BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

INFRASTRUCTURE FOR TESTING AND DEPLOYMENT OF THE REAL-TIME LOCALIZATION PLATFORM

INFRASTRUKTURA PRO TESTOVÁNÍ A NASAZENÍ REAL-TIME LOKALIZAČNÍ PLATFORMY

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MICHAL ORMOŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. VLADIMÍR VESELÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Department of Information Systems (DIFS) Academic year 2019/2020

Master's Thesis Specification

22980

Student:	**Ormoš Michal, Bc.**
Programme:	Information Technology Field of study: Information Technology Security
Title:	**Infrastructure for Testing and Deployment of the Real-Time Localization Platform**
Category:	Networking

Assignment:

1. Study the real-time location system (RTLS) architecture and the Continuous Integration (CI) and Continuous Development (CD) concepts.
2. Design CI / CD architecture for server-side software of the RTLS in one of the available cloud services. Focus also on source code style checking, include functional and stress tests.
3. Design appropriate methods to interpret data assessing accuracy, stability and performance of the RTLS.
4. Design and implement software to simulate or replay network data from RTLS anchors.
5. Implement the selected CI / CD architecture. Apply and compare its results on various server-side software commits.
6. Evaluate the outputs and discuss possible future work.

Recommended literature:

- Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps Handbook:: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution.
- Duvall, P. M., Matyas, S., & Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education.
- Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education.
- Sewio documentation web, http://docs.sewio.net

Requirements for the semestral defence:

- Items 1 to 3 completely + item 4 partially.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Veselý Vladimír, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2019
Submission deadline:	June 3, 2020
Approval date:	October 29, 2019

Abstract

This work is a case study of continuous integration and continuous deployment of real-time localization software. The objective of this thesis is to accelerate this process. The selected problem is solved with conventional testing tools, mechanisms of CI/CD Gitlab, and newly created tools for generating network traffic of real-time localization software. The contribution of this work is the acceleration of development, assurance of quality and at the same as the introduction of new methods on how to test real-time localization platforms.

Abstrakt

Táto práca je prípadovou štúdiou postupného vývoja a nasadzovania lokačného softwaru v reálnom čase. Cieľom tejto práce je zrýchliť tento proces. Zvolený problém bol vyriešený s konvenčnými testovacími nastrojmi, vlastným nástrojom pre generovanie sieťovej prevádzky lokalizačnej platformy a nástrojmi CI/CD Gitlab. Prínosom tejto práce je zrýchlenie vývoja, zaručenie kvality vyvijaného softwaru a predstavenie spôsobu ako platformu pre lokalizáciu v reálnom čase testovať.

Keywords

RTLS System, UWB, CI/CD, Indoor Localization, DevOps, Testing

Kľúčové slová

RTLS systém , UWB, CI-CD, Indoor Lokalizácia, DevOps, Testovanie

Reference

ORMOŠ, Michal. *Infrastructure for Testing and Deployment of the Real-Time Localization Platform.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vladimír Veselý, Ph.D.

Rozšírený abstrakt

Rýchly vývoj a rýchle nasadzovanie nových verzií sytémov a aplikácií je v oblasti vývoja softvéru fenomén dnešnej doby. Spoločnosti sa predbiehajú kto je schopný rýchlejšie a kvalitnejšie nový produkt vyvinúť, a bez komplikácií nasadiť. K tomu prispievajú stále lepšie technológie v oblasti nasadzovania softwaru pomocou cloudov. Vývoj viazaný na určitú technológiu alebo počítač sa pomaly stáva minulosťou a dopredu sa dostáva vzdialené automatizovanie všetkých procesov.

To všetko by nemohlo byť realizované bez stále rozvíjajúcej sa oblasti automatického testovania, ktoré poháňa rýchlosť týchto inovácií. Každý produkt softvérového vývoja spadá do kategórie automatických testov, ktoré sú pre dané potreby vyvinuté a upravené . Avšak, v každom produkte sa nájde časť, kde vytvorenie automatických testov nie je triviálna záležitosť a žiada si vytvorenie testov a ich logiky od nuly. To v každom prípade vyžaduje dobrú znalosť problému, ktorý chceme testovať a dávku kreativity.

Táto práca sa v prvom rade snaží vysvetliť a uviesť do praxe princípy postupného vývoja a postupného nasadzovania softwaru na vybranom projekte z oblasti lokalizácie v uzavretých priestoroch v reálnom čase. Celý proces vývoja sa snaží analyzovať a zaviesť praktiky spojenia vývoja softvéru a operácii nad jeho nasadzovaním, princípu DevOps.

Cieľom tejto práce je čitateľovi predstaviť a vysvetliť technológiu lokalizácie v uzavretých priestoroch predstavením konkrétnej technológie, na ktorej je práca postavená. Analýzou tejto inovačnej technológie, ktorá je aktuálne v produkcii, avšak pre konkurenciu schopnosť potrebuje svoje produkty vyvíjať a hlavne doručovať v čo najkratšom čase, sa snažíme zaviesť princípy DevOps.

Práca je zameraná výhradne na softwarovú časť tohto produktu, ktorá ruka v ruke pracuje s hardwarovou časťou. Tá potrebuje byť správne rozložená, nastavená a obsluhovaná, preto je overenie fungovania softvérovej časti veľmi náročné. Práve preto som v tejto práci vyvinuli spôsob, akým tento hardware abstrahovať a preniesť do softvérovej podoby tak, aby zodpovedal v správaní svojej hardwarovej reprezentácií.

Vytvorenie testovacej sady pre takýto software začne prvotnou analýzou akým spôsobom je tento software aktuálne vyvíjaný, testovaný a doručený zákazníkovi. Hlavný spôsob akým je software testovaný sú manuálne testy. Každý tester, ktorý tento software testuje a validuje ho musí dobre poznať, aby vedel určiť, či správanie, ktoré vykazuje je správne. Celý tento proces musí byť nahradený automatickými testami.

Aplikácia sa skladá z viacerých častí, kde sa zameriam na štyri z nich. Celá aplikácia je postavená na databáze, ktorá ukladá dáta. Tá poskytuje otvorené API pre spojenie lokalizačnej aplikácie napríklad s ďalšími aplikáciami užívateľa. Pre všetky scenáre použitia tohto API musia byť vytvorené testovacie sady, ktoré budu kontrolovať, že na správny požiadavok odpovie systém očakávanou odpoveďou.

Software je vyvíjaný len pár vývojármi, avšak ich počet narastá rovnako aj počet ľudí, ktorý pracujú na projektoch. V takom prípade do projektu musíme zaviesť štandard, ktorý každý vývojár musí dodržiavať. Tento štandard bude pozostávať z pravidiel akým zdrojový kód písať a aplikovať. Tieto pravidlá budú následne pri zasielaní zdrojových kódov automaticky vynucované a vývojár v prípade nesplnenia týchto pravidiel bude musieť kód požadované upraviť.

Hlavnou časťou celej aplikácie je lokačný algoritmus, ktorý na vstup hardwarovej časti vo formáte signálov analyzuje, vyhodnotí a prepočíta dáta, ktoré následne zobrazí vo výslednom pláne v Karteziánskej sústave súradníc. Tento proces je najnáročnejší vzhľadom na stovky dát za sekundu, ktoré musí systém spracovať a vyhodnotiť. A rovnako je aj pre

celý systém kľúčový. To je časť, kde nám komerčné testovacie princípy nepomôžu a musíme vymyslieť vlastné spôsoby testovania.

Prvou možnosťou je vziať existujúce nahrávky sieťovej komunikácie z reálnych inštalácií a následne ich prehrať aplikácii odznova. Týmto spôsobom nájdeme cestu ako oddeliť nutnosť testovať softwarovou časť len v závislosti od časti hardwarovej. Samotným prehrávaním komunikácie, ktorú už raz software spracoval a vieme, že dopadla úspešne mu môžme tieto dáta dookola prehrávať a testovať jeho rekcie. Tým zaručíme, že naprieč novými verziami a vylepšeniami bude systém lokalizovať rovnako dobre ako jeho predošlá verzia na ktorej bola táto komunikácia prvotne vytvorená. Tento spôsob je avšak limitovaný nemenosťou referenčného prostredia a nevedomosťou o reálnych vstupných dátach.

Preto zavedieme aj druhý spôsob, kde nebudeme brať dáta z reálnej komunikácie, ale tieto dáta si sami vytvoríme, teda vygenerujeme. Týmto spôsobom budeme hardware plne abstrahovať. Úspešným vytvorením takéhoto generátora sme schopný navrhnúť vlastné trajektórie, ktorých presné pozície poznáme. Tieto následne pretransformujeme do UWB signálu. A to spôsobom inverzným k spôsobu akým samotný systém tento UWB signál transformuje a tým pádom vytvoríme vierohodnú imitáciu reálnej lokalizácie. Druhý spôsob nám prinesie možnosť implementovať rôzne porovnávacie metódy a tým automaticky určovať presnosť systému oproti vstupu. Na to používame rôzne štatistické metódy. Týmto vieme presne určiť ako sa systém zhoršil alebo zlepšil.

V neposlednom rade budeme musieť prinútiť každého vývojára aby si k svojim vyvíjaným aplikáciám písal vlastné testy, ktoré kontrolujú jeho úvahy a algoritmy. Pre tieto testy bude vývojár musieť poznať presné zadanie a požadované vstupy a výstupy. Ideálnym prípadom bude, ak si napíše test ešte predtým, ako začne jednotlivý modul vyvíjať.

Záťažové testy sú poslednou a nevyhnutnou časťou našej testovacej sady. Ich úlohou bude generovať čo najväčšiu záťaž, či už na samotné API rovnako ako aj lokalizáciu. Týmto zistíme hranice systému a rovnako zaručíme, že naprieč verziami sa jeho výkon nebude rapídne zhoršoval, ideálne iba zlepšovať.

To všetko v závere implementujem do verzovacieho systému GitLab do jeho automatického zreťazeného spracovania pre postupné nasadzovanie a vývoj. Bez tejto implementácie by nám bol celý proces nanič. Hlavnou výhodou bude práve nezávislosť testov od prostredia a ich automatické spúšťanie pri každom zásahu do zdrojových kódov vo verzovacom systéme. To prinúti vývojárov aby odovzdávali vždy funkčný kód do hlavnej vetvy verozvacieho systému. Avšak celá testovacia sada bude pripravená aj na spúšťanie v lokálnom prostredí. Samotná implementácia aplikácie do verzovacieho systému a jej aplikačné testovanie tvorí rovnako náročnú a zaujímavú časť, keďže samotná aplikácia nepatrí medzi štandardné aplikácie ale je tvorená viacerými časťami, ktoré pracujú medzi sebou a predávajú si dáta.

Všetky vytvorené nástroje a spôsoby sú v záverečnej časti validované a testované v praxi. Týmto spôsobom je dokázaná ich funkčnosť a tým zaručená dôvera v ich správne fungovanie.

Celý tento proces nám pomôže zrýchliť vývoj a nasadzovanie lokalizačného softwaru vďaka redukcii času potrebného na jeho vývoj a testovanie. Rovnako nam pridá dôveru pri jeho vývíjaní a zabezpečí kvalitu naprieč verziami.

Infrastructure for Testing and Deployment of the Real-Time Localization Platform

Declaration

I hereby declare that this Master thesis was prepared as an original work by the author Bc. Michal Ormoš under the supervision of Ing. Vladimír Veselý Ph.D. The supplementary information was provided by Sewio Networks s.r.o I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Michal Ormoš
June 2, 2020

Acknowledgements

I want to express my thanks to employees of Sewio Networks s.r.o, namely Ing. Ľubomír Mráz, Ing. Tomáš Kočan and Ing. Jiří Zveřina, who helped me with their knowledge about the task I worked on, and allow me to create a thesis on that exciting and rising topic. I also would like to thanks my supervisor Ing. Vladimír Veselý, Ph.D., who help me lead this thesis through the academic environment, his guidance and feedback highly contribute to the quality of this work. Lastly I would like to thanks my family and closest friends for support they created in my academic path. As acknowledgement to all this people I would like to share my favorite banana bread recipe. Preheat oven to 165 degrees Celsius. Grease loaf pan. Combine 3 bananas, 1 cup of sugar, 1 egg, and 1/4 cup of melted butter together in a bowl. Mix 1.5 cup of flour and 1 teaspoon baking soda together in a separate bowl; stir into banana mixture until batter is just mixed. Stir 1 teaspoon of salt and small pieces of chocolate into batter. Pour batter into the prepared loaf pan. Bake in the preheated oven until a toothpick inserted in the center of the bread comes out clean, about 1 hour.

Contents

Chapter 1

Introduction

Doing work more quickly and efficiently is the desire of every person or company. Most of our time in our daily lives, we are doing repetitive work. How much more time we may have for creative things if we could find a way how to stop doing everyday work repeatedly by passing it to someone or something.

The objective of this thesis is exactly aiming for this goal in our professional programming life. We all know this as a combination of words Development (Dev) and Information-Technology Operations (Ops). What if we could find patterns in our daily repeatedly work and put them to routines in our computers, so they could perform the tasks instead of us every time we want to.

In every company and every stage of product development, things always start to be more complex, more complicated, and work harder to manage. Then we have to find new ways how to push work further and make it more efficient without losing quality. Today in every field of our modern society, we are trying to optimize everything from industry to how we use our planet. Companies and nations are trying to put prices down, get more from production and use fewer resources for the same amount of product. If we do not push these goals, we start to be obsolete and reducible.

Exactly this has motivates me to start being interested in the field of software automation. Nevertheless, this is already a well-known principle in computer science that is now well documented in every possible part. Especially accurate localization on our planet has become an inseparable part of our lives. We use GPS in outdoor environment daily, in the early stages of the industry and later for personal use. Except for indoor localization under the roof, where the GPS cannot help. Here come the UWB and other sophisticated technologies that supplements GPS with maximum precision.

How we can guarantee the quality of something that abstract as a location anywhere on our Earth? This is precisely the goal of this thesis. Create a valuable and credible test suite of the indoor localization system with continuous integration and validation of that system across the new versions and releases.

Chapter 2 describes the theoretical beginning of this thesis, where all software and hardware concepts of testing, localization, and UWB technology is explained, which are crucial for understanding the solution proposal in Chapter 3. The results are gathered in Chapter 4 and validated in Chapter 5. Chapter 6 summarizes this case study and creates suggestions for future continuation of this topic.

Chapter 2

Theory

As software and hardware products become more complex every year, and user requirements demand it to be as robust as possible day to day, in other words, high-quality code in shorter lead time. The testing principles of information technology transfer from decade to decade to more comprehensive approaches and with enthusiasm to say - philosophies.

In this chapter, We describe the underlying theory that needs to be comprehended by the reader to understand the thesis in its next chapters. The chapter starts with a brief introduction of the outdoor localization principles in section 2.1 follows with the detailed explanation of indoor localization principles in section 2.2 with explaining all necessary concepts of TDOA 2.2.3, RTLS 2.2.1 and presenting the technology used in this thesis in section 2.3. We take apart the technology in section about Anchors 2.3, Tags 2.3 and RTLS Studio 2.3. The explanation of DevOps principles in section 2.5, and CI/CD basics in sections 2.5, lastly to fundamentals of testing in section 2.6.

2.1 Outdoor Localization

The ability to localize people and assets has become an essential and indispensable part of our life from the start of the 21st century. Localization services, mobile advertising, safety, and security applications are only a few examples from a wide variety from the utilization of localization [10].

From the beginning, we will explain the principles of the outdoor localization system. These services are great and easy to comprehend to every reader of this work. We will describe how this system works and where it has its limits. Afterward, we will connect this theory with an indoor localization system and point out where and how it supplements the outdoor localization system. Understanding of outdoor localization system will give us better knowledge for easier understating of the indoor localization system.

GPS

The GPS service, which was launched in 1979 for the US military and has been fully available to the world since 1994, is now so much embedded into our infrastructure that it is called the world's utility service. Even the most people think that GPS is there to act as some navigation aid to find your way in a car, it also has found its way to almost everything that moves and other vital areas on our globe, including the electrical power grid, banking, and mobile communications [39, 43].

At the center of all of this is time, all the GPS systems, namely original US GPS version[1] or the Russian GLONASS[2], or European Galileo[3] and other purpose national systems. All work in the same principle, which is in essence, an incredibly accurate synchronized clock in the constellation of satellites spread around the globe, sending our trimming signals, which are received by GPS receivers, like the ones in your smartphones. It is the accuracy of these timing signals, which is the key of localization, and also makes very useful for so many applications [43, 14].

Each one of these satellites has an precise atomic clock on board, which is synchronized to the master clock at the Naval observatory to within ten nanoseconds. Each satellite broadcasts continuous signal which gives its timestamp and the position in space. Each satellite has to be synchronized every day to the Master ground clock [14].

To give an idea, we show an example, how crucial this timing is for precision. If there was and timing error of 1 microsecond, your car navigation would have a failure of the region of 330km [1].

Whenever on earth, you are, there are always around six satellites in direct line of sight, but only 4 with the strongest signal are used by GPS receiver [10]. GPS receiver then uses the satellites known location information and the delay in the time it takes for the signal until it arrives at the receiver of each satellite to calculate the position of the receiver. The accurate the timing the more accurate the position will be. From the most reliable references, we can determine this accuracy even as precisely as from 500cm to 30cm [1].

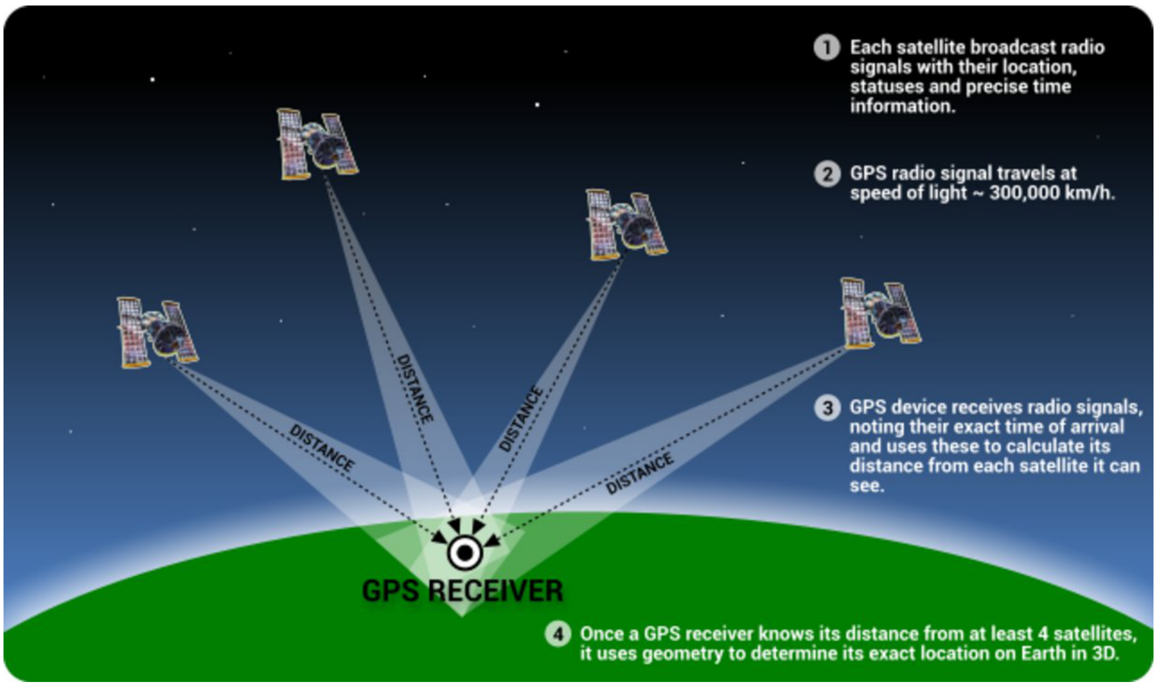

1 Each satellite broadcast radio signals with their location, statuses and precise time information.
2 GPS radio signal travels at speed of light ~ 300,000 km/h.
3 GPS device receives radio signals, noting their exact time of arrival and uses these to calculate its distance from each satellite it can see.
4 Once a GPS receiver knows its distance from at least 4 satellites, it uses geometry to determine its exact location on Earth in 3D.
DISTANCE
DISTANCE
DISTANCE
DISTANCE
GPS RECEIVER

Figure 2.1: Illustration of how GPS System works [39]

GPS is widely accepted as a universal positioning system. In most conditions, it is accurate, reliable, and available. It has supported the tremendous growth that we have seen in Location-Based Services (LSB) [26].

[1]https://www.gps.gov/
[2]https://www.glonass-iac.ru/en/
[3]https://www.gsa.europa.eu/

The full adoption of GPS in mobile devices, along with wireless networks, solved many problems of outdoor localization. However, it involves only outdoor applications. Moreover, there are a large portion of areas and applications, which require the knowledge of indoor user position [10].

Many technologies and breakthroughs have been investigated in the field of positions methods, and technologies to bridge the gap and bring positioning indoors. Such as a combination of A-GPS, accelerometer, and magnetometer, Bluetooth, Ultra-wideband, ZigBee, GSM, and others [6, 26].

2.2 Indoor Localization

In this section, we will introduce an indoor localization and Local Position System (LPS). We will briefly present Real-Time Locating Systems (RTLS) 2.2.1 and it's fundamentals. We will next talk about Local Positioning System (LPS) 2.2.2 as a superset of RTLS. And we will introduce the most important representatives namely Radio Frequency Identification (RFID), Wi-Fi and UWB in section 2.2.2. At the end of this section we will talk about main principles to understand these technologies particularly Time Difference of Arrival (TDOA), Received Signal Strength Indicator (RSSI) and Line of Sight (LOS) in section 2.2.3.

2.2.1 RTLS

A Real-Time Location System (RTLS) enables you to find, track, manage and analyze where assets or people are located. Parts of RTLS [27]:

- Tags

  A mobile device with location technology that we localize. Small enough that it can be attached to an asset or carried by people.

- Location sensors

  It is a device with a known position, which is using location sensors to locate the Tags that are attached to the assets.

- Location engine

  Software that communicates with Tags and location sensors to determine the location of the Tags.

- Middleware

  It is the connection between our RTLS technology components and business applications capable of exploring the value of this technology.

- Application

  It is the software that interacts with RTLS middleware and does the work end-users interaction.

For simplification, in the beginning, we can imagine the RTLS system as Global Positioning System (GPS). However, RTLS is a form of a local positioning system, and do not usually refer to GPS or to mobile phone tracking. The satellites of the GPS system acts as location sensors and your smartphone acts as a Tag. The Google Maps in our smartphone, which navigates as on the road home, act as RTLS Application.

2.2.2 Local Positioning System

A local positioning system (LPS) opposite to global positioning system (GPS) is a navigation system that provides location information anywhere within the coverage of the network in all conditions. If there is an unobstructed line of sight to three or more signaling devices of which the exact position on the place is known. A particular type of LPS is the real-time locating system (RTLS), which also allows real-time tracking of an object or person in a confined area such as a building [22].

We can recognize the need for RTLS technologies around many industries as tagging assets, tagging people, healthcare, manufacturing, automotive, aerospace, defense, retail, industry, education, safety, research and development, and so on. Brief introduction of RTLS technologies is shown in table 2.1 and in the next section we explain some of the in details.

Wireless technology	ZigBee	Wi-Fi	RFID		UWB
Frequency	868/915MHz 2.4GH	2.4GHz	125KHz–915MHz	3.1GHz–10.6GHz	3.1GHz–
Indoors accuracy	***	**	*	*	****
Detection range	***	**	*	**	*
Tag cost	**	**	***	**	*
Total cost	***	**	**	**	*
Ease of deployment	****	**	*	*	*
Tags autonomy	***	*	*****	**	**
Tags size	**	**	***	**	**
Security	**	***	*	*	**

Table 2.1: Comparison between indoor RTLS technologies [6].

RFID

Radio Frequency Identification (RFID) is a technology used for the automatic identification and tracking of assets and people. It requires to have an RFID Tag attached to each asset or be worn on each person we want to track and RFID readers to be installed in the mobile devices. A typical RFID system consists of three parts [23].

- Transporter (Tag)

  Holds the data on the object or person to which they are attached. Usually is containing unique code for identification.

- Reader (Location sensor)

  When the transporter is within an appropriate range of a reader, the transporter transmits data to the reader using the radio. Each reader covers a specific zone through its radio frequency signal, known as a reading field. When a Tag passes through the reading field of a particular reader, it is said that Tag is in that zone.

- Controlling Application (Engine with middleware and Application)

  After the reader receives the signal, it decodes it to the digital information, which is then relayed to a computer application that makes use of it.

This technology is widely used in many different applications such as security systems, public transports payment systems, tracking of goods, and livestock verification, etc. [23, 6].

We can most readily imagine RFID usage as a security method against theft in every store where the readers are pillars in the shop entrance, which are also our virtual zone. Transporters are located in each shop's valuable item as a small metal barcode, which looks like a circuit.

Wi-Fi

Location systems based on Wireless Fidelity (Wi-Fi) take advantage of Wi-Fi WLANs (Wireless Local Area Networks), working in the $2.4GHz$ and $5.8GHz$ ISM (Industrial, Scientific, and Medical). This technology uses modulated Wi-Fi transmission signals to detect the presence of a device. This system can triangulate the position of the device based on the signal received from the other access points (AP) [23, 6]. It uses two key features.

- The probability distributions to enhance accuracy to filter the noise nature of the wireless channels.
- Clustering of map locations to reduce the computational requirements.

The most common localization technique used for positioning with wireless access points is based on measuring the intensity of the received signal (Received Signal Strength Indication or RSSI), explained in detail in section 2.2.3 [44]. Typical parameters used to geolocate the wireless access point include its Service set identifier (SSID) and media access control address (MAC address).

UWB

Ultra-Wideband (UWB or ultraband) is any radio technology that has bandwidth exceeding 500 MHz or 20 percent of the arithmetic center frequency, whichever is lower. This makes it a good solution for near-field data transmission. Also, the high bandwidth and extremely short pulses waveforms help in reducing the effect of multipath, which makes UWB a more desirable solution for indoor positioning than other technologies [2, 36, 15]. UWB is a carrierless communication scheme. The early applications of UWB technology were primarily related to radar. A UWB-based locating system is very much like any other RTLS except that it uses UWB signals [27]. UWB is one of the most recent, accurate, and promising technologies. It is an emerging technology in the field of indoor positioning that has shown better performance compared to the other methods we mentioned above [2].

2.2.3 Main principles of these technologies

Now we will discuss technological principles connected with mainly GPS and UWB systems. Understanding these principles is not crucial for understating the RTLS systems itself. However, later in this thesis, we will look deeper into UWB location systems, and these principles will be crucial for the understating our test suite.

Triangulation vs Trilateration

For given techniques, several algorithms can be used to compute the position. Despite how GPS receivers are often confused with Triangulation (which measures angles), they do not use angles at all. They use trilateration, which involves measuring distances [27]. UWB based RTLS system can use both trilateration as well as triangulation principles.

- Trilateration

  It is a technique in which we can estimate the position of something if we know its distance to three different locations. When we use more than three locations, it is called multilateration.

- Triangulation

  It is a technique in which we can estimate the position of the object if we know the line angle between that something and three different static locations concerning a common reference line.

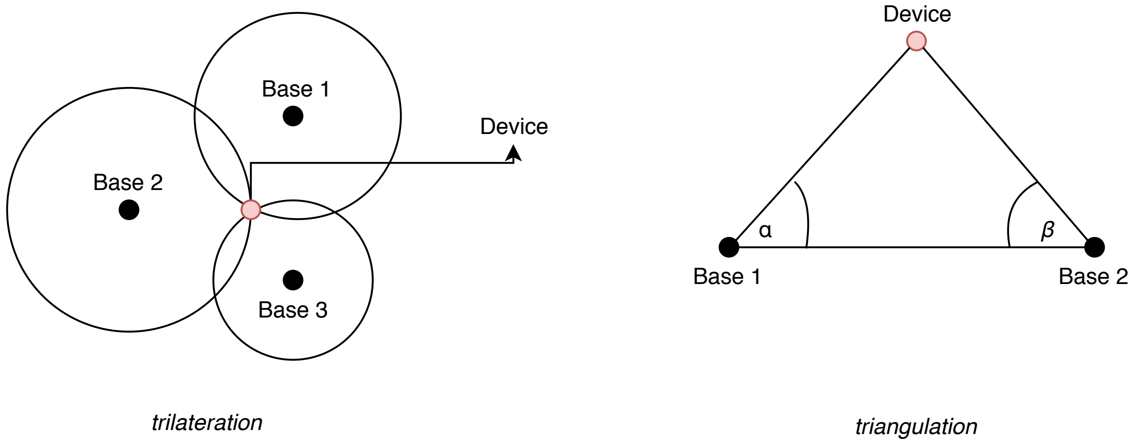


Figure 2.2: Differences between trilateration and triangulation.

Time Difference of Arrival

Time Difference of Arrival (TDOA) measures the difference in transmission times between signal received from each of the transceivers to a receiver or vice versa. TDOA is used to estimate distances between the Tag and location sensor, although the distance does not matter, it is required for TDOA measurements. That means all location sensors must be exactly time-synchronized. Otherwise, location ambiguity will occur [27, 42].

Received Signal Strength Indicator

Localization algorithms play a significant role in enhancing the utility of collected data. Much of the research work in this area assumes received signal strength indicator (RSSI) as a parameter in their localization system [27]. Typically RSSI is a measure of the power present in the received radio signal in dBm, which is ten times the logarithm of the ratio of the power (P) at the receiving end and the reference power (P_{ref}). Power at the receiving end is inversely proportional to the square of the distance [33, 26]. Because of the power levels at the start of transmission, RSSI can be used to estimate the distance signal traveled [42].

Many other UWB localization algorithms have been developed over the past few decades, time of arrival (TOA), and also the angle of arrival (AOA). The TDOA and RSSI estimation algorithms are more commonly used in the current UWB localization system due to its advantages of simple principle [42].

Line of Sight

There are many aspects that make indoor positioning different from outdoor positioning. In comparison with the outdoor environment, the indoor environment is more complex because there are many objects (such as walls, people, machines) that reflect signals and lead to multi-path and delay problems. Furthermore, due to the existence of various objects, indoor environments typically rely on Non Line of Sight (NLoS) propagation in which signals cannot travel directly in a straight route from an emitter to a receiver which causes inconsistent time delays at the receiver, as is shown in Figure 2.3 [2].

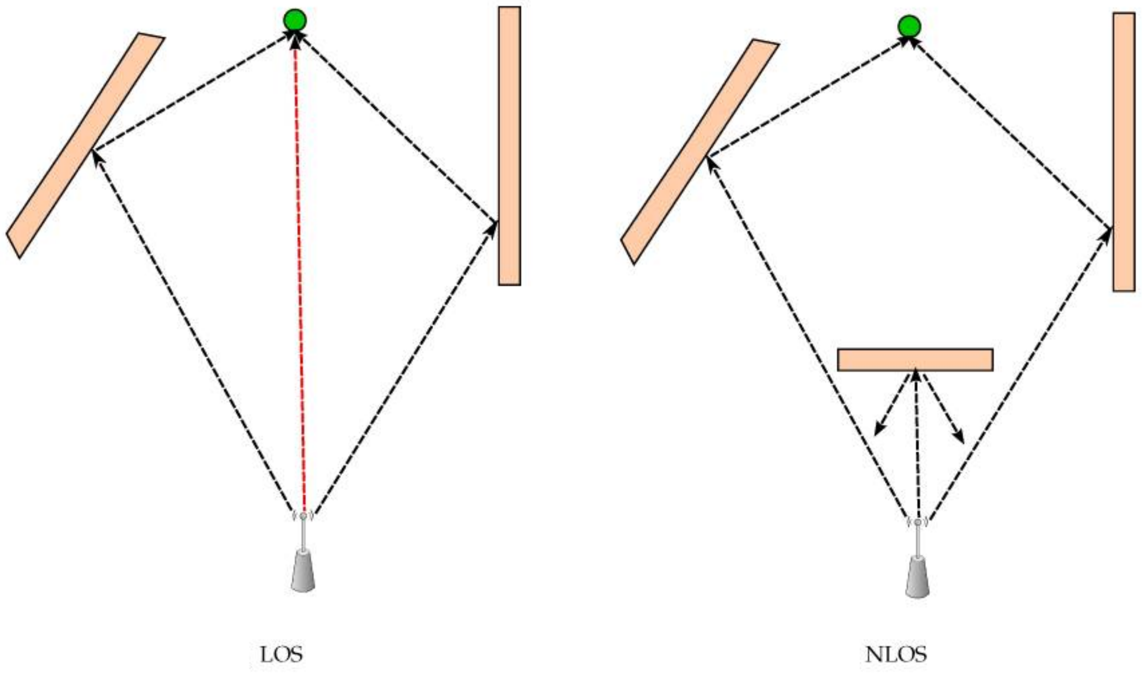


Figure 2.3: Line of sight (LOS) on the left part of the image. Non line of sight on the right part of the image. The green dot on a top is a receiver. The small station at the bottom is the emitter. The red line represents a direct path, LOS. The black lines represents multipath [2].

2.3 Existing Solutions

Hereafter in this document, we will be talking only about indoor UWB based localization systems. We will further skip RFID, Bluetooth, Wi-Fi, and other technologies in this thesis. UWB is by table 2.1 the most promising RTLS technology based on localization accuracy, cost, ease of deployment, autonomy, size, and security. RTLS solutions for industry and personal based applications around UWB happen to arise in recent years. There are several companies engaged in localization applications with real business and industry 4.0 accomplishments. As the technology adoption life cycle characterizes the start of the Slope of Enlightenment, figure 2.4. We can choose a few points that describe in which stage are RTLS companies with their UWB solution [4]:

- The market has already started to mature about this technology and be more realistic about how technology can be useful to organizations.
- Many companies have already commenced using the technology, and more organization fund pilot projects. There are more instances and real case studies on how new technology can benefit the industry[4].
- Second and third-generation products appearing.

We can say with faith that this technology is in the stage of early adopters. There are starting to be more and more companies in this field. Even big corporations are starting to formulate the teams, whose concern is creating technologies based on UWB, as future innovation and successful ideas for this companies. GPS is here from the 1960s, RFID from the 1970s, and now the UWB arrives.

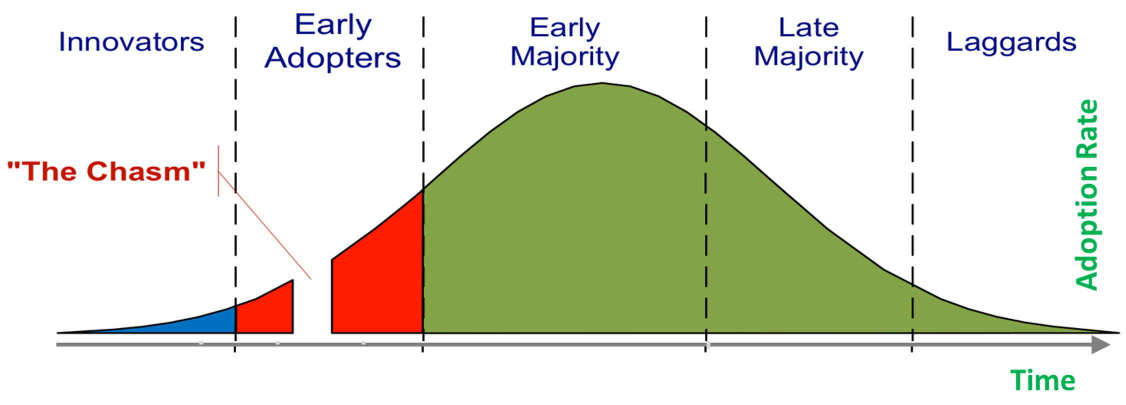


Figure 2.4: The “Technology Adoption Lifecycle”, with the current position of Model Driven Engineering indicated by a thick blue line [16].

We decided to choose and collaborate with one of these companies, and it is Sewio Networks s.r.o, which is the Czech Republic, Brno based company acting in the global market. RTLS is for precise indoor tracking. Using technology based on ultrawideband TDoA, Sewio gives you the RTLS hardware and software you need to gain accurate and actionable data and be more productive, cost-effective, and safe [38]. We will describe Sewio technology in further paragraphs, proposal and implementation are based on their technology. From theory we acquired in 2.2.1, we can define the Sewio environment into a few parts.

- We have **UWB Anchor**, which acts as Localization sensor 2.2.1. They are physical devices based on RF radio, which are connected either by Ethernet or wireless network to RTLS Application. They are immobile devices with the exact location; the location is determined in advance. They are powered up from constant power sources as PoE or USB. As Sewio documentation says [38]: *„Anchors are electronic devices that*

[4]https://www.sewio.net/customer-projects/

detect UWB pulses emitted by UWB Tag and forwarded them to the location server for calculating Tags positions. To cover the area with an indoor tracking system, a set of Anchors needs to be installed above the area to create the location infrastructure.“

- The succeeding part of the environment is **UWB Tag** . These are specifically Tags as we describe them in our theoretical section 2.3. They are small devices that are designed to be worn by people or attached to industrial machinery. Batteries usually power Tags. As Sewio documentation describes them[38]: *„Tags are small electronic devices that are attached to objects that need to be tracked. Same as a smartphone in a GPS environment. The Tags send out blinks that are received by Anchors and forwarded to the location server for calculating the Tags' position“*
- As of the last part of Sewio technology, we will describe RTLS Studio, which acts as location engine 2.2.1, middleware, and partly as final customer application. We will explain this in more detail later in this section, because this will be the aim of this thesis implementation.

From here until the end of this work if we will mention Tags, we will mean UWB Tags. The same as we mention Anchor, we will mean UWB Anchors.

RTLS Studio

The RTLS Studio is the leading software part of the RTLS Application, with many components within it. Nevertheless, it consists of three fundamental parts to use the RTLS Application as a localization platform.

- **RTLS Manager**, as its name suggests, is the manager for RTLS system, namely Anchors, and Tags. It directly communicates with Anchors as data input of their UWB data stream through UDP or TCP on ports 5000 or 5001, respectively. The central task of the RTLS Manager is to parse and preprocess UWB packets coming from Anchors, get rid of useless information, and choose only packets crucial for localization. It is also communicating with Anchors over their HTTP interface, where it can update their settings and control them in an accessible manner. Through the Anchors, Manager can also change the settings of Tags.
- After the UWB packets are parsed, preprocessed, and cleaned of useless data. They are then passed to **RTLS Server**. It has only one crucial function depend on localization, and it is to recalculate real position based on gathered UWB information. RTLS Server has many settings for UWB specialized algorithms and functions that are too complicated to describe in this thesis (if the reader has concern for this type of information, he can study them further in the documentation mentioned above). Vigour for the RTLS server is recalculating all the positions he gathers correctly and fast as possible. More Tags we have, more challenging this task is.
- They are then passed to **RTLS Sensmap** for showing data in Cartesian coordinate system for Application for final use in 2D or to Sensmap 3D for their representation in 3D.
- When all this is once processed the data and position history are stored in the main system **database**. This database also includes Open REST API for interconnection with commercial applications.

Furthermore, all this is strengthened with other RTLS Studio applications, which are not crucial for running the system but they are helping to analyze data and manage the system as a whole. Also, there are contributing to system usage. They are:

- SAGE for showing analytical data with many combinations.
- RTLS IO for connecting external HW devices for signal output in the mean of light, sound or signal.
- RTLS Monitor for monitoring system operations and performance.

2.4 DevOps

The last few years have seen much discussion of what the DevOps is and what the DevOps is not. DevOps is as much about culture than it is about tools, and the culture is all about people. Neither two groups of people guarantee to create the same sort of software in the same circumstances. Word DevOps, which stands for Development and Operations, is being more and more used in the field of creating software. DevOps is not a simple tool that could be easily explained or shown, so for a start, We would like to define what is not a DevOps [18, 41].

- DevOps is only for Open Source Software.

  Despite many success stories, we hear from the legendary open-source software projects, like Linux, PHP, MySQL, and others. By them, we imagine a massive base of autonomous contributors, by which independent lines of code are linked as one entity. Nevertheless, we can find similar success in an application written in Microsoft.NET, mainframe assembly code, and others, even with embedded systems lead by traditional software companies [20].

- DevOps is only for a start-up.

  For the nature of testing, we imagine brand new software, which is starting to be built from scratch, so naturally, we are creating some pre-written test plans before creating real production code. In the hope that this will save us more time in later stages of the project development where things will become more complex, and our DevOps will support us, such as stories of many internet 'unicorns' like Netflix, Etsy, Amazon, and others. Nevertheless, we can easily imagine a traditional organization like Google or Microsoft to transform their business to DevOps like even with their monolite and already deployed products [20].

- DevOps is only the next step and replace other principles like Agile development.

  We would rather say DevOps is compatible with Agile and complement it because its focus on small teams continually delivering high-quality code [20].

The way we deliver software is going through a wave of changes as environment operate it, as we can see in table 2.2. Market and industry needs are changing continuously, and technology is changing rapidly. The more pressure is generated to adapt to the market and delivered quickly. We can no longer afford to keep the customer waiting for new version for months or years.

Over the years, the organization adapts many processes of optimization in their software development, known as agile transformations. However, in entire evolution the focus was mainly on software development, leaving the operation side of software lagging. That is why DevOps evolves as a set of practices that are trying to bride the developer-operations gap [28].

What makes a DevOps culture. Word culture can be described by the Oxford dictionary [32] as “the arts and other manifestations of human intellectual achievement regarded collectively”. In technical terms, it is much easier to talk about what kinds of tools to use and how to use them. The tools itself are accessible as constructs [41]:

- Open Communication.

  Traditional technical teams interact through complex ticketing systems that require direct intervention. A team talking in more DevOps approach talks about products in life cycles discussing requirements, features, schedules, etc. [41].

- Respect.

  All team members should respect all other team members; nevertheless, they do not need to like each other. However, everyone needs to recognize the contribution of every person in the team. Respectful discussion and contribution [41].

- Trust.

  Is a big component of achieving the DevOps culture. The operation must trust that development is doing what it should do. Development must trust that QA is not just there to sabotage their successes. The Product Manager trust that Operations is going to give objective feedback and metrics for the next deployment [41].

- Incentive and Responsibility Alignment.

  Supposedly your teams are centralized around the core goal, creating an awesome product for the customers, whatever the product happens to be. Development is not rewarding for writing lots of code. Operations are not punished when code is not running as expected in production [41].

Development is representing software developers (programmers, testers, quality assurance engineers). Operations are representing experts who can put software into production and manage the production infrastructure, namely administration, database administration and networks. The gap between Development (Dev) and Operations (Ops) usually occurs at three different levels [18]:

- **Incentives gap** is a result of different goals of development and operations.
- **Process gap** has roots in different approaches of development and operations on how to manage changes, bring them to production, and manage them.
- **Tools gap** has origin in the fact that traditionally development and operations often use their own tools in their work.

One popular interpretation of DevOps is that it was created from the need to enable more productivity of developers because the numbers of developers grew, but there were not enough operations people to handle all the resulting deployment work [20].

	1970s-1980s	**1990s**	**2000s-Present**
Era	Mainframes	Client/Server	Commoditization and Cloud
Representative technology of era	COBOL, DB2 on MVS, etc.	C++, Oracle, Solaris, etc.	Java, MySQL, Red Hat, Ruby on Rail, PHP, etc.
Cycle time	1-5 years	3-12 months	2-12 weeks
Cost	$1M-100$M	$100k-10$M	$10k-1$M
At risk	The whole company	A product line or division	A product feature
Cost of failure	Bankruptcy, sell the company, massive layoffs	Revenue miss, CIO's job	Negligible

Table 2.2: The ever accelerating trend toward faster, cheaper, low-risk delivery of software [8].

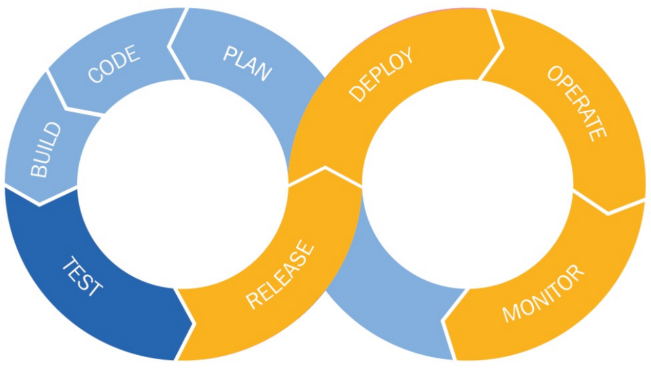


Figure 2.5: Illustration of DevOps processes we are going to implement in this thesis [11].

DevOps means culture shift toward collaboration between development, quality assurance, and operations so we can deliver software in high industry demand speed but with superior quality at the same time. DevOps can be applied to very different delivery models but must be tailored to the environment and architecture of the product [7].

Rapid deployments mean that human testing is impossible. Only even one deployment a day was necessary; this testing load would overload human testers [24].

Benefits of DevOps [28]:

- Time to get any tests set up done in just a matter of a few minutes in comparison to user driver testing to days.
- There is no longer a need to keep dedicated HW for this matter. But the cloud can be applied with further release possible to production.
- Everything is stored in the version control repository - the entire automation is portable to any setup.
- Deployment becomes a consistent, repeatable process.

- CD model enables the developer to access production-like systems and thereby doing validation on a production-like environment.
- Apparently, the high cost for adopting this principle will return as significant cost saving as the team can share the pool of resources. As deployment is automated in almost no time, there is no need to keep resources reserved. As soon as testing is done team can release the resources.

The shared goal of DevOps [20]:

- Reduce the percentage of budget spending's on product support and unplanned work.
- Ensure lead time from code check-in to production release.
- Ensure tgat release can always be performed during normal business hours.
- Integrate all the required information security control into deployment pipelines to pass all requirements.

Furthermore, we should define design patterns for our developers to help them write code to prevent abuse, such as putting rate limits for our service and graying out submit buttons after they been pressed [20].
OWASP[5] created a great cheatsheet series on this topic which includes:

- How to store passwords.
- How to handle forgotten passwords.
- How to handle logging.
- How to prevent cross-site scripting vulnerabilities.

"In high-performing organizations, everyone within the teams shares a common goal - quality, availability, and security are not the responsibility of individual departments but are part of everyone's job, every day" [20].
To summarize this section, we conclude DevOps as it assimilates development and operations teams to improve the collaboration process. A DevOps Engineer will work with IT developers to facilitate better coordination among operations, development, and testing functions by automating the integration and deployment processes. The pillars of successful DevOps are continuous integration and continuous delivery (CI/CD). To establish and optimize the CI/CD development model and receive the benefits, companies need to build an active pipeline to automate their build, integration, and testing processes.

2.5 CI/CD

Term continuous integration (CI) and continuous deployment (CD) are actively connected to DevOps. As many users can misinterpret, consider DevOps eliminating traditional IT Operations, but this is rarely a case. IT Operations became as crucial as ever, they have started to collaborate with software cycles earlier and staying with the code long after it launches to production [20].

[5]https://www.owasp.org/index.php/Main_Page

In development, CI often refers to the integration of multiple branches into one master trunk and ensuring that it passes all unit tests. However, in the context of DevOps, CI also passes production-like environments and passing acceptance and integration as well as unit tests [20]. From a DevOps perspective, continuous integration, and continuous deployment gets the most press [19].

2.5.1 Continuous Integration

Continuous integration (CI) refers to integrate early; do not keep changes localized to your workspace for long. Instead, share the code with the team through the version manager and validate how code behaves continuously. Not only within the component to which it belongs but even with the product on integration level [28].

The process of integration of software is not a new problem. It may not be much of an issue on a one-person project, but as the complexity of project increases (adding more persons or complex dependencies), there is a greater need to integrate and ensure that software components work together early and often [9].

In one popular Continuous Integration article [12], Martin Fowler describes CI as:

"[...] a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly."

Features CI should have [9]:

- Connection to a version control repository.
- Build script.
- Process for integrating the source code changes.

Challenges of CI [3]:

- Mindset - Expose developer work, changing old habits.
- Tools & Infrastructure - Code reviews, regression feedback's, test automation.
- Testing - Unstable test cases, too many manual tests, preserving quality, implementation and test dependencies.
- Domain applicability - Process sustainability, the complexity of the product.
- Understanding - Unclear goals, increased pressure, different interpretations.
- Code dependencies - Coordination of integration, dead code.
- Software requirements - Requirements breakdown, deliver feature growth, the interaction of big impact changes.

Quality in means of software is always unique to a given context. Usually, customer has own right to provide his definition of quality because he is the sponsor of delivering the product. He can personally determine what he wants to spend the money on [18].

2.5.2 Continuous Deployment

Continuous Deployment (CD) means that after the developer submits his code to the version control, this code is objected to a variety of tests, and assuming these tests passed, it passes to production. These test cases must be complete enough for good coverage [24].

Automating our deployment process means [20]:

- Creating pre-configured virtual images or containers.
- Copying packages or files onto the production environment.
- Restarting application, services.
- Generating configuration files.
- Automating database migrations.

This pipeline consists of a development environment, a staging environment, and a production environment [28].

2.6 Process of Testing

Software testing is the most frequently used method for verifying and validating the quality of software. Testing is the method of executing a program or system to detect faults. It is a significant activity of the software development life cycle. It helps in developing the confidence of a developer that a program does what it is intended to do [31].

2.6.1 Automated Testing

Automated testing addresses the significant problem, without automated testing the more code we write, the more time and money is required to test our code. This is an unscalable business for any organization [20]. Automated tests mostly fall into one of three categories, from the fastest to the slowest:

- Unit tests

  They are are designed and written to test the smallest pieces of the application as possible in isolation to the other parts as function, method, class, or small interaction between them. However, they are not touching higher structures like databases, file systems, or network if the applications have one of these. As a result, they should provide assurance for the developer that their code operates as they designed [20].

  A good practice is writing unit tests after designing architecture and before writing the code itself. To ensure that we implemented it as we designed while writing the code [20]. Unit tests should run fast and stateless; if not, we can run in the following problem as people will stop using it due to time requirements [9].

  One thing that makes the test more clean is readability. It is even more important in unit tests than it is in the production code. The thins that makes the test more readable is the same thing that make code readable, we want to say a lot with a few expressions as possible [29].

- Acceptance tests

  Contrary to the smallest units in unit tests, acceptance tests should typically test application as an entity and give us the answer to our questions "Did I get what I wanted?" and importantly, "Does it do what customer wanted?". These tests should be running in a production-like mode of the application. After our software successfully passes the acceptance test, it is then available for manual testing [9, 20].

- Integration tests

  Integration tests are where we have to ensure that our application correctly interacts with our other production applications and services, usually written by another team member, whole teams, or other companies. The purpose of this level of testing is to expose faults in the interaction between integrated units. Integration testing is usually done before system testing and it comes after unit testing. Integration tests help in better test coverage and improve test gaps. These tests are more reliable and it is easy to isolate the failures. Integration tests catch system-level issues, such as a broken database schema, mistaken cache integration, etc. [9, 20].

Usually, when project pressure starts to rise due to deadline, developers stop or slow writing test cases as part of their daily work.

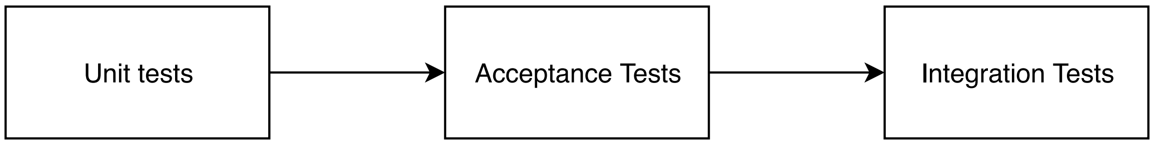


Figure 2.6: Ideal example of CI pipeline.

Additional automated tests options:

- Performance tests

  To find performance issues early, we should always evaluate performance statistics between the builds. How to write performance tests depends on used architecture and design. We will analyze this in more detail in the practical part [20].

2.6.2 Black Box Testing vs. White Box Testing

Software testing can be classified into two categories:

- Black Box Testing:

  It is a method of software testing in which the internal parts (design, structure, implementation) of the item being tested is not known to the tester. Black-box testing plays a significant role in the software testing; it overall aids functionality and validation of the system [34]. Categories of black-box testing:

  - Acceptance testing.
  - **Alpha testing** is performed by Testers who are usually internal employees. Performed at developer's site, ensuring quality of the product before moving to Beta testing.
  - **Beta testing** is performing by Clients or End Users.

  - **System testing** reveals that the system works end-to-end in a production like environment.
  - **Functional testing** is done for a finished application. This testing is to verify that it provides all of the behaviour that it is required from it.
  - **Regression testing** is a testing of the software after changes have been made. This testing is done to prove the reliability of each software release. Testing after changes has been made to ensure that changes did not introduce any new errors into the system.

- White Box Testing:

  White-box testing is also called a structural testing technique that design test cases based on the information gathered from source code. White-box tester is most often the developer of the code, who knows what the code exactly looks like and writes test cases by executing methods with specific parameters. White-box testing mainly focuses on control flow or data flow of programs [34, 25]. Categories of white-box testing:

  - Unit testing.
  - Alpha testing.
  - Regression testing.

White-box and Black-box testing are consider corresponding with each other. Many researchers underline that to test software more correctly. It is essential to cover both specifications and code actions [34, 25, 35].

When we neglect all testing practices and automated tests, it usually flows into an unstable environment where is the majority of our problems, that we discovered in production. In the results of this, we ended where we started at the beginning of the waterfall method [20].

Basic rules of testing:

- Expansive test coverage of unit tests. More the coverage, the better.
- Excellent stability of the software.
- Minimize accidental complexity.
- Compliance with system runtime quality including functionality, performance, security, availability, resilience.
- Compliance with system non-runtime qualities modifiability, portability, reusability, testability.
- Good overall cycle time.

A crucial aspect of every software engineering is measuring what we are doing. Sooner or later, we have to consider which metric is meaningful enough for development and delivery processes.

Testing Type	**Opacity**	**Specification**	**Who will do this testing?**	**General Scope**
Unit	White Box Testing	Low-Level Design Actual Code structure	Generally Programmers Code structure who write code they test	For small unit of code generally no larger than a class
Integration	White & Black Box Testing	Generally	Generally Programmers who write code they test	For multiple classes
Functional	Black Box Testing	Testing	Independent Testers will Test	For Entire product
System	Black Box Testing	Design Actual	Independent Testers will Test	For Entire product in representative environments
Acceptance	Black Box Testing	Programmers	Customers Side	Entire product in customer's environment
Beta	Black Box Testing	Code structure	Customers Side	Entire product in customer's environment
Regression	Black & White Box Testing	Changed Documentation High-Level Design	Generally Programmers or independent Testers	This can be for any of the above

Table 2.3: Testing spectrum [34].

2.7 GitLab

GitLab is a version control system for managing your code. It was first released in 2011 under the MIT license. Besides, it is a fantastic project management tool for Git projects; it also includes a GitLab Continuous Integration (CI) system. A CI system allows you to run automated unit tests on every commit; when a build is not successful, they warn you and reject the commit. It has high integration with the GitLab system itself. GitLab allows you to set up webhooks that will be triggered every time a specific event is launched. The user defines the type of the event [5, 13].

Git[6] is a free and open-source distributed version control system designed to manage everything, from small to large projects with speed and efficiency. Git is good for tracking changes in source code during software development. It is intended for coordinating work between programmers, but it can be used to track changes in any set of files. It is used to track changes in the source code. It allows multiple developers to work together. A version

[6]https://git-scm.com/

control system allows you to keep every change you make in the code repository. GitLab is a Git-based. Besides Git's functionalities, GitLab has a lot of powerful features to enhance your workflow [13].

Although, we learned that DevOps is more than just a tool, there are several tools, which are commonly used to get the desired state. One of them is the source code repository. The repository manages the many version of code developers push in, so they can collaborate on each other's projects.

Another is a build server. Building software can be done on the individual developer's workstation, but the CI pipeline dedicated build server can be used. Modern build servers, which we will also use, not just for build but they also provide advanced testing functions.

A third of them is configuration management; it is controlling the environment where CI/CD takes place. In the name of GitLab, this tool is Chef[7]. The following tool is Virtual infrastructure. Infrastructure on which software runs has always been virtual. In the cloud, virtual infrastructure is an extra layer of abstraction that represents an entire machine. Last important tool will be pipeline orchestration. The pipeline refers to an automated number of steps to get your code from inception to production after it has been checked in version control. Essentially Kubernetes[8], a platform for the roll-out and management of constraints on a large scale. It is supported by Docker[9], which means in any way, we can pack out the whole production environment in a container that could be run on any environment in the same conditions [11].

2.7.1 GitLab Runner

When you have your project along with a `.gitlab-ci.yml` file it allows you to defines what stages your CI/CD pipeline has and what to do in each stage. This typically consists of a build, test and deploy stages. Within each stage, you can define multiple jobs. For example in the build stage, you may have three jobs to build on Debian, CentOS, and Windows [11, 13]. In GitLab dictionary it means:

- `build:debian`
- `build:centos`
- `build:windows`

A GitLab runner clones the project, read the gitlab-ci.yaml file and do what he is instructed to do. So basically, GitLab runner is a process that executes some instructed tasks. When he finishes the build, he already builds data moved to the stage test. In this stage, he will run all the tests he was instructed to do and in the final stage, it will deploy. For example, create a pre-configured environment, copy packages, and set the configuration files. The output of this pipeline will be product ready to be sent to the user [13].

GitLab can also function as a container registry. By this, we mean it can store images that you create on your workstation or inside of GitLab runner pipelines [11].

[7]https://about.gitlab.com/devops-tools/chef-vs-gitlab.html
[8]https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[9]https://www.docker.com/

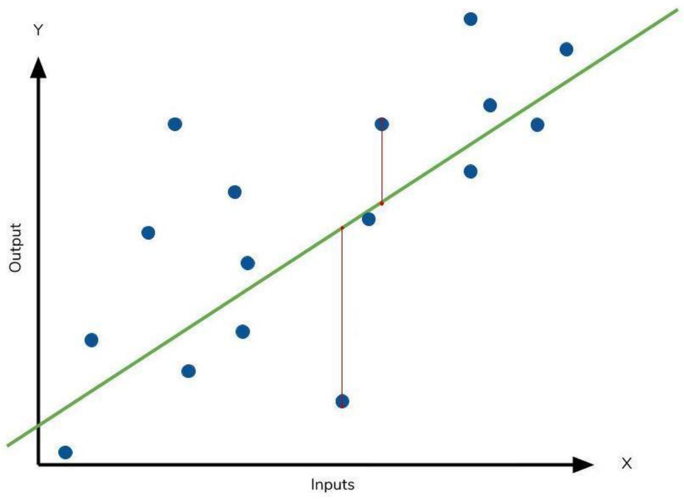


Figure 2.7: Example of forecasted values (green line) and predicted values (blue dots)

2.8 Additional Metrics

In this section, we will introduce metrics we will use in the following parts of the work and are crucial for understanding the evaluating process. Later in this work, we will talk about gathered positions representing by coordinates [x, y] in the environment. We will need some methods for the future automation of our test suite for these coordinates. Therefore we have to choose and represent some statistical methods we will later apply.

We will choose from statistical methods represented in table 2.4. Our primary requirement is the accuracy that we will later monitor and it will be our main concert. Variable $y_{predicted}$ in all equations represents predicted values, variable $y_{observed}$, observed value. Variable n in all equations means number of observations.

2.8.1 RMSE

Root-mean-square error (RMSE) (also called as root-mean-square deviation (RMSD)) is measure of the differences between predicted and observed values. In some disciplines as in our RTLS system the RMSE is used to compare differences between two data sets that may vary. RMSE is always non-negative number and a value 0, in practise almost never achieved, indicates a prefect fit for the data. In practise, the lower RMSE the better. Since the errors are squared before they are averaged, the RMSE gives comparatively huge weight to large errors. This means the RMSE is most beneficial when large errors are particularly undesirable [17, 40].

Mean squared error	MSE	$= \frac{1}{n}\sum_{t=1}^{n}(y_{observed} - y_{predicted})^2$
Root mean squared error	RMSE	$= \sqrt{\frac{1}{n}\sum_{t=1}^{n}(y_{observed} - y_{predicted})^2}$
Mean absolute error	MAE	$= \frac{1}{n}\sum_{t=1}^{n}\lvert(y_{observed} - y_{predicted})\rvert$
Mean absolute percentage error	MAPE	$= \frac{100\%}{n}\sum_{t=1}^{n}\left\lvert\frac{y_{observed} - y_{predicted}}{y_{observed}}\right\rvert$

Table 2.4: Comparison of prediction accuracy methods

2.8.2 MAPE

Mean absolute percentage error (MAPE) (also know as mean absolute percentage deviation; MAPD) expresses the accuracy as a ration defined by the formula as shown in Table 2.4. $y_{observed}$ represents the actual value and $y_{predicted}$ forecast value. The difference between $y_{observed}$ and $y_{predicted}$ is divided by the actual value of $y_{observed}$. The absolute value is then summed for every forecasted point in time and divided by a number of fitted points. Multiplying by 100% makes it percentage error. MAPE is one of the most widely used measures of forecast accuracy, due to its advantages of scale-independency and interpretability [17, 21].

Chapter 3

Proposal

The architect works with the team to select test cases. These cases must be complete enough for good coverage — both the system function and system qualities such as performance and security. Tests can consume much time, so having a minimal number of them, which provides excellent coverage for functions and qualities is critical. Choosing these test cases then becomes important [24, 28].

We already introduced, indoor localization technology of Sewio 2.3 in Chapter 2. We saw that this type of technology is not a single software application but a combination of many software applications cooperating with hardware components. Thus, in creating a test suite for this type of application, it has to be unique and created from scratch.

During the theory part, we introduced Anchors and Tags as fundamental hardware parts. They are producing localization data which creates inputs for the software applications. In this thesis, we will aim only for the test suite created for the software part of the application. By software part we mean RTLS Studio and we will leave the hardware test suite for the future. No matter this decision, an essential part of the software application tests will be this hardware input, which we have to generate to create accurate input data for testing algorithms and applications as one entity.

We already know how the system behaves, but even then, we want to keep the black-box and white-box testing equality. First of all, we want to know that the system is behaving as we designed it to behave. Furthermore, we want to be confident that it behaves correctly.

The main goal of this implementation is to ensure that releases of this localization software will keep their quality. Even the new features are added daily. We need to create the CI/CD pipeline, which ensures the swift and safe deployment of new versions. So that customers will always have the latest working version with equal or even better performance than the last version has.

We will also aim to implement ideas from DevOps principles we learned in the first sections. Namely applying CI/CD principles, dividing the roles in the team, applying the same rules and tools.

Next, in this chapter, we will discuss the actual situation of releasing new version of RTLS software 3.1 and its DevOps flow. Later we will talk about ways how we want to measure and guarantee metrics for valid testing 3.3. Lastly, we will decide which thresholds we need to set, if we want to achieve quality outputs of our software. This will lead to final test suite design representing by Figure 3.5.

3.1 Actual and Desired DevOps Attitude

The team is developing its main software localization product, RTLS Studio with other supplementary software tools for supporting indoor localization projects. We will aim only for RTLS Studio as the leading localization software.

As we have already learnt in Chapter 2.3 about RTLS Studio, it consists of closely linked software parts cooperating as a whole. These parts are regularly maintained and new features are added. Regular release is spread around the users. However, the system is mostly working in the state without the connection to the Internet, so we cannot assure regular forced updates to the newest version, if some hot-fixes are needed. Taking this fact in account, we can recognize that we have to assure the same quality of this software around releases. Ideally with newer the update the better and more precisely localization could be. Furthermore, at the same time, we have little space for errors to happen.

The tests are usually done by the user manually, which has driven access based on tester experience and information he receive. Usually the developer is the tester of his own application part. It will take a long time to test the whole RTLS system manually. Only the subjective decision of the tester is taken into account as a valid conclusion that software is behaving correctly. When the manual tests of individual testers are done, their feedback will take place, and bugs are repaired. After many iterations of manual testing of the new release, the Alpha version is created, which is deployed to internal system. After a time, this version is used, and problems or bugs are fixed, Beta version of the application is created. This Beta version is released to be chosen customers, which gives feedback on their state. After the Beta confirms proper functionality over time, the product can be released to everybody.

In the end, we can see the most time and resources are taken by initial manual testing. If the majority of these problems is found in the first stages of the testing, the more easygoing and quicker the Alpha and Beta tests will be.

As a result, we see that automated test suite is much needed. We introduced many DevOps principles in the theoretical part. Now we want to transfer them directly to the project.

We need to divide the team of developers and the team for operations correctly. We distinguish two groups of people who will develop the new features and second group, which will test the new feature. Developers usually know their outcome source code the best. They are proper for doing white-box testing of their work. However, when it comes to the black-box testing, the second person is highly needed. As a new operation person will come to the process of this new feature, he can look at it from another angle than a developer is looking. The operation person precisely knows what it should do and does not bother with the implementation. However, in this case, Operation person can do white-box testing and also try to understand the source code. However, his main goal is to black-box the new feature in the way the assignment asks, and also, his experience and imagination allow.

After we distinguish these two groups, we have to ensure the communication between these groups is fast, clean, and well documented. By this, We are proposing using GitLab Issues functionality. It is an advanced issue tracker for collaboration. Here the operation team can create issues from predefined templates and deliver the fix request back to the developer's team in a brand-new way. This will close the process and tools gap (as mentioned in Section 2.4. We have for the cooperation between the development and operations team.

Our following of DevOps principles, mainly implementing the CI/CD pipeline, will be described in the next parts. With CI/CD pipeline implemented, unit, acceptance, and integration tests should be done manually to a minimum, and the operation team should only gather the results and passed them in Issues to development team to the second iteration. The operation team should later assist in Alpha and Beta testing and create a firm connection to developers.

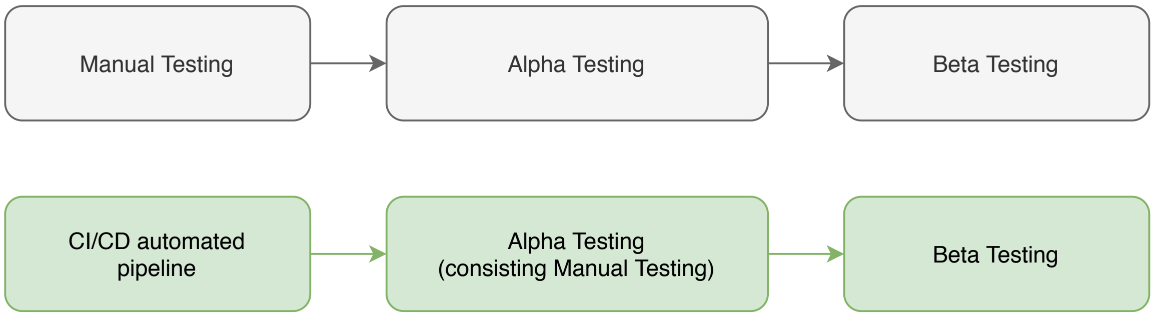


Figure 3.1: Old testing approach (above) vs. new testing approach (bellow).

3.2 Detailed Data Software Flow

For better user understanding, we describe the processing of Sewio data as follows. We have three main applications for our test suite. RLTS Manager, which is the first-line of processing data. It will receive localization data from Anchor. More accurately, packets represent localization data and process them. By processing, we mean filtration of useless information from packets and only leaving localization information, which is needed for position calculations. Then this data is sent to the RTLS Server, which will do all the calculations. Outputs of RTLS Server are localization data in the Cartesian coordinate system and other relevant information (such as timestamp and sensors data). This data is sent to RTLS Sensmap and Database for final use. Whole demonstration of this flow is shown in Figure 3.2.

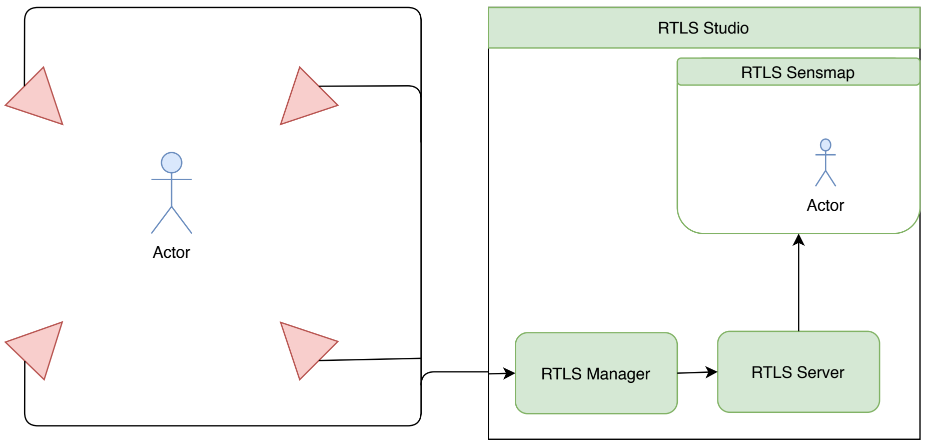


Figure 3.2: Flow of RTLS system. From real scene with Anchors (red triangles) and person wearing a Tag, through the whole system and final representation in RTLS Sensmap.

RTLS Server as the most critical part responsible for recalculating positions from UWB data and precise localization is a crucial part of the process. Created unit tests are well written, but they only validate the software within its function. Integration, acceptance, and performance tests still have to be developed.

All data are stored in the MySQL database, including localization history. Therefore, the database acts as data storage for RTLS Studio itself and a place for fetching and saving data. Nevertheless, it also provides an open REST API endpoint, to whom customers can connect and fetch data to their own applications and localization software implementation. Open REST API requires more detailed test cases for validating the behavior of every endpoint it is offering.

The system has minimal system requirements for running. However, system requirement is directly dependent on the quantity of Anchors and Tags connected to the system. More HW connected, more performance is needed. HW has its limits set by physical limitations on UWB, by how many devices can be served without the signal collisions.

As more and more developers are starting to work on the localization platform, we will also need to start standardizing code style checking. As simple as spaces vs. tabulators use further to advanced code style rules, it will be necessary to set the same standard between source files before compilation and pushing to the system.

3.3 Proposed Test Suite

As in our theory example, with GPS technology, we can explain the testing environment we have. We are moving around the city with our smartphone, and we want to use our GPS functionality to determine our location in this city. We have our real world and our movement in it, which is our first environment. The second environment is the two-dimensional copy of our world in front of the screen of that smartphone we hold. The quality of GPS service can be easily determined by the precision of position in real-world relative to the position on the smartphone screen.

The localization platform tracks the movement of assets by recording their positions in time. We can easily remodel this approach to UWB indoor localization environment. From the beginning, for the demonstration purposes, we can imagine an elementary two-dimension rectangle room, where our coordinate system is beginning in the top left corner with coordinates $x = 0$ and $y = 0$ and going to positive values to bottom right corner. Our asset, imagine it as a small ball in the room is moving around the space, this ball is representing the Tag. So the input of the system is the ball moving around the space in our space coordinate system. In our real environment, we can imagine physical ball moving around the space; in our other environment, we can imagine our imaginary ball duplicating the movement of the real ball in a precise manner. The four blue triangles are representing the Anchors deployed in the room. The red ball is our Tag.

We will be carefully watching the precision of position as our first metric. However, how can we determine precision in the automated test environment? By two manners.

- First, by generating our own input to the system, which has a predefined and well-known trajectory, by generating UWB data. Then we can compare our predefined trajectory with the real output of the RTLS system and compare how precise the RTLS output is due to our generated input. We name this method **pcap Player**.

- The second is by using well-known network recordings of a system from real installations and comparing them. If we already know where the Tag has moved in the real

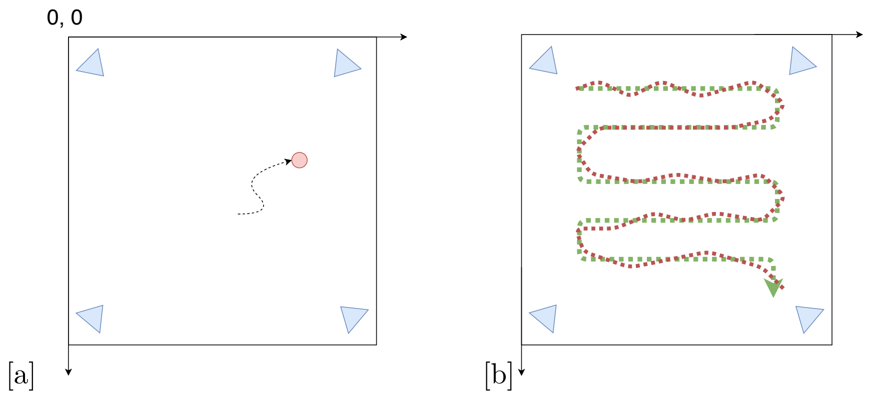


Figure 3.3: Imaginary room with four Anchors and one Tag [a]. Representation of generated input data (green line) compared to context with output of RTLS Studio (red line) [b].

environment, we can easily re-play it and compare it. We name this method **Report Generator**.

By knowing the exact position of input data which we generated (green line of figure 3.3) and comparing it with the output of RTLS Studio data (red line of figure 3.3), we could calculate the precision of localization in two axes by the accuracy of points on a green line with comparison to points on the red line.

Report Generator

After Anchors calculate their distance and precise timestamp to the Tag, they share this information with the master Anchor (i.e., a notable Anchor which, in addition to localization, communicates directly with RTLS Studio). In this Anchors–RTLS Studio connection (as shown in Figure 3.4) is precisely the spot where we will create our Report Generator. The communication between the master Anchor and RTLS Studio is serviced by UDP or TCP connection by sending reports in the payload in a hexadecimal format. Anchor has many types of payloads, but for simplicity, we will use two of them:

- Blink payload:

  Distributes the information about Tags as an asset, which is localization.

- Synchronization payload:

  Distributes the information about the time, which is crucial for localization precision.

We will study and describe this payload in detail in the implementation section of this work. Correctly understanding and generating these two payloads will be crucial for valid input to RTLS Studio. We will create a predefined trajectory that will then be parsed to blink and synchronization payloads, which we will pass to the system input and wait for output to compare it with our predefined trajectory. We could also create intended errors to blink and synchronization payloads for observing how the system will cope with errors of the environment and observe how the localization algorithms will cope with that.

pcap Player

Similarly to the Report Generator principle, we are going to implement another way of abstracting the hardware (Anchors and Tags) and environment dependencies on our way

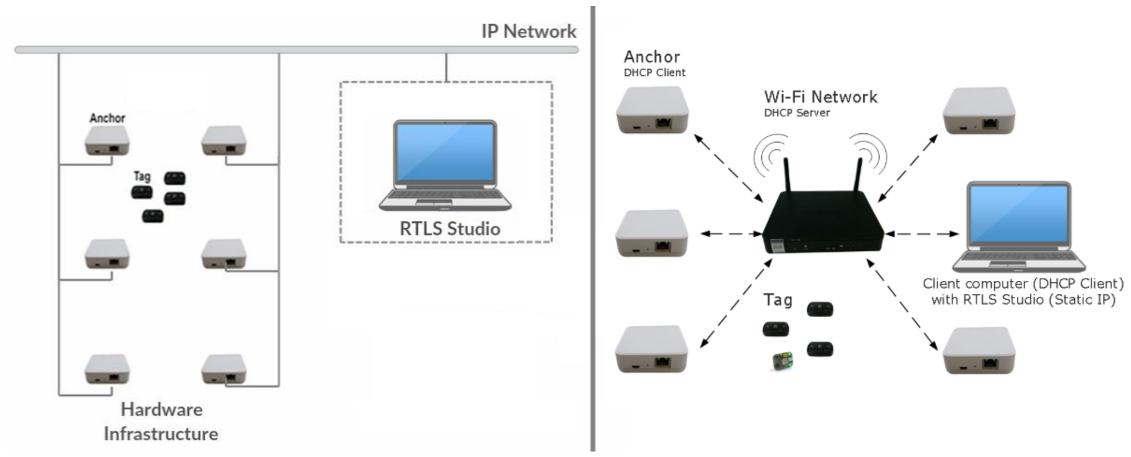


Figure 3.4: Graphical representation of connection between hardware and software parts of RTLS system over Ethernet and Wireless networks [37].

to the automated test suite. At the time that the system has already been deployed to the variety of experimental or industrial projects, where it is in daily use. We can easily create network recordings of its state called RTLS Captures. Which are nothing less than a recording of packets sent and received on network interface between the Anchors and RTLS Studio. In the pcap Player, comparable to the Report Generator, we have to send this communication to the RTLS system to create the desired movement. We will only need to record network traffic on the desired interface and later, find a way how to process it and playback to the system again and again.

Test Controller

Test controller will be part of the test suite which will control all the tests in test suite. It will decide which test to run and when it run. It also controls which parts could or could not be run together in parallel. It will also mediate all the input data to the test parts and gather all the output data for final representation.

REST Tests

Part of REST Tests will be responsible for testing API of the system by trying every endpoint that system has for GET, PUT, POST and DELETE options and expect behavior as it is defined in the API documentation.

Tests of Individual Applications

This part will rely on the fact that every developer should have written the unit tests of his functions and modules, which are usually stored in `/test` directory of the individual project and are run in conventional way as `npm test` for JavaScript based applications or `make test` for language C based applications.

Syntax Checker

Every developer should keep on predefined rules about how he should write the code on the project. This is crucial for teamwork on projects or source files where many developers contribute daily. Primarily when more developers work on the same source file, is essential that it has the same structure. For example, using a tabulator instead of space. Moreover, it follows the code rules that were defined before the start of the project. By this method, we will prevent merging conflicts when many developers push to the same repository. It is decided to write this down theoretically and also enforce automatically.

After a discussion about the right style and best practices between company developers, we decided to write down all requirements which are mentioned in Appendix B.

JavaScript is the primary language of the project application. It is a cross-platform, object-oriented scripting language for interactive webpages. It also has a more advanced server-side version of JavaScript, such as Node.js, which are used in the project. It allows you to add more functionality [30]. JavaScript has a strict syntactical superset called TypeScript, which adds optional static typing to the language. TypeScript is designed for the development of large applications transcompiles (compiled source-to-source) to JavaScript.

There are many tools called Lints (or linter) that are designed to analyzed source code to flag programming errors, bugs, suspicious constructs, and stylistic errors. There are two main Lints designs for both JavaScript and Typescript. The first tool ESLint[1] is a tool for identifying and reporting on patterns found in JavaScript code. Rules in ESLint are configurable and customized. ESLint covers both code quality and style issues. The second tool is TSLint[2], which behaves the same as ESLint but for TypeScript. Both tools have the ability to locate the issues and fix them.

We will transform and enrich our Code Style rules from Appendix B and rewrite them to our Linters to check and fix the code for these mistakes.

Performance Test

An essential part of our test suite will also be to find out where are the performance limits of the software implementation depending on the hardware organizations. The limitation of hardware parts can be easily calculated by their design and theoretical limits of the UWB. Nevertheless, as always, the software is usually built together with many third-party applications and running on different operating system distributions, so a real performance test under the load has to be performed.

We will aim for:

- REST API part and investigate how many inquiries the system can handle before losing the power to serve all the clients.
- Watching the performance of server CPU consumption and watch which system processes take the most resource while localization is working with at minimal and heavy load of Tags moving around the space.
- Clients can also connect to WebSocket for receiving the localization information, so we will also test how many clients can connect and receive on-time information before the server is some much overextended that it will delay this information.

Cloud-based GitLab CI/CD Testing

Due to actual codes are stored on the GitLab platform, we will try to use GitLab CI/CD functionality and adjust our test suite to this system. We already introduced all principles

[1]https://eslint.org/

[2]https://palantir.github.io/tslint/

of GitLab we try to use in theoretical section 2.7. With the help of the GitLab CI/CD, we build the application, tests it in on our test suite, and deploy it if needed. This could happen on every commit, every midnight, or other action we defined. If the test suite run successfully the code is accepted, otherwise, the code is rejected and the developer will be informed where was the problem with his commit. The main idea as we define it in the theory part is to test the code regularly to catch the errors early as we can.

3.4 Proposed Metrics and Methods for Validation

As a result, the test suite proposal is represented in Figure 3.5. In the white-box test group, we can recognize REST Tests, Syntax Checker, Unit tests and Performance tests. These are the methods where the validation is straightforward. The test cases are written with information about how the code should behave.

- REST API test is testing every possible request we can ask from the system. We can test successful response, unsuccessful response and what data each response returns. As a result, we can simply compare if the actual response is equal to the expected response. In return code of this action, we can compare the return value of response numbers (2xx, 4xx, 5xx) and in data response the content.

- Syntax Checker has predefined rules, if some rule will not be fulfilled the test will fail and notify which part of the code is invalid.

- Unit tests are written with information about the implementation of every module. Tests are written before the implementation. And the module is implemented in a way it should pass the tests and requirements. If this test fails the test suite announces it.

- Performance test will send the number of requests to the system at once. We will count how many responses were answered by the system and how many responses the system drop due to overwhelming services. This threshold will be set for 600 answered responses for every 1,000 requests sent.

On the other side, the white-box tests will be much more challenging to validate. Here we use the help of statistical methods we introduce in Theory 2.8.

- pcap Player

  In this type of test, we have UWB payload from real installations, which we are sending to the system repeatedly. We can never determine the original trajectory this UWB data represents. We have only knowledge about the minimal number of positions the RTLS system should generate and their approximate trajectory. Number of positions RTLS system generates will be our only metric for automation. Before the applying every test case recording, we determine the minimal number of positions every Tag should travel. The second way which is not possible to automate, is to observe this trajectory manually and see if it does not consist of any anomalies. However, if we know that the Tag was the whole time static (i.e., without the movement), we can determine the radius of positions we received and determine the approximation precision of the static objects.

- Report Generator

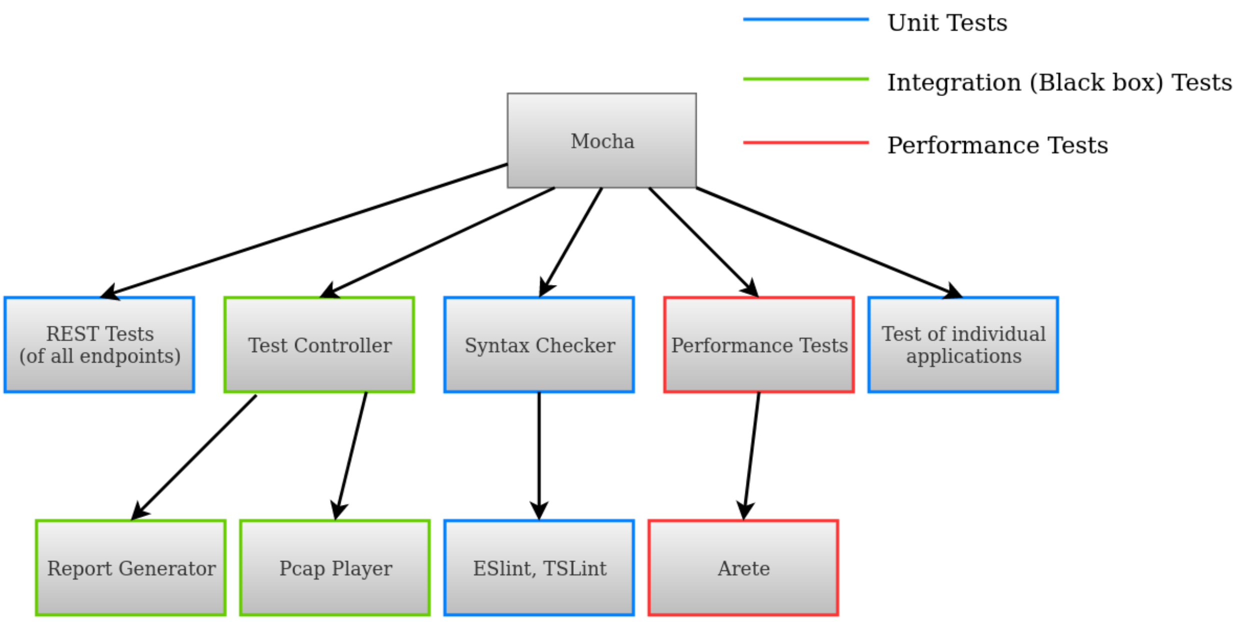


Figure 3.5: Final proposal of test suite.

In the Report Generator, the situation is much more friendly. The UWB payload is not from installation recordings but from our generated stream. As we are creators of this streams, we know how it looks before we put it to UWB payload. By this, we can use the same methods as in PCAP player plus we can use approximation methods. These methods will help us to determine the deviation of the RTLS system trajectory from the generated trajectory. By this approach, we can tell how big the error is in each plays.

Chapter 4

Implementation

In this chapter, we will take apart all pieces of our test suite and describe their implementation to details. We will start with the main control unit of this test suite, Test controller 4.1, then we move to the most difficult parts, pcap Player 4.2 and Report Generator 4.3. We will describe them in great detail with relation to localization. We as well add improvement details, namely injecting intended errors 4.3.1 as well as use our statistical methods RMSE and MAPE 4.3.2. After this two main sections we will talk about the implementation of the rest test suite modules; Code review 4.4, Performance tests 4.5, REST Tests 4.6 and Unit Tests 4.7. In the final section, we will discuss integration to GitLab 4.9 and CI/CD pipeline 4.10, where we will link all parts together and create the entire test suite.

Sewio Networks provided all information. Specifically, complete documentation of the system, which is also available online. Four network recordings of real installations, which are anatomized and detailed take apart and analyzed for use in pcap Player.

4.1 Test Controller

For executing, controlling, and gathering results of our test suite, we will use Mocha[1]. Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing easier. Mocha tests run serially, allowing for flexible and accurate reporting while mapping uncaught exceptions to the correct test cases.

A crucial part of making test valuable is this assertion of the test result. Clauses *should* are simple clauses do the assertion in the Mocha. By the use of *should* clauses, we can easily compare what should be equal to what, or what values should above, below, and so on. We will show examples of using this clauses in sections bellow.

4.2 Replaying Recorded Data

Users of RTLS Studio can create RTLS Capture, which is a recording of network communication between hardware and software components on TCP or UDP port 5000. As well as information about the system itself, namely actual backup of the whole database, actual system settings, all system logs, and plan images. Recordings of network communication are done by simple `tcpdump` command, which is then outputted to the Packet CAPturing file with suffix *.pcap*. The recording is done in real-time. If we open *.pcap* file in any network analyzer tool like Wireshark, we would see every communication which proceeded on

[1]https://github.com/mochajs/mocha

this network interface. Whole communication gives detailed information about all messages running to the server, but for our purpose, it will be more accessible to parse this data. We will get rid of all the data which are not depending on correct localization.

For this purpose, We decided to use tool *tshark* as CLI representative of well-known GUI tool Wireshark. First of all, we will read the input *.pcap* file. Then we will decide to have output data in field formats of two fields. One will represent the data and second the delta time between this data. Furthermore, most importantly, we parse only the packets which are coming from port 5000 on TCP or UDP, respectively. This filter is summarized in the code fragment below.

```
tshark \
 -r capture_*.pcap \
 -T fields \
 -e frame.time_delta_displayed \
 -e data \
 -Y "udp.port==5000 || (tcp.port == 5000 && tcp.flags.push==1)" \
 > capture_file
```

Now we have in our hand's actual recording of network stream as it was recorded with the types of data we need and without any useless information. Our recorded stream contains only two columns—the first column with the timestamp representing the time between packets and second column is the UWB payload. The payload includes all localization data needed for calculation in one long string. Timestamp determines the time when the payloads were sent between each other.

```
0.000029000    7c643838303339363231643661 7c417c30303...
```

In the TCP communication, the start and the end of the communication pay a significant role in communication. But not in our data flow. The recording of this communication was made in the middle of the TCP connection. We only extract the required data for localization in desired format for future credible processing of this communication.

Replaying this data is possible with conventional tools like *tcpreplay*[2]. But we decided to create my own module for this, because of:

- The data were not complete when we got them, they were missing the beginning and the end of the TCP communication.
- Now we even reduce them to special and minimal two-column format.
- They are mostly in the UDP communication format.
- They are in huge numbers.
- We require full control of this process.

[2]https://tcpreplay.appneta.com/

For the best performance, we will create our own pcap replay with minimal complexity. Then we send them as fast and lightweight as possible, so that we have full control of their flow.

In the next step, we need to send this data to the system. System expects data in the same format and on the same port as in the real hardware-software situation. The most important part of this controlled data transportation will be the timestamp of this data. The delta time we gathered from the previous step — the time order in which this data has initially been sent to the system.

We have the data which hardware generated, and we have time delta when these data were sent. We will send the first packet and wait exactly delta time between the first and second packet were sent, then we will send second, third, fourth, and all other packets until the end of the steam. This timing will be managed by the queue of predefined data streams and timeouts.

Nevertheless, with only the data stream system will have information about the localization. However, we will miss information about what we are localizing, where, in which scale, and with what environment settings. Here comes the part with the database dump. This dump carries all information about the RTLS system — the plan on which we localize, the scale, and the origin of the coordination system. The actual database on the testing server will be cleared, and the database from the pcap file will be loaded instead of the old one. This will give us the same physical settings as initial deployment has. Also, the configuration settings of the RTLS Server carried by the configuration file will be copied next to the RTLS Server. This configuration file includes filters applied by localization and utilized thresholds.

This pcap Player is neither following Report Generator nor can be marked as simulator due to few facts.

- RTLS system is a real-time dependent system.
- Its localization engine and the whole system is counting only with real-time time flow.
- If we speed up or slow down the time spectrum, we would brake the validity of the system as a whole because they count the data as real-time flow by milliseconds.

4.3 Generating HW Abstraction

In the previous part, we were taking about real recordings of data. Created by the Anchors and replaying them to the RTLS Studio to have repeated information for future use in the test suite. This approach will give us precise testing input data, but with fixed scope. The scope is defined by what data we were recording namely, how the Tags moved around the area. This state of every recording is unchangeable, therefore is very limited for testing purposes. Moreover, we can never precisely determine what was the original Tags movement at the place it was recorded. For the better testing purpose of the software part, it will be much handier to generate the data by ourselves and adjust them in the way we want for our desired test purpose. This means we can move with Tags virtually as we want it.

Nevertheless, before we start with this task, we have to understand how this data is created. In the previous sections, we learned that the RTLS software is on its input expecting UWB payload. The payload represents the all necessary data for localization and synchronization packed together in a buffer. The buffer is created in Anchors, and it is sent through UDP or TCP connection to the RTLS software. This buffer is then dismantled into individual parts by the software where it is analyzed and used. Calculations for positions and localization is performed. In the end, our localization results are displayed.

There are a few types of payloads, depending on what is their bearing information. For our use, two of them will be necessary, UWB Sync and UWB Blink. The first one is bearing UWB information necessary for Anchors synchronization and then localization accuracy. The second is bearing UWB information about the Tag in the environment and then contributing to localization itself.

- UWB Blink:

  An essential part of this generator will be creating UWB Blink. This Blink is bearing the information about Tag MAC address and UWB signal properties. It is addressing the data crucial for the reconstruction of that signal to the Cartesian coordinate system depending on its properties. Tags are sending UWB Blinks to every accessible Anchor. These Anchors add to the data value about how long it takes for the signal to travel to them. Then they pass this data to RTLS Studio. Where the software has to decide which UWB Blinks from which Anchors use for the most precise localization.

- UWB Sync:

  UWB sync is a piece of central information about the timestamps of individual Anchors. The timestamp is crucial to localization accuracy. As far as our Anchors are only generated abstraction, we cannot observe any time inaccuracy, as we would see in real hardware. As a result, no time synchronization is needed. However, we will describe how the signal of the precise timestamp is sent to individual Anchor.

Time Difference of Arrival (TDoA) principle consists of processing the UWB data from Anchors depended on time when they receive the signal and other factors needed for recalculating positions of Tags. In our generator principle, we will need to turn this principle inside out. We will generate the trajectory represented in the Cartesian coordinate system and then try to remake it to initial TDoA data. Anchors will have their static location, and Tags will be moving around the space dynamically.

First of all, we will need to calculate the distance between the individual Anchors and Tags at every iteration of moving the Tag. This calculation can be dismantled to a simple theorem, as shown in Figure 4.1. It is described in detail in the following equation:

$$t = \frac{\sqrt{(Tag_x - Anchor_x)^2 + (Tag_y - Anchor_y)^2}}{SpeedOfLight} \tag{4.1}$$

This is how we get the time of flight, by Pythagoras theorem of the distance between Anchor and Tag, divided by speed of light[3]. This variable is then multiplied by constants for Anchors internal clock and put to buffer as one variable. Other variables in that buffer are Anchor MAC address, sequence number, general headers, and settings of that Anchor.

[3]the speed of light = $299792458 m/s$

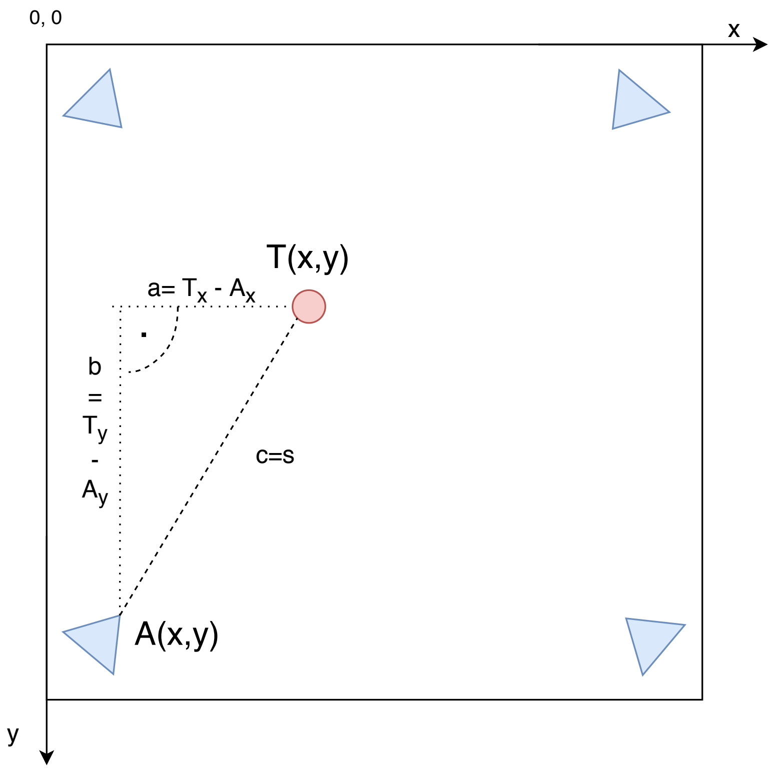


Figure 4.1: Explanation of how we get the distance between the Tag and the Anchor when the position of Tag and Anchor are predefined. In the real TDoA environment, we only know the speed and time of the equation, and the distance is then calculated. Here we know the distance and speed, and we are calculating the time. $A(x, y)$ represents Anchor position, $T(x, y)$ Tag position, c is real distance between the Tag and Anchor we want to calculate.

Now we know the signal path, velocity (speed of light), and the time it took for the signal to travel the path from Tag to each Anchor. With this information about the exact position of the Anchors, these are the information the system needs to determine the Tag location. That is the final buffer we will send to the system.

To fully describe the process of TDoA, we now shortly describe what the system does in the next steps. It then calculates the TDoA and creates hyperbolas around every Anchor. They will cross in several points but only a few points are within our environment, as shown in Figure 4.2. In our simple test environment of the small square, we have no doubts which point it is. However, in a case, we have many squares next to each other, and Tags will travel between the cells. It will be more challenging to determine which one. Nevertheless, this is the above work of this thesis, and it is depended on each TDoA implementation.

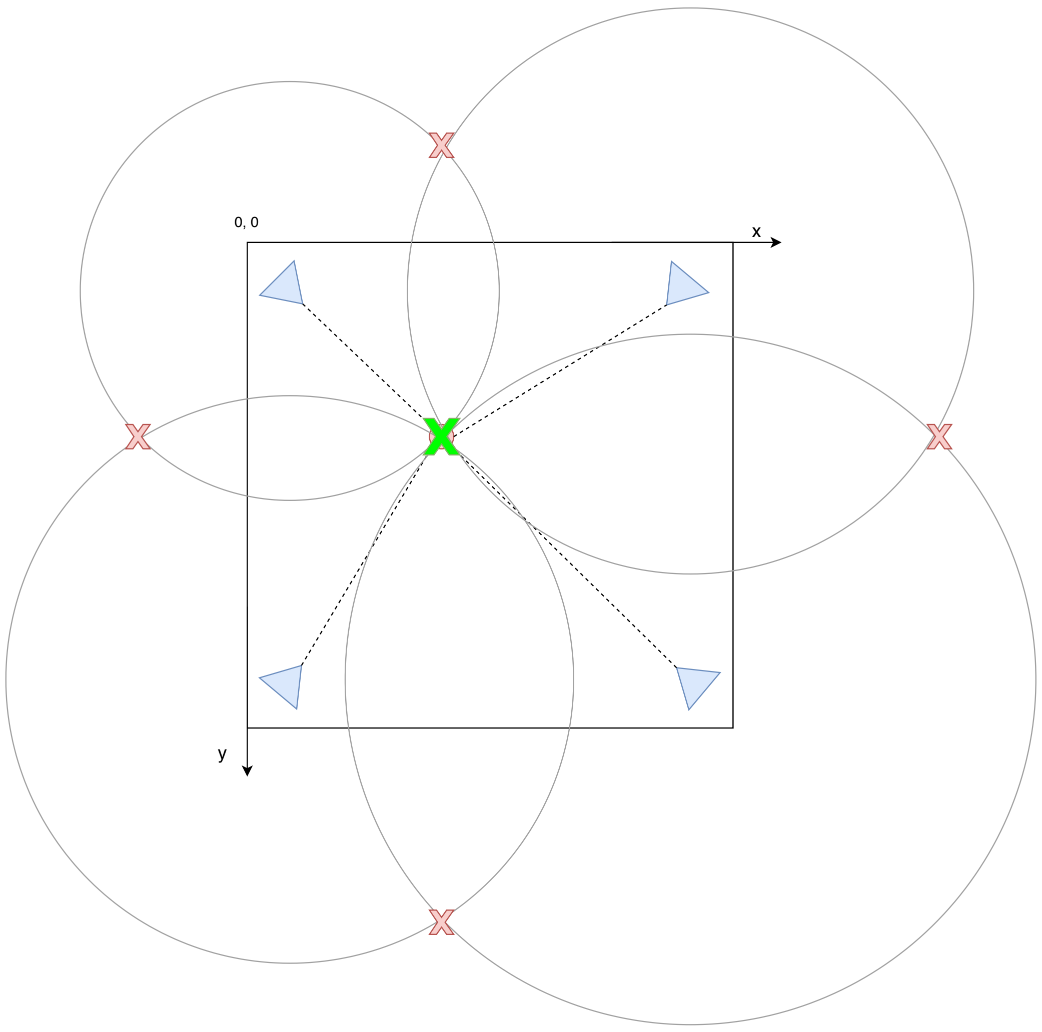


Figure 4.2: Interpretation of how the RTLS TDoA system counts the position of the Tag by knowing the distance from Tag to at least four Anchors.

We now have full control of our abstracted hardware. We can put Anchors whenever we want around the virtual space. We can put as many as we want. We can create as many Tags as we want. Then test the system under a big load. We can even create intended errors to the system by putting deviations to the final buffer, which will be sent to the system. We can later observe how the system will cope with this errors. We will discuss it more in the following part.

4.3.1 Injecting intended errors

For reminding, when we are creating our final UWB buffer, RTLS system expects that we are inserting the following items to this buffer.

- MAC address of the Tag.
- MAC address of Anchor.
- X and Y coordinate of Tag recalculated to the UWB payload.

- Sequence number.
- System time.

These pieces are then send to the buffer to RTLS Studio and it represents the data from Anchor. However, before that, we have a unique opportunity to affect what data the RTLS system gets. Especially the item *System time*. As we already learned, time synchronization is crucial for the precise localization. If this synchronization is inaccurate in even microseconds, the result localization can be very inaccurate. However, this is exactly what we are going to inject into the test suite. Something similar to HW time inaccuracy. The time accuracy of the Anchors is periodically synchronized by the RTLS system. We call this intended error variable `BLINK_ERROR`. We define it by values from zero to five. Zero will mean no intended synchronization error, and five will mean the biggest synchronization error.

The actual zero error synchronization time is calculated as:

```
system_time + anchors_clock_offset + tag_refresh_rate
```

Our specially adapted synchronization time is calculated as:

```
system_time + anchors_clock_offset + tag_refresh_rate
+ (Math.random() < 0.5 ? -1 : 1) * BLINK_ERROR
* Math.random() * one_nanosecond
```

- `Math.random()` will generate a value between 0 and 1 on 16 decimal places.
- One nanosecond represent $1e^{-9}$ of second.

`Math.random()` will generate value between 0 and 1 on 16 decimal places. If it is smaller than 0.5, we will set it to -1 if it will be greater than 0.5, we will set it to 1. This will securely create a random deviation from an interval -1 to 1. By this method we can randomly set positive and also negative drift to synchronization time. If `BLINK_ERROR` will be zero, this error deviation will be zero, if higher the `BLINK_ERROR` the higher deviation we will get. The real impact on this error for localization we will be explain in detail in Evaluation 5.

4.3.2 Applying RMSE and MAPE

The ideal result of our Report Generator will be two sets of data. The first set consist of generated data represented by x and y coordinates. Report Generator will transform this data to output format that represent the UWB payload and send them to the RTLS system sequentially. After the main calculation by RTLS system is done, we will get second set of x and y coordinates. This set is output of RTLS system. When we plot both data sets on one plan, we get the ideal comparison of localization precision by sight. However, this is not enough. Our goal is to make an automated test suite so user intervation should be minimal or none.

This is the ideal place to implement one of our methods described in the theoretical Section 2.8. We will simply take x or y coordinate from both data set and apply the statistical methods. We have to remember that in our case the deviation can occur in both axis x and axis y. Not only in one as we usually used to. Deviation on the x-axis is mostly caused by localization calculation processing delay and localization inaccuracy, and deviation on y-axis mostly by localization inaccuracy. This both deviations are then process by general RMSE equation or MAPE equation respectively:

$$RMSE = \sqrt{\frac{\sum_{N}^{i=1}(gathered\_data_i - generated\_data_i)^2}{N}} \tag{4.2}$$

$$MAPE = (\frac{1}{n}\sum\frac{|gathered\_data - generated\_data|}{|gathered\_data|}) * 100 \tag{4.3}$$

We will analyze this results in detail in Evaluation 5.

4.4 Code review

We introduced our Linters for JavaScript and Typescript in Proposal 3.3. Now we are going to explain in detail the process of rewriting these rules from Appendix B to these tools.

Both tools require configuration files. These files contain information about the desired lint settings, predefined rules, and user-defined rules. File for ESLint is called *.eslintrc.js* and for TSLint *tslint.json*. Both files have a similar structure but not the same. Their rules can be presented in user source code editor as fractures of code, which are underlined by editor default settings to inform developers about perhaps wrong code structure (they are working very well with Visual Studio Code[4]). They can also assist in finding the right fix for individual issues.

The second way of using these rules is in automated test suite, where the test controller returns the error status code about the lines which do not fulfill this predefined rules. This can warn or fix them automatically.

By best-agreed practice, we decided to use this tool in our code editors as warnings but not rely on the automatic fix of Lint tools. Later after the developers will acquaint with these tools, fixed the most lint errors, learn the rules, and learn trust Linters, we will consider turning this option on.

JavaScript Code Review

Our JavaScript lint configuration file called .eslintrc.js is consists of five parts.

- First section, called *env*, will define which ES (ECMA Script programming language) version should it follow. For our purpose, it will be ES6.
- This is closely followed by section *parseOptions*, which defines exactly *ecmaVersion* for 2018.
- The third section called *extends* is here for adding some third-party ESLint configuration files to our file from their library. We will use one extend called *airbnb-base*. Which is an ESLint configuration file from public service AirBnB, and it is the same file they are using for their web application and decided to share with the rest of the world. This package provides Airbnb's base JS *.eslintrc*[5] (without React plugins) as an extensible shared config.

[4]https://code.visualstudio.com/
[5]https://github.com/airbnb/javascript

- Fourth section is called *globals*. It consists of pre-defined ESLint variables for describing the source code.
- In the last, the most important part are the rules. Here we can create our own unique rules based on our requirements.

TypeScript Code Review

TypeScript code review is defined in file *tsling.json*. This file is much more simple and consists only of the *rules* category. This section describes all our desired clauses defined in Appendix B, for more information, follow this Appendix and files *tsling.json* on CD in Appendix C. TypeScript code reviews is much more simple due to the fact that TypeScript is a type control system for JavaScript. By obligatory use of types, while writing the code in TypeScript, we can easily catch errors in the state of compilation.

4.5 Performance and Load Testing

We introduced the performance test in the Proposal 3.3. In this test, we will focus on the throughput of REST parts. Especially how many messages from how many clients it can manage before it starts to throw them away due to overloading (the test will use short API messages as well as long API messages).

For the REST test, we will use a third-party module called Arete[6], which will help us with processing the test itself. We will prepare a detailed API call with predefined messages and settings. We will use the Arete module for sending these messages in hundreds or thousand and trailing how long it takes them to get the response or if they get the response at all.

The second test case will take place in cooperation with the pcap Player and Report Generator. As they are generating the traffic between the system applications which are connected between predefined port numbers, we can easily watch these ports for drops. This will warn us when the system starts to be overloading with the data. That means that for this number of Anchors and Tags which are creating data traffic, the server performance is not enough. By this means, we could identify HW requirements directly proportionally to system complexity.

4.6 REST Tests

Our RTLS system provides the REST API implementation, which gives developers of the third party applications programmatic access to a proprietary software application and web service. With this method, other applications can built their solution software on this RTLS system. This REST API has many calls described in the documentation[7], and every one of them has to be tested. API calls consist of GET, POST, PUT, and DELETE messages. Every possible call has to be tested before every release. In further sections we describe the automation of this part.

The result of the REST calls is best defined as a result of the HTTP Request. In our valid scenario, it should be 200. If it is something different from 200, for example, $5xx$ or others code different from $2xx$ we have a problem with our Open API. The crucial tool for

[6]https://github.com/capablemonkey/arete

[7]https://demo.sewio.net/documentation/api-rest

this is use of assertions *should*. They can easily compare the resulting output with defined output. The syntax of assertions is very friendly to use as we can see in a simple example:

```
request(url)
    .get('/api/Tags/')
    .set({
        'X-APIKey': 'sdfjk23fewce3H23qwuoc3r43'
    })
res.status.should.be.equal(200)
var output = JSON.parse(res.text);
output.Tags.length.should.be.equal(2);
output.type.should.be.equal("Tag");
```

We make a simple call for REST API endpoint for Tags of the system with authorization API Key, and we get the results. First of all, we check if the result code is 200, which means it passed successfully. After that, we test the payload of these results as we parse the JSON output of the results. Then we expect the filed of the result to have predefined length and also be equal to data we push to it before. If this all pass the test successfully, REST API call is valid. If not, we have found an invalid response with an exact call that had failed, and we can make a further investigation.

We can also test the invalid results and await if the return status would be equal, for example, to code 401, which states for *Unauthorized*, if we use the wrong API key for our test as is shown in code snippet bellow.

```
request(url)
.get('/api/Tags/')
.set({
    'X-APIKey': 'IAmGonnaHackYou'
})
res.status.should.be.equal(401)
```

We provided the implementation of the REST API for every call with testing the GET calls. Then testing the PUT call with verification by fetching the GET call again. Furthermore, we repeat this procedure with the POST call, as well as the DELETE call. The whole process can be found in Appendix C.

4.7 Unit Tests

Every developer is now responsible for his piece of code, and unit test written by him to his code is obligatory. As most of the project modules are written in JavaScript, the pipeline is set to go to every project directory where `package.json` files are located. Here the pipeline tries to run the command `npm test`, which is a conventional command for every module to consist of looking for `/test` directory and run files there.

In the C language source files, the pipeline is looking for the place where the `Makefile` is located, and run the command `make unittest`. If the pipeline finds that some module does not have this unit test written. It will notify the user about the obligation to write it.

4.8 Integration with Local CI/CD Pipeline

The initial deployment of the whole test suite was done in the virtual environment where the whole RTLS system was packed in the prepared *.vbox* image. After that, the test suite was run with the information about the IP address of that virtual machine within the computer network and with login credentials. The whole test suite was run on the CLI interface, and after the run results were shown in the final test report.

This final test report was put on an individual web server under port (6210) and it is designed to comprehend to developers as well as managers. The report is very straightforward and indicates if something misses the boundaries we set by red color. After that developer can simply enlarge the problematic part and see the console output of the problem and locate it.

This process can be try by reader with using the included CD and following the manual in Appendix C.

4.9 Integration with GitLab CI/CD

In the proposal chapter, we choose the GitLab CI/CD possibilities as our desired platform. First of all, we have to move the RTLS system from the offline virtual environment to an online access environment. For easier manipulation and fetching the newer application versions within the GitLab pipeline. After that we create *.gitlab-ci.yml* file with detailed information for the runners what to do. Now we describe this file in detail, and the whole interpretation of this file can be found on CD in Appendix C.

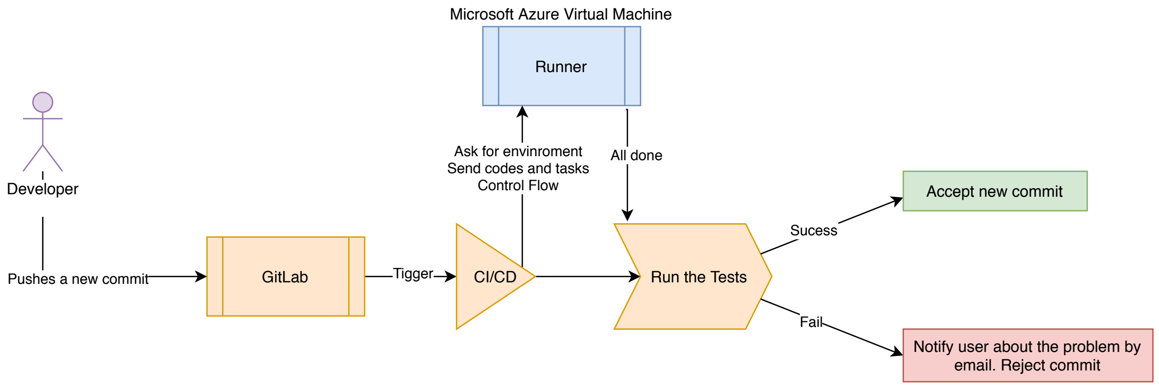


Figure 4.3: Diagram flow of CI/CD pipeline.

The RTLS system is running in the virtualized environment within the Azure cloud network. GitLab will run the pipeline and call this runner and pass him the tasks from the latest source codes from the code repository. By this approach, we have a fully isolated system in the latest version. The pipeline is divided into two parts.

- Minimal part consists of test scenarios without the PCAP Player.
- The full part consists of all test suite scenarios.

This division is done by the fact that PCAP Player is done by real-time playing, and it takes a considerable amount of time to process. The minimal part is automatically triggered to

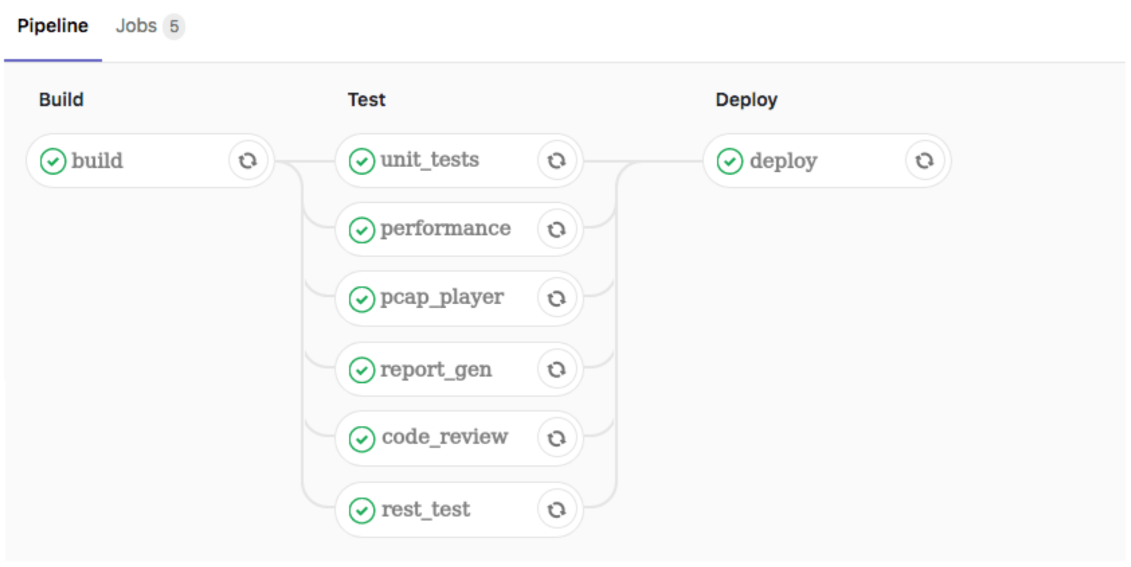


Figure 4.4: Final GitLab pipeline.

run by every push to the repository. The full test suite is run every day at midnight. This is just a test requirement and can be changed anytime.

The test suite is designed by the behavior that it can be used on local RTLS system installation as well as Cloud-based. However, also from the different servers, which tests the desired server with RTLS system. The whole process comes from detailed GitLab official manuals for setting up the pipeline[8].

The example of the final infrastructure is accessible online for the reader. Please follow the manual in the Appendix D.

4.10 Final Infrastructure

The ultimate infrastructure consists of the GitLab CI/CD testing pipeline as the main frontier of the first stage of releasing the new version of the RTLS system. This pipeline guarantees and validates the functionality of the code in the repository and also creates the usable image of the final system. This system is then submitted to Alpha Tests done by operation team within the company. After the final passage of the Alpha test, it is then committed to the Beta test, by testing on real customers and then finally ready to release candidates (RC) versions, which are final release stage of the new version of the product.

Alpha testing follows the exact set of steps to be done and tested with proper fields in every task as:

- The conditions within the test should be realized. In which state the system should be set and which settings have to be applied.
- The task that has to be done with the system.
- The expected output.

[8]https://docs.gitlab.com/ee/ci/yaml/

- Validation if the output is expected or different.

This Alpha Test is done manually. The list is regularly updated with new system features functionalities. The full manuscript of this Alpha test is included in the Appendix C.

Beta tests are done by selected end-users in their installation. These terms usually mean working with the new version and observing if everything works, as it should. If not, the rollback to the previous working version can be easily done. The release candidate is a version of the new software ready to be finally deployed.

If any problem occurs during the automation tests, Alpha, or Beta tests, they are well documented, and the issue is submitted to the code repository. After the tests are done and individual developers fixed their issues, the iteration is starting again until the product is without the observable issues.

However, from now the most work is done by the automated pipeline, which saves much time in the future Alpha and Beta testing phases.

Chapter 5

Evaluation

In the previous chapters of this work, we introduced the problem, made a proposal for the solution, and then implemented it. Now we are going to validate the outcome before we deploy it in the application workflow. Validation will be displayed in two cases. First, we introduce the problem in the native RTLS system and manual workflow to test it. Then we make the same scenario in our test suite and compare the results.

5.1 Case 1 - pcap Player

Our first evaluation case will be about the RTLS system as it is designed. We will deploy five Anchors in a square room with dimensions of `6.947m x 6.947m`. One Anchor to every corner of the room and the last Anchor in the middle of the room, close to wall and between the already deployed two Anchors, as can be seen in Figure 5.2 part 1. Anchors are marked by green signs . Walls on the plan do not play any role in these tests.

We will initialize the system. First, we open RTLS Manager and run the Anchor initialization, which links the system with the Anchors. After a while, the connection is established. Following that, we open the RTLS Sensmap. We upload the room plan image and set the dimensions of the room. With these dimensions, the system calculates the scale of the room for accurate localization. Lastly, we put the exact location of the Anchors in the room to the RTLS Sensmap. The RTLS system is now prepared and ready for localization.

We will turn on network recording. Furthermore, we also open RTLS Sensmap and turn on Tag tracking to observe the Tag in real-time, with an internal tool enabling for remembering the history. Then we take one Tag with the alias *DP Test Dynamic* and move it around the room in the symbol of tilt „U“ letter. Tags are marked by blue signs . The shape is created by walking from one window around the room back to another window and then way back. The print screen of RTLS Sensmap from this test is shown in Figure 5.2 part 1. This picture represents the valid results of the RTLS localization system. After this walking, we stopped the network recording and saved it. Now we have network a recording of the real RTLS installation in use.

As we can see, the movement in Figure 5.2 is not perfect. It is not straight line as one could expect. It is much more realistic. Walking is not always done in one line, and the body which carries the Tag shakes it a little bit without knowing. Also, the system itself has a precision to units of decimetres, which is desired.

Evaluation of pcap Player

Now we are going to use our pcap Player to replay the scenario we just recorded. Remember, from the last section that we have only the backup of the system with the recording of the network communication. We will take another clean RTLS Studio on different server. The Anchors and Tags are now not needed. We will upload the backup of the system and network recording to the pcap Player and wait for the results. After the initialization and preparing the network, which takes a few seconds, we can then refresh the RTLS Studio. Now we can see the same plan with the same properties as the real system has. In a while, the Tag starts to move and make the same trajectory as we observe in the real system. This final trajectory can be seen in Figure 5.2 part 2. Then we make another run of pcap Player, and the second result is shown in the same Figure 5.2, part 3.

By the first inspection, we can assume the trajectories on Figure 5.2 are the same on part 1., part 2. and part 3. However, when we take a closer examination, we can see the differences between them. They are mostly similar, but not identical. In each of all three cases, the RTLS system gathered similarly around 400 positions of the Tag *DP Test Dynamic*, which are plotted on Figure 5.2.

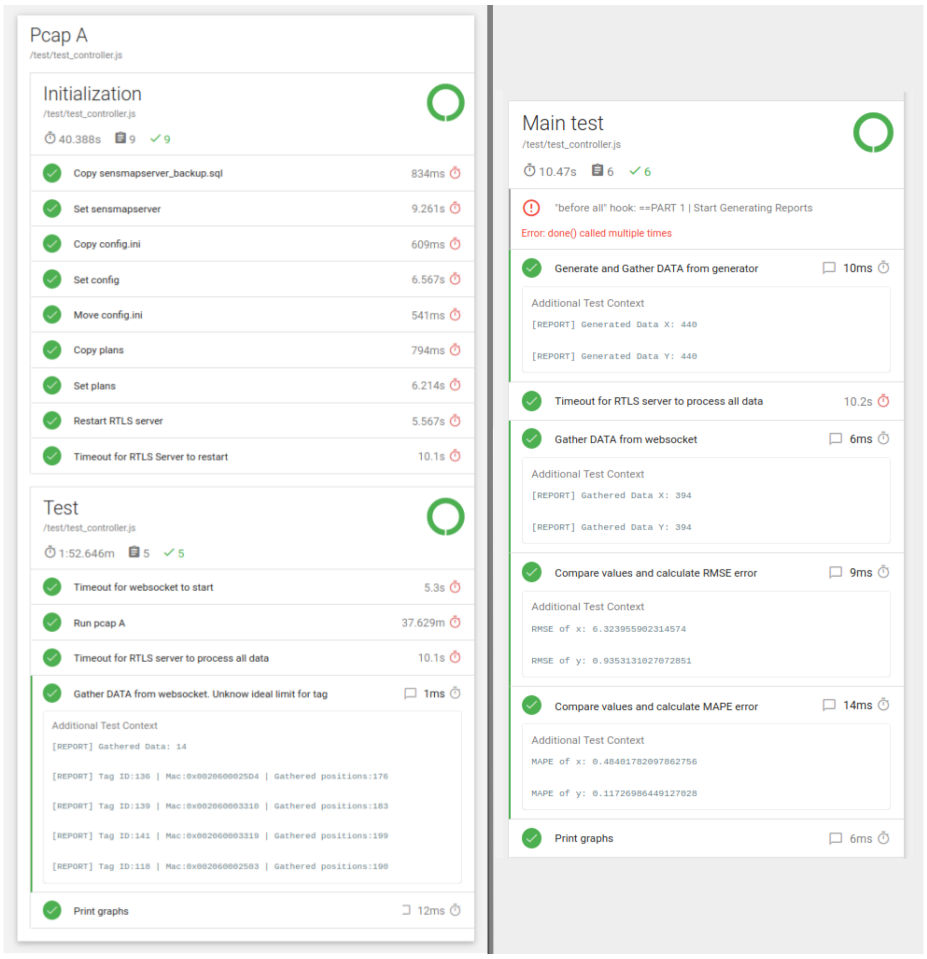
The system, as we assume, is not 100% deterministic, which is due to many approximation filters RTLS system use. As well as algorithms for choosing the best Anchors for calculations of the positions. In every run of the same network recording, the output is more or less the equivalent but never identical.

This RTLS system has many additional settings, which we will not discuss in detail in this thesis because their complexity is out of the scope of this thesis. By adjusting these settings, the system can be improved in many ways depending on the environment and desired results. Here we only want to demonstrate how these settings can affect the results. With the help of the support team of Sewio Networks, we improve the system by adjusting the settings, especially for this type of environment. The results can be seen in the same Figure 5.2 part 4. This outcome is now so much realistic as results before and much more usable in the industry. By this outcome, we want to demonstrate how useful this tool can be.

Support teams of this RTLS system do not have to travel to every installation of the RTLS system in different environments. They usually improve the system settings by traveling on the place and walking around the RTLS installation environment to see the results. Adjust the settings and walk again to see if it helped. Now they can ask someone on site to do a walk and make a recording for them. The support team then replay it in this tool by replaying and customizing the settings. Hence, support teams can adjust the system in a more convenient way and even remotely. Moreover, these settings might be exported to an existing system. That is only one example of the utilization of this tool.

The primary goal of the pcap Player for us will be in our pipeline. Four recordings and backups of the real installation of this RTLS system was provided to us, with their specifications and information about how many positions should the Tags have in these recordings. This all can be find in Appendix C. The real view how the user see this test in our pipeline is shown on Figure 5.1.

These four files will be our reference data inputs. We implement all four of them to our pipeline, and in the test suite, we can then compare how many positions we got from replaying for every Anchor with the references number of positions these Tags should have. If they are above the desired threshold, then we can conclude that the system is working correctly. If they are bellow, there is a problem somewhere. With this pcap Player test,



```
Pcap A
/test/test_controller.js

Initialization
/test/test_controller.js
40.388s   9   ✓ 9
✓ Copy sensmapserver_backup.sql                         834ms
✓ Set sensmapserver                                     9.261s
✓ Copy config.ini                                       609ms
✓ Set config                                            6.567s
✓ Move config.ini                                       541ms
✓ Copy plans                                            794ms
✓ Set plans                                             6.214s
✓ Restart RTLS server                                   5.567s
✓ Timeout for RTLS Server to restart                    10.1s

Test
/test/test_controller.js
1:52.646m   5   ✓ 5
✓ Timeout for websocket to start                        5.3s
✓ Run pcap A                                            37.629m
✓ Timeout for RTLS server to process all data           10.1s
✓ Gather DATA from websocket. Unknow ideal limit for tag   1ms
  Additional Test Context
  [REPORT] Gathered Data: 14

  [REPORT] Tag ID:136 | Mac:0x0020600025D4 | Gathered positions:176

  [REPORT] Tag ID:139 | Mac:0x002060003310 | Gathered positions:183

  [REPORT] Tag ID:141 | Mac:0x002060003319 | Gathered positions:199

  [REPORT] Tag ID:118 | Mac:0x002060002503 | Gathered positions:190
✓ Print graphs                                          12ms
```

```
Main test
/test/test_controller.js
10.47s   6   ✓ 6
"before all" hook: ==PART 1 | Start Generating Reports
Error: done() called multiple times
✓ Generate and Gather DATA from generator               10ms
  Additional Test Context
  [REPORT] Generated Data X: 440

  [REPORT] Generated Data Y: 440
✓ Timeout for RTLS server to process all data           10.2s
✓ Gather DATA from websocket                            6ms
  Additional Test Context
  [REPORT] Gathered Data X: 394

  [REPORT] Gathered Data Y: 394
✓ Compare values and calculate RMSE error               9ms
  Additional Test Context
  RMSE of x: 6.323955902314574

  RMSE of y: 0.9353131027072851
✓ Compare values and calculate MAPE error               14ms
  Additional Test Context
  MAPE of x: 0.48401782097862756

  MAPE of y: 0.11726986449127028
✓ Print graphs                                          6ms
```

Figure 5.1: CI/CD pipeline output example as the user sees it. On the left side of the Figure is output of pcap Player. On the right side is output of Report Generator with RMSE and MAPE calculated. In the last section *Print graphs*, the user can open the plan with plotted positions for manual validation too, same as we saw in Figure 5.2.

if the developer is, for example, modifying the location engine. He no longer has to fear if everything is working as it should, he can now see it by the results of this pipeline test that it is working well. Also, he can make a manual comparison of the expected trajectory by referral images and real output trajectories same as we did in Figure 5.2.

5.2 Case 2 - Report Generator

In the same manner as in Case 1, we will now evaluate our Report Generator module. This module was much harder to develop as we learned in the Implementation 4.3. However, we are expecting much greater flexibility from this module than from pcap Player.

The Report Generator used in this evaluation has predefined settings. It uses the same room with the same scale as pcap Player evaluation. It works with four abstracted Anchors with predefined positions, as can be seen in Figure 5.3. The user defines the position of Anchors, as one input parameter. The only request is they have to create a square, but they can be whenever the user wants at a reasonable distance maximally 10 meters from each other.

By the position of this Anchors Report Generator calculates the localization envelope. The envelope symbolizes the space between the Anchors, where it will move the Tag. Generator always starts generating the positions from the right upper corner, and he will move in a zig-zag manner back to the bottom right corner. The whole movement is demonstrated in Figure 5.3 Part 1., as output from the Report Generator. The Report Generator, as same as the pcap Player, prepares the system for this scenario. In Part 2. of the Figure 5.3, we can see the view from the RTLS Sensmap, and in part 3. of the Figure 5.3, we can see the result of the Report Generator results with the results of the RTLS system output combined and plotted on the one plan.

By sight, we can see that the green line and orange line are not identical. Green line represents the output of the RTLS system. The orange line represents the output of the Report Generator, which is then transformed into the input of the RTLS system. We can see that the orange line is precisely straight. The line is created by the cycle with no offset. However, on the other side, the green line of the RTLS system is much more leaping.

However, this is correct. We can even see the system missed the beginning of the packets coming and started when the Tag had already been in the left upper corner. RTLS system algorithms are always counting with some approximation. Nevertheless, we can see it is much more straightforward.

5.2.1 Scaling the synchronization

This is also a great place to test systems in a different way. We can put some intended errors to the system and observe the behavior. In the implementation section 4.3.1, we consider this and put options for injecting time synchronization errors. These errors can be defined by user from zero to five. Zero means that Report Generator do not make synchronization errors. Five means that Report Generator make the most synchronization errors but do not break the system as a whole.

In our second example, we set this value to number two and run the Report Generator. The result from RTLS Sensmap can be seen in Figure 5.4 part 1. In contrast with previous results in Figure 5.3 Part 2., the blue line is much more shabby. We will not go to too many details of this process. For our testing pipeline purpose, this is the final result. The benefit of this error making method is the ability for developer scale the localization engine for the hardware and environment synchronization errors. Also to see how the system copes with errors and improves its ability to handle these types of errors.

5.2.2 Applying comparative methods

This is a great place to apply our RMSE and MAPE methods we introduced in the Theory 2.8 and Implement in 4.3.2. As we already noticed in Figure 5.3 part 3. and Figure 5.4 part 2. there are two cases where we know precise positions of initial points running to the system. As well as the output of localization positions coming from the RLTS system. Here we have all the data we need for accurate comparison.

We have two sets of data represents by an array of sets of x and y values. One set from Report Generator which as predicted data. And second set from localization output of RTLS Studio as observed data.

MAPE		Level of intended error		
		0	2	3
Axis	X	2.39%	2.87%	3.25%
	Y	0.65%	1.05%	1.25%

Table 5.1: Comparison of intended errors using MAPE and different level of this intended error. Gathered from average of 10 runs of Report Generator.

RMSE		Level of intended error		
		0	2	3
Axis	X	6.470255	6.819870	6.928015
	Y	0.982994	0.985278	1.088032

Table 5.2: Comparison of intended errors using RMSE and different level of this intended error. Gathered from average of 10 runs of Report Generator.

Our Report Generator module can also create intended error to the predicted data by creating synchronization deviations. This will influence the observed data. We will try this intended error on a few values and observe how this affects the results.

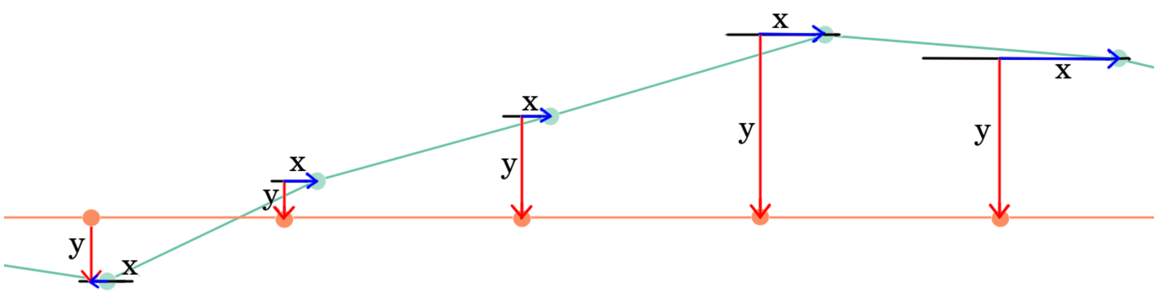


Figure 5.5: Graph displaying how we apply RMSE method on our result.

As we can see in Figure 5.5 where in our case, there is not the only deviation in one axis but in both. Deviation in the y-axis is caused by an error of precision, and deviation on the x-axis is caused by a delay in processing the data in the RTLS system. The result is then summed representation of deviation of x point in x-axis and y position in the y-axis. The both methods are applied on data similar to data from Figures 5.2 and 5.3 with different level of intended error and summarized in the Tables 5.1 and 5.2. This observation was done 10 times and data were averaged.

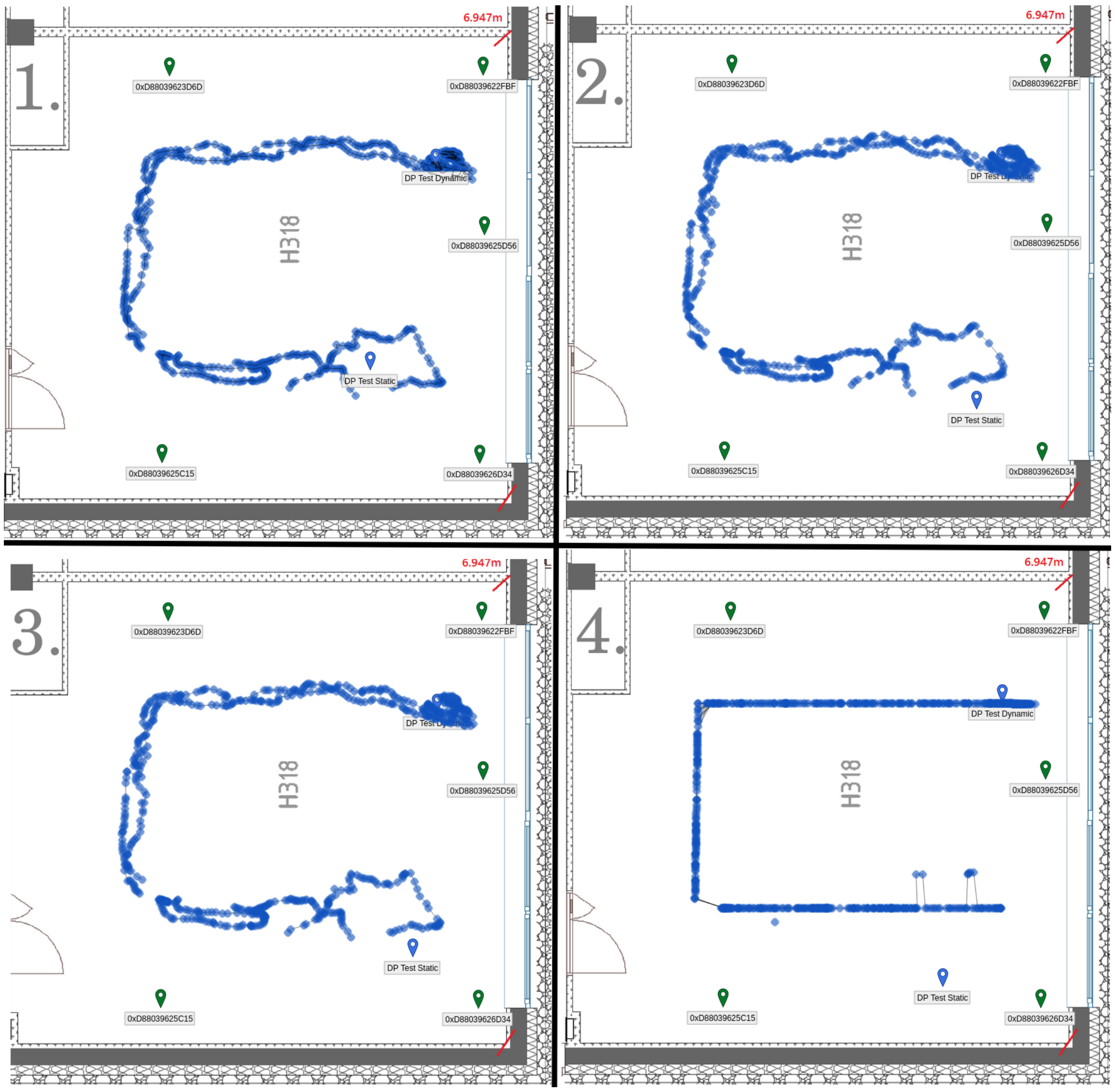


Figure 5.2: Representation of the real RTLS system and pcap Player. *1)* (upper-left corner) is result of the localization in the real RTLS system with hardware Anchors and Tags. *2)* (upper-right corner) and *3)* (bottom-left corner) is representation of replaying communication in pcap Player from original network recordings. *4)* (bottom-right corner) replaying communication with additional RTLS system settings adjustments for the best result.

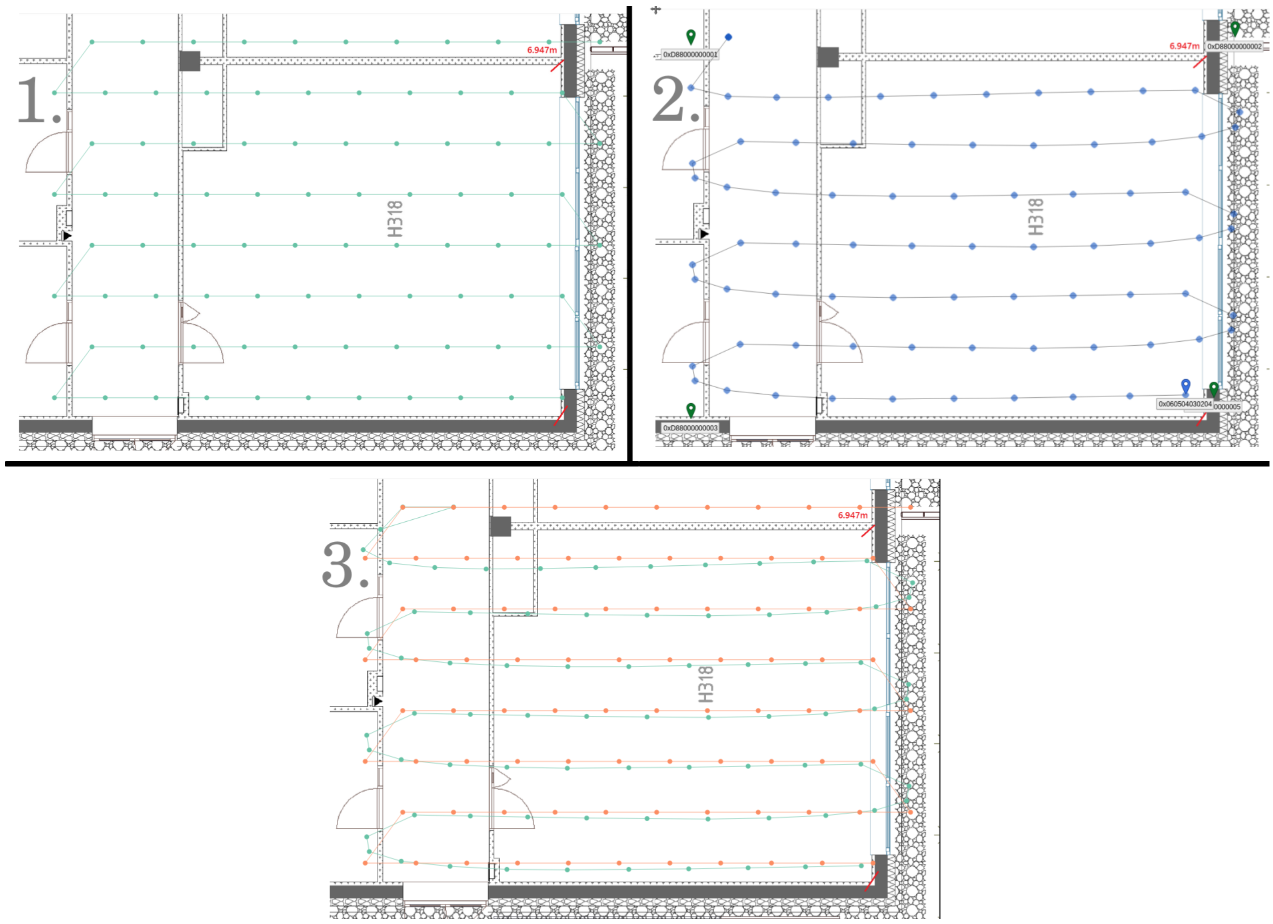


Figure 5.3: Example of using Report Generator. *1)* (upper-left corner) are data generated from Report Generator. *2)* (upper-right corner) is the result of RTLS system output for generated input from *1)*. *3)* (bottom) is combination of both for visual comparison.

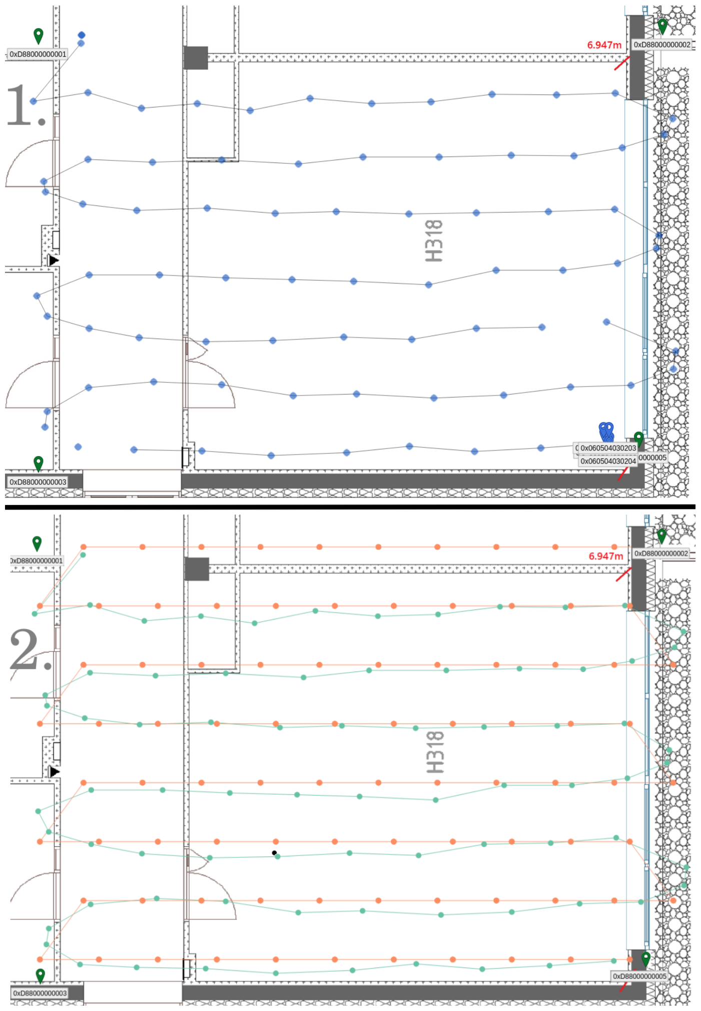


Figure 5.4: Example of using Report Generator with intended errors. *1)* (top) is result of RTLS system for generated input with synchronization errors. *2)* (bottom) is comparison of output from Report Generator (orange line) and output of the RTLS system (green line) with synchronization errors.

Chapter 6

Conclusion

The goal of this thesis was to create an infrastructure for testing and deployment of the real-time localization platform based on existing localization technology.

We learned how a real-time location system works and the differences between the various implementations. We studied the process of creating continuous integration and continuous delivery solutions, as well as many types of testing options for validating that the software works as it should. We also studied the principles of DevOps and its ideas on how to apply them to the process of software development.

We analyzed the actual process of creating and delivering real-time localization software. We then managed to develop principles of how this software should be developed, tested, and delivered.

The main accomplishments are creating the DevOps culture in the RTLS Software as well as an automatic pipeline for testing and delivery of the location software. This pipeline consists of automated tests oriented on validating the REST API, individual applications, code styles, performance, acceptance tests, and integration tests.

The most exciting parts are modules pcap Player and Report Generator, which We created especially for this real-time localization systems. These two modules allow the test to take hardware parts as software created abstraction and bring the simplicity and speed for testing these systems. These two modules were validated in this work, and their behavior and functionality were compared to a real working system with genuine hardware parts.

The important results are that these principles were implemented in the pipeline, and these tests are run at every critical change made to this system. With this pipeline, the development of this real-time localization product is more secure and straightforward. The developers do not have to be afraid to add new features to the system as this pipeline can easily and quickly tell them that everything works as it should (or even better than before).

My work could be expanded on automated tests of the application front-end , which brings even detailed and more test coverage. This will take a lot of time due to the complexity of the application and its front-end functions as every test has to be tailored to the system.

This pipeline is already deployed and making product development more accessible. This work was verbally presented in conference Excel@FIT 2020 and was awarded by the expert panel.

Bibliography

[1] *Global positioning system wide area augmentation system (WAAS) performance standard.* Standard. USA: Department of Transportation and Federal Aviation Administration, 2008.

[2] Abdulrahman, A., Abdulalik, A.-S., Mansour, A., Ahmad, A., Suheer, A.-H. et al. Ultra Wideband Indoor Positioning Technologies: Analysis and Recent Advances. *In IEEE PerCom 2003.* may 2016, vol. 16. ISSN 1424-8220.

[3] Adam, D., Mikael, D. and Richard, B. S. Challenges When Adopting Continuous Integration:: A Case Study. In:. 1st ed. Springer, Cham, 2014, vol. 8892, p. 17–32. ISBN 978-3-319-13835-0.

[4] Antonio, V. On the Industrial Adoption of Model Driven Engineering. Is your company ready for MDE? *International Journal of Information Systems and Software Engineering for Big Companies (IJISEBC).* december 2014, vol. 1, p. 52–68. ISSN 2387-0184.

[5] Baarsen, J. van. *GitLab Cookbook.* 1st ed. Packt Publishing, 2014. ISBN 978-1783986842.

[6] Barsocchi, P., Potortì, F., Chessa, S. and Knauth, S. *Evaluating AAL Systems Through Competitive Benchmarking. Indoor Localization and Tracking.* 1st ed. Springer-Verlag Berlin Heidelberg, 2012. ISBN 978-3-642-33532-7.

[7] Christof, E., Gorka, G., Josune, H. and Serrano, N. DevOps. *IEEE Software.* IEEE. 2016, vol. 33, no. 3, p. 94–100. ISSN 0740-7459.

[8] Cockcroft, A. *Velocity and Volume (or Speed Wins).* 2013. Presentation at FlowCon, San Francisco, November.

[9] Duvall, P. M., Matyas, S. and Glover, A. *Continuous Integration: Improving Software Quality and Reducing Risks.* 1st ed. Addison-Wesley Professional, 2007. ISBN 978-0321336385.

[10] Elbahhar, F. B. and Rivenq, A. *New Approach of Indoor and Outdoor Localization Systems.* 1st ed. InTech, 2012. ISBN 978-953-51-0775-0.

[11] Evertse, J. *Mastering GitLab 12: Implement DevOps culture and repository management solutions.* 1st ed. Packt Publishing, 2019. ISBN 978-1789531282.

[12] Fowler, M. *Continuous Integration)* [online]. 2006 [cit. 2020-03-06]. ThoughtWorks. Available at: https://www.martinfowler.com/articles/continuousIntegration.html.

[13] *GitLab Runner Documentation* [online]. GitLab, 2020 [cit. 2020-03-06]. Available at: https://docs.gitlab.com/ee/README.html.

[14] Grewal, M. S., Weill, L. R. and Andrews, A. P. *Global Positioning Systems, Inertial Navigation, and Integration.* 1st ed. John Wiley Sons, Inc., 2001. ISBN 0-471-35032-X.

[15] Guangliang, C. Accurate TOA-based UWB localization system in coal mine based on WSN. *Physics Procedia.* Elsevier. 2012, vol. 24, p. 534–540.

[16] *How experts explain technology adoption cycle* [online]. Technology Trend Analysis, 2020 [cit. 2020-03-06]. Available at: https://setandbma.wordpress.com/2012/05/28/technology-adoption-shift/.

[17] Hyndman, R. J. and Koehler, A. B. Another look at measures of forecast accuracy. *International Journal of Forecasting.* october 2006, vol. 1, p. 679–688.

[18] Hüttermann, M. *DevOps for Developers.* 1st ed. Apress, 2012. ISBN 978-1430245698.

[19] Kim, G., Debois, P., Willis, J. and Humble, J. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* 1st ed. Addison-Wesley Professional, 2010. ISBN 978-0321601919.

[20] Kim, G., Debois, P., Willis, J. and Humble, J. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* 1st ed. IT Revolution Press, 2016. ISBN 978-1942788003.

[21] Kim, S. and Kim, H. A new metric of absolute percentage error for intermittent demand forecasts. *International Journal of Forecasting.* july 2016, vol. 32, p. 669–679.

[22] Kolodziej, K. W. and Hjelm, J. *Local Positioning Systems LBS Applications and Services.* 1st ed. Taylor Francis Group, LLC, 2006. ISBN 978-0-8493-3349-1.

[23] Len, B. An Evaluation of Indoor Location Determination Technologies. *IEEE Software.* 1st ed. IEEE. 2018, vol. 35, no. 1, p. 8–10. ISSN 0740-7459.

[24] Len, B. The Software Architect and DevOps. *IEEE Software.* 1st ed. January 2018, vol. 35, no. 1, p. 8–10. DOI: 10.1109/MS.2017.4541051. ISSN 1937-4194.

[25] Liu, H. and Tan, H. B. K. Covering code behavior on input validation in functional testing. *Information and Software Technology.* february 2009, vol. 51, p. 546–553.

[26] Lui, G., Gallagher, T., Li, B., Dempster, A. G. and Rizos, C. Differences in RSSI readings made by different Wi-Fi chipsets: A limitation of WLAN localization. In: *2011 International Conference on Localization and GNSS (ICL-GNSS).* June 2011, p. 53–57. DOI: 10.1109/ICL-GNSS.2011.5955283. ISSN 2325-0771.

[27] Malik, A. *RTLS For Dummies.* 1st ed. For Dummie, 2009. ISBN 978-0470398685.

[28] Manish, V. Understanding DevOps bridging the gap from continuous integration to continuous delivery. In: *Fifth International Conference on the Innovative Computing Technology (INTECH 2015).* IEEE, 2015, p. 78–82. ISBN 9781467375504.

[29] Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship.* 1st ed. Prentice Hall, 2008. ISBN 978-0-13-235088-4.

[30] *MDN web docs* [online]. Mozilla, 2020 [cit. 2019-10-15]. Available at: https://developer.mozilla.org.

[31] Myers, G. J. *The Art of Software Testing.* 2nd ed. John Wiley Sons, Inc., 2004. ISBN 0-471-46912-2.

[32] *Lexico* [online]. Oxford, 2020 [cit. 2020-1-4]. Available at: https://www.lexico.com/en/definition/culture.

[33] Parameswaran, A. T., Husain, M. I. and Upadhyaya, S. Is RSSI a Reliable Parameter in Sensor Localization Algorithms. *State University of New York at Buffalo.* State University of New York at Buffalo. 2009, no. 1, p. 1–15. ISSN 0740-7459.

[34] S., N. Black Box and White Box Testing Techniques - A Literature Review. *International Journal of Embedded Systems and Applications.* june 2012, vol. 2, p. 29–50. DOI: 10.5121/ijesa.2012.2204.

[35] Saglietti, F., Oster, N. and Pinte, F. White and grey-box verification and validation approaches for safety- and security-critical software systems. *Information Security Technical Report.* 1st ed. february 2008, vol. 13, p. 10–16.

[36] Saua, S. M. *Indoor Positioning - Technologies, Services and Architectures.* 2007. Master's thesis. University of Oslo.

[37] *Sewio Public Documentation* [online]. Sewio Networks, 2019 [cit. 2019-10-15]. Available at: https://docs.sewio.net/docs.

[38] *Sewio Webpage* [online]. Sewio Networks, 2019 [cit. 2019-10-15]. Available at: https://sewio.net/.

[39] *Physics.org* [online]. IOP Institute of Physics, 2020 [cit. 2020-1-2]. Available at: http://www.physics.org/article-questions.asp?id=55.

[40] Tianfeng, C. and R., D. Root mean square error (RMSE) or mean absolute error (MAE)? *Geosci. Model Dev.* january 2014, vol. 7. DOI: 10.5194/gmdd-7-1525-2014.

[41] Walls, M. *Building a DevOps Culture.* 1st ed. O'Reilly Media, Inc., 2013. ISBN 978-1-449-36417-5.

[42] Wang, M., Xue, B., Wang, W. and Yang, J. The design of multi-user indoor UWB localization system. In: *2017 2nd International Conference on Frontiers of Sensors Technologies (ICFST).* IEEE, 2017, 2017-, p. 322–326. ISBN 9781509048601.

[43] Xu, G. *GPS Theory, Algorithms and Applications.* 2nd ed. Springer, 2003. ISBN 978-3-540-72714-9.

[44] Youssef, M., Agrawala, A. and Shankar, A. U. WLAN Location Determination via Clustering and Probability Distributions. *In IEEE PerCom 2003.* february 2003, vol. 1.

Appendix A

List of Abbreviations

- RTLS - Real-Time Location System
- UWB - Ultra-Wideband
- LoS - Line of Sight
- GPS - Global Positioning System
- RSSI - Received Signal Strength Indicator
- RFID - Radio Frequency Identification
- AP - Access Point
- TDOA - Time Difference of Arrival
- AoA - Angle of Arrival
- RTT - Round Trip Time
- CI - Continuous Integration
- CD - Continuous Deployment
- PoE - Power over Ethernet
- QA - Quality Assurance
- TCP - Transmission Control Protocol
- UDP - User Datagram Protocol
- SSID - Service Set IDentifier
- MAC address - Media Access Control address

Appendix B

Coding rules

- Avoid global variables if possible. If not possible, consult and then avoid.
- Do not abbreviate variables/function names, use full words, like *calculateZpositionOfTagBasedOnBarometer* and not *calcZposBaro*.
- 1 whitespace around expressions, e.g. $a = b + 1$ and not $a = b + 1$.
- Whitespaces around language constructions, e.g.:

  ```
   for (int i = 0; i < length; i++) {
  ```

  and not

  ```
   for (int i=0;i<length;i++){
  ```

- Opening brackes on the same line as *if/while/for/etc*, e.g. :

  ```
      while (condition) {
  ```

  not

  ```
      while (condition)
      {
  ```

- Use const keyword for parameters that are not changing in given function, e.g.

  ```
  bool calculateZpositionOfTagBasedOnBarometer(double *z,
  const TCalculationStructure *calculationStructure) {
  ```

  so one can immediately tell that z is output parameter and *calculationStructure* is not being changed, just read.

- Do not ever use magic constants. Use:

  ```
      const double speedOfLight = 3e8;
      distance = time * speedOfLight;
  ```

not

```
distance = time * 3e8
```

- Use *camelCase*. Underscores are reserved for tests and for imported libraries.
- Be defensive. Check for all possible input values. Do not rely that they will stay the same. Check for *NULLs*. Check if *malloc* was successful.
- Try to keep functions short (<100 lines of code), keep files short (< 1000 lines of code). However, this is not strict - keep functions readable (do not artificially cut one readable 250 lines of code function into three).
- Think hard before implementing something that works in multiple threads. E.g. *strtok* is not thread-safe, *strtok_r* is. Be careful with shared variables. Use mutexes. **Deadlock is not an option**.
- Always use safe string functions to avoid buffer overflow, i.e. functions that take third parameter - max len of string (*strncpy* instead of *strcpy*, *strncmp* instead of *strcmp*, ...).
- Write tests before implementing.
- Check new functionality (mainly memory using functionality) with *valgrind*. **Memory leak is not an options**.
- Always use brackets for if, even for one line body, E.g. use:

```
if (cond) {
 a = b + 1;
}
```

and not

```
if (cond) a = b + 1;
```

- Use T prefix for structs (so one can immediately tell that this struct is defined by us and is not from third-party library):

```
typedef struct TPos {
    double x;
    double y;
} TPos;
```

- Before first float/double comparison, please read this: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- Predefined TAB key behavior set to 4 spaces utilizing tabulator, applied for *.ts, .js., .c, .c++* files.

Appendix C

Content of CD - Offline Example

C.1 Files

- src/
  - test-suite/test/test_controller.js - Main test file. Control most of the other files, making the initial initialization for every test and eventual cleaning after every test.
  - test-suite/test/rest-tests.js - All test for REST API.
  - test-suite/test/lint-test.js - Lint emmiter.
  - test-suite/ml_report_generator.js - Report Generator.
  - test-suite/ws_listener - module for catching positions from RTLS system.
  - test-suite/pcap_p.js - pcap Player.
  - test-suite/config.ini - main test-suite configuration file.
  - test-suite/A/, B/, C/, D/ - Anonymized .pcap files.
  - test-suite/mape.js - Module for calculation of MAPE.
  - test-suite/rmse.js - Module for calculation of RMSE.
  - test-suite/pac.js - Position accuracy calculator.
  - test-suite/create_svg - Module for plotting the generated data and gathered data to the svg html canvas.
  - test-suite/config.ini - Configuration file of test-suite. Here the user can set which test will run and which won't.
  - alphaTestScenario.xcl - Excel Sheet with Alpha testing manual tasks for RTLS Localization system.
  - gitlab-ci.yml - Example of GitLab CI/CD pipeline orchestrator.
  - tslint.json - TypeScript syntax checker.
  - eslintrc.js - JavaScript syntax checker.
- doc/
  - Source latex files of this paper.
- demo/

  - Live demonstration usage of this work, based on src/ files with RTLS System. For more information see manual.

- README.md
  - Description of CD content.

C.2 Manual

How to use this CD.

Manual was tested on Ubuntu 18.04 (VirtualBox 6.0) and Windows 10 with VirtualBox (6.1)

1. Unzip the folder.
2. Add the `.ova` image to the VirtualBox.
3. Run the machine.
4. Plug a LAN cable to you PC, so you can simulate Bridge network. Netplan configuration is already prepared for this network. Set Bridge Adapter mode in Networking settings of VirtualBox when you will be asked. Also set Name of interface to your ethernet.
5. Login as `dpormos` with password `dpormos`.
6. Go to folder `cd test-suite/`.
7. Run command *ifconfig* to find out which IP address your network assign to this machine.
8. Open file *config.ini* in your favorite editor, for example *vim* and change first line *IP_ADDRESS* from *192.168.225.2* to IP address you found in step 7.
9. Run `npm test`. Demonstration orchestration of test will run in order Lint, REST, Performance, report generator, pcap Player (A) and Unit tests.
10. Observer behaviour on your host web browser. While test are running you can visit RTLS System on `<IP>/sensmap` with the same credentials. And see how the system generates/replay positions.
11. After finish, visit `<IP>:6210` on your host and see results of final pipeline.

Appendix D

Online Example

For the easy demonstration process of this thesis and CI/CD pipeline for the reader I prepared online interactive solution.

This solution consist of two important parts:

1. GitLab Repository, contain:
    - all source codes of this thesis.
    - example .pcap files provided by Sewio.
    - .gitlabci.yml file orchestrating the CI/CD process.
    - CI/CD pipeline.

   There is a repository containing all files as CD. This repository can be accessed at `https://gitlab.com/michal.ormos/dp`.

2. Azure Virtual Machine (runner), contain:
    - provide as Runner to the GitLab
    - sample testing RTLS System
    - computing power
    - webserver for showing the results (on port 6210)

   With the help of Microsoft Azure Cloud service we create online testing server designed for this thesis. The server can be found on `http://ormos-masterthesis-demo.westeurope.cloudapp.azure.com`. Server is running test RTLS System for our pipeline. Credentials are same as in the offline example *dpormos/dpormos*.

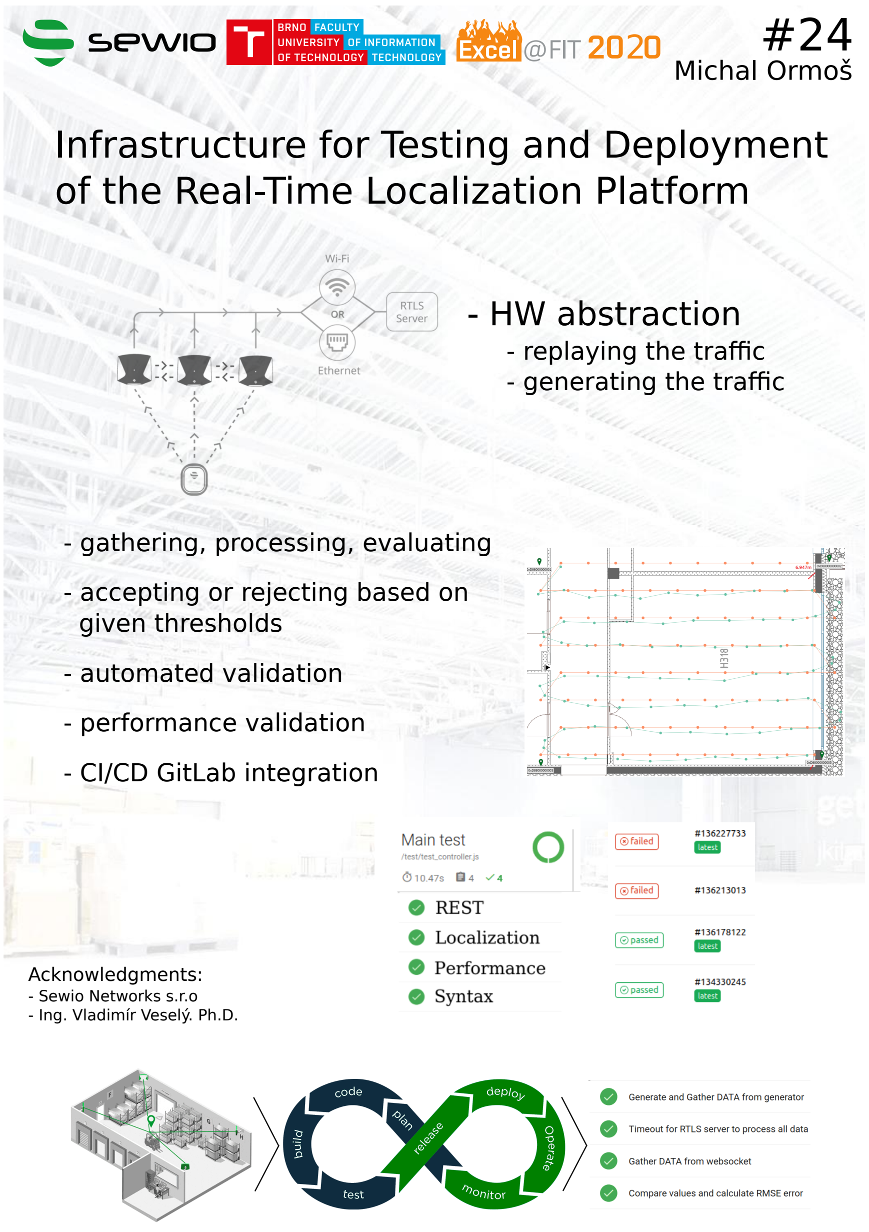
D.1 Manual

How to use online example.

- Open `https://gitlab.com/michal.ormos/dp` and observe CI/CD -> Pipelines.
- For new run click on *Run Pipeline* in the right top corner.
- Observe pipeline state in the runner and also on the azure server same as in the offline example.

Appendix E

Poster



#24
Michal Ormoš

Infrastructure for Testing and Deployment of the Real-Time Localization Platform

- HW abstraction
  - replaying the traffic
  - generating the traffic

- gathering, processing, evaluating
- accepting or rejecting based on given thresholds
- automated validation
- performance validation
- CI/CD GitLab integration

Main test
/test/test_controller.js
10.47s 4 ✓ 4

✓ REST
✓ Localization
✓ Performance
✓ Syntax

failed	#136227733 latest
failed	#136213013
passed	#136178122 latest
passed	#134330245 latest

Acknowledgments:
- Sewio Networks s.r.o
- Ing. Vladimír Veselý, Ph.D.

✓ Generate and Gather DATA from generator
✓ Timeout for RTLS server to process all data
✓ Gather DATA from websocket
✓ Compare values and calculate RMSE error