École polytechnique de Louvain

Sensor fusion at the extreme edge of an internet of things network.

Use-case : Real-time indoor person tracking using sonars

Authors: **Julien Bastin, Guillaume Neirinckx**
Supervisor: **Peter Van Roy**
Readers: **Igor Kopestenski, Etienne Rivière, Ramin Sadre**
Academic year 2019–2020
Master [120] in Computer Science

Declaration

We, Guillaume Neirinckx and Julien Bastin, hereby declare that we are the sole authors and composers of our thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, we declare that we have acknowledged the work of others by providing detailed references of said work.
We hereby also declare, that our Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Acknowledgments

This work is the result of the collaboration of a multitude of people whom we would like to warmly thank.

First and foremost, we would like to thank Professor Peter Van Roy for the countless advice and input he has given us throughout the entire work. He displayed a friendly, positive and supportive attitude during the whole process. He ensured we stayed on the right path and was able to see some real hidden opportunities on certain situations we faced.

We are equally thankful towards our Academic Advisor, Igor Kopestenski, for his support alongside Prof. Van Roy. The outcomes of the LightKone project allowed us to bring the state of the art further, as our own work could benefit from our advisors' technical experience.

We would also like to thank Peer Stritzinger for having guided us while we were improving a grisp module for the Pmod MAXSONAR.

Finally, we wanted to thank Secretary Vanessa Maons for all the efforts and administrative work she has done to exceptionally give us access, during the hard times of the COVID-19 outbreak, to a private room for our experiments in the Réaumur Building.

Abstract

We are witnessing these years an exponential increase in the number of IoT devices around the world. Consequently, the amount of data generated by sensor-equipped IoT devices is increasing a lot. Edge computing came up as a solution to offload the cloud, that couldn't keep pace anymore with this evolution. The workload has hence been shifted towards the edge of the network, closer to (but not at) the IoT devices.

With this evolution in mind, we propose an entire new approach where all the computations such as sensor fusion are done on the sensor-equipped devices themselves. We believe that this approach will be mandatory in the future, once the approach of performing all the computations at edge gateways or at the cloud won't be viable anymore with the billions of new IoT devices that are bound to come.

To show the feasibility of our approach, we have used it with a relevant use-case where sensor fusion is done on sonar-equipped IoT devices to perform indoor person tracking at near real-time. Indoor person tracking can be very useful for monitoring older persons in nursing homes.

We have implemented a resilient scalable system, using GRiSP boards as prototype IoT devices, that allows people detection of up to 2 people using trilateration. This system possesses an anti-crosstalk synchronization feature to prevent sonar crosstalk to take place. It also offers the possibility of having a live view of the surveyed area.

We have performed numerous deployments of our system under different scenarios to evaluate it. Encouraging result have been obtained with our approach, that has not constituted a limit at any time during our research. The only limit we have observed was the limited detection area of the sonars we used. The results of our use-case indicate that our approach can be very useful in the future. It could also be used as a complement to existing edge computing architectures, since it provides support for gateways.

Contents

List of Figures

List of Tables

List of Snippets

List of Algorithms

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
BEAM	**B**ogdan/Björn's **E**rlang **A**bstract **M**achine
BIF	**B**uilt-**I**n **F**unction
CDN	**C**ontent **D**elivery **N**etworks
CPU	**C**entral **P**rocessing **U**nit
CRDT	**C**onflict-free **R**eplicated **D**ata **T**ype
ERTS	**E**rlang **R**un **T**ime **S**ystem
GPIO	**G**eneral **P**urpose **I**nput/**O**utput
HTML	**H**yper**T**ext **M**arkup **L**anguage
I/O	**I**nput/**O**utput
IoT	**I**nternet **O**f **T**hings
ML	**M**achine **L**earning
OTP	**O**pen **T**elecom **P**latform
PID	**P**rocess **ID**entifier
Pmod	**P**eripheral **MOD**ule
RTT	**R**ound **T**rip **T**ime
SDRAM	**S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory
SPI	**S**erial **P**eripheral **I**nterface
TCP	**T**ransmission **C**ontrol **P**rotocol
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter
UDP	**U**ser **D**atagram **P**rotocol

List of Notations

`GRiSP`	GRiSP-base board produced by PEER STRITZINGER GMBH
`grisp`	GRiSP Erlang Runtime
`Pmod MAXSONAR`	Maxbotix Ultrasonic Range Finder
`pmod_maxsonar`	`grisp` module that interface with `Pmod MAXSONAR`
`Hera`	OTP application we developed to perform sensor fusion
`hera`	Erlang module of `Hera`
`Hera_synchronization`	OTP application we developed to perform synchronization of measurements
`hera_synchronization`	Erlang module of `Hera_synchronization`

Part I.

Foundation

1. Introduction

1.1. The increase in the number of IoT devices

The number of IoT devices around the world is growing rapidly. This number of connected devices is predicted to hit 50 billion [1] in 2020, exceeding by far the number of humans, personal computers and even mobile phones. In 2021, it is estimated that this number will rise with 35 billion new devices [2]. By 2030, it is expected that we would have around 125 billion devices around the world [3].

1.2. Limits of the classical cloud model

With high numbers like these, running applications on such an enormous network becomes very challenging. The number of IoT devices is already orders of magnitude larger than the number of computing nodes in data centers. To make it even more harder, its growth rate is also much higher than that of the number of computing nodes. Computation is not the only problem, also the network's latency and bandwidth limitations become problematic. It becomes harder to have an acceptable latency between devices and distant data centers for applications that would require immediate analysis like for an autonomous vehicle [4].

1.3. The computation shift towards the edge nodes

The solution is to make the applications run closer to the IoT edge devices [5], outside the data centers on the network's edge. Instead of making them run on distant data centers, we would run them closer to where they are needed. The advantage of this approach is that the latency is improved and that the amount of processing at the data centers is significantly reduced. This is traditionally done by offloading the cloud's computation to an entity, such as an edge gateway/device, that is situated close to the IoT edge devices and that can perform computations locally. However, it is possible however to go even further...

1.4. Sensor fusion at the edge devices

1.4.1. Sensor fusion

Sensor fusion is a certain type of computation performed by aggregating data from multiple sensors. This aggregated data is used to gain a more accurate picture of the sensors' subject or environment than can be determined by any one sensor alone [6]. Fusion of sensors' data, which is traditionally performed at the cloud, is a crucial computation that has become a very important matter in the context of Internet of things (IoT). It is a technique that enables better action/decision making by combining relevant data from different sources.

1.4.2. Proposed approach

We believe that in the future, the number of IoT devices will be so huge that it would be impractical to continue performing all the computations such as sensor fusion at the cloud (see 1.2) or at a nearby device such as an edge gateway (see 1.3). Once the number of nodes will reach this critical point, the same limitations as for traditional cloud computing will be observed again.

Instead, we propose to perform the fusion of the sensors' data on the sensor-equipped devices themselves that are situated at the extreme edge of the network. In our approach, we favor lateral sharing of data between the end user devices. Vertical data sharing with the upper layers (gateways and/or the cloud) should only be done when necessary in our view.

This implies that these devices must have enough computational power to perform relevant computations. If IoT devices are never used for anything else than sending collected data to edge servers, and spend most of the time waiting for results to come back, their processing and storage capabilities are not used even if they could be. Even with IoT devices whose computational power can be compared with that of personal computers from the 90s, the number of possible applications running on the extreme edge is enormous.

1.4.3. Approach advantages

This approach comes with great benefits, especially in the abundant cases where the need for edge gateways or the cloud can be eliminated. IoT sensor fusion-based applications following this approach won't have to rely on them anymore to work properly. The IoT sensor-equipped devices would be able to run these applications entirely on their own, acting as an autonomous entity. Consequently, edge gateways that crash temporally shouldn't have an impact anymore on the correct working of these sensor fusion based applications.

Removing the need for edge gateways is extremely cost-beneficial since less money/resources would be spent on buying/manufacturing and maintaining them. By eliminating the need for edge gateways and performing computations at the extreme edge on computation-capable sensor nodes, the number of needed devices becomes lower than with the traditional edge computing approach. This approach will come very handy in the future when the amount of edge gateways will become too high. As the amount of available resources on our planet is finite, decreasing the number of needed devices can only be advantageous.

Another big advantage is that by running the computations on the extreme edge, the latency becomes close to non-existent. This approach should therefore be very well suited for applications that require fast processing of data. Also, it adds the possibility for intra-layer data sharing at the layer made of the sensor-equipped devices. Finally, it is very good for data-privacy, as all generated data can be kept locally at this layer's network.

Nevertheless, our approach does not remove the possibility of compatibility with current existing models. Systems built with our approach can be fully interoperable with edge gateways. As a proof, the system we have built for our use-case allows to retrieve information via a gateway that is be present near the network of nodes.

1.5. Accurate real-time person tracking in a room

To show the feasibility of the approach described in Section 1.4, we have decided to experiment this approach with a relevant use-case that does sensor fusion at the extreme edge (at the sensor nodes themselves). The objective is to evaluate whether our approach can work for resource consuming applications where time matters.

1.5.1. Motivation

The total number of older persons all around the world is increasing significantly. For example, the number of persons around the world aged 80 years or over is projected to increase more than threefold between 2017 and 2050, rising from 137 million to 425 million [7]. Monitoring them while they are at home or inside buildings becomes a problem of significant importance. One of the best ways is by being able to localize these older people indoors. This could for example be used to track them when they walk out of their bed during the night (and possibly fall) [8]). It could also be used to detect and track an intruder in a protected location indoors.

1.5.2. Description

We have put into place a system that uses sensor fusion at the extreme edge with sonars to track a moving target at near real-time. The sensor fusion part is done by aggregating on each sonar-equipped IoT device the measurements originating from the other devices and from the device itself. This aggregation makes it possible to calculate the target's position. The system does near real-time computing as there is only a slight time delay introduced by data transmission and data processing. The computing is also soft real-time because the usefulness of the produced results decreases gradually over time and not immediately as in hard real-time computing [9]. This system also allows to retrieve information via a gateway to allow having a live view of the surveyed room. We have included a user manual for this system destined for any user interested in using it (see Appendix C).

1.6. Thesis contributions

Our thesis contributes in a lot of ways.

Sensor fusion at the extreme edge at near real-time Our thesis shows that the approach of performing sensor fusion at the extreme edge (i.e., at the sensor nodes) is a viable approach. It does so by using this approach with success for a relevant use-case that uses sensor fusion to compute relevant information (i.e., the position of a moving target in a room) at near real-time.

System for sonar-based indoor target tracking We have made the entirety our system/software available to the great public. This system can be used by anyone everywhere. The only requirement is to have sonar-equipped IoT devices that can run Erlang, and that possess a reasonable amount of processing power.

Scalability Any number of nodes can be part of our system. We have designed it in such a way that the number of nodes present in the system does not have an influence on the correct working of the system (see 5.2.2). The tests we have carried out with our system were done using 1 to 4 boards.

Resiliency to crashes No matter if one or more nodes in the network crash, the system would continue to work and give the position in near real-time, more or less accurately depending on the number of active nodes remaining (see 5.4.2.1 and 5.2.2).

Anti-crosstalk system for sonars During the development of our use-case, we have realized that the crosstalk problem (see 5.4.1) hindered significantly the results' accuracy (see 6.4.2.3) of our system. We hence developed a coordination system to make this problem disappear (see 5.4). This system is based on a global server that performs a round robin to tell the boards' sonars to make their distance measurement in turn. This server is located on one of the nodes of the cluster and is crash resilient. If the node on which the server is located crashes, the server is restarted on another node in the network.

Improvement of a grisp module for the Pmod MAXSONAR We have added a feature that was not available before and that we needed to avoid crosstalk between sonars. Indeed, with the current version of this module, the multiple sonars continuously emitted waves to measure a distance, which caused this problem of crosstalk. In our new implementation of this module, we have introduced the possibility to trigger the distance measurement when needed and to keep the possibility of using the continuous mode. The existing driver didn't have to be modified (see 5.5).

Interoperability with higher layers To demonstrate that it is possible for a gateway to retrieve data from our system, we have created a web application that can display at near real-time data sensed and computed by each node (see 5.6).

Live view of a person On the web application described in the previous point, we have added the possibility to visually observe in near real-time a person moving in a room, as illustrated in Figure 1. In this figure, we can see a person moving from left to right. The dark grey square represents the room. Each little red square is the location of a board and each cyan coloured square is the position calculated by one of the boards. Each board sends the position it has calculated through the network we have set up.

An *ad hoc* network Our system can establish a custom ad hoc wireless network for peer-to-peer communication, without using a pre-existing Wi-Fi network (section 5.3.4). It can also be reconfigured to connect to standard 2.4 GHz wireless access points.

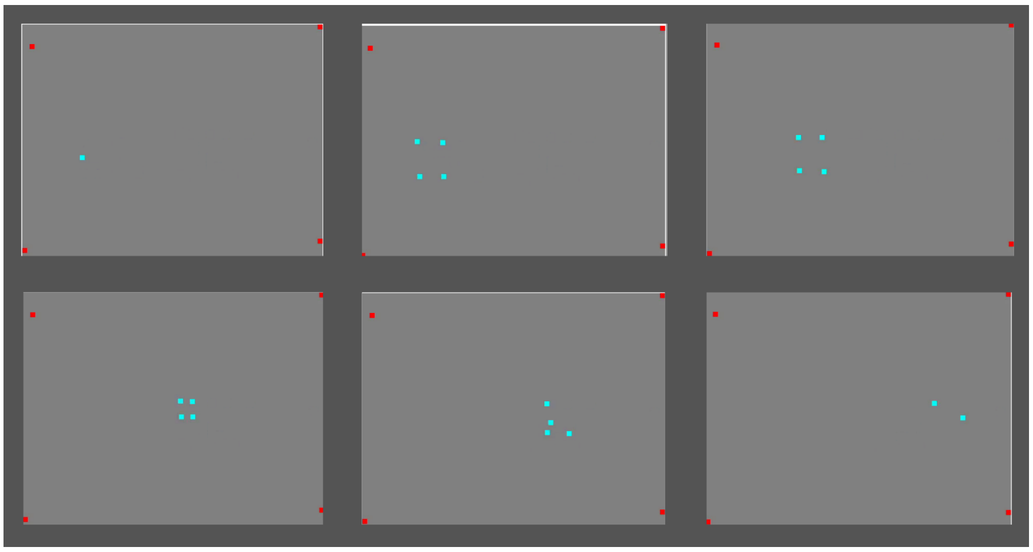


Figure 1.: Live view of a moving person in a room.

UDP multicast group communication Interoperability with a gateway is possible thanks to the ad hoc network of the IoT nodes as well as a multicast UDP group to which sensor data and calculation results are sent (section 5.3.4). To retrieve the information sent by each node, the gateway only needs to connect to the nodes' multicast group.

Genericity of our system Our system can be used by any type of IoT device supporting Erlang as well as with any type of sonar and data. It also allows to perform several measurements with more than one sensor at the same time on the same node. Multiple calculations can also be performed at the same time with our system (section 5.3.7).

1.6.1. Experimental results

What we list in this section are not contributions per se, but rather experimental results on existing technologies that may be of interest to users of these technologies.

Speed of UDP and TCP messages on GRiSP boards We have performed simple tests that can be replicated for measuring the end-to-end delay between GRiSP boards using UDP multicast and TCP. The description and results of these tests can be found at section 6.1

Technical limit of Pmod MAXSONAR We have tested the limit at which a person can be correctly and consistently detected by a Pmod MAXSONAR. The theoretical limit described in the datasheet is 8 feet (244 cm) [10]. We have shown in our results (see 6.4.2.1) how this limit is too high for applications where the sonar's readings must consistently detect a person.

1.7. Verdict

This thesis has generated a lot of results. The results we have obtained with our use-case allow us to conclude that the approach we proposed is a viable approach. We have developed a resilient system constituted of components that can each be reused for numbers of different use-cases. That system can track up to two targets in a room using sonar-equipped IoT devices, and this while performing all the sensor fusion calculations on these devices themselves. In this use-case, new sensed data needs to be generated and propagated very frequently by each device. New incoming sensed data also needs to be processed very quickly. The devices were able to do this while cooperating in a very efficient way as an autonomous entity. This allowed them to compute at near real-time (with a non-visible delay) the target's position. The approach has not shown any sign of limitation during the whole process. The only limitation we have encountered were the sonar sensors we employed, but they still were able to make the system we developed produce remarkable results.

2. Related Work

2.1. Edge and fog computing

Edge computing refers to the enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT services [11]. It appeared as a new distributed computing paradigm that brings computation and data storage closer to the location where it is needed, to improve response times and save bandwidth [12]. Its first form appeared in the late 1990s in the form of content delivery networks (CDN) to deliver web and video content by deploying edge servers close to users [13]. These edge servers then evolved in the early 2000s to also host applications and application components [14].
The term `fog computing` is a term created by Cisco that refers to a standardized way of performing edge computing [15]. Many use this standard as a jumping-off point for edge computing. It makes use of fog nodes that act as intermediary nodes between the cloud and the edge's sensor-equipped nodes, to bring the cloud services closer to the users [16].

2.2. Mist computing

Mist computing is a relatively new term that is not yet often used. It is referred to as computing at the very edge of a network, typically consisting of micro-controllers and sensors [17]. This approach is used as a complement to fog and cloud computing. Mist computing uses the computation capabilities available on the sensors to already perform some computations locally and to delegate the other computations to fog computing nodes. By for example filtering data locally, less data must be sent by that sensor node, which in turn conserves battery power as well as bandwidth [18]. The difference with our approach is that with mist computing, only a small part of the computations is performed at the very edge. This approach is still dependent on fog nodes and the cloud to function properly.

2.3. Data Fusion near the real edge

To the best of our knowledge, no literature exists yet that treat sensor fusion at the extreme edge of the network, i.e. at the sensor nodes. There are however numerous papers that treat sensor fusion **near** the extreme edge, often referring to this place as the `edge`. This term is somewhat ambiguous, since some papers consider local edge servers/gateways to be situated at the edge, while some others only consider the sensor nodes to be situated at the edge of the network. To avoid any ambiguity, sensor nodes would be referred to as "extreme edge devices" in the rest of our thesis. This paper for example [19] proposes the concept of temporal data fusion at the edge. Here, the process of sensor fusion is performed at edge servers, and not at the sensors. The only task that sensor nodes perform here, is to send data they senses to the nearest edge servers.

Our approach goes further than theirs by performing the sensor fusion at the sensor-equipped nodes themselves. In the case of a power outage between these nodes and the edge servers, our application would still be functional, while theirs would not.

2.4. Cognitive edge computing

This presentation [20] brings cognitive edge computing to the light. In this type of edge computing, edge devices near the IoT devices can perform cognitive computing. Cognitive computing is a resource-hungry task that's usually done at the cloud. The challenge of this approach is that the edge devices are short on power, networking, storage and computational resources [21]. The idea proposed by the presentation is to have an energy-efficient, collaborative sensing done by the IoT devices. It proposes to do so by having real-time coordination between them through the edge device. An ultra-low power battery-free sensor can for example be used to detect the right time to trigger a sensor such as a camera that requires more resources to use. The data produced by cameras also requires a lot of resources to be processed at the edge device. By not triggering it, this data won't be produced and the edge device won't have to process it. By making the edge device treat incoming data less often, interesting computations that are more resource-hungry and complex (for example cognitive/ML-based computations) can be done at the edge device by taking advantage of the resources spared thanks to the collaborative sensing. The difference with our approach is that with our approach, the coordination isn't orchestrated by an edge device, but by the IoT devices themselves. We will describe in Section 5.4

how we performed coordination to remove the problem of crosstalk between sonars.

2.5. The Achlys framework

Our thesis promotor Peter Van Roy and advisor Igor Kopestenski presented the Achlys framework and its applications in two papers [22, 23]. It is an edge computing framework that provides reliable storage, computation, and communication capabilities at the extreme edge of the network. It uses Partisan [24] for communication and Lasp [25] for providing efficient decentralized storage based on the properties of the CRDTs' (Conflict-Free Replicated Data Types). A CRDT is an abstract data type that is designed to be replicated at multiple processes or computers in a network [26]. They exhibit the following nice properties:

- Any replica can be updated concurrently and independently without coordination with another replica.
- It always possible to resolve inconsistencies that might result of these updates.

2.6. Trilateration

Trilateration is a method of control extension, control breakdown, and control densification that employs electronic distance-measuring instruments (EDMIs) to measure the lengths of triangle sides [27]. This method contrasts with triangulation that uses angle measurements instead. Sonars are very effective distance-measuring instruments. Assuming the sonars' locations are known, it is possible to determine a target's position by measuring the distances of the sonars to a target and performing trilateration. With the measures of only 2 sonars, we can limit the number of possible locations to 2. This figure 2 illustrates this with a vertical 2-dimensional view of 2 sonar readings r_1 and r_2. The red dots represent the 2 possible locations. In most applications, the incorrect location can easily be identified and filtered out. If that isn't the case, then one could make use of an additional sonar to compute the target's position without ambiguity.
The relationship with our work is that we use trilateration on the measurements yielded by sonar-equipped IoT devices.

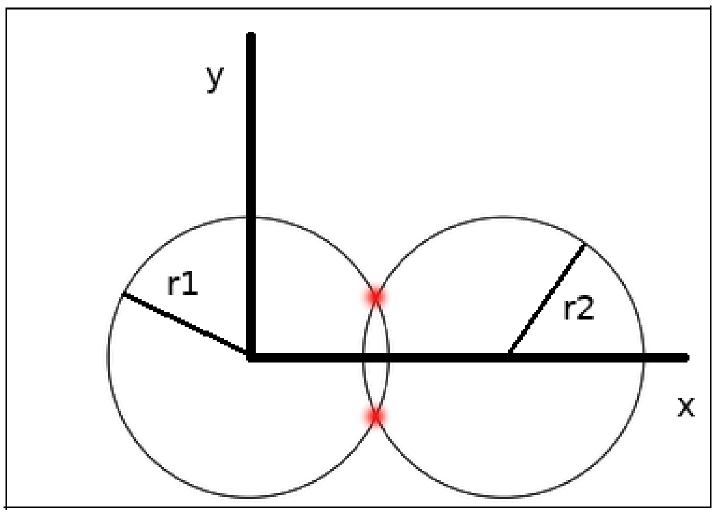


Figure 2.: Ambiguity using 2 sonars - Image was made on Gimp

Part II.

Tools

3. Resources used and methodology

3.1. Requirements for the experiment

To experiment with our thesis' use-case, we would need several hardware and software resources. The 2 sections below cover in a non-exhaustive way the hardware and software we have used to successfully develop our system and conduct our experiments.

3.2. Hardware

For the hardware part, we would need connected devices that can be equipped with sonars. They must also be physically capable of running applications entirely on their own, since this is what the thesis is about. We then of course would need sonars that are compatible with these devices. Each of these 2 hardware components are mandatory for our application. A detailed overview of our picks for these components is given below.

3.2.1. GRiSP boards

We have decided to use GRiSP boards (Figure 3) as our sensor-equipped edge devices. GRiSP boards are highly energy efficient IoT devices that are equipped with connectors to which PMod sensors and actuators can be attached. They can communicate over Wi-Fi and posess a MicroSD Socket to which standard MicroSD cards can be attached.

3.2.1.1. Motivation

There are several reasons why have chosen the GRiSP boards to represent our sensor devices located at the extreme edge. First of all, they are great to be used as prototype devices. They are the ideal candidate for the development of a multitude of sensor-based applications, since all Pmod connectors are already present on it.

Another advantage is that since we want to use Erlang (see 3.3.1) for developing our application, using GRiSP boards allows us to test our application very quickly. That is because no prior setup is required to install Erlang, as it is already included in this device by default. They are made to be used with Erlang since they boot

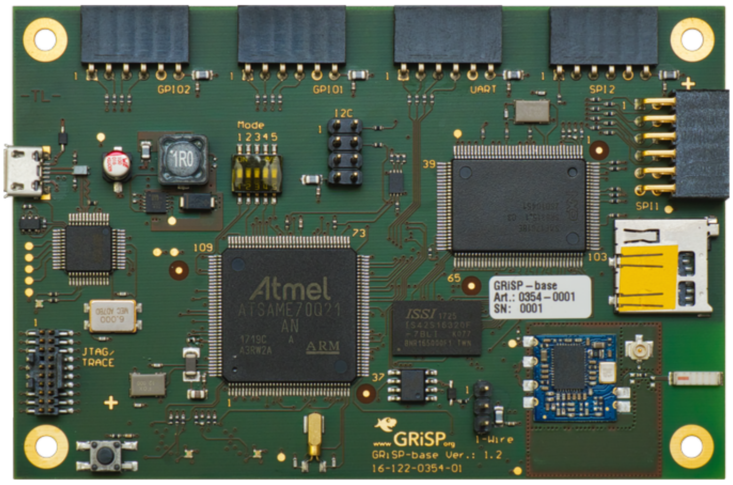


Figure 3.: GRiSP board - Image was taken from https://www.kickstarter.com/projects/peerstritzinger/grisp-2

straight into the Erlang VM running on real bare metal. Finally, this device also has the advantages of being energy-friendly and lightweight.

The price for GRiSP boards is at the time of writing €213. This price is however insignificant, as it constitutes only a small fraction of the expenses for developing an edge application. Buying cheap boards/devices for the development would be a false economy. Currently, GRiSP boards are an excellent choice to quickly develop Erlang applications on. These applications can, once they are ready, be deployed on cheaper mass-produced edge devices. These cheaper devices are permitted to be specifically designed for these applications and are allowed to only possess the hardware required for the application to work. GRiSP boards are great prototyping devices, but they are not well suited at the moment for finished products due to their price.

3.2.1.2. Specifications

The following specifications are taken from the GRiSP website[1].

CPU The GRiSP board possesses an Atmel SAM V71 processor which includes an ARM Cortex M7 core that runs at a clock speed of 300 MHz. This clock speed is comparable with that of processors from the 90s[2].

[1] https://www.grisp.org/specs/

[2] https://smoothspan.files.wordpress.com/2007/09/clockspeeds.jpg

Internal memory It hosts a 2048 Kb flash memory (used by the bootloader) and a 384 Kb SRAM.

External memory For the external memory, a 64Mb SDRAM is used, as well as a 2Kb EEPROM. The external memory also includes a standard MicroSD card that can be attached to the MicroSD Socket specifically designed for it.

Network The GRiSP board is equipped with a Wi-Fi antenna that communicates using the 802.11b protocol at a frequency of 2.4 GHz.

I/O The board comes with several connectors that are compatible with Pmod actuators and sensors manufactured by Diligent. Below is a list that details the different connectors present on the board.

- A Dallas 1-Wire via 3-pin connector
- A Digilent Pmod compatible I2C interface
- Two Digilent Pmod Type 1 interfaces (GPIO)
- One Digilent Pmod Type 2 interface (SPI)
- One Digilent Pmod Type 2A interface (expanded SPI with interrupts)
- One Digilent Pmod Type 4 interface (UART).

3.2.2. Pmod MAXSONAR sensor

The Pmod MAXSONAR (see Figure 4) is a very small sonar that can be connected to the GRiSP board via its Type 4 interface with the UART connector[3]. It makes use of the MaxBotix (MB) 1010 LV-MaxSonar-EZ1[4]. Within a certain range, users of this sensor can effectively measure how far an object or person is with an accuracy of 1 inch. Large objects can be detected by it at a distance up to 6.45 meters. It consumes very few power and it is designed to be used indoors for applications such as people detection, distance measuring, autonomous navigation for robots etc. The specifications below come from the datasheet of the MaxBotix 1010 LV-MaxSonar-EZ1 [10].

[3]https://store.digilentinc.com/pmodmaxsonar-maxbotix-ultrasonic-range-finder/
[4]https://www.maxbotix.com/Ultrasonic_Sensors/MB1010.htm?_ga=2.254695880.1132479797.1595168189-432722846.1590669867

Figure 4.: Pmod MAXSONAR: Maxbotix Ultrasonic Range Finder.

Range This sonar provides range information of objects that are situated up to 6.45 meters away with a resolution of 1 inch (2.54 cm).

Supply It operates from 2.5V to 5.5V and needs a low 2.0mA average current requirement.

Reading rate The reading rate of the sonar can go up to 20Hz. This means that one reading can be done every 50 milliseconds.

Recommended temperature range It is recommended to use this module in an environment where the temperature doesn't vary outside 0 to 60°C.

Beam pattern The datasheet shows how the detection patterns differ for tracked objects of different sizes (see Figure5). Chart A (resp., B) shows the detection pattern obtained when placing dowels of diameter 0.61 cm (resp., 2.54 cm) if front of the sonar. The pink dotted detection patterns on both charts were obtained supplying the MB1010 LV-MaxSonar-EZ1 a voltage of 3.3 volts. The pattern covers a larger area by supplying more voltage. Each square represents a surface of 30x30 cm.

People detection This sonar can according its datasheet detect people up to 8 feet (2.44 meters) away. By using a voltage of 5.5V, this detection range can me maximized. The pattern for people detection falls between sonar patterns A and B 5. During one of our experiments 6.4.2.1, we saw that this limit of 244 cm is too high for the sonar to consistently detect a person in our use-case.

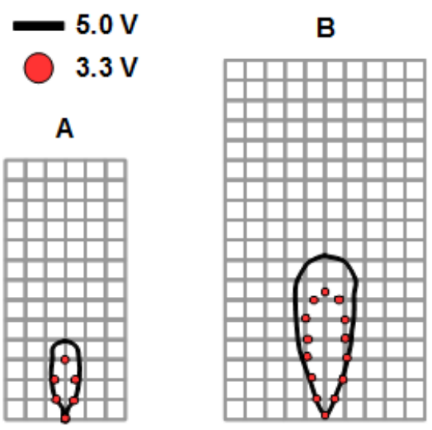


Figure 5.: MB1010 LV-MaxSonar-EZ1's beam patterns - Image was taken from its datasheet

3.3. Software

3.3.1. Erlang/OTP

3.3.1.1. Introduction

We used Erlang as our main programming language, which consists of the Erlang runtime system (ERTS), combined with OTP (Open Telecom Platform), which is a set of libraries (and design principles) for Erlang[5]. Originally, Erlang/OTP was a proprietary software of Ericsson created in 1986 by Joe Armstrong, Robert Virding, and Mike Williams. It was then released as a free open-source software in 1998 [28]. It has immutable data, pattern matching and functional programming [29].

3.3.1.2. Desirable properties

Erlang is specifically designed for systems that strive for the following traits:

- Robustness
- Soft real-time
- High availability
- Concurrency support
- Distributed

[5]https://www.erlang.org/

3.3.1.3. Modules

Erlang uses modules to allow separating functions into independent modules. Modules in Erlang are comprised of a sequence of module attributes followed by function declarations. Module attributes are used to define properties of a module. Erlang functions can be either named or anonymous. They have the capability of having guard sequences which will run the guarded function only when evaluated to true.

3.3.1.4. Processes

To support concurrency and distributed programming, Erlang makes use of lightweight processes, which are basically threads of execution that share no data with each other. Erlang processes are what allow functions contained into modules to be be executed. They can communicate/exchange information with one another in the form of messages (message passing).

3.3.1.5. Rebar3

Rebar3 is a build tool for Erlang/OTP sponsored by the Erlang/OTP team[6]. It is a very convenient tool that can be used to build, compile and deploy Erlang applications that adhere to OTP standards. It facilitates the process of creating, developing and releasing Erlang applications and libraries.

Dependencies management Each application created using Rebar3 contains its own `rebar.config` file. In Erlang/OTP, files with names ending with `.config` are configuration files. Files of this kind contain configuration parameters values for the applications in the system[7].
Dependency management using Rebar3 is very easy. To add a dependency to the application, one just has to add it to the list of the `deps` entry inside the `rebar.config` file. This dependency and the others will be resolved once the user executes the `rebar3 upgrade` command.

3.3.1.6. The rebar3_grisp plugin

Rebar3 proposes a plugin named `rebar3_grisp`[8] that provides users with a command for creating new projects targeted for GRiSP boards from a template. It also adds

[6] https://www.rebar3.org/
[7] http://erlang.org/doc/man/config.html
[8] https://github.com/grisp/rebar3_grisp

useful commands to Rebar3 for building and deploying such GRiSP applications.

3.3.2. Elixir, Mix and Phoenix LiveView

Elixir is an open source project, originally started by José Valim. It is a dynamic, functional programming language designed to run on the Erlang VM (BEAM). This allows developers to have access to Erlang's ecosystem including the battle-tested OTP framework. The language is a compilation of features from various other languages such as Erlang, Clojure, and Ruby [30, 31].

Mix is a build tool that provides tools for creating, compiling and testing Elixir projects, managing their dependencies and even more. It is the equivalent of Erlang's Rebar3 [32].

Phoenix is an elixir framework that allows to build rich, interactive web applications quickly, with less code and fewer moving parts [33].

Phoenix LiveView is a Phoenix library that provides rich, real-time user experiences with server-rendered HTML [34].

3.4. Methodology

We followed an incremental approach for implementing our application. Every time we faced an issue or had to implement a new feature, we brought change to our application and thoroughly tested the changed component. Before testing new functionalities on the sensor devices, we first tested them when possible with device emulations on the shell. The objective of this testing part is to filter as much development errors as possible.

Testing functionalities on the devices themselves is a time-consuming process since it includes the deployment on the device and the device's startup. We therefore prioritized testing it on emulations first. If this stage is successful, we then tested the new functionality by deploying it on the sensor devices. We have also frequently used regression testing on our previously developed and tested functionalities to check whether they still perform well after some changes.

Part III.

Experimentation

4. Algorithm

4.1. Foreword

Before diving directly into the implementation, it is better to first describe in natural language how the whole procedure works. By making abstraction of the tools (GRiSP boards, Pmod MAXSONARs, Erlang/OTP, Rebar3, ...) and details about techniques we use (trilateration, filters, measurement synchronization, ...), it becomes easier to reason about how the whole system works.

As a reminder, the goal is to track a moving target indoors using multiple sonar-equipped connected devices. It has to be done in such a way that these devices perform all the computations themselves. They must keep working and collaborating correctly even in the case they are disconnected from the external world.

4.2. Description

The way the whole system comprised of sonar-equipped nodes works is as follows. They collaborate in such a way that only one node at a time triggers a sonar measurement. Each node, turn by turn, performs a sonar measurement. By doing so, the problem of cross-talk (see 5.4.1) between sonars is solved. These sonar measurements are done very frequently (more than one per second) to detect changes in the target's position in a quick manner. Every time a sonar measurement has been made by a node, the obtained information is shared with all other nodes by making the node send it to them.

Since for our use-case, all the nodes of the network are in the same room, we assume that non-crashed nodes have the ability to contact each other at any time. A consequence of this assumption is that we do not have to worry about possible network partitions.

To ensure that the different nodes synchronize between each other to together perform only one sonar measurement at a time, we make use of a synchronization server. This server runs on a single node in the network. On top of that, a client part runs on each node of the network and is in charge of contacting the server. Algorithms 1 and 2 describe the general functioning of the client side and the server side.

At the same time as the measurements are made, calculations can be performed at a user-specified frequency. The algorithm 3 describes how calculations are performed. The algorithm 4 shows how the detection of up to 2 persons is performed. More explanation about this algorithm can be found at 5.2.2.1.

What we aim to achieve by doing so on each node, is that each node has access the most recent sonar reading from every node part of the system. Each node can then, based on the obtained sonar readings from every node including itself, calculate the position of a static/moving target using trilateration.

Algorithm 1 Synchronization of measurements, client side.

```
MeasurementOfAllNodes ← new_dictionary()
CurrentNodeId ← get_current_node_id()

procedure PERFORM_MEASUREMENTS(Iterations, FilterFunction)
    send_measurement_request_to_server(CurrentNodeId)
    $I \leftarrow 1$
    while I $\leq$ Iterations do
        wait_from_server(perform_measurement_permission)
        MeasurementResult ← make_measurement()
        if I $\neq$ Iterations then
            send_ack_to_server(CurrentNodeId)
        end if
        if not(FilterFunction(MeasurementResult)) then
            send_to_other_nodes(MeasurementResult)
        end if
        add_or_replace(MeasurementOfAllNodes, CurrentNodeId,
MeasurementResult)
        I := I+1
    end while
end procedure

procedure RECEIVE_MEASUREMENT(NodeId,Measurement)
    add_or_replace(MeasurementOfAllNodes, NodeId, Measurement)
end procedure

function ADD_OR_REPLACE(Dictionary, Key, Value)
    if Dictionary.is_key(Key) then
        Dictionary.replace(Key, Value)
    else
        Dictionary.add(Key, Value)
    end if
end function
```

Algorithm 2 Synchronization of measurements, server side.

```
 1: NodeOrder ← new_queue()
 2:
 3: procedure Receive_measurement_request(NodeId)
 4:     NodeOrder.put_last(NodeId)
 5: end procedure
 6:
 7: procedure Dispatch_turn
 8:     FirstNode ← NodeOrder.remove_first()
 9:     FirstNode.send(perform_measurement)
10:     wait_ack(FirstNode)
11:     NodeOrder.put_last(FirstNode)
12: end procedure
```

Algorithm 3 Calculations process

```
 1: procedure Perform_calculation(Iterations, FilterFunction, WaitTime)
 2:     $I \leftarrow 1$
 3:     while I ≤ Iterations do
 4:         sleep(WaitTime)
 5:         Measurements ← get_all_measurements()
 6:         Results ← calculation(Measurements)
 7:         if not(FilerFunction(Result)) then
 8:             send_to_other_nodes(Result)
 9:         end if
10:     end while
11: end procedure
```

Algorithm 4 Algorithm for the detection of up to 2 persons

```
 1: procedure Detect_up_to_2_persons(Sonars, Measurements)
 2:     if Measurements.length() >= 4 then
 3:         ContiguousSonarPairs ← get_contiguous_sonars(Sonars)
 4:         Positions ← new_list()
 5:         for SonarPair in ContiguousSonarPairs do
 6:             Positions.append(trilateration_2_measurements(SonarPair, Measurements))
 7:         end for
 8:         Positions := fuse_close_positions(Positions)
 9:     end if
10:     Positions ← detect_one_person(Sonars, Measurements)
11:     return Positions
12: end procedure
```

4.3. Trilateration

With known sonars' distances to an object, and with sonars placed on different positions, we can calculate the position of that object using trilateration (described in 2.6). We have developed 2 different trilateration formulae that are used depending on how much recent sonar measurements are available (coming from different sonars).

4.3.1. Trilateration using 2 available measurements

We consider a two-dimensional Cartesian space for defining the sonars' positions. The sonars' positions can differ in abscissa and/or in ordinate. As shown in Section 2.6, performing trilateration with 2 sonar measurements renders 2 different possible positions, or only one in the rare cases where the target's position forms a straight line with the 2 sonars' positions. In some cases where 2 possible positions are obtained, the incorrect position can easily be identified and filtered.

4.3.1.1. Formula derivation

During our research, we have come across very few formulae for doing trilateration with 2 measurements. The few of those we encountered (for example this one [35]) could only be used in the case the two sonar's ordinate are the same, which is not always the case in our application. One example in which this does not hold would be when 2 sonars sit at the opposite corners of a room. That is the reason why we had to derive our own formulae.

The formula we have derived is based on the Pythagorean theorem. To make it clearer, an illustration 6 is provided. We consider two sonars that each produce measurement ranges r1 and r2. We suppose that the first sonar is situated at position (0,0). The second sonar's position varies from the first sonar's position with a certain horizontal separation u and vertical separation v. The target's position is at the unknown point (x,y). On the illustration 6, it is appears twice since it is a case where there is ambiguity regarding the target's position after applying trilateration. The Pythagorean theorem yields the following two equations.

$$r_1^2 = x^2 + y^2 \tag{1}$$

$$r_2^2 = (u - x)^2 + (v - y)^2 \tag{2}$$

The first (resp., second) equation was obtained using the known coordinates of the

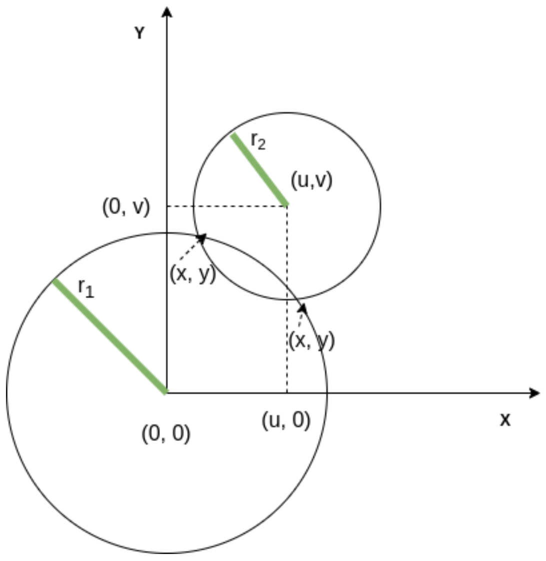


Figure 6.: Formula derivation illustration - Image was made on draw.io

first (resp., second) sonar. Using Wolfram[1], we were able to quickly solve these two equations by expressing the unknown parameters x and y using the known parameters r1, r2, u and v. When v equals 0 (i.e., the 2 sonars are at the same ordinate), x and y are given as follows.

$$\begin{gathered} x = \frac{r_1^2 - r_2^2 + u^2}{2u} \\ y = \pm\sqrt{r_1^2 - \frac{(r_1^2 - r_2^2 + u^2)^2}{4u^2}} \end{gathered} \tag{3}$$

Below is when they are not placed at the same ordinate ($v \neq 0$).

$$\begin{gathered} x = \frac{r_1^2 u - \sqrt{-v^2(r_1^4 - 2r_1^2(r_2^2 + u^2 + v^2) + (-r_2^2 + u^2 + v^2)^2)} - r_2^2 u + u^3 + uv^2}{2(u^2 + v^2)} \\ y = \frac{r_1^2 v^2 + u\sqrt{-v^2(r_1^4 - 2r_1^2(r_2^2 + u^2 + v^2) + (-r_2^2 + u^2 + v^2)^2)} - r_2^2 v^2 + u^2 v^2 + v^4}{2v(u^2 + v^2)} \end{gathered} \tag{4}$$

or

$$\begin{gathered} x = \frac{r_1^2 u + \sqrt{-v^2(r_1^4 - 2r_1^2(r_2^2 + u^2 + v^2) + (-r_2^2 + u^2 + v^2)^2)} - r_2^2 u + u^3 + uv^2}{2(u^2 + v^2)} \\ y = \frac{r_1^2 v^2 - u\sqrt{-v^2(r_1^4 - 2r_1^2(r_2^2 + u^2 + v^2) + (-r_2^2 + u^2 + v^2)^2)} - r_2^2 v^2 + u^2 v^2 + v^4}{2v(u^2 + v^2)} \end{gathered} \tag{5}$$

[1] https://www.wolframalpha.com/

Two different (x,y) pairs are produces most of the time. It is assumed that the sonars aren't placed at the same position ($u^2 + v^2 \neq 0$). It also works for when the first sonar (the one that produces r_1) isn't positioned at (0,0). All one has to do then is to add to each resulting (x,y) pair the position of the first sonar. If for a certain resulting (x,y) pair, the first sonar's position is (a,b), then the resulting target's position would be (x+a,y+b).

4.3.2. Trilateration using 3 available measurements

When we want to detect the position of a single person with 3 or more measurements (originating from different sonars), we only select 3 of them and perform trilateration with those 3 measurements. Since there are 3 of them, there cannot be ambiguity regarding the target's position as there would be with only 2 measurements (figure 6). The trilateration formula 6 we use here is another one that we have also derived using the Pythagorean theorem.

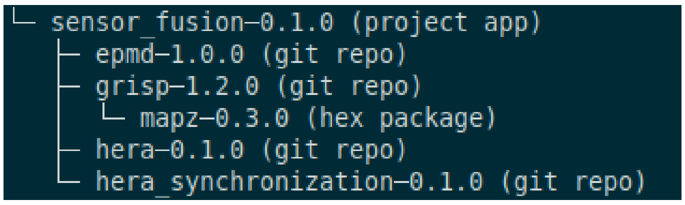
This formula calculates, for a node with id i, the position $(x_{p,i}, y_{p,i})$ of the person p in the room. Part of the known parameters in this equation are n, the number of sonar-equipped nodes present in the system, and (x_i, y_i), the coordinates of node i. The other known parameters are the coordinates $(x_{(i-1)\%n}, y_{(i-1)\%n})$ and $(x_{(i+1)\%n}, y_{(i+1)\%n})$ of the 2 neighbouring nodes with ids $i-1$ and $i+1$. Parameters v_i, v_{i-1} and v_{i+1} are the sonar measurements obtained by the node i and its two neighbours.

$$\left\{\begin{array}{lcl} x_{p,i} & = & \frac{C_iE_i - F_iB_i}{E_iA_i - B_iD_i} \\ y_{p,i} & = & \frac{C_iD_i - A_iF_i}{B_iD_i - A_iE_i} \\ A_i & = & -2x_{(i-1)\%n} + 2x_i \\ B_i & = & -2y_{(i-1)\%n} + 2y_i \\ C_i & = & v_{(i-1)\%n}^2 - v_i^2 - x_{(i-1)\%n}^2 + x_i^2 - y_{(i-1)\%n}^2 + y_i^2 \\ D_i & = & -2x_i + 2x_{(i+1)\%n} \\ E_i & = & -2y_i + 2y_{(i+1)\%n} \\ F_i & = & v_i^2 - v_{(i+1)\%n}^2 - x_i^2 + x_{(i+1)\%n}^2 - y_i^2 + y_{(i+1)\%n}^2 \end{array}\right. ,\forall i \in NodeIds \; ; n = NbNodes \tag{6}$$

5. Implementation

5.1. Architecture

We have developed four applications for our system: one `grisp` application[1], two `Erlang OTP` applications [36] and one `Phoenix LiveView` application[2]. You are invited to consult Appendix A to obtain the links to each GitHub repository of these applications. The installation instructions are included also included in Appendix C. The two `Erlang/OTP` applications we have developed are called `Hera` (section 5.3) and `Hera_synchronization` (section 5.4). We have designed them in such a way that it permits doing sensor fusion on any kind of node that supports Erlang. The `grisp` application is called `sensor_fusion` (section 5.2) and allows us to deploy `Hera` (section 5.3) and `Hera_synchronization` (section 5.4) on `GRiSP boards` for doing computations at the extreme edge using these two applications. Finally, the LiveView application is called `sensor_fusion_live_view` (section 5.6) and gives us the opportunity to have a view of what is sensed by the nodes at near real-time. You can find the dependencies between the two `Erlang/OTP` applications and the `grisp` application in Figure 7.



```
└─ sensor_fusion─0.1.0 (project app)
   ├─ epmd─1.0.0 (git repo)
   ├─ grisp─1.2.0 (git repo)
   │  └─ mapz─0.3.0 (hex package)
   ├─ hera─0.1.0 (git repo)
   └─ hera_synchronization─0.1.0 (git repo)
```

Figure 7.: Dependencies between our `Erlang/OTP` and `grisp` applications

5.2. Sensor_fusion

This `grisp` application has been made to perform sensor fusion at the extreme edge of the network, more precisely at sensor nodes. We have included in this application a module containing the necessary implementation of our use-case (Section 1.5). This application works with the `Hera` sensor fusion framework we have designed. It

[1]`https://github.com/grisp/grisp/wiki/Creating-Your-First-GRiSP-Application`
[2]`https://hexdocs.pm/phoenix_live_view/Phoenix.LiveView.html`

also makes use of `Hera_synchronization` for synchronizing the sonar measurements between the nodes.

To be able to use the above two `Erlang/OTP` applications we designed, it is necessary to include them as dependency applications inside the `rebar.config` file of the project (see 3.3.1.5). We have also includeded `grisp` as a dependency in order to be able to deploy our application on `GRiSP` boards. We have a fourth dependency on our application, called `epmd`[3], which is necessary for the proper functioning of the program on `GRiSP` boards. This last dependency is added by default when creating a new `grisp` application. The code snippet 5.1 illustrates these dependencies.

```
{deps, [
    {hera, {git , "https://github.com/guiste10/hera.git" , {branch , "master"}}},
    {hera_synchronization, {git, "https://github.com/bastinjul/hera_synchronization", {branch, "master"}}},
    {grisp, {git, "https://github.com/bastinjul/grisp.git", {ref, "a1e3f2c"}}},
    {epmd, {git, "https://github.com/Erlang/epmd", {ref, "4d1a59"}}}
]}.
```

Snippet 5.1: Sensor_fusion dependencies

5.2.1. Target tracker app

The implementation of our use-case is located inside the `hera_position` module. You can find the API functions of this module in Appendix A.1.

This API gives access to two principal functions. The function `launch_hera` is used to start the sonar measurements and the trilateration calculations on a single board. The other function `restart` servers to restart the procedure by changing some parameters. The first function can take from five to seven parameters, depending on what you want to do. Notice that the function `launch_hera` must be called on each board with the correct parameter values in order to make our system work correctly. At least 2 nodes must be used in order to be able to calculate a target's position with or without ambiguity, depending on how they are placed (see 5.2.4.1). The more nodes are used, the greater the room's surveyed area can become by spreading them intelligently across the room.

launch_hera/5 takes as parameters `PosX` and `PosY`, representing the (x,y) position of the node inside the room. The nodes can be placed on any location as long as the nodes' abscissa and ordinate are non-negative.

[3] https://github.com/Erlang/epmd

The `NodeId` parameter represents the id of the node. All identifiers that are part of the cluster must be different and start at 0. Adjacent nodes must have adjacent identifiers (except node 0 which has node 1 as its neighbor and the node with the highest id if there are more than 2 nodes in the cluster). When using for example 3 nodes, the id's would have to range from 0 to 2.
The `{MinX, MinY}` and `{MaxX, MaxY}` parameters express the minimum and maximum coordinates respectively where a person or object can move inside the room. How they are used is explained in 5.2.4.1.

If you choose to start the trilateration on a board with this function of 5 parameters, a calculation (see 5.3.6) of the position will be made every 50 ms and the measurements (section 5.3.2) of the sonars will be synchronized (see 5.4) between boards. The measurements and calculations will go on forever.

launch_hera/6 takes the same parameters as `launch_hera/5` with the extra parameter `MaxIterations`, that represents the maximum number of calculations and measurements to be made before the node goes to sleep. The frequency of the calculations stays at 50 ms and the measurements of sonars are also synchronized.

launch_hera/7 takes the same parameters as `launch_hera/6` with the extra parameter `Frequency`, the frequency at which the calculations of the position of the object and the sonar measurements are made. With this arity, the measurements are set to be non-synchronized.

If you choose to use one of the previous functions with arity 6 or 7, the measurements and calculations will stop once the maximum number of iterations is reached. You can restart new calculations and measurements with the second function `restart/2` which takes `Frequency` and `MaxIterations` as parameters.

5.2.2. Measurement and calculation functions

We have included in this module the 2 necessary functions that make measurements and calculations, as well as the two functions needed to filter the values resulting of these functions. These 2 functions are defined in order to interface with the `hera` module from the `Hera` app. The measurement function makes use of the `pmod_maxsonar` module that interfaces with the `Pmod MAXSONAR` sensor to return a value expressed in centimeters.

5.2.2.1. Calculation function

The calculation function we have included is used to calculate the position of one to two targets in the room. This calculation function uses different trilateration formulae (see 4.3.1 and 4.3.2) depending on how much recent measurements coming from different nodes are available, and depending on whether the user wants to detect only one or up to two persons. As mentioned in 4.3.2, for the detection of one person, if more than 3 recent measurements are available, the calculation function will only select 3 of them to perform trilateration. This ensures that the calculation function will compute a position for any number of nodes higher than 3. The more nodes are used, the higher the chance of having at least 2 or 3 of them detect a target.

For the detection of up to two persons, as shown on the algorithm 4, at least 4 sonars must be used and placed in such a way that they form a rectangle. This allows our system to effectively use a filter (described in5.2.4.1) to remove the ambiguity on each position calculated with each pair of contiguous sonars. Our algorithm works in 2 steps, it first uses the trilateration formula for 2 measurements (see 4.3.1), with the measurements produced by each pair of contiguous sonars, to calculate a position for each pair. It then returns 2 positions in the case 2 different positions have been computed that are separated from each other with more than 40 cm. In the other cases, it returns only one position or none if no person has been detected. Our algorithm is rudimentary however and doesn't work in all cases, especially when the 2 people are situated more or less in the center of the room. Figure 8 illustrates such a problematic case. Here, 4 different positions would be generated that are separated from each other with more than 40 cm. The blue persons are where the 2 persons are actually located.

5.2.3. Sonar measurement filter

Filter functions are passed from the `hera_position` module of `Sensor_fusion` to the `hera_filter` module of the `Hera` application. They are explained in subsections 5.2.3 and 5.2.4 We have implemented a filter that has been specifically designed for the sonars that are part of our use-case. This one is strongly based on the generic filter described in 5.3.3.1. In our use-case, we typically set the upper bound value to 0.28 cm/ms. This corresponds to a speed of ±10 km/h. This assumes that a walking person's distance to the sonar never varies faster than this limit. This filter differs from the generic filter 5.3.3.1 with only a few differences.

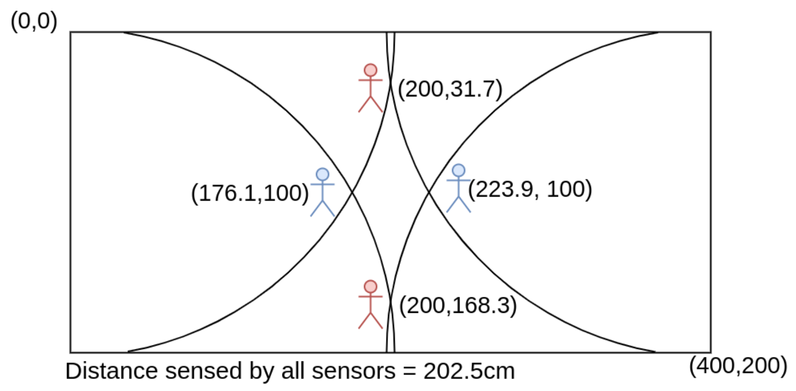


Figure 8.: 4 points are possible for 2 persons in the room - Image was made on draw.io

It makes use of the default measured value of the sonar that was measured in the warm-up phase (see 5.3.2.1). The warm-up phase is used to define the maximum distance that can be detected by a sonar, e.g., the distance to the closest wall in front of the sonar. During the warm-up phase, no target is allowed to be present in the range of the sonars.

If a measured value is very close to or bigger than this default measured value, the value measured by the sonar will be filtered out. This has as consequence that only the relevant measured values are kept, i.e. values that represent the distance of the sonar to the target, and not the distance to a static object or wall. This assumes that with a moving target in a sonar's range, the distance can only be smaller than the value produced during the warm-up phase.
This filter only permits the measurement to change faster than the upper bound in the case that the previous measurement couldn't detect the target. If the measurement's value has decreased significantly in a non-gradual way, then that would mean that the target has just come in the range of the sonar. This measurement should thus pass the filter.

5.2.4. Position filtering

Each position (or pair of positions in the case of ambiguity) that was generated with trilateration must pass the following 2 filters.

5.2.4.1. MinX, MinY and MaxX, MaxY filtering

After having obtained one or two possible target positions with trilateration, we make them pass through a filter that only keeps the positions that are inside the `(MinX, MinY)` and `(MaxX, MaxY)` bounds specified by the user in `launch_hera` (5.2.1). Both the target's abscissa and ordinate must be inside these bounds for the position to be valid. The bounds are defined as `[MinX, MaxX]` for the abscissa and `[MinY, MaxY]` for the ordinate.

This filtering of the positions obtained with the trilateration of 2 measurements is very useful for eliminating the incorrect position. It is especially so in the case where a pair of sonars is used for the trilateration, and that these sonars are placed against the same wall in a rectangular room. With well specified bounds, the incorrect position can always be filtered out. Thanks to this, it is possible to calculate the target's position using only 2 sonars.
Figure 9 illustrates this. Here, one would use {0,0} as `{MinX, MinY}` and {4,2} as `{MaxX, MaxY}` values for the parameters in `launch_hera`. The 2 sonars are placed on positions (0,2) and (4,2) respectively. The arcs represent all the possible positions of the target when each sonar's measurement is considered individually. Using trilateration, the positions obtained are those 2 where the arcs intersect. The filtering would only keep as possible position the one that is inside the rectangle, as the other one violates the `MaxY` bound.

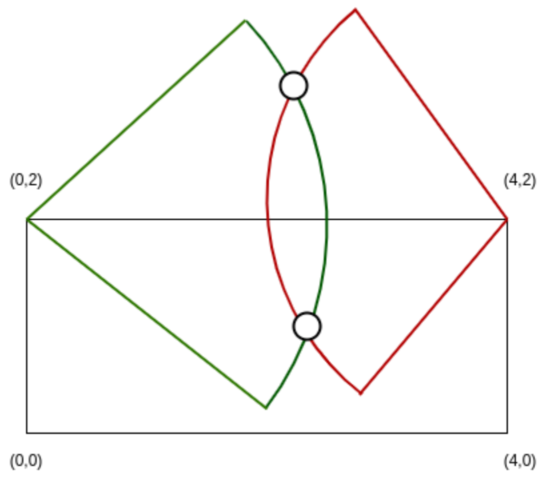


Figure 9.: Incorrect position identification - Image was made on draw.io

5.2.4.2. Speed filtering

In addition to the room position filter, we have added a speed filter. This filter is entirely based on the generic filter (described in5.3.3.1). In the same way as the sonar filter, we have placed an upper bound of 0.28 cm/ms, which corresponds to a maximum speed of 10 km/h. Based on the previous calculated position that has passed the filter, if the current position violates that speed limit, that position is filtered.

5.3. Hera

`Hera` is an OTP application we have created that allows fusion of sensor data to happen on multiple nodes. Thanks to its support for concurrency (using Erlang processes, 3.3.1.4), it provides the possibility to fusion in a concurrent way several types of data generated from different sensors. We have designed this application to be as generic as possible so that it can be used with any type of sensor data for any use-case. It allows for example to perform sensor fusion on data generated by sonar sensors, light sensors etc. It is important to point out that the sensor fusion only considers the most recent obtained data. Calculations using data to perform sensor fusion only have access to the most recent data from each sensor that is present in the system. This isn't such a limitation since there are countless use-cases where only the most recent data is needed, such as in ours. It also removes the risk/possibility of having the memory using more and more space until the memory becomes full. The number of possible applications using `Hera` is huge. This application isn't only destined for GRiSP boards. It can also be used on any device supporting Erlang, as long as this device supports connectivity and can be equipped with sensors.

5.3.1. Supervision tree

In Erlang, a supervision tree is a tree of processes. There are two kinds of processes in such a tree. There are the workers and the supervisors. The workers are the modules containing the code specific to the application and the supervisors are used to manage the workers or other supervisors. If a child process of a supervisor fail, the latter can restart it. In order to be a supervisor, a module must implement the `supervisor` behaviour [37, 38, 39].

There are four types of possible restart strategies for a child of a supervisor. We only use two of them in our tree, the `rest_for_one` and the `one_for_one`.

The rest_for_one strategy is used in the case where you want to restart some processes that are dependent on other processes. If you specify this strategy, whenever a child process fails, all the other child processes that have been started by the supervisor after this process are terminated. Then, the failed process is restarted, followed by all other child processes that have been started (and forcefully terminated) after this failed process. For example, given the list of processes to be started in line 7 of this code snippet 5.2, if the process B fails, then the processes C and D would be restarted after the process B.

The one_for_one strategy is used whenever a process is independent from other processes. Only the failed process is restarted using this restart strategy.

You can also specify the `intensity` and the `period` of the restarts. Assuming the values `MaxR` for `intensity` and `MaxT` for `period`, then, if more than `MaxR` restarts occur within `MaxT` seconds, the supervisor terminates all child processes and then itself [39]. The code snippet 5.2 also illustrates how to specify such values.

```
  init(A,B,C,D) ->
2     MaxRestarts = 6,
      MaxSecondsBetweenRestarts = 3600,
4     SupFlags = #{strategy => rest_for_one,
          intensity => MaxRestarts,
6         period => MaxSecondsBetweenRestarts},
      {ok, {SupFlags, [A,B,C,D]}}.
```

Snippet 5.2: `rest_for_one` strategy illustration

We only start filter (5.3.3), measurement (5.3.2) and calculation (5.3.6) processes on `GRiSP boards`, not when using an emulation of GRiSP boards on PC. We use this emulation to make the PC act as a gateway for our LiveView application. You can find the supervision tree in both cases in Figure 10 and in Appendix 29 respectively. At each level of the tree, processes that are started first sit on the top of the level. For the restarting strategies, the `hera_supersup` process for example has defined a `rest_for_one` restart strategy for its two children. This has as consequence that if the process of the `hera_sensors_data` module fails, then then all other processes of `hera` are restarted. The module `hera_supersup` is the one in charge of assembling the supervision tree. Concerning the filter, measurement and calculation modules, we use a pool of processes in order to have the possibility of starting processes of these types at runtime with custom arguments. The implementation of these pools is inspired of `https://learnyousomeErlang.com/building-applications-with-otp`.

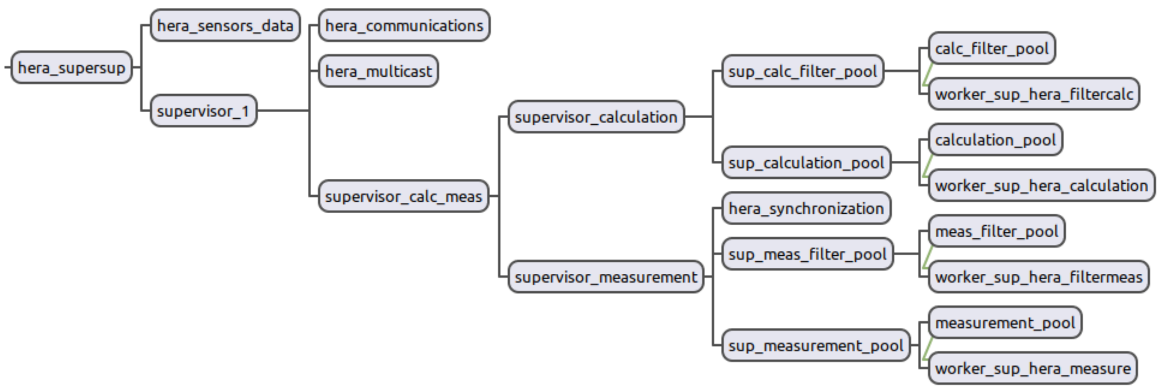


Figure 10.: Supervision tree of `Hera` on `GRiSP boards`

5.3.2. Measurements

The module `hera_measure` allows to perform at the same time multiple types of measurements. Indeed, one process is started per kind of measurement the user desires to perform on a device.

We make a distinction between two types of measurements: the measurements that are synchronized between the nodes and the non-synchronized measurements. These two types are each characterized by several elements.

The synchronized measurements are characterized by a `name` which uniquely identifies a kind of measurement, a `measurement function` that allows the module to perform the measurements, and a `maximum number of iterations` which can be a finite number or the atom `infinity`. It defines the number of times the `measurement function` will be called. They are also characterized by an `upper bound` value that indicates the maximum allowed variation per milliseconds between two successive measurements, and by a `filtering function` that returns a boolean value to indicate when the result of the measurement has to be filtered out. The upper bound value is supposed to be used by the filter. If the measurements do not need to pass through a filter, the filtering function can be replaced with the atom `undefined`. In this case, the upper bound value can take any value as it won't be used by the measurement filter. The calls to the `measurement function` are synchronized between all nodes thanks to `hera_synchronization` (Section 5.4).

The non-synchronized measurements have the same characteristics as the synchronized ones, plus a `frequency` value that indicates in milliseconds the interval between two calls to the `measurement function`.

Once the module has received the measurements from the execution of the `measurement function`, and the value has successfully passed the filter function, it sends this value to all nodes that are part of the cluster using a UDP multicast datagram (see 5.3.4).

When the number of iterations done reaches the maximum number of iterations, the process enters a hibernation phase where its memory allocation has been reduced as much as possible [40]. The process can be restarted by calling a `restart` function that allows to either keep the same parameters (maximum number of iterations, upper bound, ...) or to change them. The only characteristic that cannot be changed is the name of the measurement. You can also indicate when restarting a measurement phase whether you want to perform a warm-up phase or not.

5.3.2.1. Warm-up

When a series of measurements starts or restarts with the `WarmUp` parameter set to true, a warm-up phase of 100 measurements begins. This phase allows `Hera` to retrieve the default value of the sensor. At the end of the phase, the module calculates the median of all sensed measurements. This value is then passed to the filter described in the next section (5.3.3) before the series of measurements starts. This allows the filter to use that value for filtering the upcoming measurements.

5.3.3. Filtering of measurements and calculations

Since sensors in general are not accurate 100% of the time, false measurements can be made. In order to eliminate a significant portion of them, we have implemented measurement filters in our app. Even by using values that have successfully passed a measurement filter, it may happen that calculations yield a result that is not consistent with the result previously calculated, or that the result is simply not possible in the environment in which the system is located.

Since both measurements and calculations need a filter, we have created a single module called hera_filter. When a calculation or measurement is started by Hera in a process, a filter process is started immediately with the associated filter function.

5.3.3.1. Generic filter

Most of the erroneous measurements/calculations are easy to be detected for applications where sudden considerable variations are very unlikely to happen. We have thus implemented a generic filter that is based on this fact. This filter can be used for a multitude of applications (measurement or calculation of temperature, speed, humidity, etc.).
All the user has to do is to specify an upperbound on the difference that can have successive values based on the time difference between them. The upper bound must be expressed in difference_in_value / milliseconds. This difference in value is expressed in the same unit as the unit of the result returned by the measurement or calculation function. For instance, in an application where the measurement function returns a distance expressed in meters, the user must express the upper bound in meters/milliseconds.
If a measurement or calculation value is filtered out, and the next value is exactly the same as that of the filtered one, it may not also be filtered out since the time difference between them has increased. The consideration of time difference between measurements and calculations allows the filter to eventually trust a value if it is measured or calculated repeatably, even after first having changed faster than the upper bound permits it.

5.3.4. Send messages : UDP multicast

At the beginning of the development, we firstly used the `Achlys` tool [22, 23], and more specifically `Lasp` [25] to propagate nodes' data measurements across all nodes part of the cluster using CRDTs. We quickly found out that this method doesn't allow to have real-time calculations due to its low speed (see [41, Chapter 11] for results about the mean convergence time of `Lasp` on `GRiSP` boards). Also, as we wanted real-time calculations, we prioritize fast unreliable communication above slow reliable communication. That is how we concluded that using `UDP` as canal of transmission was the best way as it is faster than `TCP` and `Lasp` (see 6.1.2 for the results and conclusion of our tests with UDP and TCP on `GRiSP`). Since lots of measurement and calculation results are sent very frequently, the few occasional losses of UDP datagrams do not have a visible impact on the correct working of applications similar to our use-case.

Th necessary code to send messages via UDP is inside the `hera_multicast` module. We make use of an UDP multicast group in order to be able to send data to all nodes in the cluster even if the topology of the cluster is not known. When the module starts on a node, the node joins the UDP multicast group with the address 224.0.2.15. We have chosen this address because it is part of the AD-HOC Block I addresses group and it is unassigned [42]. We connect a node to this group using the parameter {add_membership, {MultiAddress, InterfaceAddress}} passed to the gen_udp:open/2[4] function, where MultiAddress is the multicast group address and InterfaceAddress is the address of the board. All the parameters needed to correctly open a connection to this group can be found in Appendix A.4.

Once connected to this multicast group, each node can send data to this group. All other nodes connected to this group will then receive this data in the form of a datagram message. When sending data via UDP (a measurement or calculation result), we put inside a message the name of the data (measurement or calculation), the name of the node that is sending the value and then finally the data. Including the node's name is important for organizing the storage of data on each node 5.3.5.

5.3.5. Data storage

Once other nodes' data has been received via UDP multicast, it first needs to be stored before it can be used for calculations. As we focus on real-time functionalities, we only store the last data sent by each node. The module in charge of storing is hera_sensors_data. The process started from this module keeps in its state one dictionary[5] per type of measurement. In each dictionary, the data is stored by node name (the key of the data is the node's name). The data is stored with a sequence number and a monotonically increasing timestamp taken by the node at the moment it stores the data.

The module provides two methods for retrieving the stored data, namely get_data/1 and get_recent_data/1. Both take as argument the name of the data to be retrieved. The first one returns the whole dictionary containing the data of all nodes and the second returns the dictionary containing all data obtained and stored less than 1 second before. This second function can be used by calculations that must ignore stored values that haven't been updated for a while, to only retrieve recent data.

[4]http://Erlang.org/doc/man/gen_udp.html#open-2
[5]http://Erlang.org/doc/man/dict.html

5.3.5.1. Data logger

The module `hera_sensors_data` also provides logging functionalities with the functions `log_measure` and `log_calculation`. These two functions each respectively create a directory named `measurements` or `calculations` where the corresponding data will be logged when received. In order to not exceed the low storage capacity of IoT type nodes (`GRiSP board` in our case), we only log data on a computer, member of the multicast group acting as a gateway device. One file is created per node that sends the data. Each file takes the name of the node, prefixed by the data name. For our use-case, a file named `sonar_node1` is created inside the directory named `measures` on our computer in the case it receives a sonar measurement from the node named `node1`.

On the boards, we only log the possible errors thrown during execution of the code. The configurations required for the loggers can be found in Appendix A.2 for the boards' logger configuration, and in Appendix A.3 for the computer's logger configuration.

5.3.6. Calculations

As it is the case for the measurements, different calculations can also be executed in a concurrent manner. Each specified calculation will start one process. Once a node has stored the received measurement (5.3.5), this value becomes available for calculations.

We have defined a type called `calculation` that is characterized by the `name` of the calculation, the `calculation function` to be used, the `frequency` to which that calculation function is executed, the `number of iterations` to be made (which can be infinity) and finally a `filter function` and an `upperbound` value for this filtering function. This filtering function is used to filter results obtained with the calculation function.

Once the result of the calculation successfully passes the filter function, it is sent to the UDP multicast group.

5.3.7. Genericity

Hera is a framework that can be used for an unlimited number of other use-cases that perform any calculation on the most recent measurements. It has been designed in such a way that the only task that is left for the user to do is to specify one or more measurement and calculation functions. These measurement and calculation functions are allowed to return anything as long as it is an Erlang data type. The user can also specify its own filtering functions to filter the results of these functions.

A user of our `Hera` framework can for example define the measurement function in such a way that it returns the value obtained with a temperature sensor. He can then define a calculation function that calculates the maximum temperature of all stored temperatures. If the temperature sensor doesn't always yield correct measurements, the user can make use of the generic filter or define his own filter function for the measurements and/or calculations.

5.3.8. Starting the measurements and calculations

The main module `hera` contains the `launch_app` function that is used to start the measurements and calculations. It takes two arguments, a list of measurements (of type `measurement`) and a list of calculations (of type `calculation`).

5.4. Hera_synchronization

We use this app as an Erlang distributed application[6] to synchronize the measurements between the nodes. To use this application, one must add the application to the `rebar.config` file of the main application as a dependency 3.3.1.5.

Also, in order to synchronize the right measurements, the lines of the snippet 5.3 must be added to the `sys.config` file of the main application. At the `measurements` entry, all the measurements to be done must be inside of tuples of form `{MeasurementName, Boolean}` where `MeasurementName` is the name of the measurement and `Boolean` is set to true if this series of measurements must be synchronized between the nodes.

```
{hera_synchronization, [
  {measurements, [{MeasurementName, Boolean}]}
]},
```

Snippet 5.3: Lines to add to the `sys.config` file to inform which measurements must be synchronized

5.4.1. Crosstalk between sonars

During our experiments, we have faced a major issue of non-consistent sonar readings that became only worse the more sonars we used. Making measurements with only one

[6] http://erlang.org/doc/design_principles/distributed_applications.html

sonar is not a problem, but by placing for example two sonars in front of each other, the measurements would vary a lot even when the target is not moving (the results of section 6.4.2.2 illustrates this problematic). This problem is called `crosstalk` and occurs when a sonar receives the signal of another board [43].
The first purpose of this application is to resolve the crosstalk between sonars for our use-case, but it can also be used in any other cases where measurements also need to be synchronized.

5.4.2. Solution

To solve this issue, coordination between the IoT devices is needed. We make use of globally centralized registered processes that indicate to all nodes of the cluster when to perform their measurements (see 5.4.2.2 for details of how they works).

The default implementation of the `grisp` module called `pmod_maxsonar` doesn't allow its users to trigger individual sonar readings. It only gives its users access to the latest sonar reading. It refreshes by default the latest sonar reading every 50 ms, performing a new measurement each time. Users here have no control of when to trigger them. We therefore had to modify the existing implementation to add a new non continuous mode where the users themselves perform the individual triggers of sonar measurements. This extension combined with synchronization allows a group of sonar-equipped nodes to make only one measurement at a time to avoid crosstalk.

5.4.2.1. Distributed application

The default behavior of a centralized algorithm in a cluster of IoT nodes is that if the node running the algorithm crashes, the functionality offered by this algorithm is lost. Thankfully for us, `Erlang` has already developed a solution to this problem, called `distributed applications`. Thanks to this solution, if the node running the application goes down, the application is restarted on another node of the cluster [44].

To use an OTP application as a distributed application for a certain application, it must be included as a dependency inside that application's `rebar.config` file. It must also be included inside the `relx` section of the same file for it to be included in the application's release. Unlike other dependency applications, it must not be part of the `.app.src` file inside the project. If it were the case, the application would start on all nodes and not only on one node.

Each node must know the cluster's topology in advance in order to be capable of restarting the distributed application on another node. For this purpose, one must add the snippet 5.4 inside the `sys.config` file. The first line contains, after the keyword `distributed`, the name of the distributed application, the number of milliseconds to wait after the node running the application has crashed, and a list. The list is composed of the name of the first node that must execute the application first, and of all other nodes that will run the application once all previous nodes in the order of enunciation have crashed.
The second line refers to all nodes that must be started to run the application. The third one on the contrary refers to all other nodes that are not mandatory to be started. Finally, the fourth line is used to indicate within which time all the mandatory nodes must be started. If the timeout value has elapsed and all specified mandatory nodes have not started, the node running the application terminates [44].

```
    {distributed, [{application_name, 2000, [node1, {node2, node3}]}]},
    {sync_nodes_mandatory, [node1, node2, node3]},
    {sync_nodes_optional, [optional_node]},
    {sync_nodes_timeout, infinity},
```

Snippet 5.4: Distributed application specification

If the node currently running the application crashes or is no longer contactable, the next node in the order of enunciation will start the application. If a node that precedes the current node in the order of enunciation becomes available again afterwards, the application will stop running on the current node and will restart on that previous node. Concerning our application `hera_synchronization`, we have put into place a restart policy that allows all nodes to continue executing normally without the need for human intervention. This restart policy is explained in the following section.

5.4.2.2. Global processes

We make use of several global processes in order to make coordination happen by synchronizing the measurements of all nodes. A global process is a process globally registered on all nodes of a connected network, i.e., all nodes are aware of this process. In Erlang, you can retrieve all nodes that are connected to a single node by executing on that node the Erlang Built-In Function (BIF) `nodes()` on an Erlang shell. To start a connection between two nodes, you must execute the `erl_inet:ping(NodeName)` function[7], where `NodeName` is the hostname of the node [45, 46].

[7]http://erlang.org/doc/man/net_adm.html#ping-1

Thanks to the fact that hera_synchronization is a distributed application, all started nodes that are part of the cluster are already connected between each other, therefore no connection must be performed manually.

Before having turned the application into a distributed application, we designed an automatic node discovery procedure via UDP multicast. The process is simple: each node sends on the UDP multicast group a hello message with its hostname every 2 seconds. When a node receives this message, it will try to ping the sender node back using the hostname given inside the message and will potentially succeed doing so. The entire code of this functionality can be found inside the branch called node-discovery-via-udp of our Hera repository[8].

In Erlang, it is possible to communicate with a process if you know its process identifier (PID). To retrieve the PID of a global process, you can call the function global:whereis_name(Name)[9] where Name is the globally registered name of the process. We have attributed to each of these processes a specific name. These names can either be found in the header file hera.hrl, or are sent by the process itself to the nodes.

We have defined two kinds of global processes: a global process that each node can contact for indicating that it wishes to start a series of measurements, and a global process per kind of measurement that has to be synchronized. The latter is used to send a message to the nodes indicating that it must perform its measurement. The first kind of process is implemented inside the hera_global_sync module, with its global name being the same as its module name. The second kind is implemented inside the hera_global_dispatch module. There, the name of each process is the result of the concatenation of the atom hera_global_dispatch_ and the atom representing the name of the measurement to be performed. For our use-case for example, we have a global process named hera_global_dispatch_sonar.

The whole procedure that is followed for performing series of synchronized measurements on multiple nodes is explained below.

First, each node wishing to perform a series of measurements sends via a process started from the hera_synchronization module a message {make_measurement_request, Name}. Name here is the name of the measurement the node wants to

[8]https://github.com/guiste10/hera/tree/feature/node-discovery-via-udp
[9]http://erlang.org/doc/man/global.html#whereis_name-1

perform that must be synchronized with the `hera_global_sync` process. The `hera_global_sync` process then records in a queue corresponding to these measurements, both the node's name that has sent the message and the process identifier of the node that has sent the message. This process maintains one queue per measurements that must be synchronized.

Then, each `hera_global_dispatch` process checks the queue linked to the measurements it oversees. If the queue is empty, it waits for 2 seconds before rechecking the queue. Otherwise, it removes the first element from the queue and sends a message to the process of the corresponding node, asking it to perform a measurement. This message contains the global name of the process itself as well as the name of the measurement to be performed. Once this measurement has been performed, the node will recontact the process indicating either that it wishes to continue to perform measurements or that the measurement it has just performed is the last one it had to perform in its series. In the first case, the global process will add the node to the end of the queue before repeating the whole operation.

Since we want the synchronization to be usable for near real-time applications, we have set a time limit for the global process to wait for the response from the node that indicates that the measurement has been performed. Based on tests we ran to find out the average response time, we have decided to set this maximum time to 200 milliseconds (see the results of our tests at Section 6.3). If a node does not respond before the end of the time limit, the `hera_global_dispatch` process restarts the whole procedure with the node at the beginning of the queue. If the process nevertheless receives the response from the previous node after the 200 milliseconds, it will add this node to the end of the queue and that without impacting the 200 milliseconds timeout for the currently followed node.

If the application is restarted on another node, that node will send to all nodes a message saying that they must resend a message asking to start their measurement. By acting this way, the new global process `hera_global_sync` will complete its queues and the measurements will continue correctly (but maybe not in the same order as before).

5.5. Pull request to the GRiSP Erlang Runtime repository

The initial implementation[10] of the module which is used to control a `Pmod MAXSONAR` on a `GRiSP` board only allows to use the free run mode, giving its user no control over when to trigger a sonar reading. As explained before, this mode allows interference to take place between multiple sonars. The sonar's datasheet [47] however mentions that a trigger mode is possible on the `Pmod MAXSONAR`. We have modified the implementation of the module in order to be able to use this triggering mode.

The basic implementation only allows to use one method, the `get/0` method, which returns the last value given by the sonar. With the free run mode, a measurement is given every 50 milliseconds to the `GRiSP` board via pin 5. This measurement is then stored in the process state in the `last_val` variable.

In order to use the trigger mode of the `Pmod MAXSONAR`, the pin number 4 of the sensor must be connected to a logical low. This logical low is represented in the grisp software by the atom `output_0`. To trigger a value, one just has to connect pin 4 to a logical high. By default, pin 4 is connected to the value `output_1` which is a logical high and corresponds to the free run mode [47].

In our implementation, we bring a new `set_mode/1` function that allows to change the sonar's working mode. It takes as parameter an atom whose value can either be `single`, `disabled` or `continuous`. The single mode is the trigger mode mentioned above, the continuous mode is the free run mode and the disable mode puts the sonar to sleep, making it stop emit anything. The get function always fetches the last value felt by the sensor in continuous and single mode. In the disabled mode, the `undefined` atom is returned instead of a numerical value. In the continuous mode, the value is retrieved in the same way as in the basic implementation. In single mode, when the get method is called, a trigger is performed, i.e. pin 4 is connected to a logical high and then directly to a logical low. Our full implementation can be found in Appendix A.5.

We have done some tests to verify that our new implementation works well. Details of the tests and results can be found at 6.2 and 6.2.2.

[10] `https://github.com/bastinjul/grisp/blob/1.2.0/src/pmod_maxsonar.erl`

5.6. Sensor_fusion_live_view

We have designed this application to allow having a graphical representation of what the nodes sense (or measure) and calculate when Hera is used.

We have integrated a basic but general representation of the data that can be used for any use case of Hera as well as a graphical representation specific to our use-case.

The installation guide is available in the `README.md` file of our github project[11]. Please note that the application is not to be run on an IoT node, but on a computer acting as a gateway that is connected to the node's network.

The website includes three pages for displaying data. Two pages are accessible via the paths /sonars/measurement and /sonars/calculation at localhost:4000. The first (resp., second) page displays live measurements (resp., calculations) sent by each node. An illustration of this page is available in Appendix 31 (resp., 30). A third page can be accessed via the path /sonars/room and displays a bird's eye view of the room. Inside this room, sonars (resp., estimated target's positions) are represented by red (resp., cyan) squares. An illustration of this representation is available in Appendix 32. To show how the live view behaves when a person moves around a room, we have presented figure 1 as a comic strip. It shows a person moving around a room from left to right and vice versa. Each cyan square represents a calculation performed by a single board. If there are 4 squares, it means that all 4 boards have managed to calculate a position with the measurement's data of the other nodes. A cyan square only moves if the board in question sends a different (x,y) position than the previous one. It is therefore possible that if several calculation or measurement results are filtered out, the square will stay in place for a long time. In the same way, the red squares can only move when the board sends its new position. It is also possible that 2 green squares appear instead of a cyan square. These squares represent the case where there are two possible positions for the person in the room as in Figure 6.

To display the data sent by the nodes, we connect the application to the UDP multicast group of the nodes in the same way as in `Hera` (see 5.3.4). It should be noted that the sole purpose of this application is to present a live graphical representation of the data and that nothing is calculated by the application. All data are computed by the nodes themselves.

[11]https://github.com/bastinjul/sensor_fusion_live_view/blob/master/README.md

6. Experiments

6.1. Data sending between `GRiSP` boards

6.1.1. Setup

In this section, we detail the experiences we have made to help us determine the best solution to send information between GRiSP boards to achieving our near real-time application.

To find the best solution, we needed to determine the average time it takes for a message to arrive from one node to another. On the other hand, we also needed to evaluate the percentage of data loss.

As mentioned previously in section 5.3.4, we prioritize a fast unreliable transmission method over a slower transmission method that has a lower percentage of loss.

To determine the average end-to-end delay, i.e. the average time it takes for a message to be sent from one board to another, we cannot estimate it by first looking at the GRiSP board's clock that sends the message and then at the clock of the receiver when the message is received. Indeed, the clocks on each board contain different values. To fix this problem, we first calculate the average round trip time (RTT), i.e. the time it takes for a message to be sent plus the time it takes for the acknowledgement of this message to come back [48]. Once the average RTT is known, we can simply divide it by two to obtain an estimate of the average end-to-end delay [49].

To calculate the average RTT, we make board A send a message to board B containing the timestamp T1 of the time at which the message is sent. Once board B has received this message, it immediately sends the message back to board B without modifying it. Once board A has received the reply of board B, it calculates the RTT of the message by taking a new timestamp and subtracting T1 to it. Once this is done, we use the Erlang logger to save it in a file. The operation is repeated a large number of times (we have chosen series of 1000 iterations) to obtain a good estimation of the average RTT. The mathematical formula giving the average end-to-end delay can be found at formula 7.

$$EndToEndDelay = \frac{\sum_{i=1}^{n=1000} T2_i - T1_i}{2} \tag{7}$$

We have used this technique to calculate the average end-to-end delay for the UDP multicast medium as well as for the TCP medium. We have also varied the message sending frequency to see if it has an influence on the calculated value.

In addition to the frequency, we also varied the number of boards to which messages are sent. For TCP, we have experimented with 2 and 4 boards. For UDP we only have experimented with two boards. There are technical reasons for this difference.

There is a difference on how UDP and TCP messages are sent and received in Erlang. A UDP message is sent with a single datagram to the multicast group it belongs to. Conversely, a TCP message must be sent once to each node of the cluster. With TCP, the number of packets that must be sent is proportional to the number of other nodes present in the cluster. Moreover, the only way to send the same message to all nodes via TCP is to iterate on all sockets that are open to the other nodes and ask each socket in turn to send the message[1]. Sending a message to all the cluster's nodes must therefore be faster with UDP than with TCP. The only downside is that UDP is not 100% reliable.

Before explaining how we have obtained the results concerning the reception of messages with more than 2 boards, we will first explain how sockets work in Erlang. Both for TCP and UDP, an Erlang process is assigned to each socket. This process is responsible for processing the messages received through this socket. As a reminder, processes run concurrently with each other in the Erlang virtual machine.

As mentioned earlier, there is one open socket per connection to other boards with TCP. It is therefore possible to receive and process concurrently messages coming from other boards. With UDP on the other hand, there is only one socket for communicating with all other boards. There is also only one process that is responsible for processing one by one, in order of arrival, incoming UDP messages. Figure 11 shows the situation encountered during the simultaneous reception of 3 messages coming from three different nodes. We can see that if we perform tests with three boards for UDP and take the timestamp of the reception of the message, only the message M1 will have the correct timestamp. The other 2 won't because of the delay introduced by the reception of the previous messages. With TCP, the three messages will have a nearly correct timestamp as there are 3 processes working concurrently. Under these conditions, it is possible to recover the average RTT with TCP by having more than two boards in the cluster, but not with UDP.

[1]`https://github.com/guiste10/hera/blob/0bfe764ed7ecbcb447e9f8a29eb2b6bd25e261fc/src/hera_test_tcp.erl#L34`

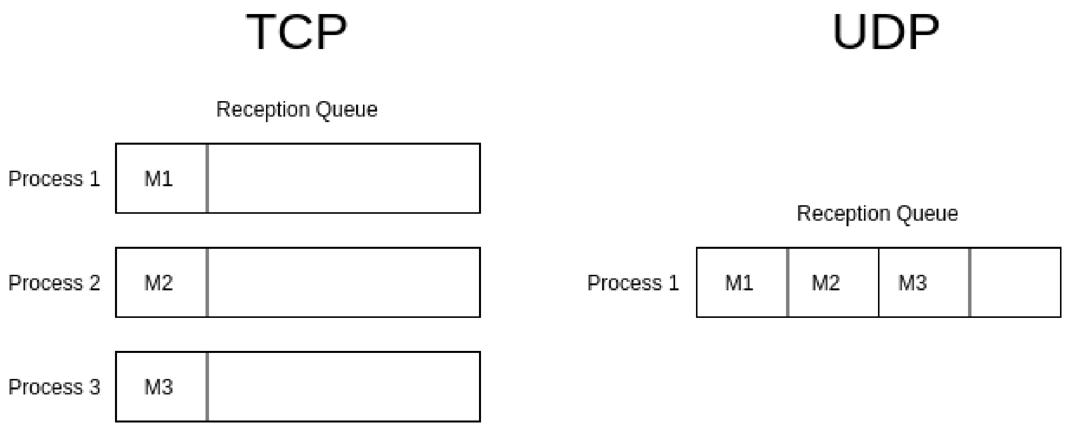


Figure 11.: Situation at the reception of messages from 3 nodes at the same time with TCP and UDP.

In the case of 4 boards with TCP, we have calculated the average RTT in another way than with 2 boards. For the results with 4 boards, we only consider the highest RTT value of the 3 messages that were sent to the other boards from board A. We have opted for this way to take into account the possibility of having a node that receives all the messages with a lot of delay compared to the others. Because the messages are sent one by one, the end-to-end delay of the last sent message is the average RTT for that last message minus the average end-to-end delay between two boards, as seen in the time-sequence diagram in Figure 12.

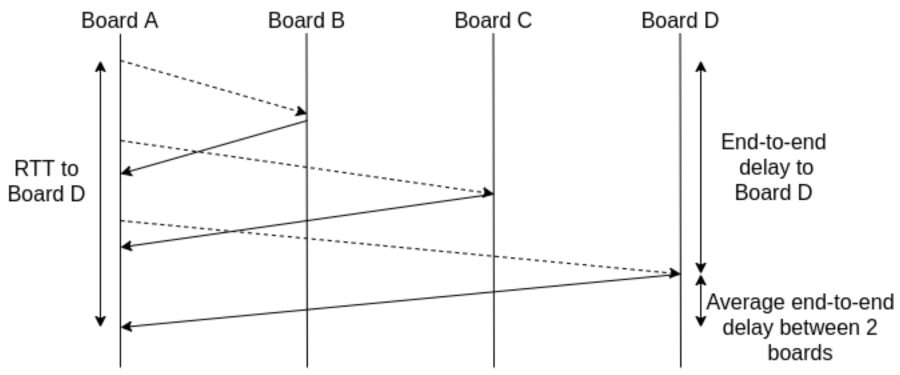

Board A
Board B
Board C
Board D
RTT to Board D
End-to-end delay to Board D
Average end-to-end delay between 2 boards

Figure 12.: Sending with TCP a message to 3 boards, represented with a time-sequence diagram.

To calculate the loss percentage of UDP messages, we simply make board A send a predefined number of messages to board B and then count the number of messages board B has received in total. We then calculated the percentage using these two obtained values. We have repeated this experiment 4 times with sending frequencies of 50 and 100 milliseconds.

6.1.2. Results

6.1.2.1. UDP multicast

Between 2 GRiSP boards Table 1 shows for different message sending frequencies the average end-to-end delay obtained during our experiments. The graphs showing the detailed results of our experiments can be found in Appendix B.3.1.1.

We can deduct from these results that the frequency has almost no effect on the end-to-end delay and that whatever the frequency, the delay is between 10 and 13 ms on average and 90 percent of messages are delivered within a delay of 18 ms. These results are in line with our objective of near real-time computing.

Message sending frequency	50 ms	75 ms	100 ms	200 ms
Average end-to-end delay	11.86 ms	12.8 ms	11.7 ms	10.60 ms
Median end-to-end delay	10.5 ms	11.5 ms	10.5 ms	9.5 ms
90th **percentile**	18 ms	18 ms	16.5 ms	15.5 ms

Table 1.: Average end-to-end delay per message sending frequency

Loss percentage We have performed this experiment twice with a few minutes between each measurement and after restarting the boards between the two series of measurements. During the first series of measurements, the average percentage of message loss with a frequency of 50 ms was 31.5% and 36.4% with a frequency of 100 milliseconds. During the second series of measurements, they were 24.68% for a 50 ms frequency and 25.04% for a 100 ms frequency.

This difference being quite large, we cannot conclude anything about the percentage loss between boards because the factors that influence percentage can be numerous and are unknown to us. Especially since the percentage of loss observed in the tests carried out to track a person was much lower than the percentages obtained with those experiments.

6.1.2.2. Full mesh TCP

Since the frequency does not have a great influence on the end-to-end delay, we have performed the measurements for TCP with a message frequency of 200ms. The graphs showing the detailed results of these two experiments can be found in Appendix B.3.1.2.

Between 2 GRiSP boards We have observed an average delay between two nodes of 14.83 ms, which is a bit higher than the UDP delay between two boards. This could be explained by the three-way handshake used by TCP. The 90^{th} percentile is 21.5 ms. The upper limit for 90% of messages is therefore higher than that of UDP.

If our system only used 2 boards, TCP, which is only a few milliseconds slower, would be a better alternative to UDP because of the percentage of message loss being zero.

Between 4 GRiSP boards We expected that the end-to-end delay for 4 boards would be ±3 times the end-to-end delay for 2 boards. Practice shows that the average total RTT is 144.28 ms, the median is 138 ms, and that the 90^{th} percentile is 181 ms. If we subtract from these values the value of the end-to-end delay between two boards, we obtain an average delay of 128.44 ms.

This delay with TCP is far too high compared to the delay obtained by using UDP multicast in order to be usable for near real-time applications. Indeed, if messages are sent with a frequency of 100 milliseconds, up to 8 messages can be received via UDP by the last board to receive the messages, compared to 4 for TCP. We therefore opted to use UDP in our system.

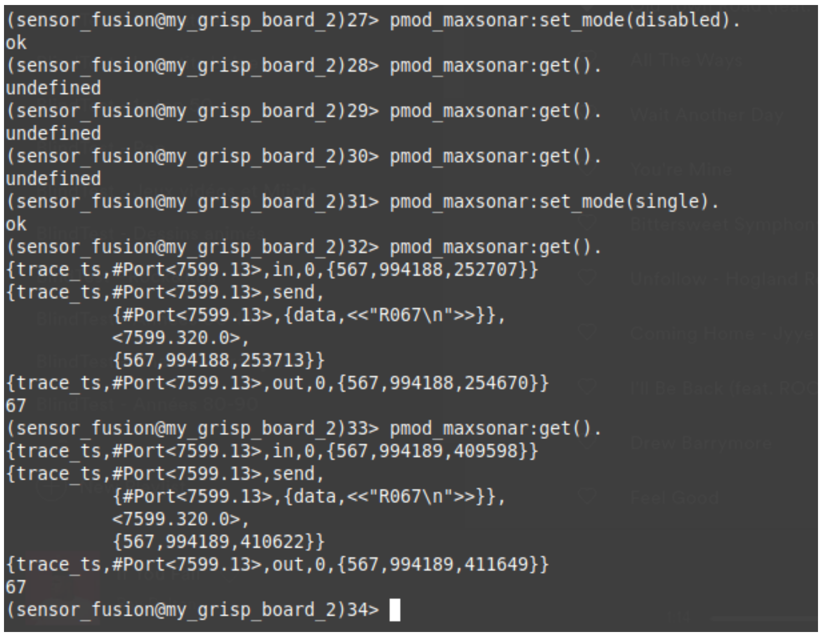
6.2. Triggering of the Pmod MAXSONAR

6.2.1. Setup

We needed to know if our new implementation of the `pmod_maxsonar` module is working properly. To do this, we used Erlang's dbg tool to trace in the terminal the messages that were sent by the Pmod MAXSONAR to the GRiSP board. By default, the Pmod MAXSONAR sends one data every 50 milliseconds, i.e. every time a wave has been transmitted. At this speed, we can see a lot of messages scrolling through the terminal. In our new implementation, this case corresponds to the `continuous` mode. In the `disable` mode the GRiSP board should not receive a message under any circumstances and in the `single` mode the GRiSP board should only receive a message after calling the `pmod_maxsonar:get()` function.

6.2.2. Results

Figure 13 shows a screenshot of the shell when the `disabled` and `single` modes were activated in turn. We can see that when the `disabled` mode is enabled, no message arrives from the sonar, even when the `get` function is called. When we turn on the `single` mode and call the `get` function, a message will arrive after the call. In `continuous` mode, messages scroll through the shell in the same way as with the basic implementation of the module. We can therefore conclude that our implementation behaves as expected.



```
(sensor_fusion@my_grisp_board_2)27> pmod_maxsonar:set_mode(disabled).
ok
(sensor_fusion@my_grisp_board_2)28> pmod_maxsonar:get().
undefined
(sensor_fusion@my_grisp_board_2)29> pmod_maxsonar:get().
undefined
(sensor_fusion@my_grisp_board_2)30> pmod_maxsonar:get().
undefined
(sensor_fusion@my_grisp_board_2)31> pmod_maxsonar:set_mode(single).
ok
(sensor_fusion@my_grisp_board_2)32> pmod_maxsonar:get().
{trace_ts,#Port<7599.13>,in,0,{567,994188,252707}}
{trace_ts,#Port<7599.13>,send,
          {#Port<7599.13>,{data,<<"R067\n">>}},
          <7599.320.0>,
          {567,994188,253713}}
{trace_ts,#Port<7599.13>,out,0,{567,994188,254670}}
67
(sensor_fusion@my_grisp_board_2)33> pmod_maxsonar:get().
{trace_ts,#Port<7599.13>,in,0,{567,994189,409598}}
{trace_ts,#Port<7599.13>,send,
          {#Port<7599.13>,{data,<<"R067\n">>}},
          <7599.320.0>,
          {567,994189,410622}}
{trace_ts,#Port<7599.13>,out,0,{567,994189,411649}}
67
(sensor_fusion@my_grisp_board_2)34> 
```

Figure 13.: View of the terminal when running the dbg tool and the modes `disabled` and `single` are tested.

6.3. Average response time to global processes on GRiSP boards

6.3.1. Setup

We have performed tests to estimate the average time it takes for the global server to send to a node the request to perform a measurement, plus the response time of the node performing that measurement (see section 5.4.2.2).

To obtain the average time, for each measurement that must be done by the boards, we took on the global server the timestamp T1 when sending the `perform measurement` message and the timestamp T2 when receiving the response from the board. We then subtracted T2 by T1 to obtain the RTT time. We used the Erlang logger to record the results in a file. We have performed this test with 4 boards that each perform 1000 measurements.

6.3.2. Results

Figure 14 illustrates the results of this experiment. The mean response time is 114.72 ms, the 90th percentile is 144 ms and the 98th percentile is 177 ms. With those results, we have concluded that a timeout of 200 ms was the best choice as 98% of the boards' answers to the global server have taken less time.

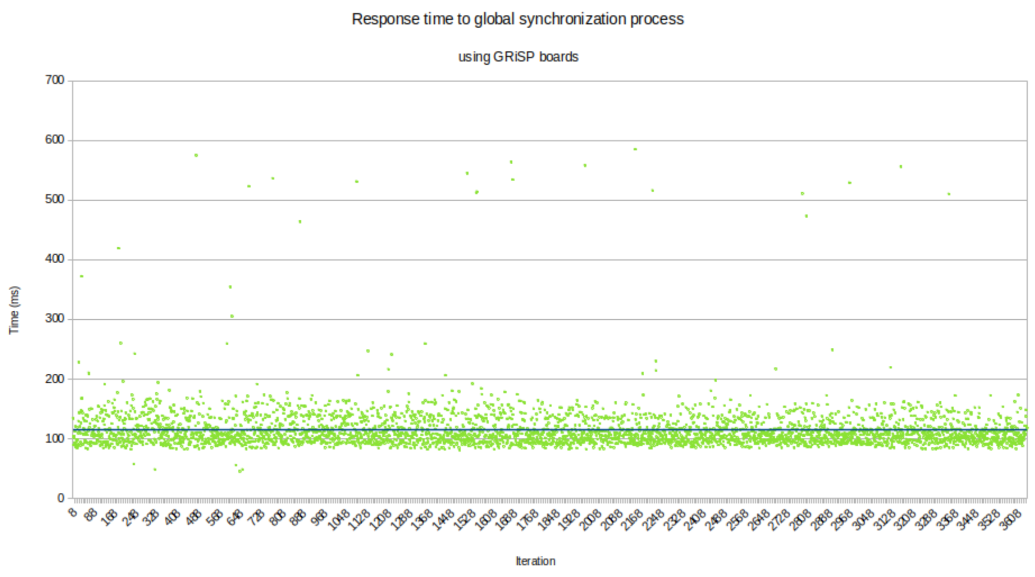


Figure 14.: Average respond time to a global server using GRiSP boards

6.4. Person tracking

6.4.1. Setup

This subsection explains the different ways we followed to set up the system and obtain results for person tracking. Each configuration consists of a unique combination of the hardware and software settings that are described below.

6.4.1.1. Fixed setup description

All our experiments were performed in an empty room of dimensions 3 x 4 meters. Each node is a GRiSP embedded system equipped with a Pmod MAXSONAR sensor connected on its UART port. Each node is placed at a different location in the room at the same height of ±70 cm above the ground. Each sonar is horizontally directed towards the center of the room with a certain angle, depending on the experiment.

All nodes start performing measurements at the same time. They then each send measurements and the resulting calculations to an UDP multicast group. To retrieve the exchanged information, a computer must be part of this UDP multicast group. The computer will create one file per type of calculation and measure, and per node. It then logs to the corresponding file the measures or calculations sent by each node. We have based the results of 6.4.2 on the information obtained with the computer via UDP. There is hence a small loss percentage of exchanged measurements/calculations.

For example, if we instruct a cluster of 2 nodes to calculate the position of a single person in a room, three files will be created per node by the computer. There will be one file to log the position calculated by the node in the room, one file to log each distance sensed by the node's sonar and one file to log the calculation of the person's position in the room.

The plots that we will show are based on those files.

6.4.1.2. Variable setups of nodes and tracked persons

Paragraph 6.4.1.2 describes the different ways we can setup the nodes. Paragraph 6.4.1.2 describes the different ways the tracked person can behave. Section 6.4.1.2 introduces some typical configurations we have used by combining different setups of the nodes with different behaviors of the tracked person.

Different node setups The nodes' setup can differ in three ways. Below contains an explanation for each way.

- **Number of nodes** We have conducted experiments with one to four nodes.
- **Position (x,y) in the room** We consider the room we use as a two-dimensional plan "seen from above". Each node is assigned a coordinate (x,y) based on its position in the room. This coordinate is used to calculate the coordinate of the tracked person in the room. As mentioned before, we assume that all nodes are positioned at the same height above the ground.
- **Direction/angle** The angle of the sonars to the ground is always the same as the nodes' sonars are directed horizontally. We have only brought variations to the horizontal angle of the sonars. For example, when using 4 nodes, we have directed their sonars with a certain angle to make them point horizontally to the center of the room.

Object physical characteristics variations We can make variations to the tracked human(s) in 2 ways.

- **Number of tracked humans** We have made experiments with one and two humans. The number of humans the system can identify and follow is proportional to the number of nodes composing it. For tracking 2 persons, we would need at least 4 nodes.
- **Movement or position** The humans can either move or remain static. Different moving patterns are possible, for example moving straight, making a circle, moving randomly etc. The human can also remain static and stay at the same position. This can be useful to learn about the sonar's detection pattern for humans.

Typical setups We have tried different configurations by combining the different ways of setting up the nodes with different behaviors of the tracked person.

- **Single node** We made multiple kinds of tests with a single node. In order to have a better idea of the people detection pattern of the Pmod MAXSONAR, we have put a single non-moving person in front of this sonar at different positions. We have also performed tests with a person that moves in front of the sonar.

Other tests were done successfully with and without the measurement filter to evaluate how useful it is for our system.

- **From 2 to 3 nodes** For more than one node, we have imagined two different space situations in order to test the effect of crosstalk. A first situation where the sonars are facing each other, and a second situation where the are not. In the case of 3 nodes, we have tested the latter by making the directions to which the sonars point form an equilateral triangle.
- **4 nodes** Using 4 nodes, we would place them in the form of a rectangle and make the sonars point to the center of it.

6.4.1.3. Software configuration

We can parameterize our system to use or not the following functionalities.

Crosstalk avoidance When sonars are facing each other, crosstalk can occur if the sonars are emitting waves at the same time and thus produce falsified measurements. In order to evaluate the importance of the crosstalk avoidance feature, we have performed tests with and without activating it at runtime.

Measurement filtering Even without crosstalk, the Pmod MAXSONAR may yield inaccurate measurements as some glitches may occur randomly. To avoid sending these glitches to other nodes, we have implemented a measurement filter. To test the efficiency of this feature, we have performed tests with and without activating it at runtime.

6.4.2. Results

Using different setups for person tracking 6.4.1, we have conducted several experiments with the application we developed. Below are the results we obtained for tracking a human with a different number of sonar-equipped GRiSP boards.

6.4.2.1. Using 1 board

Performing experiments with only one board first is very useful for studying the individual behavior of the Pmod MAXSONARs. It has enabled us to make important observations that must be taken into account when using more than one board for tracking a moving target. The Pmod MAXSONAR's datasheet [10] has given us a lot of valuable information regarding the sonar's beam pattern. We have experimented with different series of measurements to have a better understanding of the sonar's actual performance for people detection.

Maximum distance for people detection The sonar's datasheet mentions that this model of sonar can detect people up to 8 feet (244 cm). We have performed several series of measurements with a nonmoving human placed at different distances (150, 200 and 250 cm) to see what happens in practice. For each distance, we have also evaluated the impact of placing the human sideways to the sonar instead of frontally. Table 2 contains the percentage for the number of times the sonar has yielded a correct measurement for each case. Even though we have repeated the experiment for each case several times with a high number of measurements, the percentage isn't very accurate as the environment will always have an impact on these results. A sonar will have it easier to track a big boned person rather than a skinny person for example.

	150 cm	**200 cm**	**250 cm**
Front	75%	68%	23%
Side	52%	30%	6%

Table 2.: Sonar's accuracy for people detection

As we can see on Table 2, the orientation of the tracked person has a huge impact on the sonar's accuracy. If the person's body (feet and belly) is orientated to the sonar, that person can be very well detected at a distance of 200 cm. This plot 15 shows that 150 cm is close enough to achieve good results in the case a nonmoving person is orientated sideways to the sonar. Based on the results we have obtained, regardless of how the person is orientated to the sonar, we can say that 150 cm is a good range limit for applications where the detection of people must be consistent.
For our use-case, this does not necessarily mean that the sonars have to be separated from each other with 150 cm or less. Suppose for example that a person stands exactly between 2 sonars that are separated from each other with 3 meters, and that this person's position forms a straight line with these 2 sonars. The measured distances by the sonars should in this case be less than 150 cm.
The datasheet is correct saying that the Pmod MAXSONAR can detect people up to 244 cm. It however didn't mention how well/consistent it could do so.

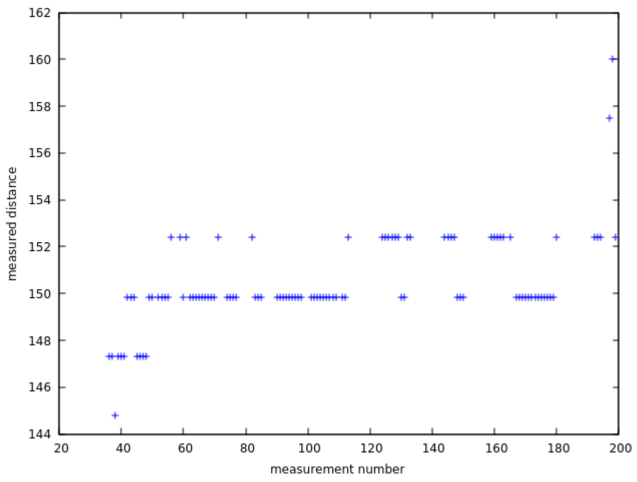
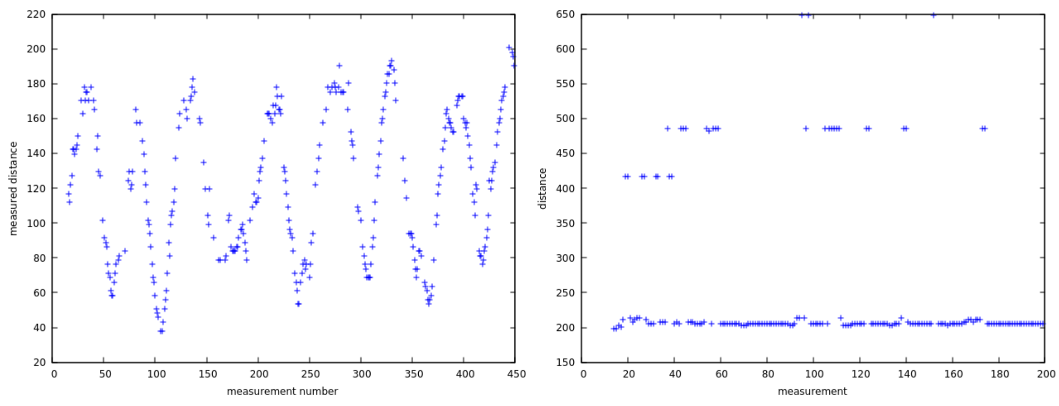


Figure 15.: Distance to a person standing sideways at 150 cm - Obtained with gnuplot

Horizontal sonar beam width The sonar's beam width we have observed matches with the one specified in the Pmod MAXSONAR's datasheet [10] on chart B (page 8). At 150 cm right in front of the sonar, the observed sonar's beam width was (only) 80cm. At more than 1 meter away from the sonar, the sonar's beam does not become notably wider as one might expect from such a device. It is hard to cover a great area in a room with this beam's width.

Moving person By taking the sonar's beam width into account and not surpassing a distance limit of 200 cm, we wanted to see how good the sonar could track a human that moves (and rotates) randomly inside an area that respects those limits. On the obtained plot (16a), we can see how well the sonar is able track that moving person. The measurement filter was used to filter out measurements that couldn't detect the moving target. In total, a bit more than 72% of the measurements were preserved by the filter to attain that plot.

Measurement filter We have conducted successive experiments with the measurement filter activated and deactivated. We saw that without filter, the sonar either gives the distance to the target, the distance to the background/wall, or in the worst case a distance of ±647 cm in the case even the background couldn't be detected. We have obtained this plot 16b with the measurement filter deactivated by making a person stand face to the sonar at 200 cm. As explained in 6.4.2.1, we can see that a person is well detected in this particular case.



(a) Moving randomly - Obtained with gnuplot **(b)** Standing at 200 cm - Obtained with gnuplot

Figure 16.: Distance to a person

6.4.2.2. Using 2 boards

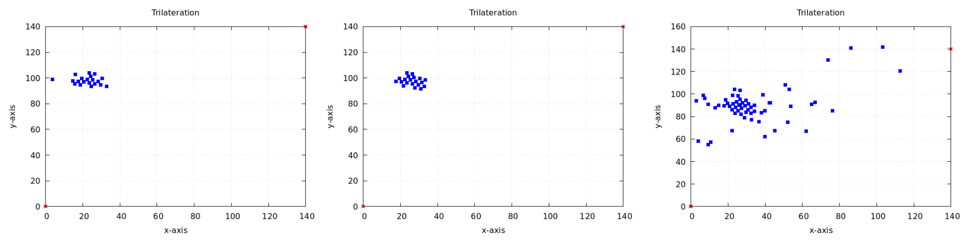
The results mentioned below were carried out with a person that does not move in a room. The person is always in more or less the same position in the room for all experiments with two boards.

We have also conducted tests with a moving person, but given the very small space where the person can be detected by both sonars at the same time, too little data has passed the filters for us to be able to draw any conclusion about the data.
In the graphs proposed for 2, 3 and 4 boards, each blue square represents a position estimated by one board and each red star represents the position of a sonar.

Facing sonars By comparing graphs of Figure 17, we can see that when the synchronization of measurements is activated, there is not much difference in the positions calculated for tests carried out with or without measurement filters. The points are more or less all grouped together in a 30 x 30 cm square. On the other hand, we can see that when the measurement filter and the synchronization are off, the calculated coordinates are much more dispersed throughout the chart. This is due to the presence of crosstalk between the 2 sonars.

This is a first example of the importance of synchronization to avoid crosstalk between sonars.

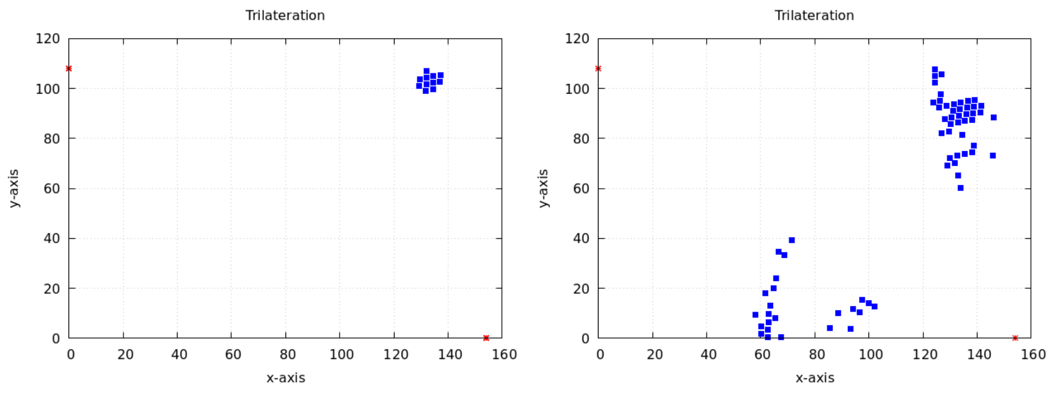
Other graphs about the experiments' results with synchronization and without filters as well as without synchronization and without filters are available in Appendix B.3.2.1. These graphs show the same trend as those mentioned above.



(a) With filter and synchronization **(b)** Without filter and with synchronization **(c)** Without filter and synchronization

Figure 17.: Trilateration of a non-moving person inside a room with 2 facing sonars - Obtained with gnuplot

Non-facing sonars The same philosophy emerges from experiments with sonars that are perpendicular to each other, as evidenced by the plots of Figure 18. The rest of the experiments are in Appendix B.3.2.1.

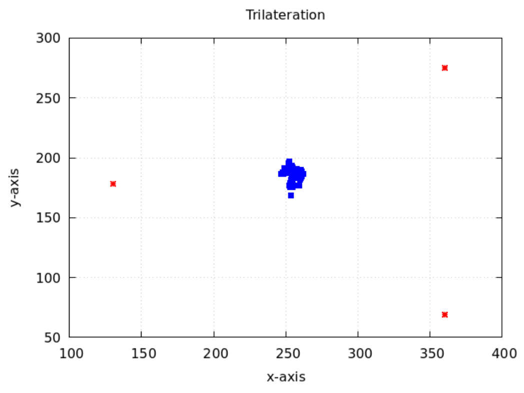


(a) With filter and synchronization **(b)** Without filter and synchronization

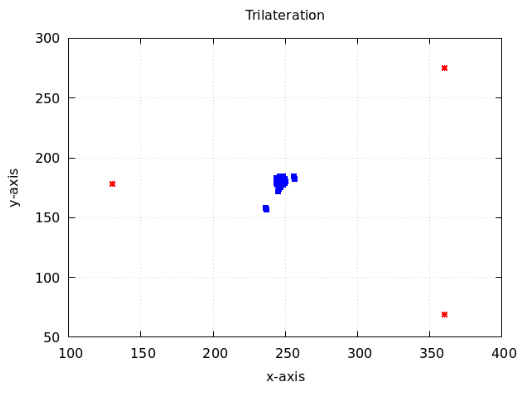
Figure 18.: Trilateration of a non-moving person inside a room with 2 facing sonars - Obtained with gnuplot

6.4.2.3. Using 3 boards

Non-moving person with anti-crosstalk Figure 19 shows us once again that the results obtained with and without the measurement filter are not very different from each other. Moreover, the person is quite well detected. However, some results in Appendix B.3.2.2 show that even with a filter, some deviations from reality remain. This shows that the filter is not perfect and needs improvement. We will discuss these possible improvements at Section 7.2.



(a) With filter and synchronization



(b) Without filter and with synchronization

Figure 19.: Trilateration of a non-moving person inside a room with 3 sonars - Obtained with gnuplot

Non-moving person without anti cross-talk Figure 20 shows the same experience of a person at the center of 3 sonars, but this time without the synchronization feature enabled. We can see once again that this feature is essential because the results obtained without it are disastrous. The more sonars are used without synchronization, the more crosstalk is present.

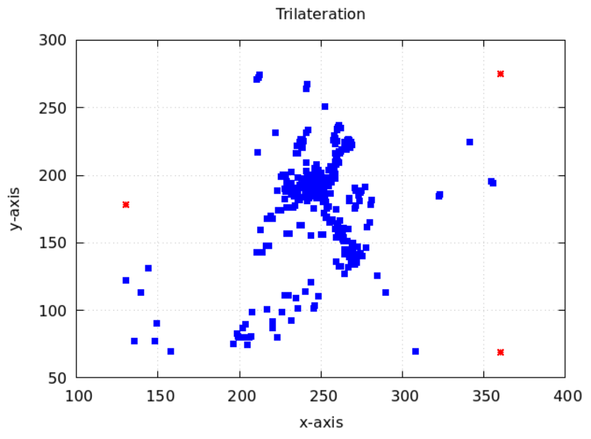


Figure 20.: Trilateration of a non-moving person inside a room with 3 sonars, with filter and synchronization deactivated - Obtained with gnuplot

Moving person Figure 21 shows the trajectory of a person that describes a circle obtained in the inner space of the 3 boards.

In the next graphs representing one (or two) person(s) moving, a colour scale is located on the right side of the graph. This scale represents the calculations' iterations made by one board. As a reminder, there is a 50 ms interval between each iteration. For example, in graph 21, the positions on the graph were obtained from a 4 second time interval that starts two seconds after the position calculations are started. The purpose of these graphs is to show the trajectory as seen on the live view.

We can see in this figure that the trajectory taken by the person is well detected by our system.

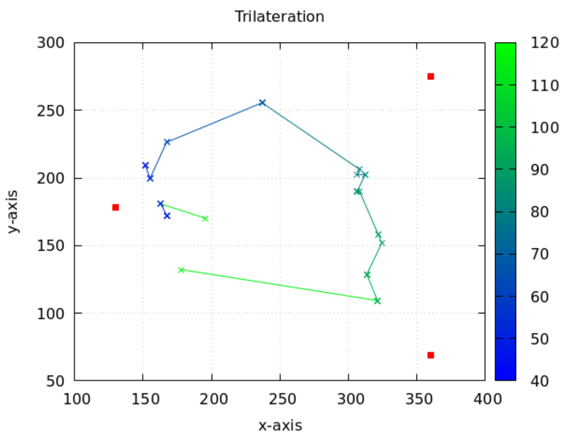


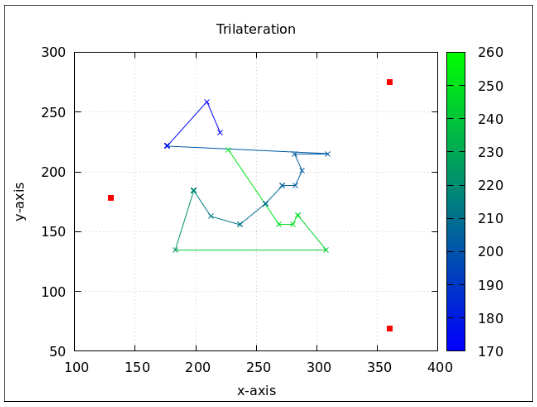
Figure 21.: Trilateration of a moving person inside a room with 3 sonars, with filter and synchronization activated. The person is describing a circle between the 3 board - Obtained with gnuplot

We have also conducted another experiment with a moving person. Figure 22 shows on the right the movement performed by the person and on the left the obtained result. We can see once again that the movement of that person is more or less well detected.

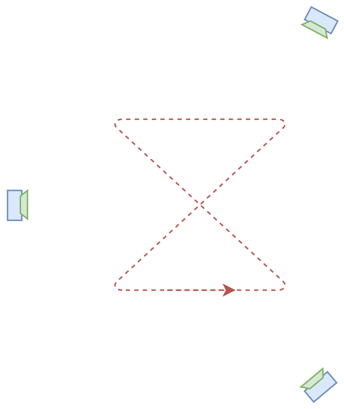
6.4.2.4. Using 4 boards

The tests in this section have been performed with the measurement filter and anti-crosstalk features activated. The tests with 2 and 3 sonars have already shown the importance of having the anti-crosstalk feature activated.

Non-moving person Figure 23 shows two tests we have performed with a person that does not move and stands at the center and at the left side of the room respectively. We can see that in both cases, the person is detected and that its position is correctly calculated. The results of these two tests are the best we have observed. Indeed,



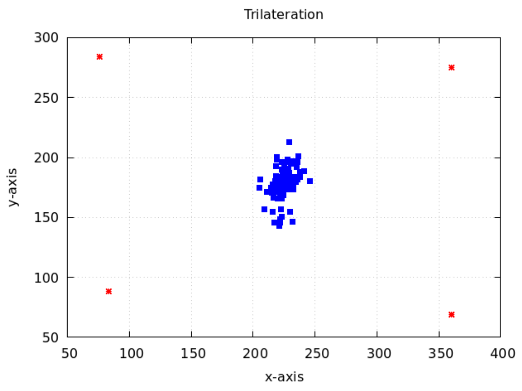
(a) Trilateration with filter and synchronization



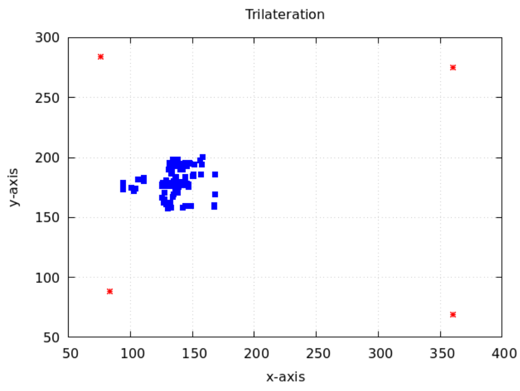
(b) Path followed by the person

Figure 22.: Trilateration of moving person inside a room with 3 sonars - Obtained with gnuplot

figure 24 shows that the results are not always as good, even by using a filter. We believe that this lack of accuracy shows the limitation of the sonar we use. On the right of this figure, the person standing on the top side of the room is less in the range of the sonars than when standing on the left as on the previous figure (23). This would explain the difference in detection quality between Figures 23b and 24b. We believe that this problem would disappear by using sonars that possess a wider detection beam (as explained in 7.2).



(a) Person on the center of the room



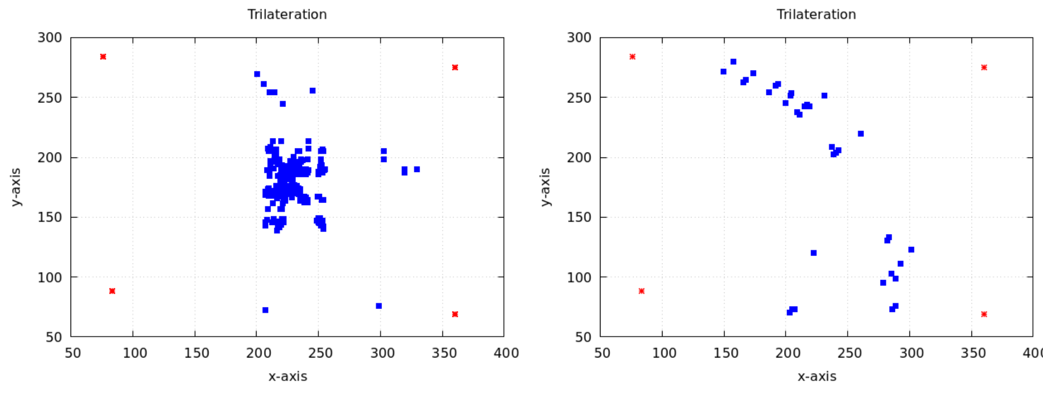
(b) Person on the left side of the room

Figure 23.: Trilateration of a non-moving person inside a room with 4 sonars - Obtained with gnuplot

Another problem that can be seen in Figure 24b, is that because the person is in an area that is less covered by sonar, less sonars' measurements were used during the calculation. A lot of the time, only 2 sonar measurements coming from 2 opposite could be used. The trilateration resulted in 2 possible positions that were not filtered by this filter 5.2.4.1 as their coordinates don't violate the allowed limits. This explains why we can see 2 groups of positions: on on the bottom and one on the top.

This test shows the importance of having a large sonar coverage area and why it is preferable that at least 3 boards detect the person, so that there is only one possible position.

The rest of the results we have obtained for a person that is not moving are in Appendix B.3.2.3. We can draw the same conclusions for those other results.

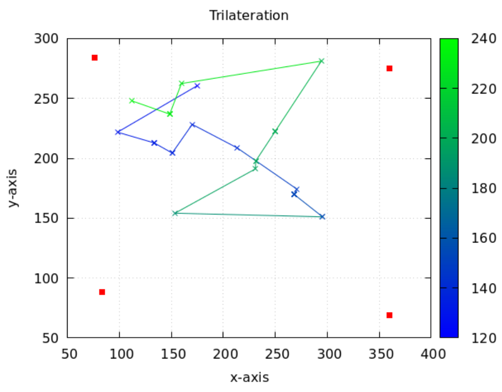


(a) Person on the center of the room **(b)** Person on the top side of the room

Figure 24.: Trilateration of non-moving person inside a room with 4 sonars - Obtained with gnuplot

Moving person Figure 25 shows the tested pattern on the right, and the path we have obtained with a person trying to mimic that pattern. The resemblance between those two is striking. Figure 26 shows three tests we have carried out where the tested pattern can be well recognized on the resulting plots. In figure 26b we can see that the person moves from right to left, in Figure 26a from bottom to top and in Figure 26c that person makes a circle. That last case is a little less precise than the two previous ones but it is still well identifiable.

The rest of the results we have obtained for a person that is moving is in Appendix B.3.2.3. The plots included in these results contain all the positions computed during each experiment. They do not allow to see the path taken and thus to have an idea of the order in which those positions have been visited. We can draw the same conclusions from those results.



(a) Trilateration of a moving person

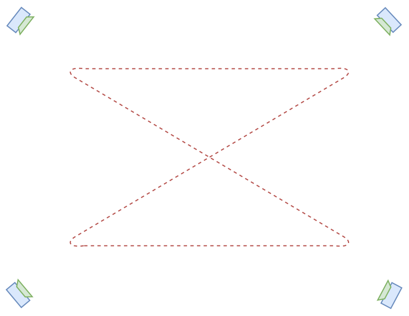
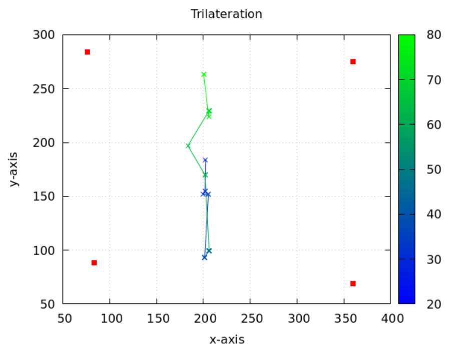
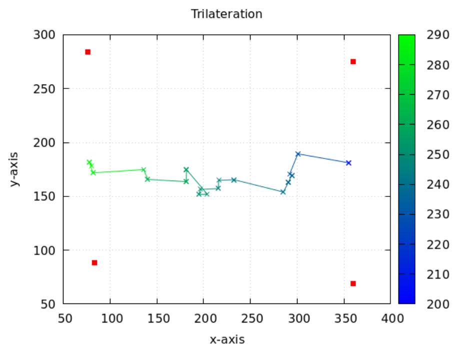

(b) Path followed by the person

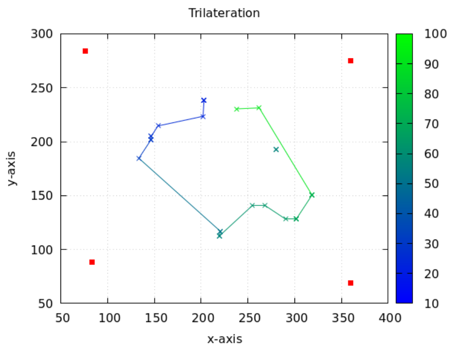
Figure 25.: Trilateration of moving person inside a room with 4 sonars - Obtained with gnuplot



(a) Movement from top to bottom



(b) Movement from left to right



(c) Person describing a circle

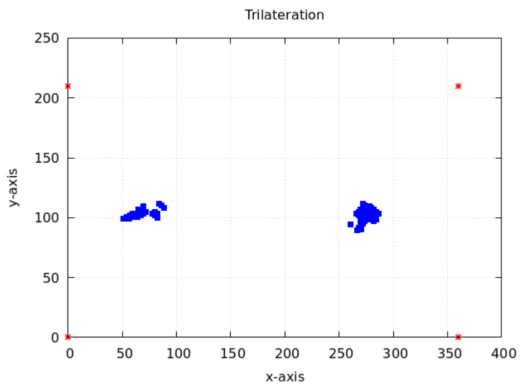
Figure 26.: Trilateration of moving person inside a room with 4 sonars - Obtained with gnuplot

Two persons The experiments for this configuration were performed in a different room than during the previous tests. Figure 27 shows the results we have obtained with two people in a room. In Figure 27a we can see that our system is able to distinguish two nonmoving people very well and that the results obtained are very grouped. The reason why the detection is very good, is because the 2 persons were standing not far away from the sonars (at a bit less than 150 cm) right in front of the sonars.

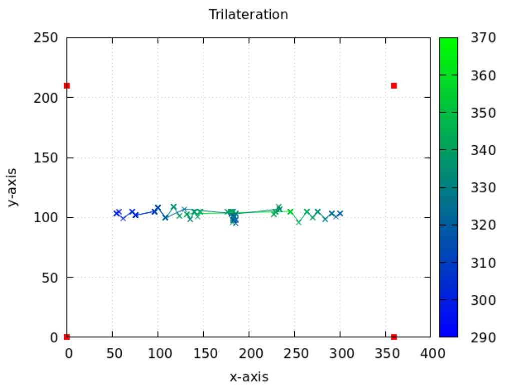
The results in Figure 27b were obtained with two people that start walking from either side of the room and meet in the middle. We have chosen this moving pattern for 2 moving people because it covers an area that is well detected by the sonars. On the graph, we can easily recognize this pattern. We can clearly see in the middle the point where our system stops considering them as 2 persons because of their proximity and considers them as one person instead. As explained in section 5.2.2.1, when the distance between two detected positions is less than 40 cm, we consider them as a single entity and fusion the two detected positions.

We can also observe that people start moving on both sides of the room. Indeed, blue dots start on the left and right, which is the beginning of the movement phase. They then gradually turn into green dots in the middle of the graph.

We have also experimented with two people each standing between two adjacent boards. Unfortunately, as the locations of the two people were outside the sonars' detection zones, no result could be obtained for this experiment.



(a) Two non-moving persons away from each other



(b) Two moving persons synchronously walking to the center

Figure 27.: Trilateration of two persons inside a room with 4 sonars - Obtained with gnuplot

Part IV.

Conclusion

7. Conclusion

Thanks to the rapid and complete development capabilities offered by GRiSP boards, we were able to create and test a generic and scalable system that is resilient to crashes. This indoor person system tracking performs sensor fusion on the sensor-equipped devices at near real-time. We also have made it available to the great public. Our system can be used on any sonar-equipped device supporting Erlang. It is made of reusable components (such as the `Hera` framework) that can each be reused for an unlimited number of other use-cases.

Through our thesis, we have demonstrated that the approach we proposed of performing calculations at the extreme edge of an IoT network is feasible. The approach has not shown any sign of limitation during the whole process. Furthermore, we have shown with our LiveView application that our system is interoperable with current approaches that use edge gateways.

7.1. Results

Chapter 6 presents the results of the most relevant experiments we have performed. A brief summary of those results is given hereafter.

Good tracking results for one to two persons With 2 or more sonars, the results obtained for one to 2 persons that move or remain static are very good, provided that they stand in an area that is well covered by the sonars. By not taking into account the sonars' detection area, the results may become significantly less accurate (or non-existent in the worst cases).
Moving patterns that stay inside the well-covered area are well detected. When testing with a person that moves in a straight line or in a circle, the pattern that the person follows is well-recognizable on the obtained graphs. It is also the case for two moving persons that stand not too close from each other.

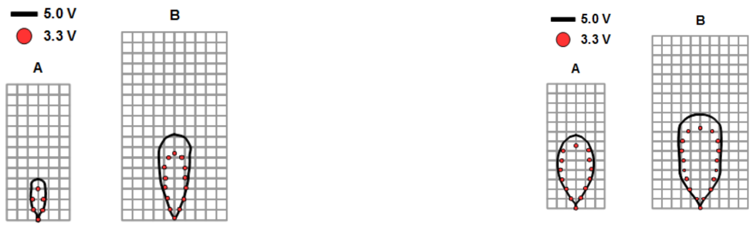
Limitations and strengths of the used sonar As we saw in the sonar's datasheet and in our results, the area covered by a sonar is quite limited. Once a person leaves an area that is well covered by at least 2 sonars, the results' quality is strongly decreased. The sonar however yields very good results for humans that stand/move inside well-covered zones. This explains the good results we have obtained for those cases.

The very powerful and unavoidable anti-crosstalk system The results with versus without anti-crosstalk are without call. With this feature disabled, the results we obtained using more than one sonar are disastrous and do not allow to determine with a certain consistency where the person is really located. This is even more true the more sonars are used in the same room.

Necessity of the LiveView It is very hard (if not impossible) to visualize a person's location live using incoming raw logs on a computer of that person's position. This is where the LiveView becomes essential, since it renders a nice representation of the room (including the tracked target(s)) at near real-time.

7.2. Future work

Perform tests with more powerful sonars As mentioned in the previous section, the range of the Pmod MAXSONAR we used for our experiences is not large enough to well cover a large area. Better results would be obtained with better sonars. Pmod MAXSONARs make use of the MaxBotix 1010 LV-MaxSonar-EZ1. MaxBotix Inc. has manufactured another sonar, the MaxBotix 1000 LV-MaxSonar-EZ0, that is much better. This version, according to the datasheet, can detect a person up to approximately 10 feet (3 meters) and has a much larger coverage area, as shown in Figure 28 [10]. It is possible that other much more powerful miniature sonars will be produced in the future.



(a) Model MB1010 (b) Model MB1000

Figure 28.: LV-MaxSonar-EZ1's beam patterns - taken from its data-sheet

Store more than the last obtained measurement For the moment, we only store on each node the last measurement obtained from all nodes of the cluster. We have made this choice because our use-case only requires this functionality in its current version. To improve our system, we could store more past measurements to allow the better filter them.

Store and aggregate calculation results For the moment, the results of the calculations sent by the other boards and itself are not stored in memory. It is currently therefore not possible to perform operations on them. By adding this feature, it could for example be interesting to gather the position calculations of all the nodes in one point and to perform splines using past calculations to get a smooth trajectory on the live view.

Test our system on GRiSP 2 We are very excited about the increased performance announced for the second version of the GRiSP. Unfortunately, this version is not yet available, we were hence not able to test our system on these new devices.

Automatically determine the position of each board At the moment, with our implementation, we have to specify on each board the position measured by hand of each board in the room. Moreover, if we change the position of the board, we would have to encode its new position and send it to the other nodes of the cluster to ensure that the tracking results remain correct.

With GRiSP boards, we can use other sensors in addition to the Pmod MAXSONAR. We can for example equip GRiSP boards with the Pmod NAV which includes a 3-axis accelerometer and a 3-axis gyroscope [50]. With this sensor, it is possible to calculate in real-time the position of each board. Thanks to Hera's genericity, one just has to specify in our system a measurement that uses motion data from the sensor and a calculation that will calculate the position of the board from this information. Each board should also start the calculation of the position from the same position in the room. This way each board would be able to calculate its exact (x,y) position in the room.

This would also make it possible to move the boards without having to specify new coordinates.

Improve trilateration formulas by integrating the angles of each sonar With the formulas we use for the trilateration of a person, the sonars cannot be tilted with respect to the ground, otherwise the result of the calculation would be degraded. This represents a limitation. By integrating the angles in relation to the vertical and horizontal planes, it would be possible for example to place the sonars in the same way as surveillance cameras, i.e., in the upper corners of a room.

By combining this with the idea of the previous point, it would be possible to calculate in real-time the exact position of a person while having moving boards (we can imagine boards fixed on drones that would go from room to room to track a person moving in his house).

Possibility to choose the communication method We have showed for our use-case that we needed to use UDP rather than TCP to perform computations at near real-time. We therefore equipped Hera with this only means of communication between nodes. However, not all use-cases need to operate at near real-time. Some of them rather need no data to be lost and therefore should use TCP as a way to communicate. It would be good to add the possibility of choosing between UDP and TCP to transmit data through the cluster.

Easy to assemble live view page for any use-case For the moment, only 3 pages of our LiveView app allow to see live what the nodes are sending to the UDP multicast group. One of these pages is specific to our use-case and the two others are just text information. As we have explained before, having a graphical representation is very important in a live application. It would therefore be very useful to have an API that allows users to compose with little code their own live view page, like we did for the representation of a room.

Improve the detection of more than one person It is hard to determine the exact number of sonars required to track a fixed number of persons. How many sonars are needed to detect 2 persons? This is a question we asked ourselves when we started talking about tracking a second person in a room. As discussed in section 5.2.2.1, with 4 sonars and two people, there are situations where there are 4 possible positions where the two people can be. We have therefore concluded that we would need at least 5 sonars (and a better algorithm) for the detection of two people.

It would be interesting to be able to determine, for a given number of individuals in a room, the minimum number of sonars required as well as their location in order to be able to detect all of them correctly.

A. Source Code

A.1. sensor_fusion

The full source code of this application can be found on our Github repository : https://github.com/bastinjul/sensor_fusion. The section 5.2 explains this source code.

A.1.1. hera_position

```
-spec launch_hera(PosX :: integer(), PosY :: integer(), NodeId :: integer(), {
    MinX :: integer(), MinY :: integer()}, {MaxX :: integer(), MaxY :: integer
    ()}) -> any().
launch_hera(PosX, PosY, NodeId, {MinX, MinY}, {MaxX, MaxY}) ->
    pmod_maxsonar:set_mode(single),
    Measurements = [
        hera:get_synchronized_measurement(sonar, fun() -> sonar_measurement()
    end, fun(CurrVal, PrevVal, TimeDiff, UpperBound, DefaultMeas) ->
    filter_sonar(CurrVal, PrevVal, TimeDiff, UpperBound, DefaultMeas) end,
    0.14, infinity),
        hera:get_unsynchronized_measurement(pos, fun() -> {ok, #{x => PosX, y
    => PosY, node_id => NodeId}} end, undefined, 0.28, 3, 500)
    ],
    Calculations = [hera:get_calculation(position, fun() -> calc_position(
    NodeId, {MinX, MinY}, {MaxX, MaxY}) end, 50, infinity, fun(CurrVal,
    PrevVal, TimeDiff, UpperBound, Args) -> filter_position(CurrVal, PrevVal,
    TimeDiff, UpperBound, Args) end, 0.28)],
    hera:launch_app(Measurements, Calculations).

launch_hera(PosX, PosY, NodeId, MaxIteration, {MinX, MinY}, {MaxX, MaxY}) ->
    pmod_maxsonar:set_mode(single),
    Measurements = [
        hera:get_synchronized_measurement(sonar, fun() -> sonar_measurement()
    end, fun(CurrVal, PrevVal, TimeDiff, UpperBound, DefaultMeas) ->
    filter_sonar(CurrVal, PrevVal, TimeDiff, UpperBound, DefaultMeas) end,
    0.14, MaxIteration),
        hera:get_unsynchronized_measurement(pos, fun() -> {ok, #{x => PosX, y
    => PosY, node_id => NodeId}} end, undefined, 0.28, 3, 500)
    ],
    Calculations = [hera:get_calculation(position, fun() -> calc_position(
    NodeId, {MinX, MinY}, {MaxX, MaxY}) end, 50, MaxIteration, fun(CurrVal,
    PrevVal, TimeDiff, UpperBound, Args) -> filter_position(CurrVal, PrevVal,
    TimeDiff, UpperBound, Args) end, 0.28)],
    hera:launch_app(Measurements, Calculations).
```

```
launch_hera(PosX, PosY, NodeId, Frequency, MaxIteration, {MinX, MinY}, {MaxX,
    MaxY}) ->
    Measurements = [
        hera:get_unsynchronized_measurement(sonar, fun() -> sonar_measurement
    () end, fun(CurrVal, PrevVal, TimeDiff, UpperBound, DefaultMeas) ->
    filter_sonar(CurrVal, PrevVal, TimeDiff, UpperBound, DefaultMeas) end,
    0.14, MaxIteration, Frequency),
        hera:get_unsynchronized_measurement(pos, fun() -> {ok, #{x => PosX, y
    => PosY, node_id => NodeId}} end, undefined, 0.28, 3, 500)
    ],
    Calculations = [hera:get_calculation(position, fun() -> calc_position(
    NodeId, {MinX, MinY}, {MaxX, MaxY}) end, 50, MaxIteration, fun(CurrVal,
    PrevVal, TimeDiff, UpperBound, Args) -> filter_position(CurrVal, PrevVal,
    TimeDiff, UpperBound, Args) end, 0.28)],
    %Calculations = [], % no calculation
    hera:launch_app(Measurements, Calculations).

restart(Frequency, MaxIterations) ->
    restart_measurement(MaxIterations),
    restart_calculation(Frequency, MaxIterations).

restart_calculation(Frequency, MaxIterations) ->
    hera:restart_calculation(position, Frequency, MaxIterations).

restart_measurement(MaxIterations) ->
    hera:restart_measurement(pos, false),
    hera:restart_sync_measurement(sonar, MaxIterations, false).
```

Snippet A.1: API of hera_position

A.1.2. Configuration files

The section 5.3.5.1 refers to the following code snippets.

```
    {logger_level, notice},
    {logger, [
        %% Console logger
        {handler, default, logger_disk_log_h,
           #{config => #{
             file => "logs/notice",
             type => wrap,
             max_no_files => 10,
             max_no_bytes => 1048576, % 10 x 1mb
             filesync_repeat_interval => 3000
           },
           filters => [{notice, {fun logger_filters:level/2, {stop, neq,
    notice}}}],
           level => notice,
           formatter => {logger_formatter , #{single_line => true, max_size =>
     1024}}}
        },

        %% Disk logger for error
        {handler, disk_log, logger_disk_log_h,
           #{config => #{
                file => "logs/error",
                type => wrap,
                max_no_files => 10,
                max_no_bytes => 1048576, % 10 x 1mb
                filesync_repeat_interval => 3000
             },
             level => error,
             formatter => {logger_formatter , #{single_line => false}}}
        }
    ]}
```

Snippet A.2: Configuration of loggers on IoT devices

```
  {logger_level, all},
  {logger, [
        %% Console logger
        {handler, default, logger_std_h,
           #{level => notice,
            formatter => {logger_formatter , #{single_line => false}}}
        },

        %% Disk logger for warning
        {handler, disk_log, logger_disk_log_h,
           #{config => #{
                file => "logs/warning",
                type => wrap,
                max_no_files => 10,
                max_no_bytes => 512000 % 10 x 5mb
             },
             level => warning,
             formatter => {logger_formatter , #{single_line => true, max_size
    => 256}}}
        }
    ]}
```

Snippet A.3: Configuration of loggers for emulation on computer

A.2. Hera

The full source code of this application can be found on our Github repository : https://github.com/guiste10/hera. Section 5.3 talks about this application.

A.2.1. hera_multicast

The section 5.3.4 is talking about UDP multicast group connexion.

```
{ok, Sock} = gen_udp:open(?MULTICAST_PORT, [
  binary,
  inet,
  {active, true},
  {multicast_if, OwnAddr}, %specify the network interface to use to send
  multicast
  {multicast_loop, false}, %no return of the packets
  {reuseaddr, true},
  {add_membership, {?MULTICAST_ADDR, OwnAddr}} %join a multicast group and
  use the specified network interface
]),
```

Snippet A.4: Entire parameters to open a multicast connection

A.3. hera_synchronization

The full source code of this application can be found on our Github repository : https://github.com/bastinjul/hera_synchronization. Section 5.4 talk about this application.

A.4. sensor_fusion_live_view

The full source code of this application can be found on our Github repository : https://github.com/bastinjul/sensor_fusion_live_view. Section 5.6 talk about this application.

A.5. pmod_maxsonar module

We talk about this new implemetation in the section 5.5.

```
% @doc
% <a href="https://store.digilentinc.com/pmodmaxsonar-maxbotix-ultrasonic-
  range-finder/">
% Pmod MAXSONAR
% </a>
% module.
%
% The Pmod MAXSONAR cyclically sends measurements via the UART interface.
% This module converts and stores the latest measurement.
%
% Start the driver with
%  ```
%  1> grisp:add_device(uart, pmod_maxsonar).
%  '''
% @end
-module(pmod_maxsonar).

-behaviour(gen_server).

% API
-export([start_link/2]).
-export([get/0]).
-export([set_mode/1]).

% Callbacks
-export([init/1]).
-export([handle_call/3]).
-export([handle_cast/2]).
-export([handle_info/2]).
-export([code_change/3]).
-export([terminate/2]).

%--- Records
    -------------------------------------------------------------------

-record(state, {
  port,
  last_val,
  callers :: list(),
  mode :: disabled %% in this mode returns undefined when getting a value
  | single %% Triggering mode
  | continuous %% Free run mode
}).

%--- API
    -----------------------------------------------------------------------

% @private
start_link(Slot, _Opts) ->
  gen_server:start_link({local, ?MODULE}, ?MODULE, Slot, []).

% @doc Get the latest measured distance in inches.
-spec get() -> integer().
get() ->
  gen_server:call(?MODULE, get_value).

-spec set_mode(disabled | continuous | single) -> any().
set_mode(Mode) ->
  gen_server:call(?MODULE, {set_mode, Mode}).

%--- Callbacks
    -----------------------------------------------------------------
```

```
% @private
init(Slot = uart) ->
  Port = open_port({spawn_driver, "grisp_termios_drv"}, [binary]),
  grisp_devices:register(Slot, ?MODULE),
  grisp_gpio:configure(uart_2_txd, output_1), %% by default continuous mode
  {ok, #state{port = Port, mode = continuous, callers = []}}.

% @private
handle_call(get_value, _From, #state{mode = disabled} = State) ->
  {reply, undefined, State};
handle_call(get_value, _From, #state{last_val = Val, mode = continuous} =
    State) ->
  {reply, Val, State};
handle_call(get_value, From, #state{mode = single, callers = Callers} = State)
     ->
  case length(Callers) of
    0 ->
      grisp_gpio:configure(uart_2_txd, output_1),
      grisp_gpio:configure(uart_2_txd, output_0);
    _ -> ok
  end,
  {noreply, State#state{callers = State#state.callers ++ [From]}};
handle_call({set_mode, disabled}, _From, State) ->
  grisp_gpio:configure(uart_2_txd, output_0),
  {reply, ok, State#state{mode = disabled}};
handle_call({set_mode, single}, _From, State) ->
  grisp_gpio:configure(uart_2_txd, output_0),
  {reply, ok, State#state{mode = single}};
handle_call({set_mode, continuous}, _From, State) ->
  grisp_gpio:configure(uart_2_txd, output_1),
  {reply, ok, State#state{mode = continuous}}.

% @private
handle_cast(Request, _State) -> error({unknown_cast, Request}).

% @private
handle_info({Port, {data, Data}}, #state{port = Port, mode = continuous} =
    State) ->
  {noreply, State#state{last_val = decode(Data, State)}};
handle_info({Port, {data, Data}}, #state{port = Port, mode = single, callers =
     Callers} = State) ->
  lists:map(fun(C) -> gen_server:reply(C, decode(Data, State)) end, Callers),
  {noreply, State#state{port = Port, callers = []}}.

% @private
code_change(_OldVsn, State, _Extra) -> {ok, State}.

% @private
terminate(_Reason, _State) -> ok.
```

```
% @private
decode(Data, State) ->
  case Data of
    % Format of response is 'Rxxx\n' where xxx is the decimal
    % representation of the measured range in inches (2.54cm)
    % (left-padded with zeros - so there are always three digits)
    <<$R, D1, D2, D3, $\n>> when $0 =< D1, D1 =< $9,
      $0 =< D2, D2 =< $9,
      $0 =< D3, D3 =< $9 ->
      % Val is given in inches
      (D1 - $0) * 100 + (D2 - $0) * 10 + (D3 - $0);
    % Sometimes for no obvious reason we receive
    % a different value from the sonar.
    % Instead of $R we get two garbage characters
    <<_, _, D1, D2, D3, $\n>> when $0 =< D1, D1 =< $9,
      $0 =< D2, D2 =< $9,
      $0 =< D3, D3 =< $9 ->
      % Val is given in inches
      (D1 - $0) * 100 + (D2 - $0) * 10 + (D3 - $0);
    _ ->
      State#state.last_val
  end.
```

Snippet A.5: Entire parameters to open a multicast connection

B. Figures

B.1. Sensor_fusion

B.1.1. Supervision tree

We talk about the supervision trees of Hera in section 5.3.1.

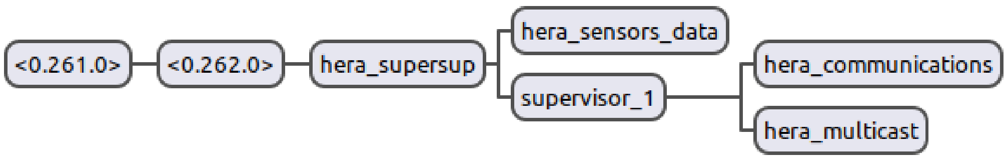
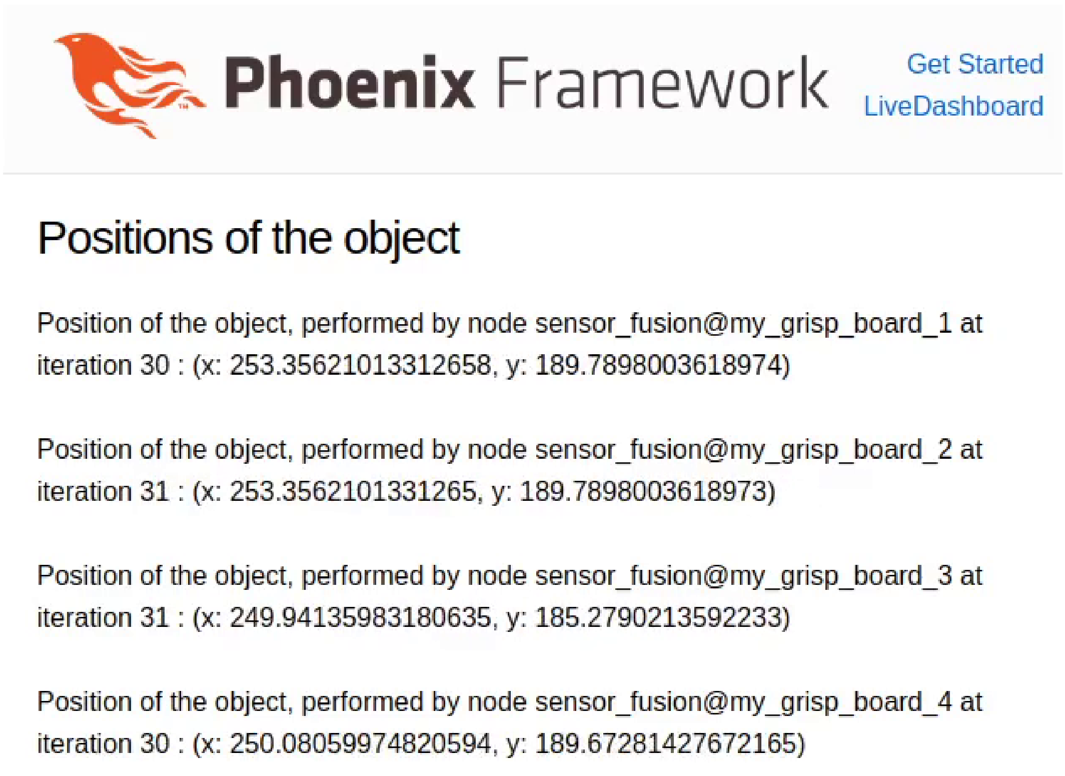


Figure 29.: Supervision tree of `Hera` on a computer on a emulation of `GRiSP boards`

B.2. LiveView

We talk about these illustrations of our LiveView at section 5.6.



Positions of the object

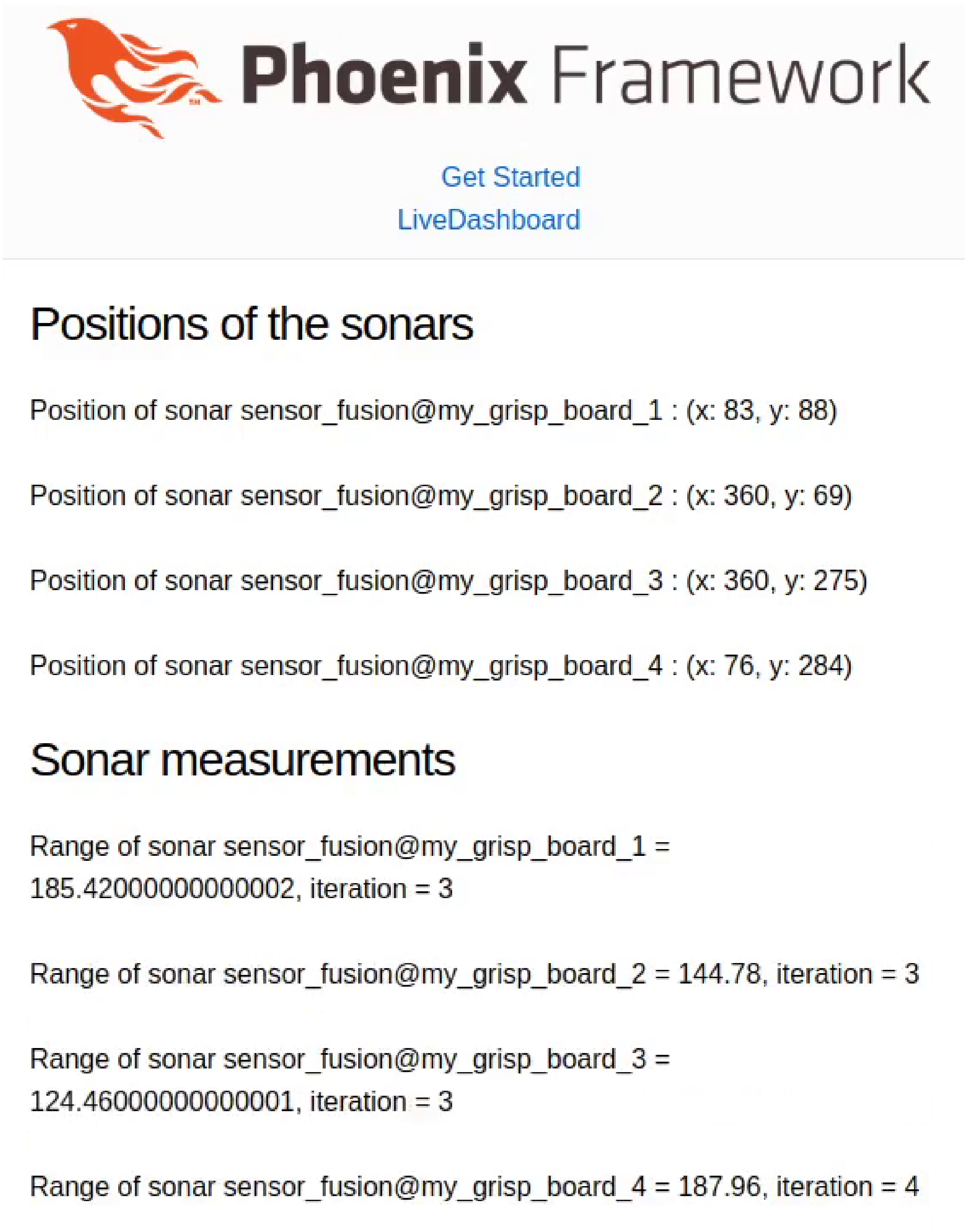
Position of the object, performed by node sensor_fusion@my_grisp_board_1 at iteration 30 : (x: 253.35621013312658, y: 189.7898003618974)

Position of the object, performed by node sensor_fusion@my_grisp_board_2 at iteration 31 : (x: 253.3562101331265, y: 189.7898003618973)

Position of the object, performed by node sensor_fusion@my_grisp_board_3 at iteration 31 : (x: 249.94135983180635, y: 185.2790213592233)

Position of the object, performed by node sensor_fusion@my_grisp_board_4 at iteration 30 : (x: 250.08059974820594, y: 189.67281427672165)

Figure 30.: Visual of the calculations LiveView page



Positions of the sonars

Position of sonar sensor_fusion@my_grisp_board_1 : (x: 83, y: 88)

Position of sonar sensor_fusion@my_grisp_board_2 : (x: 360, y: 69)

Position of sonar sensor_fusion@my_grisp_board_3 : (x: 360, y: 275)

Position of sonar sensor_fusion@my_grisp_board_4 : (x: 76, y: 284)

Sonar measurements

Range of sonar sensor_fusion@my_grisp_board_1 = 185.42000000000002, iteration = 3

Range of sonar sensor_fusion@my_grisp_board_2 = 144.78, iteration = 3

Range of sonar sensor_fusion@my_grisp_board_3 = 124.46000000000001, iteration = 3

Range of sonar sensor_fusion@my_grisp_board_4 = 187.96, iteration = 4

Figure 31.: Visual of the measurements LiveView page

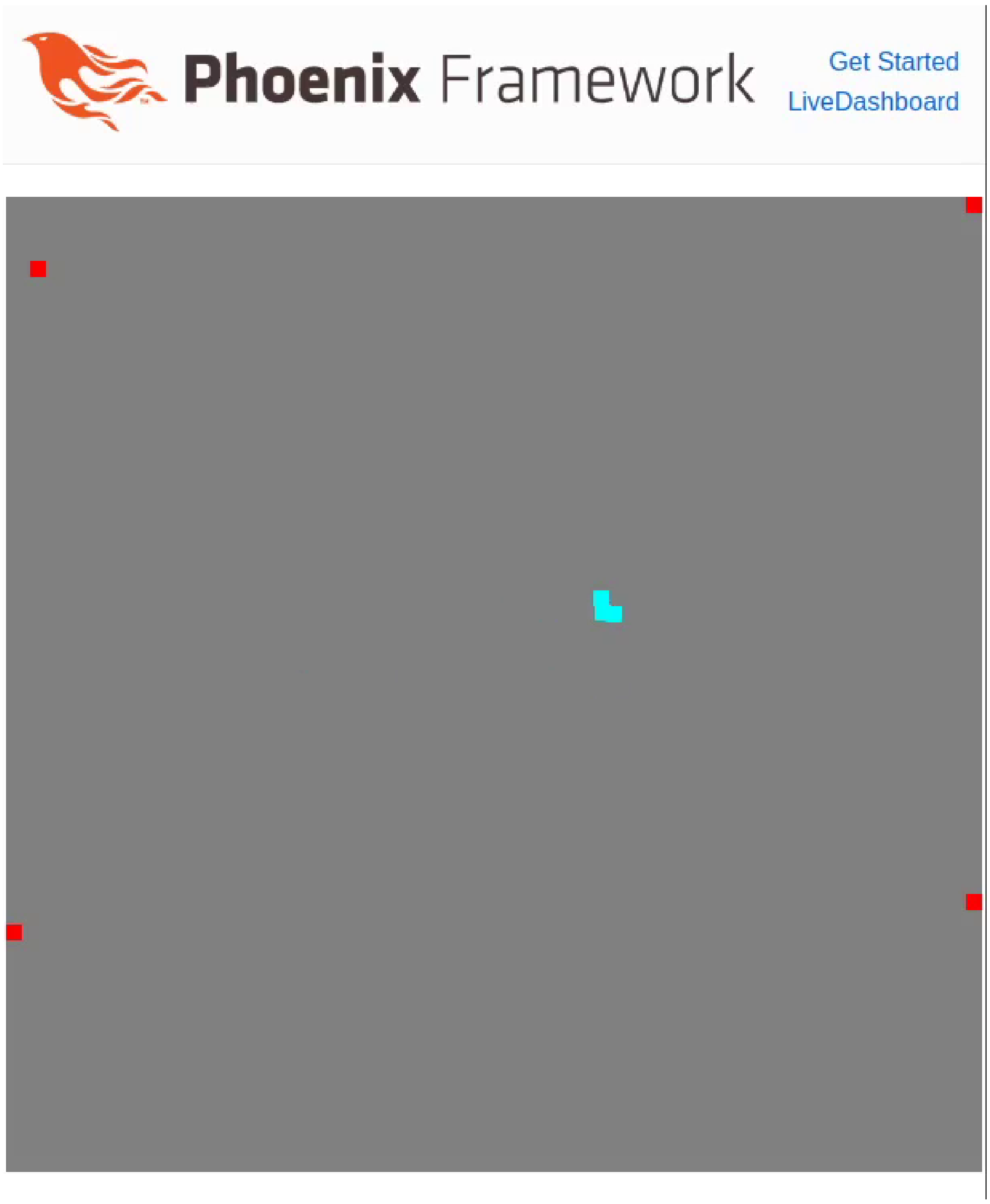


Figure 32.: Visual of the room LiveView page

B.3. Experiments

B.3.1. Data sending between GRiSP boards

The section 6.1.2 talk about the results obtained by the following graphs.

B.3.1.1. UDP multicast

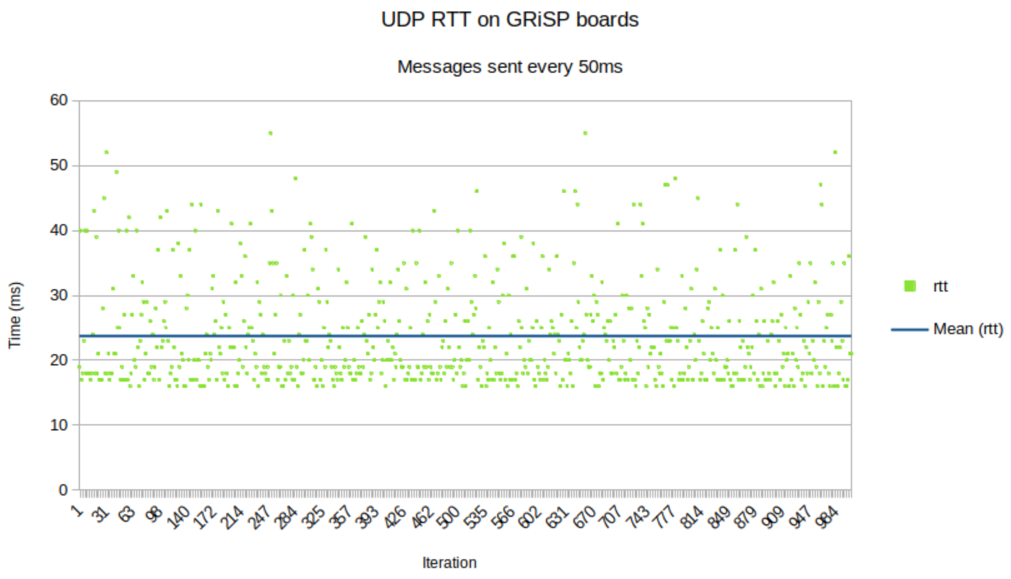


Figure 33.: RTT between two GRiSP boards using UDP multicast group by sending messages every 50 ms

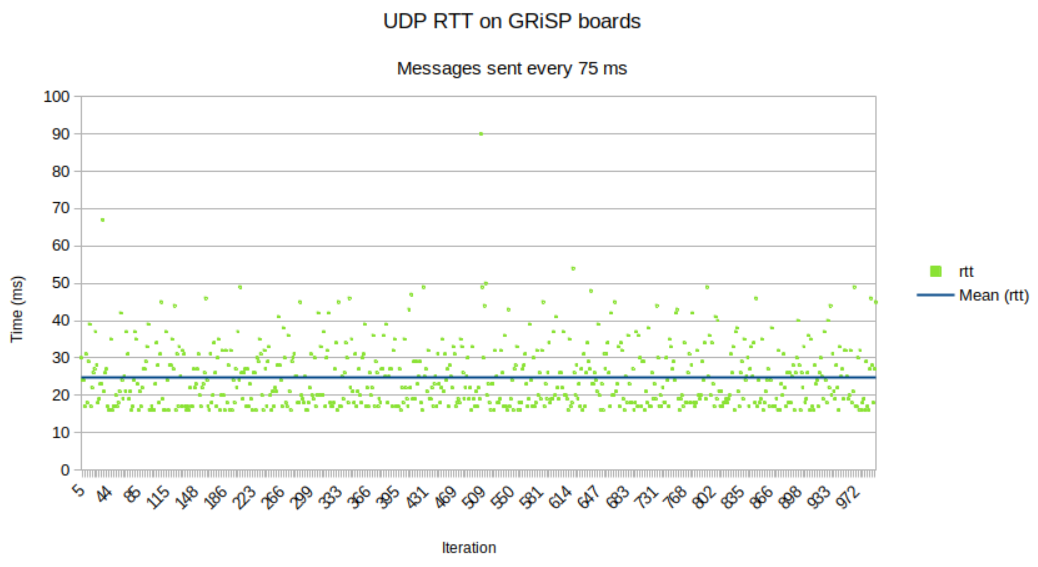


Figure 34.: RTT between two GRiSP boards using UDP multicast group by sending messages every 75 ms

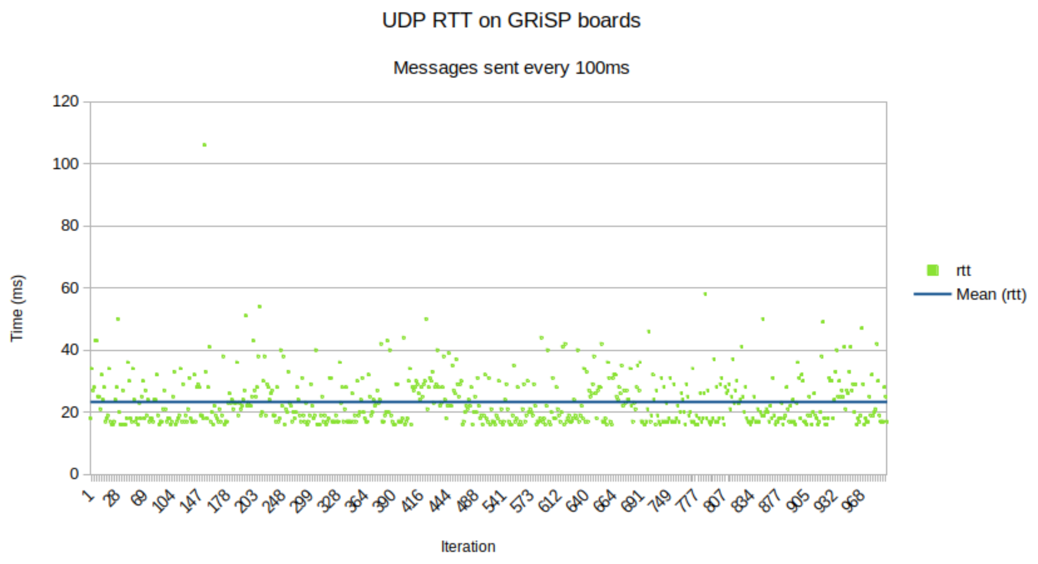


Figure 35.: RTT between two GRiSP boards using UDP multicast group by sending messages every 100 ms

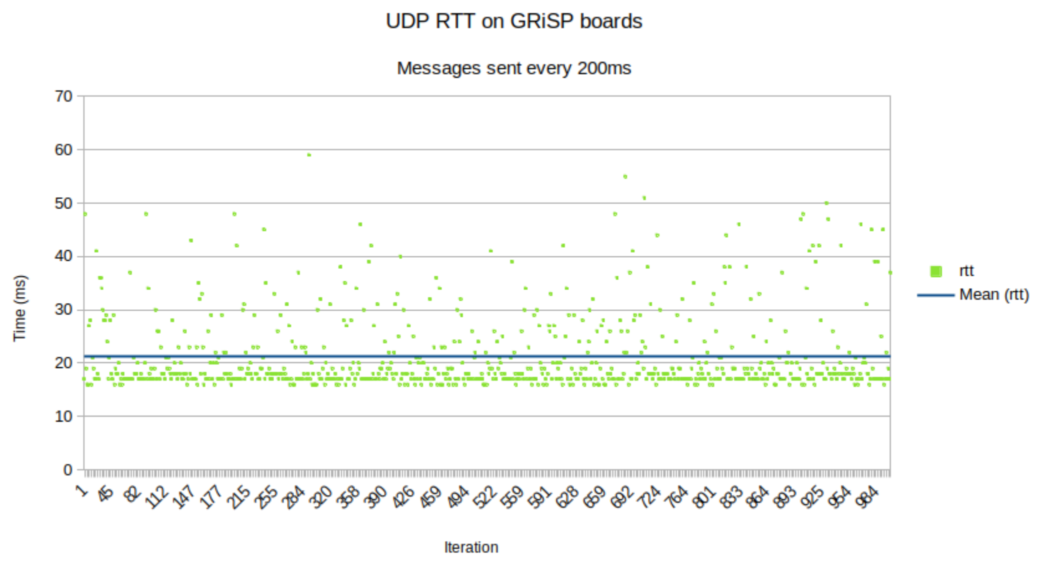


Figure 36.: RTT between two GRiSP boards using UDP multicast group by sending messages every 200 ms

B.3.1.2. TCP

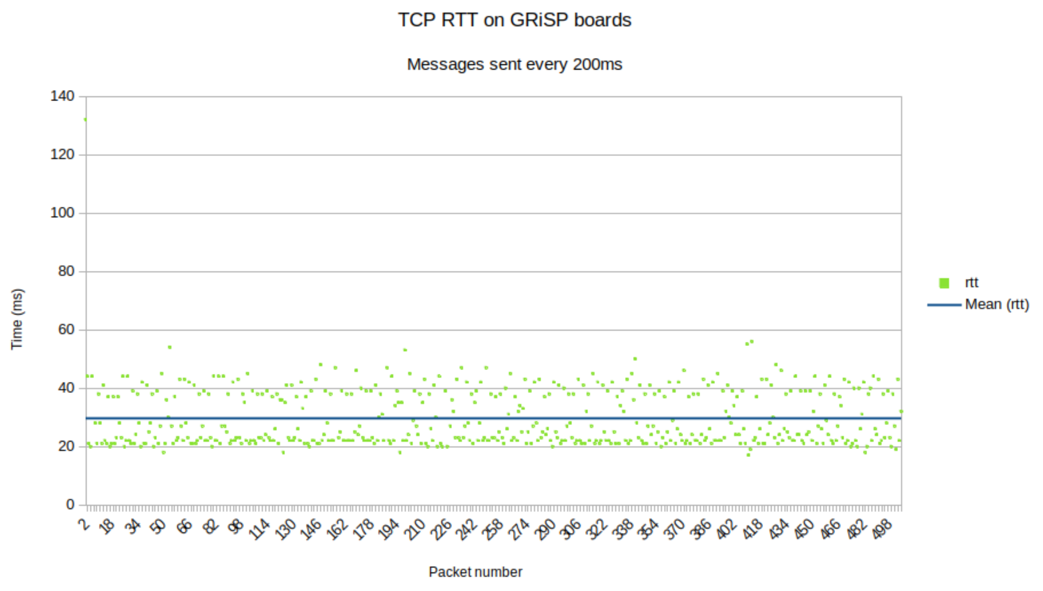


Figure 37.: RTT between two GRiSP boards using TCP by sending messages every 200 ms

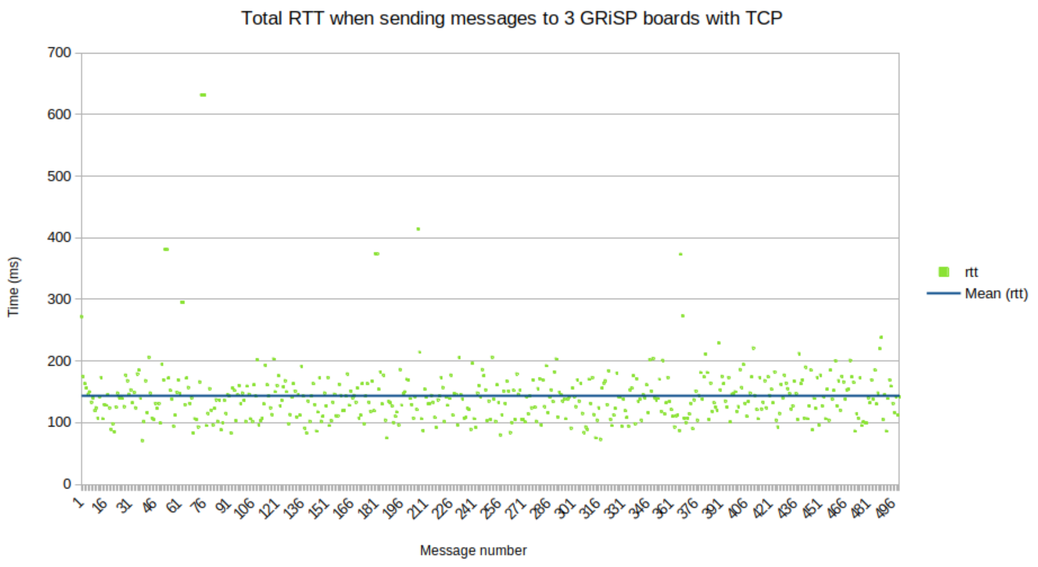
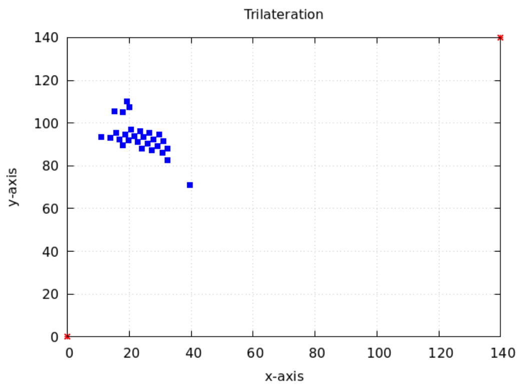


Figure 38.: Total RTT between four GRiSP boards using TCP by sending messages every 200 ms

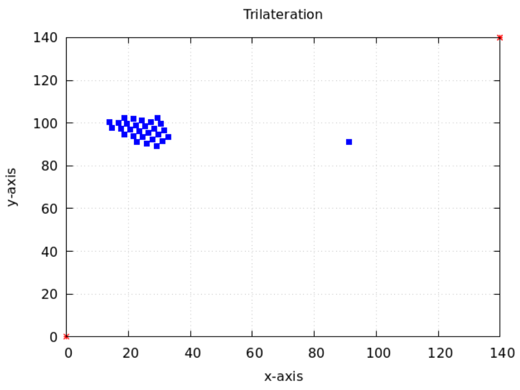
B.3.2. Person Tracking

B.3.2.1. 2 sonars

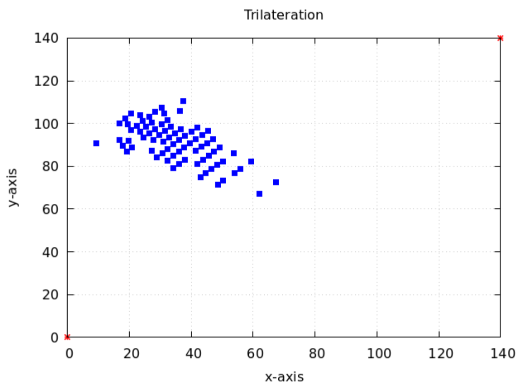
The results of the following results are discussed in section 6.4.2.2.



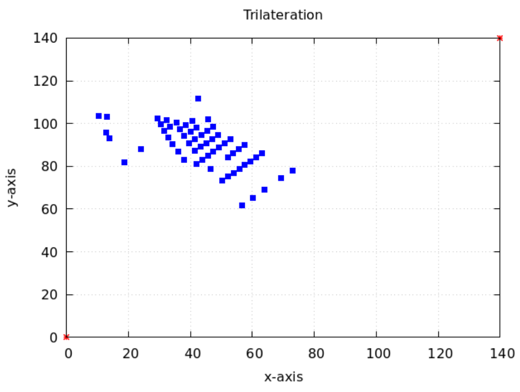
(a) With filter deactivated and synchronization activated



(b) With filter deactivated and synchronization activated



(c) With filter and synchronization deactivated



(d) With filter and synchronization deactivated

Figure 39.: Trilateration of a non-moving person inside a room with 2 facing sonars, with filter deactivated and synchronization activated. Obtained with gnuplot

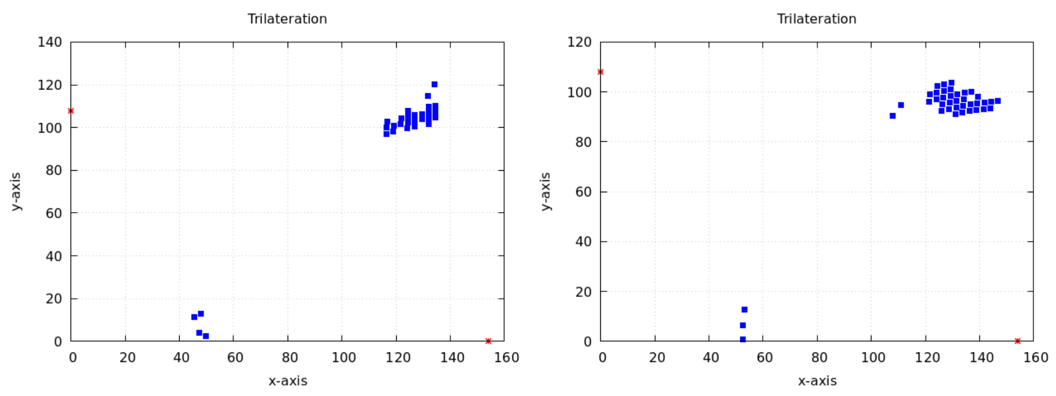


Figure 40.: Trilateration of a non-moving person inside a room with 2 non-facing sonars, with filter and synchronization activated. Obtained with gnuplot

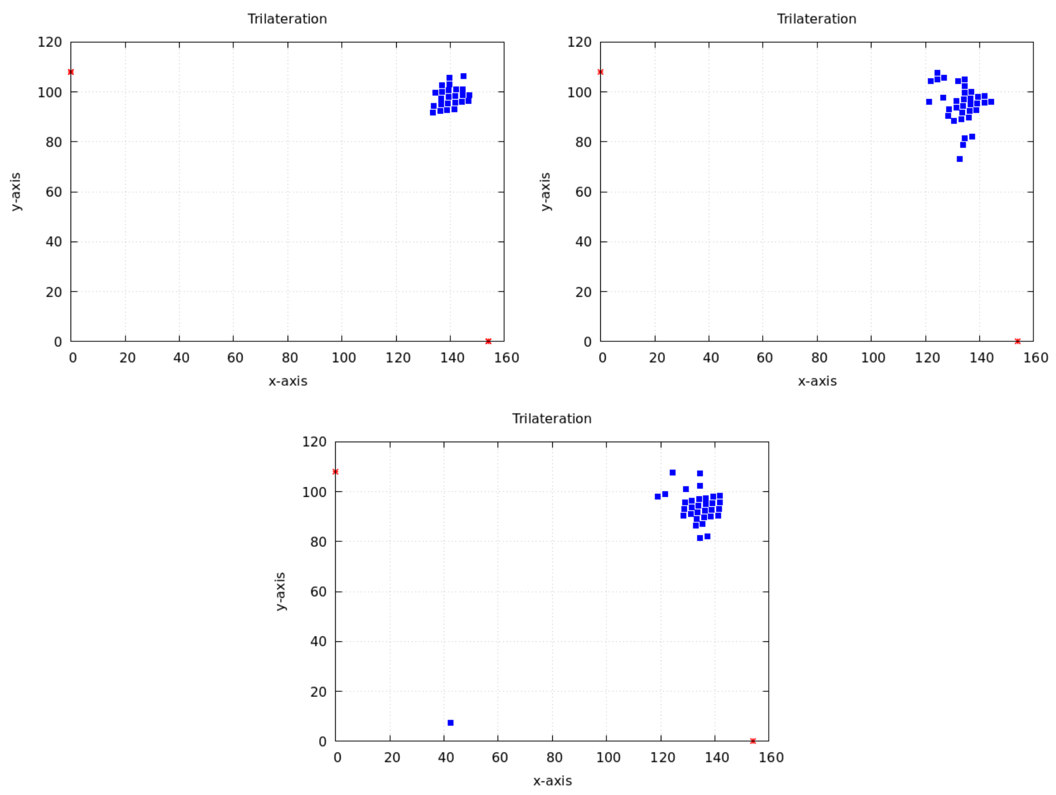


Figure 41.: Trilateration of a non-moving person inside a room with 2 non-facing sonars, with filter deactivated and synchronization ativated. Obtained with gnuplot

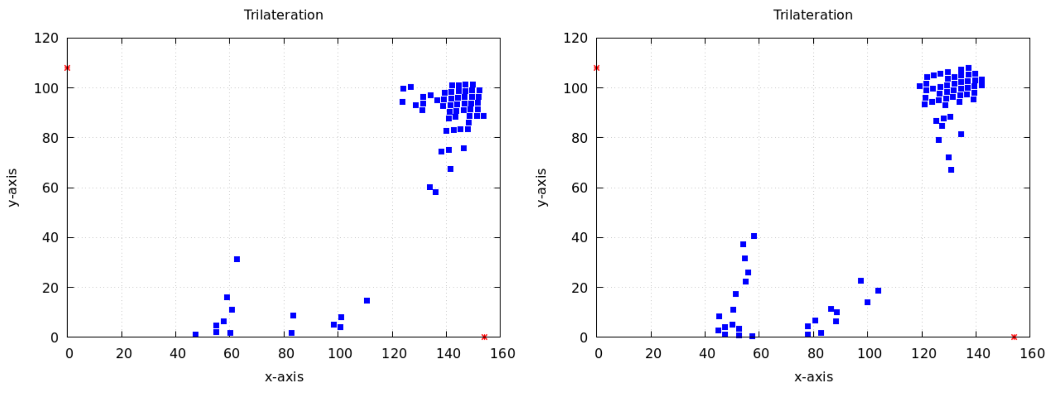


Figure 42.: Trilateration of a non-moving person inside a room with 2 non-facing sonars, with filter and synchronization deactivated. Obtained with gnuplot

B.3.2.2. 3 sonars

The results of the following results are discussed in section 6.4.2.3.

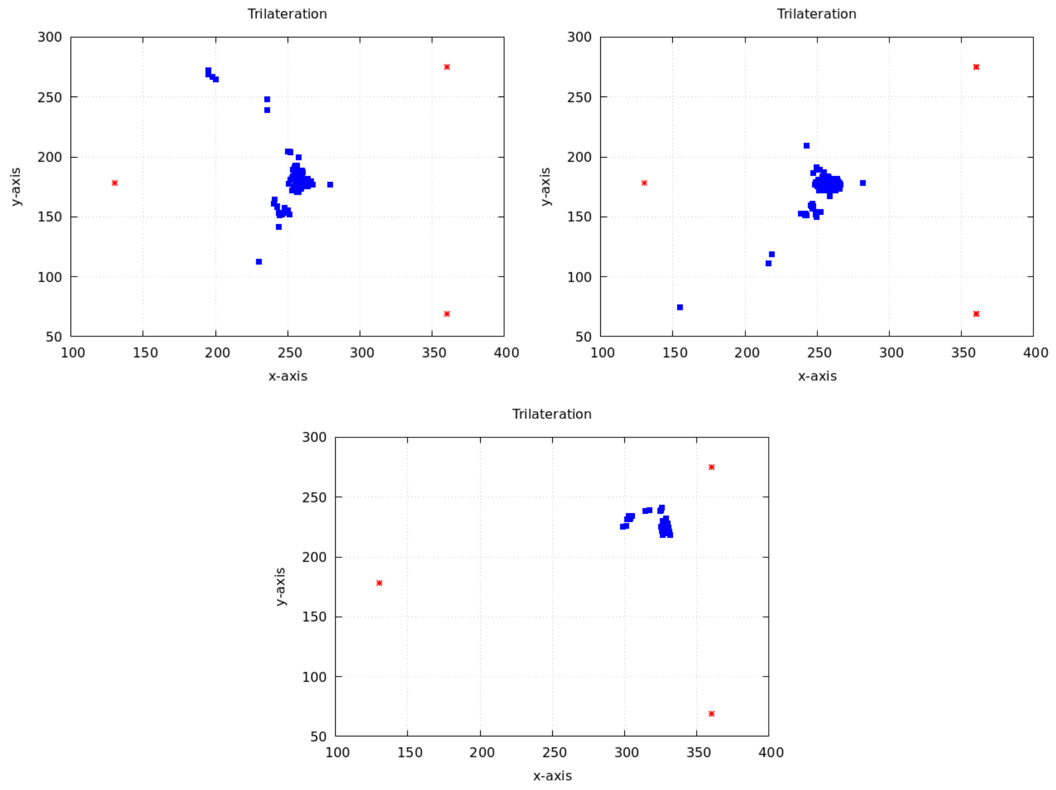


Figure 43.: Trilateration of a non-moving person inside a room with 3 sonars, with filter and synchronization activated. Obtained with gnuplot

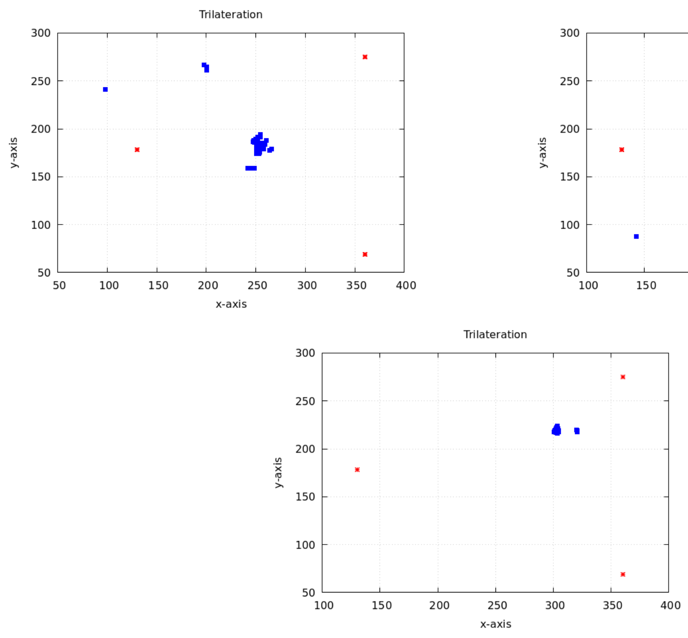


Figure 44.: Trilateration of a non-moving person inside a room with 3 sonars, with filter deactivated and synchronization activated. Obtained with gnuplot

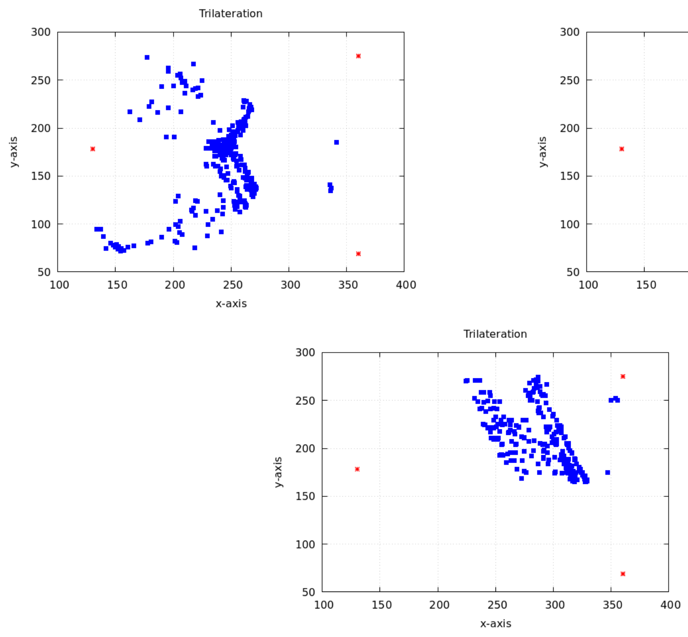
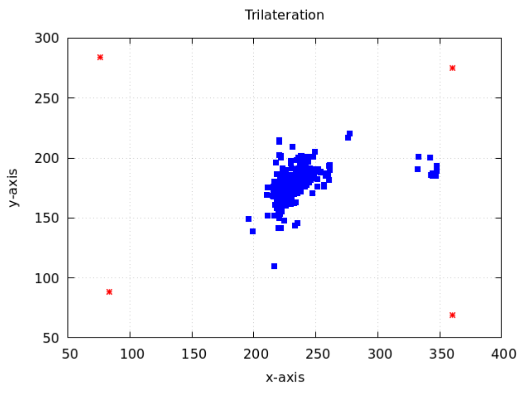


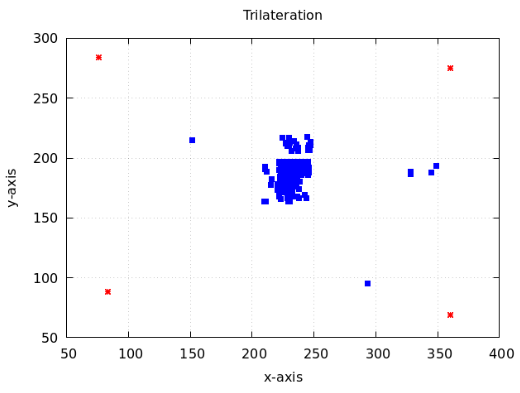
Figure 45.: Trilateration of a non-moving person inside a room with 3 sonars, with filter and synchronization deactivated. Obtained with gnuplot

B.3.2.3. 4 sonars

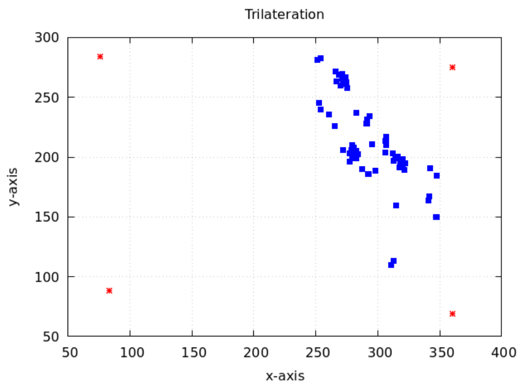
The results of the following results are discussed in section 6.4.2.4.



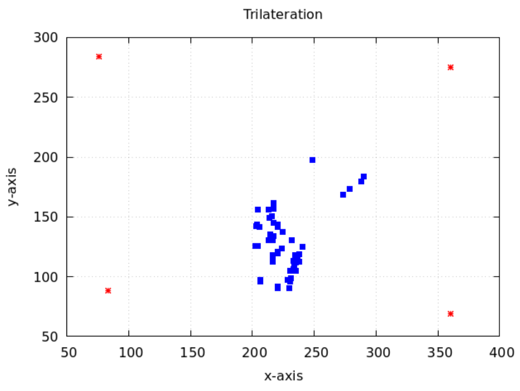
(a) Person on the centrer of the room.



(b) Person on the centrer of the room.

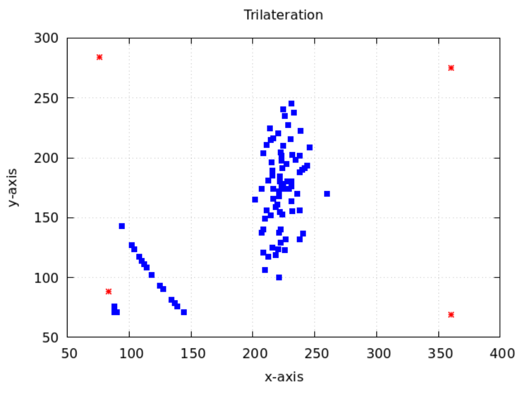


(c) Person on the right side of the room.

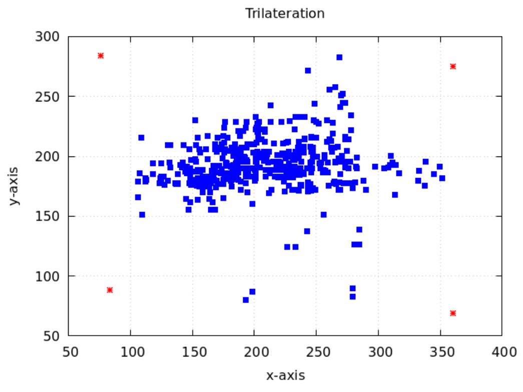


(d) Person on the bottom side of the room.

Figure 46.: Trilateration of a non-moving person inside a room with 4 sonars. Obtained with gnuplot

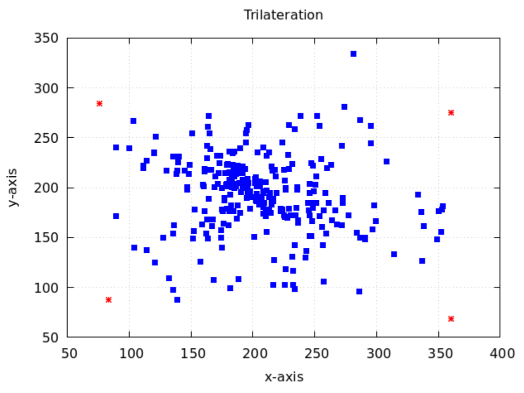


(a) Person going from the bottom to the top of the room.

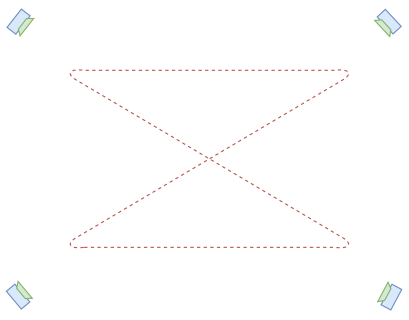


(b) Person going from left to right the room.

Figure 47.: Trilateration of a moving person inside a room with 4 sonars. Obtained with gnuplot

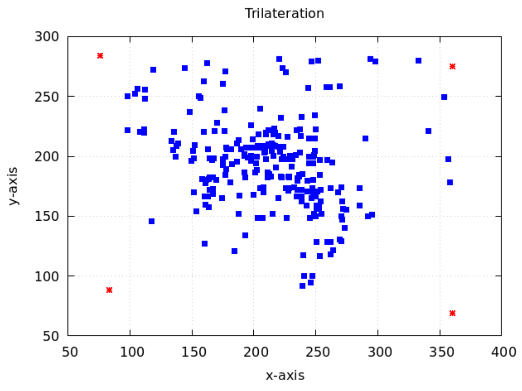


(a) Trilateration of a moving person

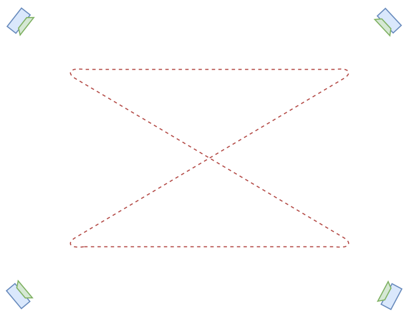


(b) Path followed by the person

Figure 48.: Trilateration of moving person inside a room with 4 sonars. Obtained with gnuplot

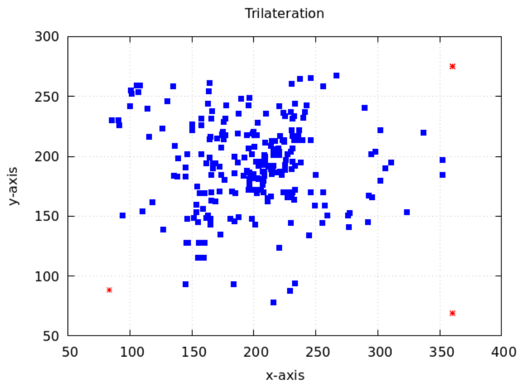


(a) Trilateration of a moving person

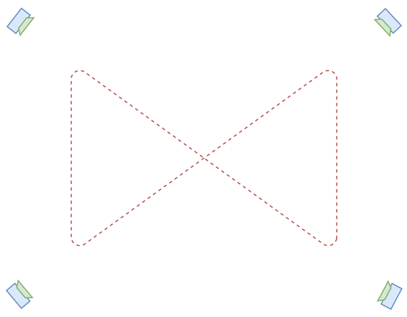


(b) Path followed by the person

Figure 49.: Trilateration of moving person inside a room with 4 sonars. Obtained with gnuplot



(a) Trilateration of a moving person



(b) Path followed by the person

Figure 50.: Trilateration of moving person inside a room with 4 sonars. Obtained with gnuplot

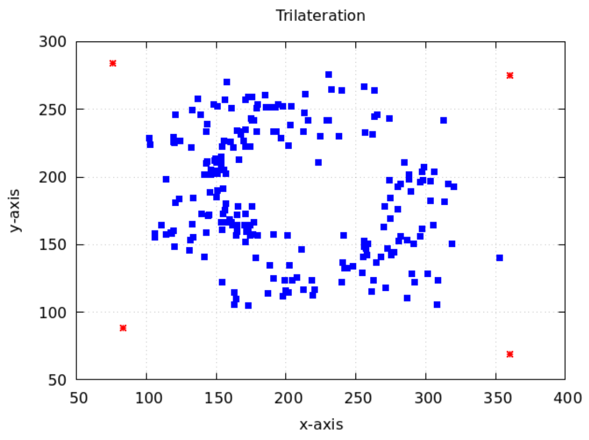


Figure 51.: Trilateration of a moving person inside a room with 4 sonars. Person describing a circle in the room. Obtained with gnuplot

C. User manual

C.1. Installation

Since our system works on GRiSP boards, you have to do the complete installation of `grisp`[1]. Then deploy our system on GRiSP boards, you need to clone our `Sensor_fusion` project[2]. To be able to see a person moving live on your pc, you need to clone our `Sensor_fusion_live_view` application[3]. Then please follow the steps to install the necessary tools in the following sections.

C.1.1. Software versions

When developing our system, we were using precise versions of `Erlang`, `rebar3`, `rebar3 plugins`, `Elixir` and `mix`. You can find these versions on the table 3.

You can quickly verify all your installed current versions using the following command :

```
rebar3 --version && rebar3 plugins list && mix --version
```

Erlang	rebar3	rebar3_hex	rebar3_grisp	Elixir/mix
Erlang/OTP 22.0	3.13.0/lastest	6.9.6	1.3.0	1.10.3

Table 3.: Software versions

C.1.2. Updating versions

If your current versions of `Erlang` and `rebar3 plugins` are lower (or greater in the case of `Erlang`) than the ones listed above and the deployment on the `GRiSP boards` is not working, we recommend you to follow the procedures below in order to update your versions.

[1] https://github.com/grisp/grisp/wiki/Setting-Up-a-Development-Environment
[2] https://github.com/bastinjul/sensor_fusion
[3] https://github.com/bastinjul/sensor_fusion_live_view

C.1.2.1. Erlang 22.0

We highly recommend to use the `asdf` tool in order to be able to install multiple versions of `Erlang` on your device. Once `asdf`[4] and its `Erlang plugin`[5] are installed, you just have to run the following command to install and use the right Erlang version :

```
asdf install erlang 22.0 && asdf global erlang 22.0
```

You can verify that the version is correctly set by running `asdf current`.

C.1.2.2. Rebar3 plugins

The following course of action has taken place during the development of our system, when we were facing a problem of versions that caused troubles with the logging functionality (section 5.3.5.1).

Firstly, execute the below command

```
rebar3 as global plugins upgrade <plugin_name>
```

in order to upgrade [51] the versions of the two `rebar3 plugins` listed above. Secondly, remove the directory containing the `rebar3 plugins` and update `rebar3` using this command

```
rm -r ~/.cache/rebar3/plugins && rebar3 update
```

in order to download effectively the right versions of the plugins.

Lastly, before deploying our system on `GRiSP boards`, remove the `_build` directory located inside the `sensor_fusion` repository if this directory exists.

That's it! You can now deploy our system on the `GRiSP boards` (section C.2).

C.1.3. LiveView

In order to be able to see the live movements of a person in a room via our `sensor_fusion_live_view` application, it is necessary to do the entire installation

[4] https://asdf-vm.com/#/core-manage-asdf-vm
[5] https://github.com/asdf-vm/asdf-Erlang

of the phoenix tool. The complete tutorial to perform this installation can be found at the following link: https://hexdocs.pm/phoenix/installation.html.

The procedure to follow to complete the installation of our application is included in the README of our github repository[6].

C.2. Deployment

There are two possible deployments of our application. It can either be deployed on `GRiSP boards` or on PC's. The deployment on a PC is used for emulating a `GRiSP board` [7] that can recover `GRiSP boards`' data such as measurements and results of calculations.

There exists several differences between the two deployment at the `Hera` implementation level. Depending on the hardware, there are differences in functionalities, particularly in terms of the modules that are launched and for the logging of data (see sections 5.3.1 and 5.3.5.1 for more information).

There are two ways for the boards and computer to communicate between each other: either via an existing wifi network or via an ADHOC wifi network. The different ways to deploy are explained inside the `README.md` file of our project[8].

C.3. Launching the system

C.3.1. Sensor_fusion

Once our `sensor_fusion` application is deployed on the GRiSP boards and the GRiSP boards are started (the two leds on each board turn red when the GRiSP boards have finished their boot phase and our application is started), you have to connect to each of them to launch the application. The command to connect to the GRiSP board is the following:

```
$ erl -sname <shell_name> -remsh sensor_fusion@my_grisp_board_X
  -setcookie MyCookie
```

Your have to replace `<shell_name>` by the name you want to give to your shell. Note that two shells cannot have the same name. You also need to replace the "X"

[6] https://github.com/bastinjul/sensor_fusion_live_view/blob/master/README.md
[7] https://github.com/grisp/grisp_emulation
[8] https://github.com/bastinjul/sensor_fusion/blob/master/README.md

in the name of the board by the corresponding number.

Once connected, you just have to start the application with one of the available commands described in section 5.2.1.

C.3.2. LiveView

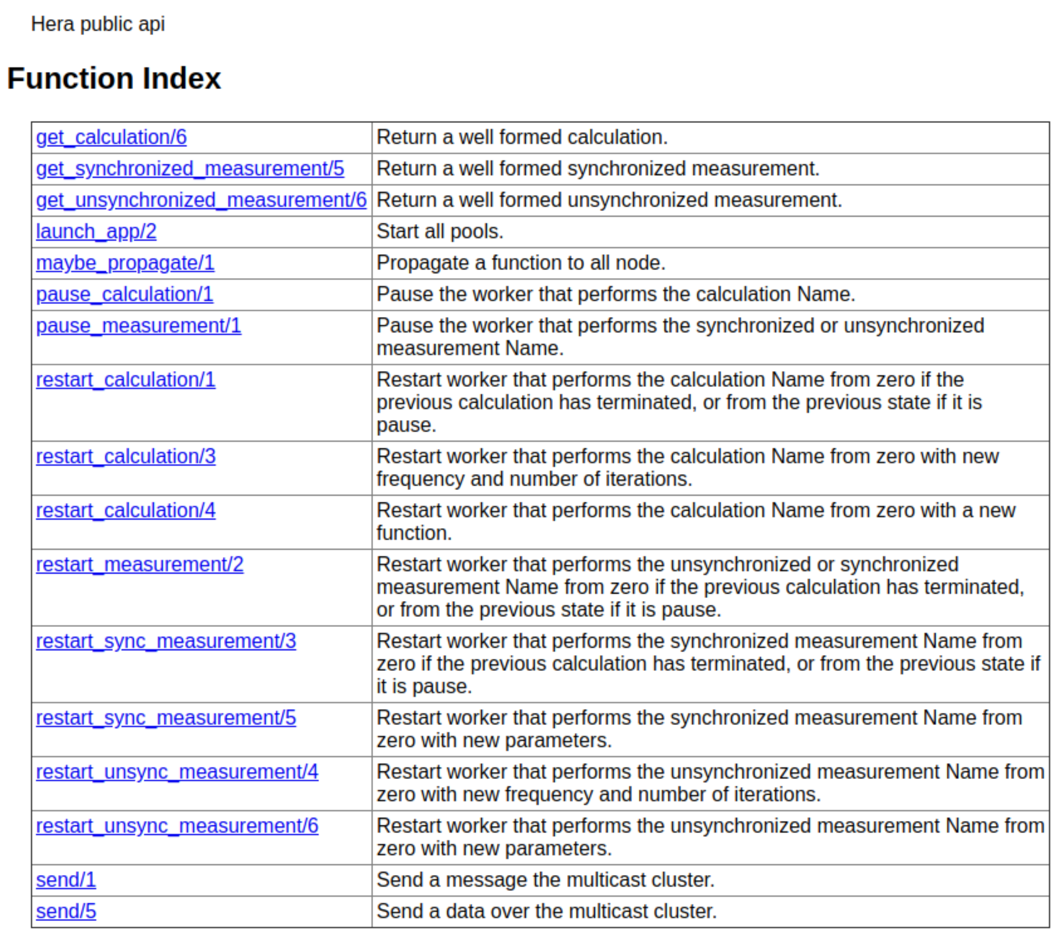
You only need to run the command `mix phx.server` in order to launch the LiveView application. A view of the three web pages can be found in appendix A.4 and are explained in the section 5.6.

C.4. Using Hera and Hera_synchronization for your own project

If you want to use Hera and Hera_synchronization to perform your own sensor merge, please follow the configuration examples for your project in sections 5.2 and 5.4.

C.5. Hera Runtime API

The figure 52 illustrates the different functions of the `Hera` application. For more information about all functions, open the file `<hera_root>/doc/hera.html` of Hera.



Hera public api

Function Index

get_calculation/6	Return a well formed calculation.
get_synchronized_measurement/5	Return a well formed synchronized measurement.
get_unsynchronized_measurement/6	Return a well formed unsynchronized measurement.
launch_app/2	Start all pools.
maybe_propagate/1	Propagate a function to all node.
pause_calculation/1	Pause the worker that performs the calculation Name.
pause_measurement/1	Pause the worker that performs the synchronized or unsynchronized measurement Name.
restart_calculation/1	Restart worker that performs the calculation Name from zero if the previous calculation has terminated, or from the previous state if it is pause.
restart_calculation/3	Restart worker that performs the calculation Name from zero with new frequency and number of iterations.
restart_calculation/4	Restart worker that performs the calculation Name from zero with a new function.
restart_measurement/2	Restart worker that performs the unsynchronized or synchronized measurement Name from zero if the previous calculation has terminated, or from the previous state if it is pause.
restart_sync_measurement/3	Restart worker that performs the synchronized measurement Name from zero if the previous calculation has terminated, or from the previous state if it is pause.
restart_sync_measurement/5	Restart worker that performs the synchronized measurement Name from zero with new parameters.
restart_unsync_measurement/4	Restart worker that performs the unsynchronized measurement Name from zero with new frequency and number of iterations.
restart_unsync_measurement/6	Restart worker that performs the unsynchronized measurement Name from zero with new parameters.
send/1	Send a message the multicast cluster.
send/5	Send a data over the multicast cluster.

Figure 52.: Hera API

Bibliography

[1] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *Cisco IBSG White Paper*, 2011.

[2] "How many iot devices are there in 2020?." https://techjury.net/blog/how-many-iot-devices-are-there/#gref.

[3] "The internet of things: a movement, not a market." https://news.ihsmarkit.com/prviewer/release_only/slug/number-connected-iot-devices-will-surge-125-billion-2030-ihs-markit-says.

[4] "The drive towards intelligent edge computing?." https://medium.com/datadriveninvestor/the-drive-towards-intelligent-edge-computing-f6db3c425332.

[5] "Iot: Understanding the shift from cloud to edge computing." https://internetofbusiness.com/shift-from-cloud-to-edge-computing/.

[6] "What is sensor fusion?." https://whatis.techtarget.com/definition/sensor-fusion.

[7] D. o. E. United Nations and S. Affairs, "World population ageing 2017 - highlights," 2017.

[8] C. P. R. Yosry S. Morsi, Anupam Shukla, *Optimizing Assistive Technologies for Aging Populations.* page 173, AMTCP.

[9] "What does real-time mean and when is it used?." https://www.microcontrollertips.com/faq-real-time-mean/.

[10] "Lv-maxsonar®-ez$^{\text{TM}}$ series - data-sheet." https://maxbotix.com/documents/LV-MaxSonar-EZ_Datasheet.pdf?_ga=2.24468986.1132479797.1595168189-432722846.1590669867.

[11] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, pp. 637–646, Oct 2016.

[12] "What is edge computing: The network edge explained." https://www.cloudwards.net/what-is-edge-computing/.

[13] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, "Globally distributed content delivery," *IEEE Internet Computing*, vol. 6, no. 5, pp. 50–58, 2002.

[14] E. Nygren, R. Sitaraman, and J. Sun, "The akamai network: a platform for high-performance internet applications.," *Operating Systems Review*, vol. 44, pp. 2–19, 01 2010.

[15] "Edge computing vs. fog computing: Definitions and enterprise uses." https://www.cisco.com/c/en/us/solutions/enterprise-networks/edge-computing.html.

[16] F. Bonomi and R. Milito, "Fog computing and its role in the internet of things," *Proceedings of the MCC workshop on Mobile Cloud Computing*, 08 2012.

[17] D. R. Barik, A. Dubey, A. Tripathi, T. Pratik, S. Sasane, R. Lenka, H. Dubey, K. Mankodiya, and V. Kumar, "Mist data: Leveraging mist computing for secure and scalable architecture for smart and connected health," *Procedia Computer Science*, vol. 125, pp. 647–653, 01 2018.

[18] "Cloud vs fog vs mist computing, which one should you use?." https://radiocrafts.com/cloud-vs-fog-vs-mist-computing-which-one-should-you-use/#:~:text=Mist.

[19] L. Yang and B. Liu, "Temporal data fusion at the edge," 07 2019.

[20] A. Misra, "The cognitive edge: Promoting energy-efficient, collaborative sensing by iot personal devices." https://www.cloudwards.net/what-is-edge-computing/, SmartEdge 2019.

[21] "Cognitive edge computing." http://cogedge.ece.utexas.edu/.

[22] I. Kopestenski and P. Van Roy, "Achlys: Towards a framework for distributed storage and generic computing applications for wireless iot edge networks with lasp on grisp," pp. 875–881, 2019.

[23] I. Kopestenski and P. Roy, "Erlang as an enabling technology for resilient general-purpose applications on edge iot networks," 08 2019.

[24] C. Meiklejohn and H. Miller, "Partisan: Enabling cloud-scale erlang applications," 02 2018.

[25] C. Meiklejohn and P. V. Roy, "Lasp: a language for distributed, coordination-free programming," in *PPDP*, 2015.

[26] M. S. Nuno Preguiça, Carlos Baquero, "Conflict-free replicated data types," 02 2018.

[27] C. F. Sturgess B.N., "The surveying handbook, chapter 12:802.11b/g/n trilateration," pp. 234–270, 1995.

[28] J. Armstrong, "A history of erlang," 01 2007.

[29] "Hitchhiker's tour of the beam." http://www.erlang-factory.com/upload/presentations/708/HitchhikersTouroftheBEAM.pdf.

[30] "Elixir." https://elixir-lang.org/.

[31] S. Juric, *Elixir in action*. Manning Publications Co., second edition ed., 2019.

[32] "Mix." https://hexdocs.pm/mix/Mix.html.

[33] "Phoenix framework." https://www.phoenixframework.org/.

[34] "Phoenix liveview." https://hexdocs.pm/phoenix_live_view/Phoenix.LiveView.html.

[35] "True_range_multilateration: Two cartesian dimensions, two measured slant ranges." https://en.wikipedia.org/wiki/True_range_multilateration#Two_Cartesian_dimensions,_two_measured_slant_ranges_(Trilateration.

[36] J. Armstrong, *Programming Erlang*. Pragmatic Bookshelf, second edition ed., 2013.

[37] "Who supervises the supervisors?." https://learnyousomeerlang.com/supervisors.

[38] J. Armstrong, *Programming Erlang*. Pragmatic Bookshelf, second edition ed., 2013.

[39] "Module supervisor." https://erlang.org/doc/man/supervisor.html.

[40] "Process hibernation." http://erlang.org/doc/man/erlang.html#hibernate-3.

[41] A. Carlier, I. Kopestenski, and D. Martens, *Lasp on Grisp : implementation and evaluation of a general purpose edge computing system for Internet of Things*. PhD thesis, UCL - Ecole polytechnique de Louvain, 2018.

[42] "Ipv4 multicast address space registry." https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml#multicast-addresses-3.

[43] "How to use multiple ultrasonic sensors." https://www.maxbotix.com/tutorials1/031-using-multiple-ultrasonic-sensors.htm.

[44] "Distributed applications." http://erlang.org/doc/design_principles/distributed_applications.html.

[45] "global." http://erlang.org/doc/man/global.html.

[46] "Distributed erlang." http://erlang.org/doc/reference_manual/distributed.html.

[47] "Controlling a maxsonar sensor." https://www.maxbotix.com/tutorials2/036-controlling-a-maxsonar-sensor.htm.

[48] Wikipedia contributors, "Round-trip delay — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Round-trip_delay&oldid=971345861, 2020. [Online; accessed 7-August-2020].

[49] Wikipedia contributors, "End-to-end delay — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=End-to-end_delay&oldid=926525261, 2019. [Online; accessed 7-August-2020].

[50] "Pmod nav." https://reference.digilentinc.com/reference/pmod/pmodnav/start.

[51] "Rebar3 plugins upgrade." https://www.rebar3.org/docs/using-available-plugins#upgrading-plugins. Library Catalog: www.rebar3.org.