University of Magdeburg

School of Computer Science

Digital Engineering Project

Mixed-reality Simulation of Quadcopter-Swarms

Author:

Lukas Mäurer

Vladimir Velinov

Dezember 15, 2015

Advisors:

Prof. Sanaz Mostaghim

Department of Intelligent Systems

Christoph Steup

Department of Intelligent Systems

Mäurer, Lukas and Velinov, Vladimir:
Mixed-reality Simulation of Quadcopter-Swarms
Digital Engineering Project, University of Magdeburg, 2015.

Contents

List of Figures

List of Tables

List of Code Listings

List of Acronyms

AHRS	Attitude Heading Reference System
GUI	Graphical User Interface
IMU	Inertial Measurement Unit
PID	Proportional-Integral-Derivative
UAV	Unmanned Air Vehicle

1. Introduction

1.1 Motivation

The work presented in this document was carried out at the Swarm Lab at the Otto-von-Guericke-University Magdeburg. The research focus of the working group lies on implementing and investing swarm algorithms in practice by using small indoor quadcopter. The used FINken quadcopter were developed in association with the working group and are small, but powerful and are highly extensible. As the research focus on swarm intelligence puts an interest on autonomous behaviour, the copter fly without any external reference and rely solely on onboard sensors.

The copter were developed in the working group and were designed with a focus on modularity, they are under constant change. Therefore, testing new changes of the copter, e.g. new control or behavioural algorithms alway poses a certain risk to the hardware due to crashes caused by bugs. Thus, a simulation tool for the quadcopter to test new software is desirable to be able to do a safe first evaluation of newly implemented ideas. A less abstract solution than a pure simulation could be a mixed reality simulation, where the behaviour of real and simulated quadcopter could be directly compared.

The idea is to build a mixed reality simulation environment where one or multiple real copter can fly together with one or multiple virtual ones. This would provide a testing possibility for new behaviour and enhancements like inter-copter communication models as well as making upscaling of swarms more easy. Simulated quadcopter can be added arbitrarily (enough computation power assumed) without increasing cost and damaging risk as with additional real quadcopter.

In contrast to existing approaches for the use of mixed reality simulation as in [CMW11], our focus lies not on hardware development. Instead, we focus on increasing situation complexity by computation power instead of more cost intensive real hardware.

The authors of [HMS+15] describe a similar approach, even using similar tools. They as well use mixed reality to enhance quadcopter swarms. However, they're implementaion of swarms rely heavily on a precise external camera tracking system. This limits the application to situations were such a tracking system can be provided. Our goal is, to provide a setup that can operate independently of a tracking system. Still, if some kind of tracking system is available, it could be used to improve the simulation.

Burgbacher, Steinicke and Hinrichs sketch a possibility to use mixed-reality simulations for the development in real world multi-robot projects in [BSH11]. They describe, how a simulation can be enriched with real elements when the hardware successively becomes available during the progress of the project.

1.2 Problem Statement

do The goal of the project is to provide a realistic, fast and scalable simulation of the FINken quadcopter. Simulated quadcopters should be connectable to a real flying FINkens, receiving it's Inertial Measurement Unit (IMU) data and behave like it's physical counterpart. The communication should work in both ways, so that the real FINken can react to simulated objects.

The physical quadcopter simulation is going to be done in the robotic simulation framework V-REP[CR]. The FINken Copter run the Paparazzi[ppr] software, which already includes a communication link between a PC and the copter. Paparazzi uses an Ivy-Bus as a communication link, so the missing part between the Ivy-Bus of Paparazzi and V-REP will be handled by a dedicated Java program.

During this project, the FINken needs to be modelled in V-REP, a communication between V-REP and Paparazzi needs to be established and the FINken firmware needs to be extended with a possibility to send data from the simulation to the real Quadcopter.

1.3 Outline

After we stated the motivation and the actual task in Section 1.1 and Section 1.2, we will explain the theoretical foundation of the work carried out later.

This theoretical part in Chapter 2 consists of the physical model of the FINken in Section 2.1 and Section 2.1.2, followed by a description of the used simulation environment in Section 2.2 and an explanation of the existing communication interfaces that the Java communication bridge will need to satisfy in Section 2.3.

In Chapter 3 we describe how we implemented the virtual FINken (Section 3.1.1) and the software structure behind it in Section 3.1.3. Next, we show the detailed structure and functionality of the Java communication application in Section 3.2. At the end of Chapter 3 in Section 3.3 we give an overview over what we needed to do on the real FINken to integrate it into the mixed reality simulation

The evaluation of our project is done in Chapter 4, beginning with the results of our first tests of the simulation and communication in Section 4.1. Subsequently we formulate the findings regarding performance of the simulation and communication link in

Section 4.2. We conclude the evaluation in Section 4.3 with the most important part, the accuracy of the FINken's position and movements in the simulation.

Finally, we sum up the achieved goals in Section 5.1 and finish with Section 5.2 with some ideas for future work based on this project .

2. Theory

2.1 FINken Modeling

2.1.1 Quadcopter modeling

For the simulation, at first, the basic physics behind a quadcopter have to be identified. As V-REP provides a physics engine, in our case bullet [bul], we will not build a complete physical model of the quadcopter in flight, but keep to what is necessary to simulate it in V-REP. A quadcopter is an aircraft with 4 rotors. In our simple case, the rotors are identical, mounted fix to the quadcopter body in the same xy-plane and have parallel thrust vectors pointing in the same direction. At start of the simulation, the inertial coordinate system and the copters body coordinate system has identical x, y, z-axis. However, when the copter moves, it's body coordinate system moves as well, keeping the copters center of mass at its origin, then denoted with x_b, y_b, z_b.

When the rotor i are powered, it turns with the angular velocity ω_i, creating a force in the direction of the rotor axis, which is equivalent to the quadcopter body axis z, and a torque τ_i around the rotor axis.

$$F_i = k\omega_i^2, \tau_i = d\omega_i^2 + I_M\dot{\omega}_i \tag{2.1}$$

The constant k depends on air density and rotor geometry. d is the drag constant for the rotor drive train and I_M is the moment of inertia of the rotor which adds a torque during angular acceleration. However, with the small diameters and lightweight plastic rotors, this contribution to the overall torque is comparatively small and can be omitted.

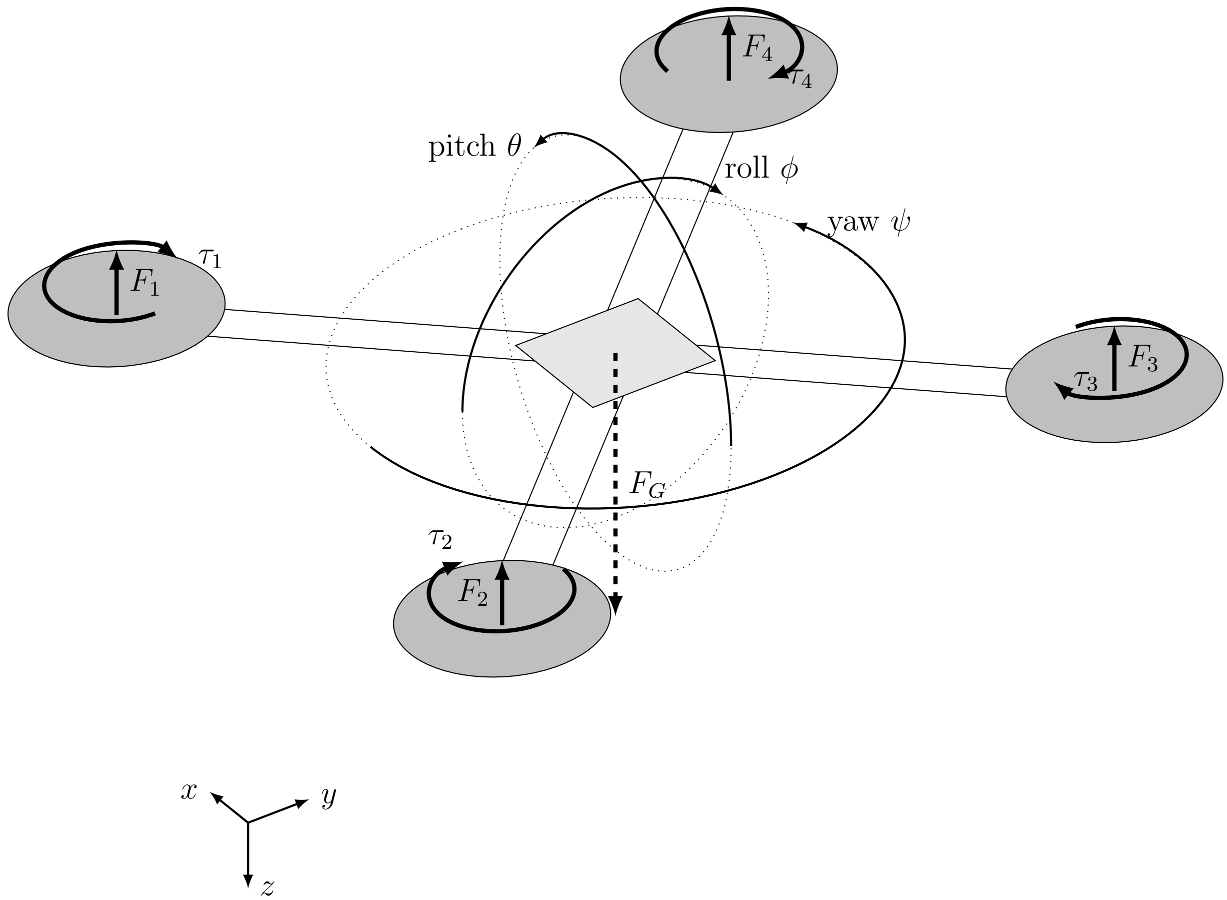

Figure 2.1: Forces and torques of a quadcopter

For the whole copter, we get the combined force F_{sum} with $F_{sum} = \sum_{i=1}^{4} F_i$ and the resulting thrust F_b relative to the body with $F_b = (0, 0, F_{sum})^T$. For the torque in body frame angles, the rotation direction of the rotor have to be taken into account.

$$\tau_b = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} cos(45)lk_{torque}(F_1 + F_2 - F_3 - F_4) \\ cos(45)lk_{torque}(-F_1 + F_2 + F_3 - F_4) \\ \sum_{i=1}^{4} \tau_i \end{bmatrix} \tag{2.2}$$

The rotor are mounted in distance l from the copters center of mass and the copter arms form a 45° angle to the x_b- and y_b-axis, resulting in a distance of $cos(45)l$ to the axis which their thrust creates a force around[Luu11].

To keep the copter in air, the forces generated by the thrust of the 4 rotors have to compensate the force F_G generated by the weight of the quadcopter.

$$F_G = F_1 + F_2 + F_3 + F_4 \tag{2.3}$$

Now, that the forces and torques on the coper are modeled, the model can be integrated into V-REP, as the physics engine will compute the according movements.

2.1.2 Rotor Modeling

Due to manufacturing tolerances and external influences as air stream, the forces F_i and torques τ_i generated at a certain angular velocity ω as described in Equation 2.1 is different for each rotor. In the previous section Section 2.1.1, we assumed, that all the rotors are identical. As this assumption doesn't hold, therefore a particle simulation was used to simulate the forces F_i and torques τ_i of the rotors. Using four particle objects with identical parameters, the model of section Equation 2.1 can be used, but the particle simulation adds some noise which makes the copter behavior more realistic. The particle simulation was already included in V-REP's example quadcopter model and was only slightly modified.

The particle simulation is used to simulate the airstream generated by the rotor. The particle object can be configured with particle size s_{px}, particle density ρ_{px} and maximum number of particles n_{px} it can hold. A simulation of rotors spinning in a particle cloud would take to much computation time, so the particles are generated below the rotor modeling the air stream. In the following the physics behind this simulation are shown, assuming that the copter is hovering, that the free stream velocity v_0 of the air around the quadcopter is zero and that the air is incompressible, which is valid as long as stream velocity are well below the speed of sound [Lau11]. Also, a homogeneous stream velocity under the whole rotor area is assumed which is sufficient accurate for this case.

Based on our assumptions, Momentum Theory gives us the thrust F_i of a single rotor as a product of the mass flow rate $\dot{m}$ and final speed v_{final} of the air accelerated by the rotor

$$F_i = \dot{m} v_{final} \tag{2.4}$$

This means, to simulate the thrust F_i with the particle object, the mass of particles and the final stream velocity is needed.

Neither the mass of the airstream nor the final stream velocity is easy to measure, but the thrust F_i when hovering is easily calculated from the weight of the copter.

$$F_i = \frac{F_G}{4}, F_G = m_{copter} * g \tag{2.5}$$

The mass flow rate $\dot{m}$, though not directly measurable, can be obtained from the air density ρ_{air}, and the volumetric flow rate $\dot{V}$ through the rotor as shown in Equation 2.6. The air density is constant (we assume standard conditions), and the volumetric flow rate depends on the area A covered by the rotor and the air velocity in the rotor plane v_{rotor}.

$$\dot{m} = \rho_{air} \dot{V} = \rho_{air} A v_{rotor} \tag{2.6}$$

Note, that the air stream velocity v_{rotor} in the rotor plane is different from the final air stream velocity v_{final} the air reaches behind the rotor. This can be shown, as by the conservation of energy, the power P the rotor puts into the air stream has to equal the energy E_{kin} the air stream carries per time as in Equation 2.9 and Equation 2.10. For the first derivative of the kinetic energy E_{kin} in Equation 2.8 note that the velocity is considered constant during hovering.

$$P = F_i v_{rotor} \tag{2.7}$$

$$E_{kin} = \frac{1}{2} m v_{final}^2, \dot{E}_{kin} = \dot{m} \frac{v_{final}^2}{2} \tag{2.8}$$

$$P = \dot{E}_{kin} \tag{2.9}$$

$$F_i v_{rotor} = \dot{m} \frac{v_{final}^2}{2} \tag{2.10}$$

Inserting Equation 2.4 into Equation 2.10 shows the relation between v_{rotor} and v_{final}.

$$\dot{m} v_{final} v_{rotor} = \dot{m} \frac{v_{final}^2}{2} \tag{2.11}$$

$$v_{rotor} = \frac{v_{final}}{2} \tag{2.12}$$

With the air velocity v_{rotor} in the rotor plane, the thrust F_i of a rotor can be calculated from the rotor area A and air density ρ_{air} which are known.

$$F_i = 2 \rho_{air} A v_{rotor}^2 \tag{2.13}$$

Equation 2.13 and Equation 2.12 together give a formula to determine the velocity v_{rotor} when the copter hovers.

$$v_{rotor} = \sqrt{\frac{F_i}{2 \rho_{air} A}} \tag{2.14}$$

As written in the introduction, the particle simulation includes the parameters particle density ρ_{px}, particle size s_{px} and rate $\dot{n}_{px}$, meaning how many particles are created per time. Particle density and particle size should be constant, as the air stream is considered incompressible, so when leaving poise, the particle rate has to change according to Equation 2.6 [Dee06].

The particles are spherical, with the particle size s_{px} as the sphere's diameter, so the mass m_{px} of a single particle can be calculated as in Equation 2.15.

$$m_{px} = V_{px} \rho_{px} = \frac{\pi}{6} s_{px}^3 \rho_{px}, V_{px} = \frac{\pi}{6} s_{px}^3 \tag{2.15}$$

The mass flow rate $\dot{m}$ of the particle is the product of particle rate $\dot{n}_{px}$ and the particle mass m_{px}.

$$\dot{m} = \dot{n}_{px} m_{px} = \dot{n}_{px} \frac{\pi}{6} s_{px}^3 \rho_{px} \tag{2.16}$$

During flight, if the air stream velocity v_{final} changes, the mass flow changes as well according to Equation 2.6, so the mass flow rate needs to be expressed as a function of air stream velocity.

Equating Equation 2.16 with Equation 2.6 in Equation 2.17 relates particle rate n_{px} to the already known parameters particle mass m_{px}, air density ρ_{air}, rotor area A and the final air stream velocity v_{final} in Equation 2.18.

$$\dot{n}_{px} m_{px} = \rho_{air} A v_{rotor} \tag{2.17}$$

$$\dot{n}_{px} = \frac{\rho_{air} A}{m_{px}} v_{rotor} = \frac{\rho_{air} A}{2 m_{px}} v_{final} \tag{2.18}$$

Now, the particle simulation can be parameterized based on the copter hovering. But, all parameters except for air stream speed are constant. Therefore, the copter's dynamics can be simulated by connecting the air stream velocity to the throttle, so the copter's thrust will be adjusted accordingly.

The thrust of the rotor can be expressed as Equation 2.19 by inserting Equation 2.16 into Equation 2.4

$$F_i = \dot{n}_{px} \frac{\pi}{6} s_{px}^3 \rho_{px} v_{final} \tag{2.19}$$

Particle size s_{px}, mass m_{px} and rate $\dot{n}_{px}$ can be arbitrarily chosen, as long as Equation 2.17 is satisfied. See Section 3.1.1 for the calculated values.

2.2 V-REP

V-REP is a versatile, highly customizable simulation environment, mainly developed for robots. It provides a rich set of functionalities which we use only a part off. We make use of it's integration of the bullet physics engine, including a particle simulation, it's external Java API, the communication structure via signals, the possibility of Lua scripting inside the simulation, it's provided sensor-simulation and the scene visualization.

The V-REP main client application provides the basis for the simulation. V-REP bases on scenes in which the simulation settings and scenes are saved. To run a simulation, the necessary objects need to be added to the scene. An object can be a dummy object with no physical properties, a 3-D shape either imported via an .stl-file or created directly in V-REP. V-REP already provides a library of objects, starting with sensors

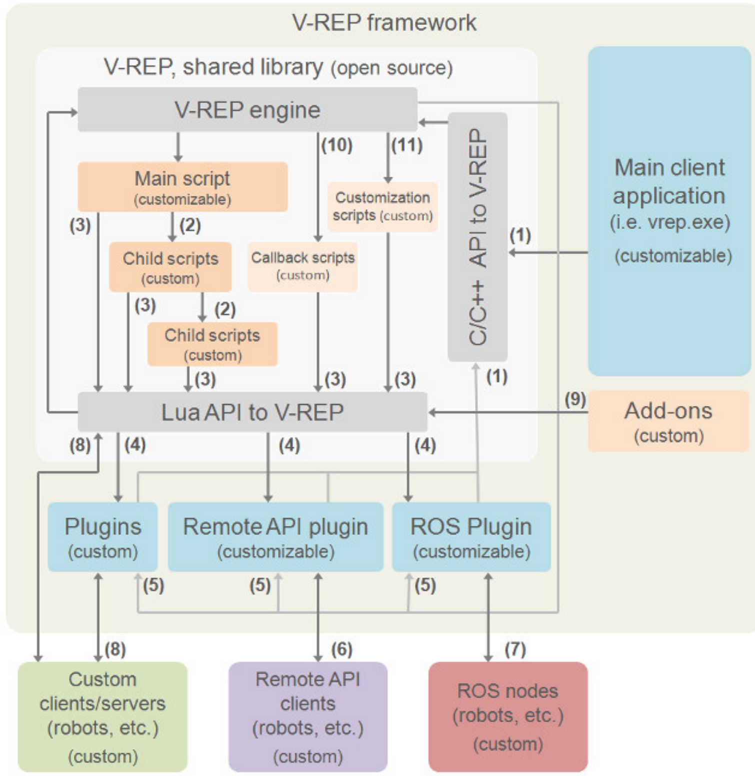

Figure 2.2: Structure of the V-REP-framework from Coppelia Robotics (13.10.2015)

and continuing with whole robots. An example are force sensors which can be used to connect object. Then, the force sensor transmits the resulting forces and torques during the simulation and can be set to break when exceeding a certain threshold. Connecting objects without force sensors can be used to separate physical simulation and visual representation. The connection becomes static, so a complex, but visually appealing mesh shape can be attached to a simpler shape with appropriate physical parameters. Thereby, physical simulation calculation are performantly done on the simple, and as V-REP provides several visibility layers, the user sees only the complex shape.

V-REP provides several possibilities to extend the simulation programmatically. The easiest and best integrated way are Lua-scripts. Lua scripts can be attached to simulation objects and are handles within the simulation environment. When choosing non-threaded child scripts, they get executed every simulation step and can influence scene objects via a comprehensive API. A V-REP simulation step consists of several sections. To execute parts of the script only at certain points, the current simulation status can be checked before executing code. The downside of these Lua-scripts is, that they are stored inside the binary scene file. Thus, it's difficult to put them under version control or do collaborative work. As the Lua-Engine inside V-REP provides full Lua support, this can be avoided by using the internal scripts only to import and call the actual scripts that are stored outside the scene. In section Section 3.1.3 is described how we handled this problem in detail.

If Lua doesn't offer the needed performance or functionalities, the second way to build deeply integrated software is the internal C++ API for plugins. The most flexible, but least performant way is the remote API of V-REP. Providing an API for Java, Matlab, Python and Urbi, it interacts with many programming languages. However, the functionality provided by the remote API is limited compared to the internal API for Lua or C++. For the remote API, V-REP needs to start a server that the client connects to. This is also possible over network, so the computation load can be distributed between different machines.

For communication inside V-REP, global custom variables can be used. As the access to those is not supported by the remote API, a more flexible way are *Signals*. *Signals* can be of String, Integer or Float type and are globally accessible in the current scene.

2.3 Communication V-REP-Quadrocopters

***Goal*:** our mixed reality simulation needs a dependable link of communication between the V-REP simulation environment and the flying quadrocopters. The Quadcopter needs to stream its telemetry data in real-time to the V-REP, and the reverse communication is needed as well.
The simulated quadrocopters that we have in the V-REP are divided into two categories: real and virtual representations.
The real are replicating the physical flying quadrocopters. They should perform the same flying maneuvers as those flying in the real environment. In order to make the model replicate this behavior, the flying quadrocopter must send its linear and angular velocity, its pitch yaw and row and other parameters in real time to its representation model in V-REP. The simulation model should also send the readings from its proximity sensors to the flying quadcopter thus providing it with information from a virtual sensor.
The virtual quadrocopters are purely simulation quadcopter objects, that exist only in the V-REP simulation environment. Their purpose is to interact with the real quadrocopters for example to avoid collisions and thus making the first steps in the swarm research. The virtual quadrocopters have to be seen in the Paparazzi ground station as if they are real physical quadrocopters. It means that all the ground station agents like message logger, the signal plotting, attitude indicator, artificial horizon and other displays have to be updated with adequate information coming from the virtual quadrocopters in the V-REP.

The communication between the V-REP quadrocopter models and the physical quadrocopters passes through several software components, which are depicted on figure Figure 2.3 and discussed in the next chapters.

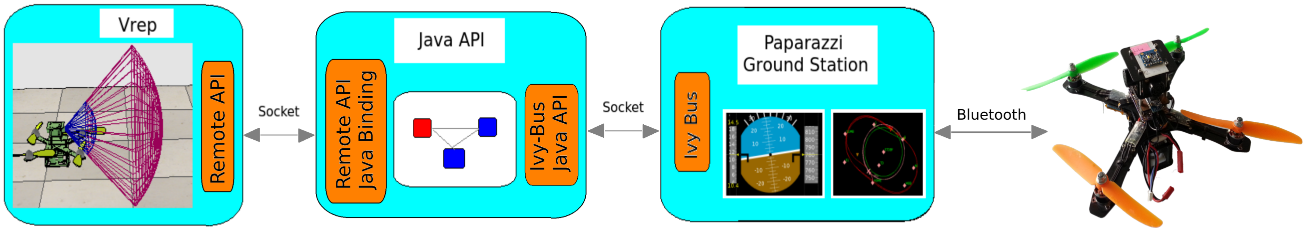

Figure 2.3: communication V-REP - Quadrocopter

2.3.1 V-REP Remote API

V-REP provides several means of communication with an external application. One of them is the Remote API, which allows to control a simulation (or the simulator itself) from an external application or a remote hardware (e.g. real robot, remote computer, etc.). The V-REP remote API is composed by approximately one hundred functions that can be called from a C/C++ application, a Python script, a Java application, a Matlab/Octave program, an Urbi script, or a Lua script. The remote API functions are interacting with V-REP via socket communication in a way that reduces lag and network load to a great extent.

2.3.2 Java API

Java API is the external program, that we have implemented to communicate with V-REP through the Remote API. We have chosen to implement our external program, communicating with the V-REP, in the Java programming language regarding the following advantages: Java's platform independence allows to run the external program even on different machine with different operating system than the one used for running the V-REP environment. Java is object-orientated which favors the use of design patterns and highly abstraction layers, which allows us to write an API that is modular, reusable and can later be easily extended to support other mixed-reality scenarios. Java also associates documentation with the actual code. The JavaDoc produces browsable documentation from the comments written in the code, which will be useful for anybody who wants to extend the project

The implementation and architecture of the Java API is discussed in details in Chapter 3. The purpose of the Java application is to serve as a communicating bridge between the Paparazzi Ground Station and the V-REP. It detects all quadrocopters in the V-REP simulation, builds their virtual representations and feeds the models with real-time data.

2.3.3 Ivy Bus

Ivy Bus is a simple protocol and a set of open-source (LGPL) libraries and programs that allows applications to broadcast information through text messages, with a subscription mechanism based on regular expressions. Ivy libraries are available in C, C++, Java, Python and Perl, on Windows and Unix boxes and on Macs.

The Paparazzi Ground Station uses the Ivy Bus as a means of communication between the different software components. Figure 2.4 depicts the communication structure in the Paparazzi Ground Station, in which the different agents communicate with each other by sending messages on the Ivy-Bus.

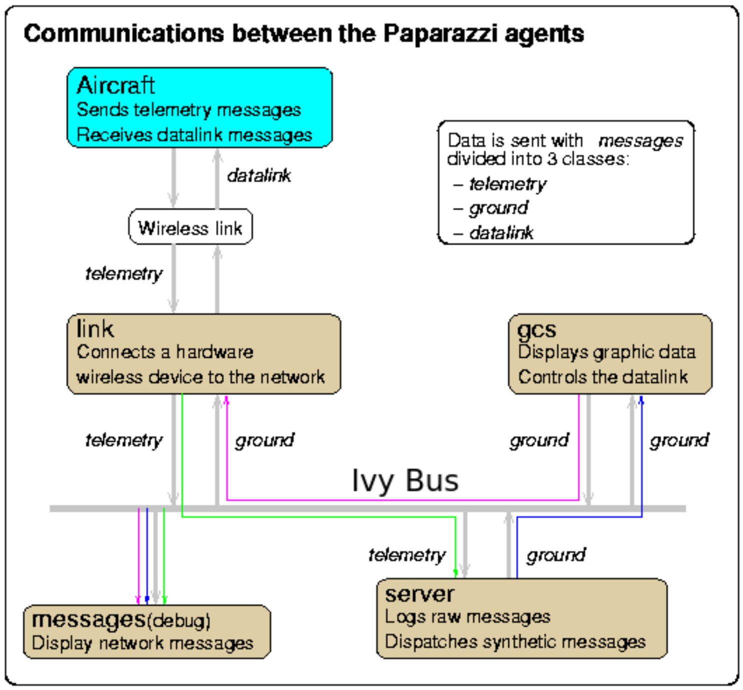

Figure 2.4: Ivy-Bus in Paparazzi Ground Station

The Unmanned Air Vehicle (UAV) (in blue) is streaming its telemetry data to the ground control station, which is received by the link. The **link** agent manages the ground-based radio modem and distributes the received messages to the other agents across the Ivy-Bus.

The Ivy Bus is an example of a publisher-subscriber protocol, in which senders of messages, called publishers, does not explicitly specify the address of the receiver, but just send the message on one, shared by all nodes, bus. The recipients, called subscribers, which are interested in the message will accept it and the others will ignore it. The publisher-subscriber is a many to many communication model in which publishers are loosely coupled to subscribers - there is no space, flow and time coupling. This means that the publishers does not have to know the addresses of the subscribers and even does not need to know of their existence. Each can operate normally without the other and can continue its thread of execution regardless if the subscriber has received the message or not. It also provides scalability, which means that we can “attach” our Java

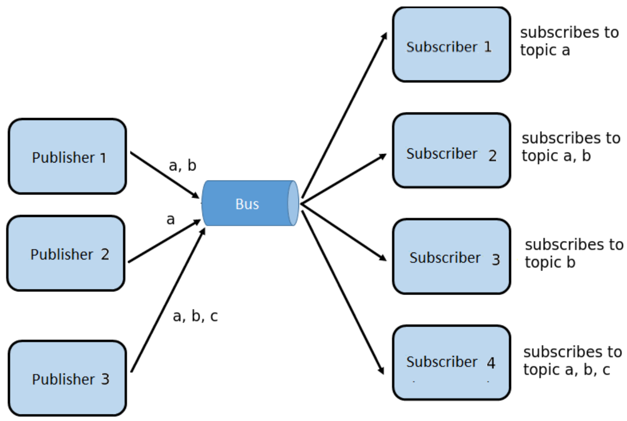

Figure 2.5: Topic-based publisher-subscriber

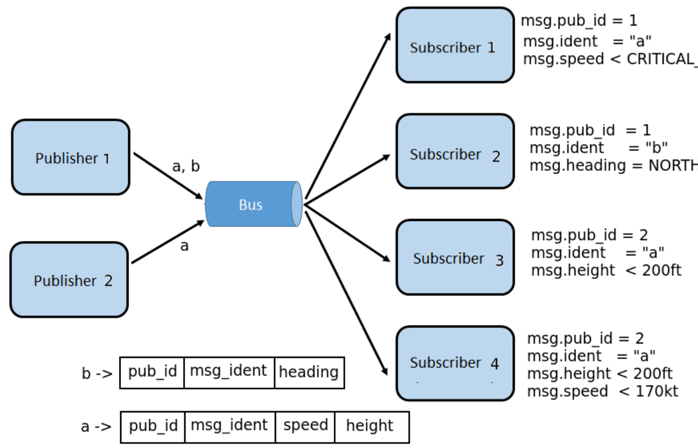

Figure 2.6: Content-based publisher-subscriber

API to the Ivy-Bus and start publishing and listening for messages without changing any line of source code in the Paparazzi Ground Station software.

In the publisher-subscriber model, subscribers typically receive only a subset of the total messages published. The process of selecting messages for reception and processing is called filtering. There are two common forms of filtering: topic-based and content-based. In a topic-based system, messages are published to "topics" or named logical channels. Subscribers in a topic-based system will receive all messages published to the topics to which they subscribe, and all subscribers to a topic will receive the same messages. The publisher is responsible for defining the classes of messages to which subscribers can subscribe. In a content-based system, messages are only delivered to a subscriber if the attributes or content of those messages match constraints defined by the subscriber. The subscriber is responsible for classifying the messages. Both filtering techniques are depicted in figures Figure 2.6 and Figure 2.5.

The Ivy Bus is a content-based publisher-subscriber and uses regular expressions for the message filtering.

2.3.4 Communication

The communication between V-REP and the quadrocopters passes through the Java API, which serves as a bridge between them. In fact the Java API does not communicate directly with the quadrocopter, but connects to the Ivy-Bus in the Paparazzi GS and can thus subscribe to the messages caring the telemetry data and also publish messages which will eventually be send to the quadrocopter by the link agent. On the other hand it uses the V-REP Remote-API to exchange information and provide the V-REP quadrocopter models with the data retrieved from the Ivy-Bus messages.
Since we wanted to create a modular and reusable API, that can be used for other mixed-reality scenarios, the Java API was created with the idea in mind to be distributed,

modular and rely on many abstraction layers.
The API development has followed the requirement-driven principles and its main tasks are described in the following paragraphs. The implementation of the described bellow requirements in described in Section 3.2 of Chapter 3.

2.3.4.1 V-REP-Remote API specific requirements

Below is a description of the requirements for the Java API which is responsible for the connection and communication with the V-REP environment. Its implementation is described in Section 3.2.2 of Chapter 3.

Connection to V-REP through the Remote API

There should be implemented a mechanism that enables to establish connection to the V-REP simulation. The connection should be able to be disconnected and reconnected at any time. Since the Remote API is based on a socket connection, it should be possible to connect to a V-REP server, situated on another machine, by specifying its IP address and port number. This will allow to run the V-REP simulation on a remote, powerful computer and thus release the local computer from having to deal with the simulation and communication program at the same time.

V-REP scene object and scene representation

The V-REP scene objects have to be represented as individual classes in order to be able to easily distinguish them and work with their properties. The class representing the scene object must have as fields the properties describing the simulation object - linear and angular velocity, position and orientation. The V-REP scene object hierarchy should also be implemented by using inheritance and composition object oriented techniques.
The Java API should provide methods to retrieve all scene objects from the V-REP scene and store them in a virtual scene representation for later use.
Since our quadrocopters are represented as shape objects and typically a V-REP scene contains at least 20 shape objects by default, there should be implemented a scanner that retrieves the quadrocopters from the scene. The scanner must be able to retrieve the virtual and real quadrocopters in a separate containers.

Continuous data exchange between quadrocopter instances and V-REP

As mentioned at the beginning of Section 2.3 we divide our quadrocopters in virtual and real representation. The virtual representations exist only in the V-REP scene, but they have to be visible in the paparazzi ground station as if they are real flying drones. It means that all the quadrocopter flying parameters have to be sent to the paparazzi ground station as messages. It becomes clear that the virtual quadrocopters have to be provided continuously with live data from the V-REP - linear/angular velocity, position and orientation. The real quadrocopter representations does not need to be provided

with V-REP parameters, since its velocity, orientation and height are coming from the physical flying drone, but it has to be provided with data from its proximity sensor scene objects. The update with V-REP data should have a frequency equal to the simulation step used - 50 ms by default.
The real quadrocopter representation get provided with velocity, orientation and height from the physical flying quadrocopter and this data have to be provided to the V-REP quadrocopter object in order to fly like the physical one. This is the inverse communication from quadrocopter to V-REP and is realized with V-REP signaling mechanism. The update is done each time a new message has been received by the physical drone and its frequency depends on how often the messages are streamed from the copter and the communication latency. In order to achieve a realistic flying maneuvers and have minimum drift the update should not exceed 20 ms.

2.3.4.2 Ivy Bus specific requirements

Ivy Bus connection

Each participant which wants to exchange information on the Ivy-Bus is described as a singular and independent bus node. In our program the participants that want to exchange data on the Ivy-Bus are the virtual and real representations of the quadrocopters. There should be designed a class, which allows the connecting on the bus, publishing and subscribing to messages. The quadrocopter representations have to inherit from it or composite it and thus become independent bus nodes.

Message retrieval and subscription

All the messages that the paparazzi software uses are described in a xml file called *messages.xml*, residing in the */paparazzi/conf/* directory. The following listing is an example of the messages file containing two messages.

Quelltext 2.1: Message Xml definition

```
<msg_class name="telemetry">
 <message name="AIRSPEED" id="54">
    <field name="airspeed" type="float" unit="m/s"/>
    <field name="airspeed_sp" type="float" unit="m/s"/>
    <field name="airspeed_cnt" type="float" unit="m/s"/>
    <field name="groundspeed_sp" type="float" unit="m/s"/>
  </message>

  <message name="SONAR_ARRAY" id="216">
        <field name="sonar_front" type="uint16" alt_unit="cm"/>
        <field name="sonar_right" type="uint16" alt_unit="cm"/>
        <field name="sonar_back" type="uint16" alt_unit="cm"/>
        <field name="sonar_left" type="uint16" alt_unit="cm"/>
  </message>

          . . . .

</msg_class>
```

In order to make the subscription and publishing of messages on the bus easier, each message should be represented as a corresponding class containing the message fields with its type and units.

There should be implemented a class, which reads the *messages.xml* file and creates an instance of the message class for each message.

Once the messages are retrieved, the Ivy-Bus node should be capable of subscribing and publishing of any of the retrieved messages. The subscription should by dynamic and allow to subscribe to a new messages even at runtime.
Once a message has been received, the Ivy-Bus node should provide the instance of the received message and its fields should contain the actual sensor values.
Since our mixed reality scenario can involve many flying quadrocopters, there will be published the same messages but from different copters. In this case the messages are identified with the drone id number. Another requirement is that the Ivy-Bus node have to subscribe just to the messages published from the copter that it was assigned to and not receive the same messages from the other copters.

2.3.4.3 Application specific requirements

Aircraft retrieval

The paparazzi software stores important information about the quadrocopters in an Xml file stored in */paparazzi/conf/conf.xml*

```
<conf>
  <aircraft
    name="Quad_Lia_ovgu_01"
    ac_id="1"
    airframe="airframes/ovgu/free_flight.xml"
    radio="radios/spektrum.xml"
    telemetry="telemetry/ovgu/tmp_christoph.xml"
    gui_color="white"
  />

  <aircraft
    name="Quad_Lia_ovgu_02"
    ac_id="2"
    airframe="airframes/ovgu/free_flight_flow.xml"
    radio="radios/spektrum.xml"
    telemetry="telemetry/ovgu/free_flight.xml"
    gui_color="#b34c14805f44"
  />

</conf>
```

Each quadrocopter is described as an aircraft, which has its name, id and other attributes. The parameter that is of specific importance for us is the id number, which is added to each message sent by the quadrocopter. The content based filtering of the messages is based on this aircraft id number. It is required to create a XML reader that parses the XML document and returns an instance of each aircraft class for each aircraft entry in the XML file.

Virtual and Real Quadrocopters representation

The virtual and real quadrocopters have to be described by an appropriate class.
Since they contain shared data and properties, an abstract parent class representing the quadrocopter will be an appropriate solution. The abstract quadrocopter should contain the aircraft it belongs to, so the name of the quadrocopter will coincide with the name of the aircraft. Since the id of the aircraft it represents is also contained, the quadrocopter will be able to send and receive the messages that only belong to the quadrocopter it represents.
Since the abstract quadrocopter represents also a V-REP scene object itself, it should

contain the *Shape* object it represents. Thus the orientation, rotation and velocity of the quadrocopter will be retrieved directly from the *Shape* object.
The proximity sensors of the quadrocopters should also be modeled and the abstract quadrocopter should provide methods to retrieve the distances measured by the sensors.
The abstract class should also contain an IvyBus node, discussed in Section 2.3.4.2, which will allow to connect/disconnect the quadrocopter from the IvyBus and to send and subscribe to messages filtered by the aircraft id it has.
In general the abstract quadrocopter should be a highly polymorphic class, being a scene object, aircraft and IvyBus node at the same time. It should have an interface allowing to connect/disconnect from the IvyBus, return object name, position, orientation, velocities and proximity sensor values.
The real and virtual quadrocopters have to be subclasses of the abstract one, implementing their specific behavior. For example the real quadrocopter subscribes to the messages coming from the quadrocopter it represents and on message arrival retrieves the relevant parameters like pitch, yaw, row, thrust and updates the V-REP model using the signals mechanism. The virtual quadrocopter on the other hand takes the parameters from the scene object, that is represents, packs them in messages with the id of the aircraft it represents and publishes them on the IvyBus.

Real and Virtual Quadrocopter retrieval

There should be implemented an automatic mechanism, that retrieved the instances of the real and virtual quadrocopters from the V-REP scene. The retrieval should be based on the scene objects retrieval and the aircraft parsing introduced in the previous paragraph.
After getting a list of all V-REP scene shape objects and all Aircrafts, it could be iterated in both lists and checked if any of the shape object's names matches the name of an aircraft. If the names match there should be created an instance of virtual or real quadrocopter with the corresponding Shape and Aircraft object. The decision if it is a real quadrocopter or a virtual one depends on the object name. The real quadrocopters have the standard FINKEN drone names like ***Quad_Lia_ovgu_01***, ***Quad_Lia_ovgu_02*** and the virtual have an arbitrary name with s prefix *Virtual*.

3. Implementation

3.1 Simulation Environment

The FINken consists of the body, the rotors and the sensors. The body contains the behavioural and flight control, the thrust simulation is handled by scripts connected to the rotor structure.

3.1.1 Scene Modeling

The body consists of multiple objects, while visual representation and the physical behaviour are split. For visual representation, a *.stl file from the CAD-Model of the original FINken was imported. This shape is rather complex, which makes the simulation time consuming. To speed up the simulation, the shape of the finken was remodelled in V-REP, using only simple rectangular shapes. Making the complex shape static, establishing a fixed connection and hiding the simple shape, the result is a a visual appealing simulation object with good simulation performance.

A dummy-object is used as a singular measurement reference point in the middle of the FINken. This has to be taken into account when targeting physical objects, as the FINken will not be able to reach the target completely without colliding with the object. This dummy object should always be used when refering to the FINkens position, orientation or movements, as it is aligned to the global coordinate system while especially imported shapes may be rotated in the simulation.

V-REP provides several pre-configured sensor type, the real FINkens sensors were modeled by using existing ultrasound distance sensors. The FINken is equipped with Maxbotix MB1232: I2CXL-MaxSonar-EZ3 sensors. According to the datasheet, they have an opening angle of 30° at maximum range when detecting smaller objects and maximum range of $2.65m$ for larger objects as walls. The datasheet states a rather complex beam shape, but for the simulation a cone with an opening angle of 30° and a height of 7.65m will be used, as the alternative sensor earlier had a range of 7.65m and in the simulation it is very easy to internally cap the values. Finally, a dummy target objects belongs the virtual FINken. This object can be moved manually in the scene and the simulated quadcopter will fly towards it. As the real FINken doesn't have the ability to fly to predefined points, we only used this feature for development.

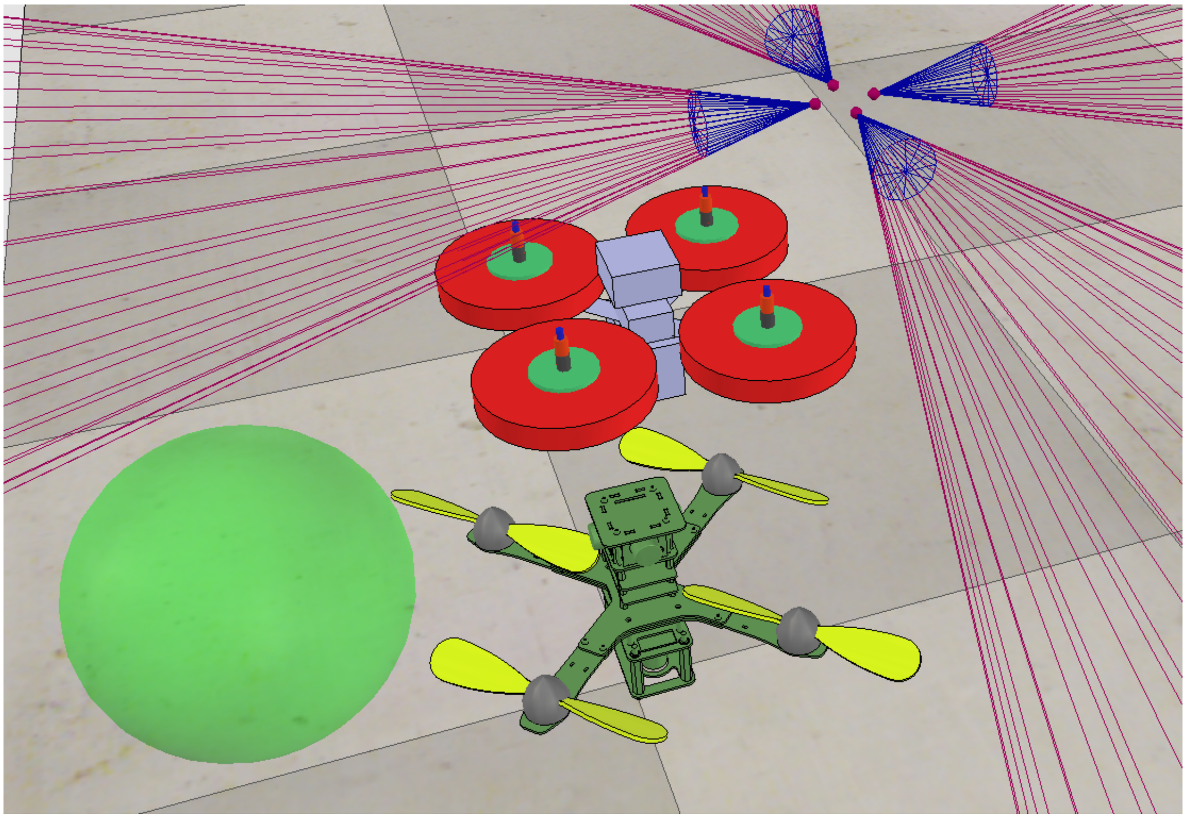

Figure 3.1: Parts of the simulation object

The measured weight of the real FINken was applied to the simulation object. As no advanced modelling like calculating moments of inertia was done, the weight of the motor was applied separately to the rotor model in V-REP. The motor assembly adds significant weight outside the FINkens center of gravity and therefore has a rather large influence on its behaviour.

Part	Weight FINken2	Weight FINken3	Weight in Simulation
Body	178g	214g	296
Motor & rotor	4x 15g	4x 25	4x 25
Battery	50g	82g	-

Table 3.1: FINken weights

The linear damping factor of the FINken body material was set to 0.3, to decrease drift and to model the air resistance of the real FINken, which is minimal but nevertheless existent.

Parameter	Value
Physics engine	Bullet
Dynamics settings	Accurate (default)
Simulation time step	50 ms (default)
Real-time mode	enabled

Table 3.2: V-REP simulation parameters

The rotor model in V-REP handles the thrust simulation and thus a huge part of the physical behaviour of the quadcopter model. Again, visual representation and physical simulation are separated.

The visualisation is done with a static shape of a rotor, that is connected to a joint and rotates with a fixed speed. During flight, the rotation speed does not really change visually noticeable, a dynamic adaption of the rotation would only increase computation time without much benefit. Of course, the rotor shape could be made non-static and rotate in a particle stream, and apply the thrust force according to the particle collisions, but again, this would mean a massive increase of computation power and is not needed for our purposes.

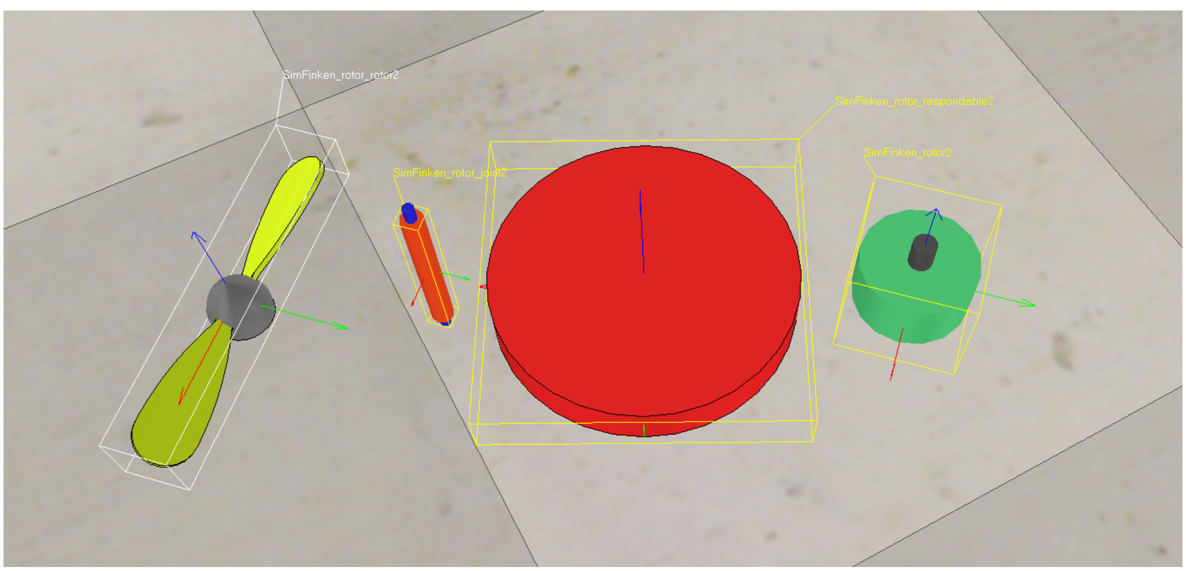

Figure 3.2: Parts of the simulated rotor

The rotor model is attached to the FINken body via a force sensor which applies the forces and torques calculated in the physical rotor model to the FINken body. The rotor is represented by a cylindrical shape which resembles the area swept by a real rotor and uses a particle simulation to emulate the airstream. The handling of the particle simulation is done inside a lua child script attached to the rotor model.

During simulation, the particle velocity is the parameter used to control the finken. This rotor model was already included in V-REP as an example and was used with slight modifications, the theory behind it is shortly explained in Section 2.1.2.

The particle velocity was calculated as in Equation 3.1 to be $11.077\frac{m}{s}$ under standard conditions.

$$v_{final} = 2 * \sqrt{\frac{F_i}{2\rho_{air}A}} = 2\sqrt{\frac{0.096kg * 9.81\frac{m}{s}}{2 * 1.2041\frac{kg}{m^3} * \frac{\pi}{4} * 0.1274^2 m^2}} = 11.0774\frac{m}{s} \tag{3.1}$$

The particle flow rate was initially set to $430\frac{1}{s}$. Our simulation step size is $50ms$, so this would mean 21.5 per simulation step. As shown in Table 3.3, the maximum step size with a particle object with a capacity of 50 particles would be $100ms$ and simulation times below $10ms$ might pose problems because of to few particles per step. The initial flow rote thus shows a good compromise between accuracy and computation time and was further used.

With a particle rate of $\dot{n} = 430\frac{1}{s}$ and the final velocity $v_{final} = 11.0774\frac{m}{s}$, the particle mass can be calculated as in Equation 3.2

$$m_{px} = \frac{\dot{m}}{\dot{n}} = \frac{\frac{F_i}{2v_{final}}}{\dot{n}} = \frac{\frac{0.096kg*9.81\frac{m}{s}}{2\sqrt{\frac{0.096kg*9.81\frac{m}{s}}{2*1.2041\frac{kg}{m^3}*\frac{\pi}{4}*0.1274^2m^2}}}}{430\frac{1}{s}} = 1.9771 * 10^{-4}kg \tag{3.2}$$

Continuing with a particle diameter of $5mm$, the particle density can be obtained as in Equation 3.3.

$$\rho_{px} = \frac{\dot{m}}{\dot{n}V_{px}} = \frac{\frac{0.096kg*9.81\frac{m}{s}}{2\sqrt{\frac{0.096kg*9.81\frac{m}{s}}{2*1.2041\frac{kg}{m^3}*\frac{\pi}{4}*0.1274^2m^2}}}}{430\frac{1}{s}\frac{pi}{6}0.005^3m^3} = 3020.8119\frac{kg}{m^3} \tag{3.3}$$

Using the particle simulation has the advantage of providing a random factor that causes noise and it simulates the airstream which will influence other copters in scenarios with multiple quadcopters.

The existing rotor model in V-REP did work, but contained minor inaccuracies. The first issue was the constant particle creation rate. As shown in Equation 2.6, the mass flow rate also depends on the air velocity. Thus, the air velocity has a quadratic influence on the thrust, when only a linear one was modeled. This was corrected as in Equation 3.4 based on Equation 2.18 that was derived in Section 2.1.2.

Simulation step size	Particles per step
5 ms	2.15
10 ms	4.3
25 ms	10.75
50 ms	21.5
100 ms	43
200 ms	86

Table 3.3: V-REP particles per step

Parameter	Original Value	new Value
Particle Size	$5mm$	$5mm$
Particle densitiy	$8500\frac{kg}{m^3}$	$3020.81\frac{kg}{m^3}$
Initial velocity	$4\frac{m}{s}$	$11\frac{m}{s}$
Particle count per second	$430\frac{1}{s}$	$430\frac{1}{s}$
Maximum number of particles	50	50
Particle life time	$0.5s$	$0.5s$

Table 3.4: Particle simulation configuration parameters

$$\dot{n}_{px} = \frac{\rho_{air} A}{2m_{px}} v_{final} = \frac{1.2041\frac{kg}{m^3}\frac{pi}{4}0.1274^2 m^2}{2 * 1.9771 * 10^{-4} kg} v_{final} = 38.8180\frac{1}{m} v_{final} \tag{3.4}$$

The torque on the quadcopter was only dependant on the air velocity as well, while Equation 2.2 shows it's linear dependance on the thrust. Thus, a torque factor of $k_{torque} = 200$ was introduced to relate torque to thrust. As the torque is applied on the rotor model, V-REP's physics engine handles the leverage of the rotor torque on the quadcopter body.

3.1.2 Flight controller in simulation

The V-REP example quadrocopter already included a position control with a subsidiary attitude control out of the box, both implemented with PID controllers. As the real FINken does not have an absolute position control, the control of the simulated FINken was changed to a pure attitude control. The external input therefore was changed from a position to pitch, roll, yaw and throttle, as it is the case for the real quadcopter. Thus, the controllers needed to be restructured and due to a different weight, the gain values had to be adjusted. The gain values were only manually tuned. The FINkens reaction to different control inputs were observed and the PID gain values were adjusted according to the behaviour. In a nutshell, the adjusments can be summed up as follows:

- K_P: Increase, if the reaction is to slow, decrease if overshooting
- K_I: Increase to reduce steady state errors, decrease if reaction is to slow

- K_D: Increase if reaction to changes is to slow, decrease if overshooting for changes

As previously mentioned in Section 3.1.1, the virtual FINkens position and orientation is received from a dummy Object in the middle of the simulation model, comparable to the Attitude Heading Reference System (AHRS) of the real FINken.

The throttle can be directly controlled, by feeding the command signal directly to the rotors. Then, only a factor is applied, to get the quadcopter to hover at 50% throttle. However, as no feedback loop exists, the copter will drift away. Because the real FINken does not compensate the battery voltage, the hover throttle will change during flight, making this type of control unsuitable for a mixed reality simulation.

As a solution, the height sensor of the real FINken can be used to set a target height. Then, a PID controller in the simulation can adjust the throttle accordingly. For this purpose, the position controller of the original simulation with only a proportional factor was reused. Opposite to the real FINken, the height controller of the virtual one includes a proportional factor for the vertical speed.

$$thrust = 22.175 * throttle/100 + 2 * e - 2 * v_{vertical} \tag{3.5}$$

The with only a proportional factor is symmetric, therefore the pitch and roll controller are identical. The rotation matrix of the IMU dummy object is compared against the target orientation. The resulting error is forwarded to the respective instance of the PID controller class, together with the current time step. The PID controller computes the correction, which is subsequently applied to the rotor according to Equation 2.2.

Parameter	Original Value	new Value
K_P	0.25	0.2
K_I	0	0.1
K_D	2.1	1.5

Table 3.5: PID gain values for pitch and roll

The yaw controller's gain parameters was estimated to the values in Table 3.6. The

Parameter	Original Value	new Value
K_P	0.25	0.2
K_I	0	0.1
K_D	2.1	1.5

Table 3.6: PID gain values for yaw

original yaw controller did not take into account, that the same orientation could be described by a positive and a negative angle. To address this issue, it has to be ensured that $-\pi < e_{yaw} < \pi$, as in the following Lua code.

```
local errorYaw=euler[3]-yawTarget*(math.pi/180)
if errorYaw < -math.pi then
        errorYaw = 2*math.pi+errorYaw
else if errorYaw > math.pi then
                errorYaw=yawTarget*(math.pi/180)-euler[3]
        end
end
```

To prevent the effect of windup of the integral part of the controller, the integral part is reset for all used controllers when the target is reached and the sign of the error changes.

3.1.3 Simulation Software Structure

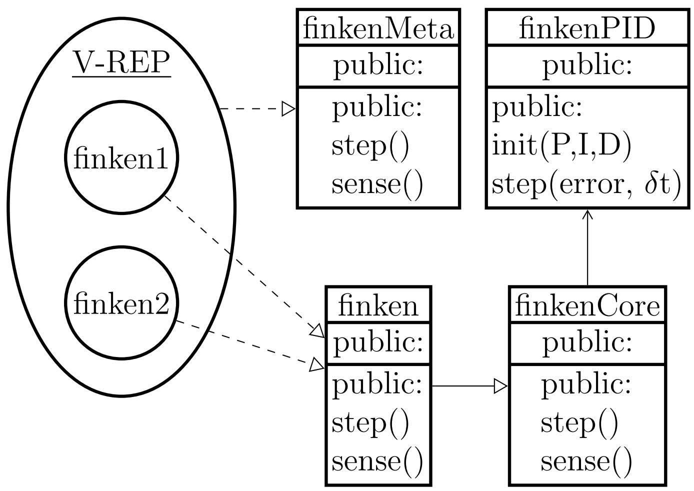

Figure 3.3: Software structure of the FINken Simulation

The simulation software is written in Lua and runs inside a non-threaded child script of each FINken quadrocopter in V-REP.

The base class is the finkenCore, which contains the main flight control algorithm and a steering interface as well as an interface to the FINkens sensors. The input commands for controlling the FINken are set via the signal communication of V-REP. Signals inside V-REP are the most versatile communication possibility which can be accessed from anywhere inside V-REP and any remote API, they can be seen as a kind of global variable. Signals can be of integer, float or string type, where custom data types can be sent in text form as string signals.

The finkenCore needs to be initialized at simulation start the for each simulated FINken. It starts the V-REP remote API server and creates the API signals. Also, the PID controllers for flight control are initialized. During simulation, calling the *FINken-Core.step()* method handles the flight control. The input values are read from the signals *pitch*, *roll*, *yaw* and *throttle*. When the signal *height* has a positive value, the

FINkenCore will control the thrust so that the simulated FINken will keep the height provided by the signal, otherwise only the *throttle* signal is used directly. The previously initialized PID controllers are used to compute the target air stream velocity for each rotor to move the simulated FINken to the target orientation.

To keep a quadcopter at a certain height without an external reference requires an exact equilibrium between gravity and thrust. The current real FINkens do not compensate the battery voltage, so the same throttle corresponds to different thrust forces during flight. Also, each FINken unit has slightly different base thrust. Therefore, the thrust in the simulation is tuned with a logisitc curve Equation 3.6 to decrease the influence of the throttle close to the hover thrust.

$$throttle_{tuned} = \begin{cases} -\frac{a*|throttle|}{a-|throttle|+50} + 50 & throttle < 0 \\ \frac{b*throttle}{b-throttle+50} + 50 & throttle >= 0 \end{cases} \tag{3.6}$$

The default values for the throttle tuning function are $a = 1$ and $b = 1$.

The simulated FINken is equipped with 4 distance sensors which resemble the 4 ultrasound sensors of the real FINken. The FINkenCore module contains a *sense()* function that reads the 4 distance sensors and writes the values to the signal *sensor_dist*. As signals only support limited data types, *sensor_dist* is a string signal containing a packed float array. The array contains the distances in the order *front*, *left*, *back*, *right*. If the FINken gets more sensors in the future, their evaluation has to be added to this method. By writing the distance values to a signal, they can be propagated to the Ivy-Bus and thus to the real FINken by our Java communication app, which enables the real FINken to detect virtual objects, as simulated FINken or walls that only exist in the Simulation.

While the FINken is normally controlled by setting the pitch, roll, yaw and throttle directly, an alternative is to specify a target object. This method was originally used by the example quadcopter model of V-REP, though the function works differently in the FINkenCore now. When calling *setTarget(targetObject)*, the differences in x, y and z coordinates of the FINken base to the target objects are given to three PID controllers which calculate the pitch, roll and throttle to move the FINken to the target objects position. Those values are then published via the control signals for the FINken and regularly processed by it's internal controls. Therefore, *setTarget(tragetObject)* needs to be called every time before *step()*, when a target object should be approached.

In the previous sections, only the signal names for the first simulated FINken were used for better readability. As the simulation is scalable, more than one FINken can be added, therefore a naming scheme for signals is needed to prevent collisions of the globally visible signals. The signals corresponding to the first FINken are *pitch*, *roll*, *yaw*, *throttle*, *height* and *sensor_dist*. For the second FINken, when the first one is copied, V-REP automatically adds "#0" to it's name. Following this convention, but leaving the '#' to prevent problems with special characters when forwarding the signals through our interface, we add '0' to the signal names for the second FINken and consecutively enumerate following FINkens.

The finken.lua module provides an basic structure for further enhancements. It's contained functions like *step()* and *sense()* are called called inside V-REP during the appropriate simulation step sections. By calling custom functions inside those functions, the behaviour of the FINkens can be customised without any need to change the lua scripts inside V-REP. If a heterogenous swarm of FINkens with different functionalities should be implemented, the different finken-scripts have to be imported by the different FINkens in the simulation, which needs to be done in the lua scripts inside V-REP.

FinkenMeta.lua is loaded in the child script of a dummy object in the scene to have an API to the simulation that is not connected to a single FINken. This can be used to dynamically manipulate the environment during simulation, e.g. by adding objects like other FINkens.

3.2 Communication V-REP - Quadrocopters

This chapter describes the implementation of the requirements on the Java-API, which ware discussed in Section 2.3.4 of Chapter 2. It begins with an overview of the software architecture and continues with the explanation of the created projects, classes and their use. It helps understanding how the communication between the V-REP and the quadrocopters is implemented and how to use the API or extend it in order to implement other mixed-reality scenarios.
Note that this is just a brief explanation of the Java-API implementation. If you want to go in details refer to the Javadoc which is also provided as an attachment to this paper.

3.2.1 Software architecture

The software architecture of the Java-API, which serves as a communication bridge between the Paparazzi software and the V-REP simulation, is designed to be as modular as possible in order to facilitate the further development of the project. Its reusable components should also serve as a building blocks for the students that want to develop future mixed-reality projects.

On Figure 3.4 is depicted an raw overview of the Java-API software architecture. The final program is assembled from independently developed components. Each part is independent and provide well-defined exported interfaces so that the other parts can use them. The first layer contains the projects *JavaV-REP*, *JavaIvyBus* and *JavaXmlSax*. The *JavaV-REP* is the implementation of the requirements concerning V-REP, that ware discussed in Section 2.3.4.1 of Chapter 2. At the heart of this project is the V-REP Remote-API binding for Java provided by Coppelia. The *JavaVrep* project extends this library and provides further utility methods for establishing connection to remote V-REP servers as well as retrieving and manipulating scene objects. The *JavaV-REP* project is described in more details in Section 3.2.2.

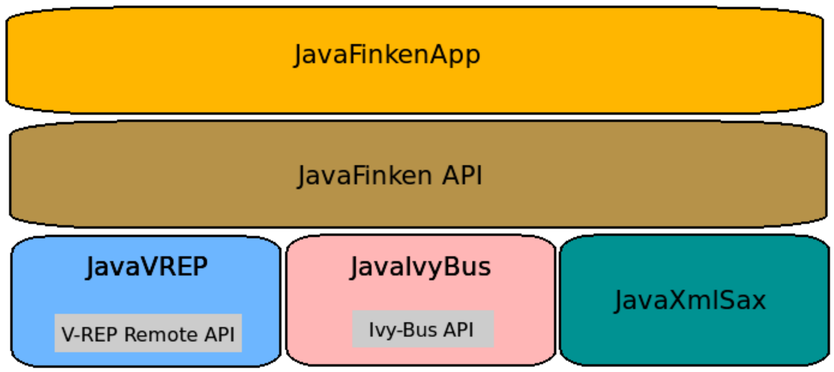

Figure 3.4: Java-API architecture

JavaIvyBus is a Java project, that imports the Ivy-Bus library and implements the requirements regarding Ivy-Bus, which ware discussed in Section 2.3.4.2.
It extends the functionality of the the Ivy-Bus library and provides the programmer with the possibility to create Ivy-Bus nodes by just instantiating an object which takes as constructor parameter the name of the bus-node. It facilitates the connection or disconnection from the bus by just calling a function the the created bus-node object. The abstractions on which this project rely also allow to easily create bus messages or just parse them from xml file and subscribe to them even dynamically. The *JavaIvyBus* project is discussed in more detail in Section 3.2.3.

The *JavaXmlSax* project is a small API, that allows the programmer to easily create a custom xml reader that parses any xml document and retrieves the instances of the objects defined by the xml. It consists of several abstract classes, that provide the template for the custom xml readers.
A short explanation of how this API can be used is included in Section 3.2.4.

On the next layer in the hierarchy is the *JavaFinken* API. It uses the APIs from the layer below to provide further abstractions and utilities for our FINKEN project. It consists of classes that describe the aircrafts defined in the paparazzi, defines the basic classes that represent our virtual and real quadrocopters and their sensors, defines a representation of the telemetry and the V-REP signals. With the help of *JavaIvy-*

Bus API, lying on the layer below, the Ivy-Bus nodes, specific for the virtual and real quadrocopters are represented. The abstraction provided by *JavaXmlSax* is used to create a custom Xml readers for parsing the telemetry, messages and aircrafts from the Xml files. A more detailed description of the API is included in Section 3.2.5

On the top of the hierarchy is situated the actual application of our project called *JavaFinkenAPP*. It uses *JavaFinken* API to bind the provided utility classes in a specific application meeting our project requirements. The application has a simple GUI that facilitates specifying the IP and Port number of the V-REP simulation server, specifying path to the paparazzi software, a button for establishing the connection and other UI elements.

3.2.2 Java V-REP

All the functionality, that concerns V-REP and was discussed in Section 2.3.4.1 of Chapter 2 is implemented as a single Java project called *JavaVREP*.
The project uses the V-REP Remote-API Java binding - package *coppelia* (containing 12 Java classes) and the *libremoteApiJava.so* or *libremoteApiJava.dll* (depending if the platform is Linux or Windows). The *libremoteApiJava.so* should be placed in the Java home directory, e.g */usr/lib/jvm/java-8-oracle/jre/lib/amd64*, in order for the project to be compiled.
The main class is the *VrepConnection.java*, which is a wrapper of the *remoteApi.java* class provided by the V-REP remote API. Its singleton instance can be retrieved by calling:

```
VrepConnection connection = VrepConnectionUtils.getConnection();
```

The above expression loads the remote API library and returns the instance of *VrepConnection* on which the Remote API functions are called. For example to retrieve all objects in a scene the following function have to be called on the *VrepConnection* instance:

```
connection.simxGetObjects();
```

The interfaces *VrepServer* and *VrepClient* and their implementations *StandardVrepServer* and *StandardVrepClient* describe the two end-points of the communication. The *VrepServer* describes the IP address and the port number of the machine on which the V-REP is running. In order to connect to a V-REP server we have to create in instance of the *VrepServer* and open the client:

```java
VrepConnection connection;
VrepClient     client;
VrepServer     server;

connection = VrepConnectionUtils.getConnection();
client     = VrepClientUtils.getClient();
server     = new StandardVrepServer("127.0.0.1", "19999");

client.connectToServer(server);

if (!client.isConnected()) {
// error in connection
}
```

The above example shows how to connect to a V-REP server. The IP *127.0.0.1* specifies that the server is running on the same machine. The port number can be chosen arbitrary, but have to match on both client and server site.
In order to close the connection just the method *client.close()* has to be called.

The *VrepClient* conforms to the Java Beans specification and can thus fire events when the connection has been established or disconnected. Any class who is interested in catching these events asynchronously, for example GUI, will have to implement *PropertyChangeListener* and register. See https://docs.oracle.com/javase/tutorial/javabeans/writing/events.html for more information.

Each V-REP scene object is represented by the interface *VrepObject* and the *AbsVrepObject* represents an abstract scene object from which all types of object derive. The abstract scene object has private properties like *Position*, *Orientation*, *LinearVelocity* and *AngularVelocity*, which represent its inertial parameters taken from the V-REP. *VrepObjectType* is an Enum, that specifies the scene object type like Shape, Path, Proximity sensor etc. The name of the scene object is represented by the class *VrepObjectName*, which consists of a base name and an index. If an object is copy-pasted (multiple instances of an object), then each instance of the object receives the following name, according to V-REP naming scheme: ***base_name#index***. For example if we want to have three quadrocopters, their names in V-REP will be represented as follows: ***Quad_Lia_ovgu_01***, ***Quad_Lia_ovgu_02#0*** and ***Quad_Lia_ovgu_03#1***.

The V-REP scene is represented by the class *VrepScene*, which retrieves all objects and hold a collection of them for further use. Since we always have one V-REP scene the class is designed as a Singleton pattern. The following example shows how this class is used for loading the scene and retrieving all shape objects.

```
VrepScene    scene;
List<Shape> shapeObjects;

scene = VrepSceneUtils.getVrepScene();
scene.loadScene();
shapeObjects = scene.getAllShapeObjects();
```

The project *JavaVrep* also defines the interface *ObjectUpdator* and its abstract implementation *AbsObjectUpdator*, which is used to update the *VrepObject*s with real-time parameters from V-REP, like their *Position*, *Orientation*, *LinearVelocity* and *AngularVelovity*.

3.2.3 JavaIvyBus

The requirements regarding the Ivy-Bus, that ware stated in Section 2.3.4.2 of Chapter 2 are implemented in a stand-alone Java project called *JavaIvyBus*.
The project requires the *ivy-java.jar* library to be on its class path in order to be compiled.
The project contains class definitions of the Paparazzi messages that are defined in a Xml file. See listing 2.1 of Chapter 2. The interface *Message* and its abstract implementation *AbsMessage* describe such a message with its name, period at which the message is sent, identifier and *MessageField*s.
The interface *IvyBusNode* represents a single independent node communicating on the common bus. By inheriting from its abstract implementation *AbsIvyBusNode*, one can create a custom bus-node. The methods *IvyBusNode.connect()* and *IvyBusNode.disconnect()* are used to attach the particular node to the bus and disconnect it. The *IvyBusNode* also conforms to the Java Beans specification and and fires asynchronous notifications each time the node joins or leaves the bus. After obtaining an instance of a *Message*, parsed from the Xml file or a custom-created, the bus-node can subscribe to this *Message* by invoking the method *IvyBusNode.subscribeToMessage(Message msg)* and thus receive all the messages of this kind or use the method *IvyBusNode.subscribeToIdMessage(Message msg, int id)*, which subscribes to the messages published just by this quadrocopter ehich has the same id.
Once a *Message* to which the bus node has subscribed has been received, the bus node fires an notification and gives the instance of the received *Message*, with all of its *MessageField*s initialized with the actual message values. The following listing shows an example hot to subscribe to a message and get asynchronous notification when the message is received.

```java
class TestBusNode implements PropertyChangeListener {

  private IvyBusNode node;
  private Message    message;

  public TestBusNode(IvyBusNode node, Message msg) {
    this.node    = node;
    this.message = msg;

    this.node.addPropertyChangeListener(this);
    this.node.subscribeToMessage(this.message);
  }

  @override
  public void propertyChange(PropertyChangeEvent event) {
    Message receivedMessage;

    // the message has been received

    receivedMessage = (Message)event.getNewValue();
  }

}
```

3.2.4 JavaXmlSax

The project *JavaXmlSax* was created with the idea in mind to provide a small, modular API, that gives the developer an abstract building block for fast and easy development of custom XML file readers. It uses the SAX Java API and extends it in order to provide a template for fast creation of specific XML readers.
The basic class in this module is the abstract class *AbsSaxXmlReader*. It encapsulates all necessary classes needed for creating and handling of the XML parsing and defines abstract methods, which allow its subclasses to provide their specific parsing criteria. Thus the developer can concentrate on the actual parsing logic and don't have to take care of setting up and managing the necessary input streams and files.

To create a specific XML reader we have to create a class, which extends the *AbsXmlReader* and provides implementation of the abstract methods *onStartElementRead* and *onStopElementRead*.These methods are called when the *AbsSaxXmlReader* encounters start or end XML tag. It is the role of the subclass to fetch the necessary attributes and create an instance of the XML element, when *onStartElementRead* is called and store the instance in some sort of collection, when the closing tag of the element is reached. In order to start the parsing process the method *parseXmlDocument* of the parent class *AbsSaxXmlReader* has to be invoked. A XML parser reads the whole XML file and returns the instances of all elements. In some case not all elements may be required and to save time and memory it would be preferred to stop the parsing after an element of interest has been found. Since the original Java SAX API does not provide the possibility to stop the parser after it has been started, we have implemented an mechanism for this. In order to stop the parsing, the method *stopParsing()* has to be invoked. The function throws a custom defined Exception - *StopParsingException*, that is handled in such a way, that the method *parseXmlDocument()* returns immediately.

The code below shows an example of how the XML parser can be used.

```
MessageXmlReader  msgReader;
List<Message>     messages;

msgReader = new MessageXmlReader("/home/paparazzi/conf/messages.xml");

msgReader.parseXmlDocument();

messages  = msgReader.getMessages();
```

The *MessageXmlReader* is a class that extends the *AbsXmlSaxReader* and provides implementation of its abstract methods. It returns all instances of messages contained in the */home/paparazzi/conf/messages.xml* file. The need to retrieve all messages was discussed in Section 2.3.4.2.

The *MessageXmlReader* gives us a significant flexibility. Since it returns a list containing the instances of all messages, we can choose a messages of interest and subscribe/unsubscribe even dynamically at run-time to them. This can be extremely useful if some quadrocopter needs some message just for a limited time, for example calibration messages, and then needs to unsubscribe from this message. Subscribing to less messages on the Ivy-Bus can save computation time.

Another example of XML reader, that extends the *AbsXmlSaxReader* is the *AircraftXmlReader* class. It is used to retrieve all aircrafts and its parameters from the */paparazzi/conf/conf.xml* file and has been discussed in Section 2.3.4.3.

3.2.5 JavaFinken

The *JavaFinken* API is a project that relies on the previously introduced projects *JavaV-REP*, *JavaIvyBus* and *JavaXmlSax*. It represents the implementation of the application requirements introduced in Section 2.3.4.3 and provides the basic classes and utilities upon which our application can be built.

The class of a special importance is the *AbsFinkenDrone*, which represents the abstract quadrocopter and is described by the interface *FinkenDrone*, which on itself extends

the *VrepObject* interface.
The virtual and real quadrocopters are represented by the classes *StandardRealFinkenDrone* and *StandardVirtualFinkenDrone*, which extend the abstract *AbsFinkenDrone*. The class *FinkenDroneScanner* is used to retrieve the instances of the quadrocopters and can be used as follows:

```
List<StandardRealFinkenDrone>    realDrones;
List<StandardVirtualFinkenDrone> virtualDrones;
FinkenDroneScanner               droneScanner;

droneScanner  = new FinkenDroneScanner();
realDrones    = droneScanner.retrieveRealDrones(this.scene, this.client);
virtualDrones = droneScanner.retrieveVirtualDrones(this.scene, this.client);
```

3.2.6 JavaFinken App

The final application, that has been created upon the described above modules: JavaVREP, JavaIvyBus, JavaXmlSax and JavaFinken is the *JavaFinkenApp*. The application has a small GUI, that allows to specify an IP address and Port number of the machine, where the V-REP simulation is executed and also the IP address of the PC, where the Paparazzi ground station is running. This allows to distribute the Simulation on different machines like shown in Figure 2.3. By default the the local machine IP Address *"1.0.0.127"* is chosen.

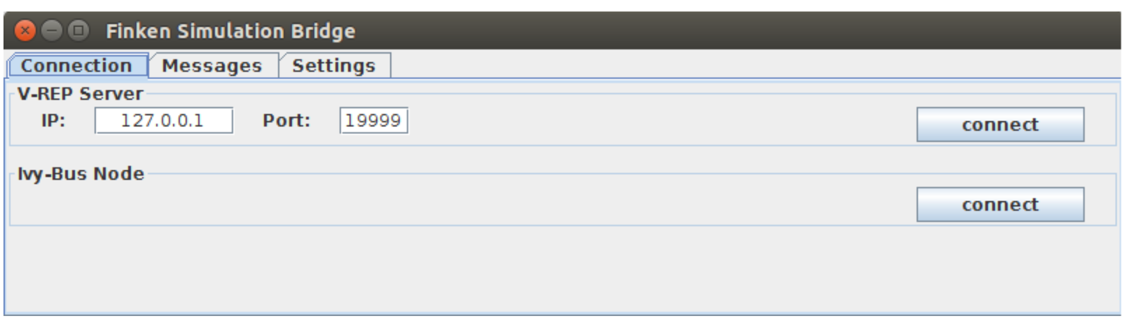

Figure 3.5: FinkenSimulationBridge GUI connection panel

On the above figure Figure 3.5 you can see the connection panel of the GUI, where the IP addresses have to be specified. The path to the *conf.xml* file, which stores the aircrafts and the path to the telemetry file have to be specified. The GUI provides separate tab for this. See figure Figure 3.6.

After choosing the path to the telemetry file, the telemetry modes are parsed from this file and are provided for choosing in the combo-box of the *Telemetry mode* sub-frame.

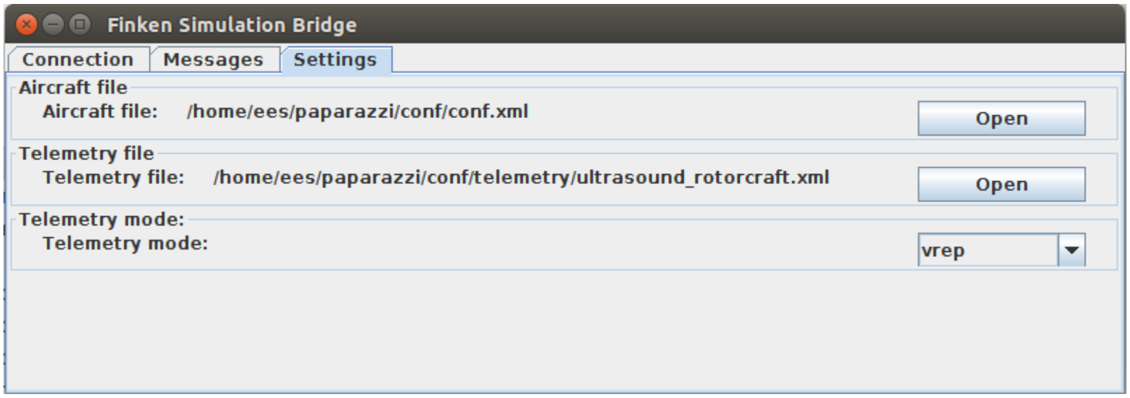

Figure 3.6: FinkenSimulationBridge GUI telemetry panel

Thus a custom telemetry mode, defining the messages that the quadrocopter needs to subscribe to, can be chosen.

The Model-View-Controller (MVC) design has been used to implement the *JavaFinkenApp*. The main class is the *FinkenSimBridge*, which contains the main method, where the program starts. The *FinkenSimBridgeView* defines the View - GUI components. And the *FinkenSimBridgeController* plays the controller role.

In order to start the communication, the "connect" button has to be clicked. The V-REP simulation must had been started in order to establish a connection. The listing below shows the method, executed by pressing the connect button.

```
void onConnect(VrepServer _server) {

  this.vrepClient.connectToServer(_server);

  if (this.vrepScene.isLoaded()) {
    return;
  }

  this.vrepScene.loadScene();

  this._retrieveVirtualDrones();

  this._retrieveRealDrones();

  this._initVirtualDrones();
  this._initRealDrones();

}
```

The function is triggered when the connect button is clicked and takes as input argument the *VrepServer*, which contains the IP Address and port number of the V-REP simulation. Then the *VrepScene* is loaded and all V-REP objects are retrieved. After the scene is loaded, the virtual and real drones are fetched from the scene and stored in a list structure. The function *initRealDrones()* makes the final initialization of the drones.

```
private void _initRealDrones() {
  for (RealFinkenDrone drone : this.realDrones) {
    drone.joinIvyBus();
    drone.loadTelemetryData(new File(TELEMETRY_FILE));
    drone.startPublish();
  }
}
```

First the drone is attached to the Ivy-Bus, then the telemetry data is loaded and the drone subscribes to the messages that reside in the specified telemetry-mode of the telemetry XML file. Eventually the the real drones start to get updated with the proximity-sensor data and to publish it as a message on the Ivy-Bus. The function *initVirtualDrones()* does the same initialization for all the virtual drones retrieved from the V-REP scene.

3.3 Quadcopter

This project was about to build a mixed reality simulation around the FINken quadcopter, so there weren't made many changes to the quadcopter and the firmware itself. A few changes were needed, regarding calibration, the message link and the communication from V-REP to the FINken. At the start of the project, the FINken II was used, but when the FINken III was released, it could directly be used as all our needed functions were compatible. In fact, the new telemetry link of the FINken III made a much faster communication possible, which helped some delay issues with V-REP as put in Section 3.3.2

3.3.1 FINken calibration

The pitch, yaw and roll angles of the quadrocopter are measured from the Inertial-Measurement-Unit (IMU). The IMU sensor may not be mounted perfectly level to the airframe due to construction issues, imperfect soldering of the sensor on the board or just a factory defect. Such issues will cause a big offset in the sensor readings and will be critical for our simulation. This was the case with the very first experiments. The quadrocopter in the V-REP simulation started to drift heavily from the real one at the very beginning of the simulation. As a result the quadrocopter was flying away from the simulated arena, although the real one was flying stable in the physical arena.
Fortunately, the Paparazzi project already provides a IMU calibration software, which we used to calibrate the accelerometer. The calibration is implemented as a python script, that has to be run on the log file containing the accelerometer calibration data. The calibration procedure was easy to perform and we had to follow the following procedure:

- Flash the board with our firmware
- Switch to the "raw sensors" telemetry mode via GCS->Settings->Telemetry and launch "server" to record a log
- Move the IMU into different positions to record relevant measurements for each axis.
- Stop the server so it will write the log file
- Run the Python script on it to get the calibration coefficients and add them to the airframe file

The listing below shows our airframe file - */paparazzi/conf/airframes/ovgu/6imucalib.xml* with the calibration coefficients resulting from the calibration.

```
<section name="IMU" prefix="IMU_">
  ...
  <define name="ACCEL_X_NEUTRAL" value="10"/>
  <define name="ACCEL_Y_NEUTRAL" value="12"/>
  <define name="ACCEL_Z_NEUTRAL" value="34"/>
  <define name="ACCEL_X_SENS" value="4.86375969658" integer="16"/>
  <define name="ACCEL_Y_SENS" value="4.86831208708" integer="16"/>
  <define name="ACCEL_Z_SENS" value="4.90201085317" integer="16"/>
  ...
</section>
```

When a new firmware is flashed to the quadrocopter, the paparazzi software also flashes the calibration coefficients from the airframe file so that IMU gets calibrated.
We observed a big improvements in the simulation performance after calibrating the IMU. Although the quadrocopter was still drifting to some extend, the strong drifts have disappeared.
The calibration software also provides calibration for the gyroscope sensor. Although the drift in the gyro sensor is also very critical for the simulation performance, we did not perform any calibration on the gyroscope. The reason was, that the calibration of the gyroscope is very sophisticated and required a moving platform for each sensor axis, which was not present at the time of the project. However we believe, that calibration of the gyroscope will improve the results to a significant extend and should be performed in the future work of the project.

3.3.2 FINken message link

In the beginning of this project, the second iteration of the FINken, the FINken II was used. There, the telemetry link was implemented with Bluetooth LE. It showed, that the data rate was barely sufficient when the quadcopter was next to the computer running the ground station and dropped heavily with more distance. The simulation runs with a time step of $50ms$, so it is desirable to get the telemetry message at the same rate to provide new data for each simulation step. This was not possible with the bluetooth link.

The FINken III that was introduced during this project features 802.15.4 based communication. This evolution has two major benefits. First, in allows to send telemetry data with every $21ms$, even multiple messages, so it could be ensured that at least on average, a new message would be available for each simulation time step, even if one package is dropped. Second, in the new communication, the quadcopter broadcasts it's data. This allows to have multiple quadcopter connected to the same groundstation and the groundstation receives data as soon as the FINken is switched on. The latter is a valuable improvement over the necessary linking for the bluetooth communication of the FINken II.

3.3.3 V-REP to FINken communication

The focus of the project lied on the communication from the real hardware to the simulation. However,to make virtual objects visible to the real FINken, they need to be send somehow from the virtual scene to the real world. The FINken relies on the ultrasound sensors to detect object, therefore especially the virtual distances to objects needs to be transmitted. Paparazzi provides a two way communication with the telemetry link. Thereby the existing solutions could be used to send the messages, as we as well implemented the communication structure bidirectional.

The modules of the FINken's firmware can subscribe to telemetry messages. Since the ultrasound sensors are handled in one module, it is easy to include a subscription to our virtual distance data into the module. At the moment, the distance sensor are only used for collision avoidance, so only the closer object is interesting. Therefore, the integration of the virtual values was done by using the minimum value of the real and virtual distance measurements. This ensures, that the real FINken will try to avoid any detected object, be it detected by its own sensors or by the sensors of its virtual counterpart.

4. Evaluation

4.1 Testing with Joystick

In order to evaluate, at an early stage, how flyable and responsive to external commands the V-REP Quadcopter was, and how reliable the communication link between the V-REP and the Ivy-Bus system is, a Hardware-in-the-loop (HIL) set-up was build. Figure 4.1 shows the HIL set-up, consisting of a joystick, which is attached on the Ivy-Bus and controls the V-REP Quadcopter.

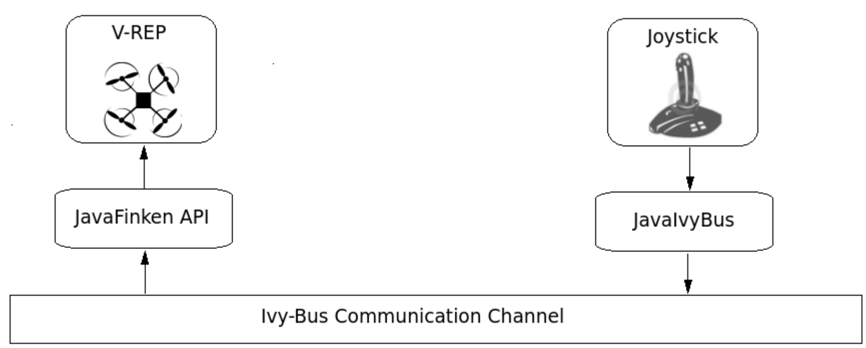

Figure 4.1: Finken Hardware-in-the-loop

The HIL evaluation was implemented as a separate project - *JavaFinkenSimHil*. It uses the Java external library JInput for reading the inputs from the joystick. The already created *JavaIvyBus* module, which was discussed in Section 3.2.3, is used to connect the Joystick to the Ivy-Bus.

The *Joystick* class has as a local variable the class *JoystickBusNode*, which extends the abstract *AbsIvyBusNode*, thus being able to communicate on the common Ivy-Bus network.

The *Joystick* class polls the data from the Joystick device at 50 MHz and sends them to the *JoystickBusNode*, which on the other hand encapsulates the data in a FINKEN_ROTORCRAFT_FP message and publishes it on the bus.

The FINKEN_ROTORCRAFT_FP is the message that the real Quadcopter uses to publish its pitch, yaw and roll angles. Since the Joystick fakes the real Quadcopter, the JavaFinkenApp receives the message as if coming from the real one, thus not even a single line of code had to be changed in the JavaFinkenApp, in order to use the HIL evaluation.

As a result of the evaluation we came to the following conclusions:

- The controlling of the Quadcopter was very precise and accurate. We ware able to make any maneuver and flight path as desired.

  When flying with the Joystick, one can get a real feeling of the Quadcopter flight dynamics and behavior.

- We experienced challenges with keeping the Quadcopter at a level height, using the throttle levers of the Joystick. It turned out, that keeping a level height was a matter of eye-hand coordination.

  Even the slightest change to the thrust when hovering resulted in a strong acceleration in z-axis. Even if this behavior corresponds to the real quadcopter, we identified it as a problem, as the hover thrust of the real quadcopter changes during flight time with the battery voltage. As a result, we decided to tune the throttle response of the virtual Quadcopter with a logistic curve as described in Equation 3.6.

4.2 Simulation and Messaging Performance

4.2.1 V-REP performance

The V-REP simulation is quite performant, running even on an old $2.4GHz$ Core 2 Duo (T8300, 4GB RAM) close to real time. On this machine, running OS X, the execution of the Lua scripts takes the most time with typically $32ms - 42ms$. The distance sensor handling takes $14ms - 18ms$ and the Bullet physics engine accounts for $10ms$⁻$12ms$. A modern computer even with a low-voltage i7 runs the simulation in real time without problems. A second computer, running Windows 10 with a i7-6650U CPU and 16GB RAM computes the Lua scripts in $26ms - 31ms$, the distance sensor handling in $6ms - 7ms$ and the Bullet physics in $6ms - 8ms$. These times are obtained by the V-REP internal profiler.

Profiling inside the Lua scripts showed, that the quadcopter main script, with the controllers and e.g. logging functions only takes 1-5 ms according to time measurements with *simGetSystemTimeInMs()*. The rotor scripts need $1ms - 5ms$ each, which might be optimizable. However, it seems that the biggest factor is the V-REP internal handling of Lua. Optimizing here might be possible, but requires detailed information about the internal structures.

4.2.2 Timings in the Communication link

The rate at which the V-REP simulated quadcopter receives the telemetry data from the flying quadcopter is crucial for the mixed reality simulation. The period at which a message is sent is configurable in the telemetry file, but there are several factors, that can influence the message frequency. The paparazzi ground station link module distributes the received telemetry on the Ivy-Bus and a slight delay could be expected there as a result of message buffering. The Ivy-Bus is a topic based publisher subscriber communication protocol and the processing of the regular expressions can be expensive. On the final link of chain is our Java communication bridge.

Considering the above factors, that can introduce a delay in the message transmission, we decided to measure how frequently the messages are sent to V-REP. Since V-REP does not provide any methods to measure the communication lag, we decided to measure how fast the Java communication bridge is sending the parameters to the V-REP. Figure 4.2 shows the results of the measurements with a single quadrocopter flying. The message is configured to be sent every 22 ms. What we see on the graph is that the average time at which the Java communication bridge sends the message to the V-REP quadcopter is around 27 ms. There are some messages that are sent even above 60 ms, but their number is relatively small and the delay scarcely exceeds the V-REP simulation step. A surprising fact is that some messages are sent even more frequently than the configured 22 ms. Since we could not find any explanation of this, we assumed that the paparazzi link uses some buffering strategy that leads to the observed fact.

In order to evaluate the performance of the Ivy-Bus when there are more agents, we did the same test with two flying quadrocopters. On Figure 4.3 you can see that the graph shows the same pattern as in Figure 4.2, but the average value is slightly increased.

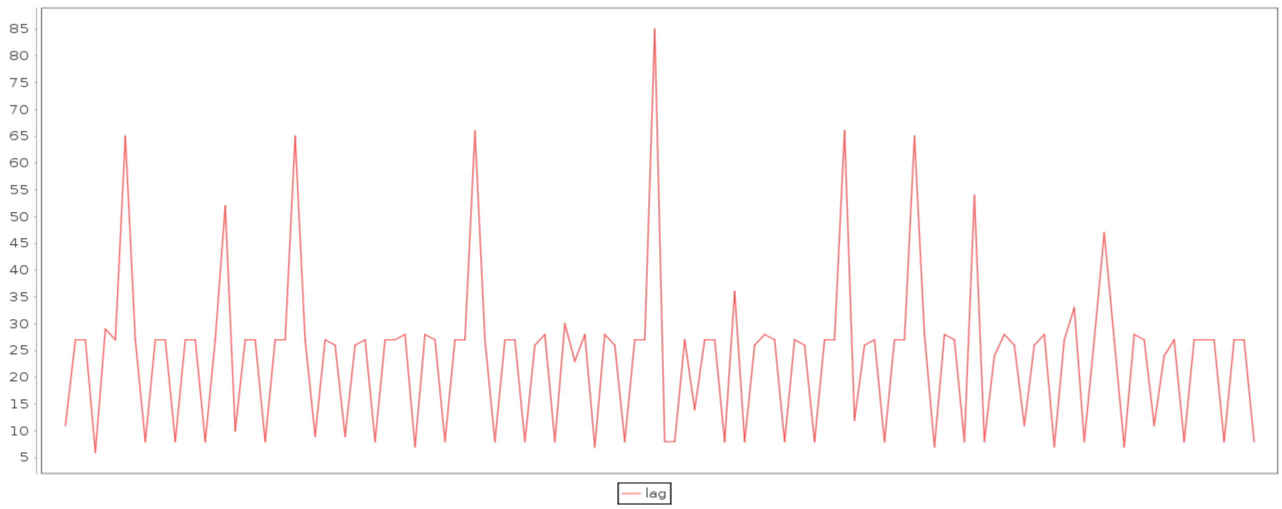

Figure 4.2: message lag with one quadrocopter

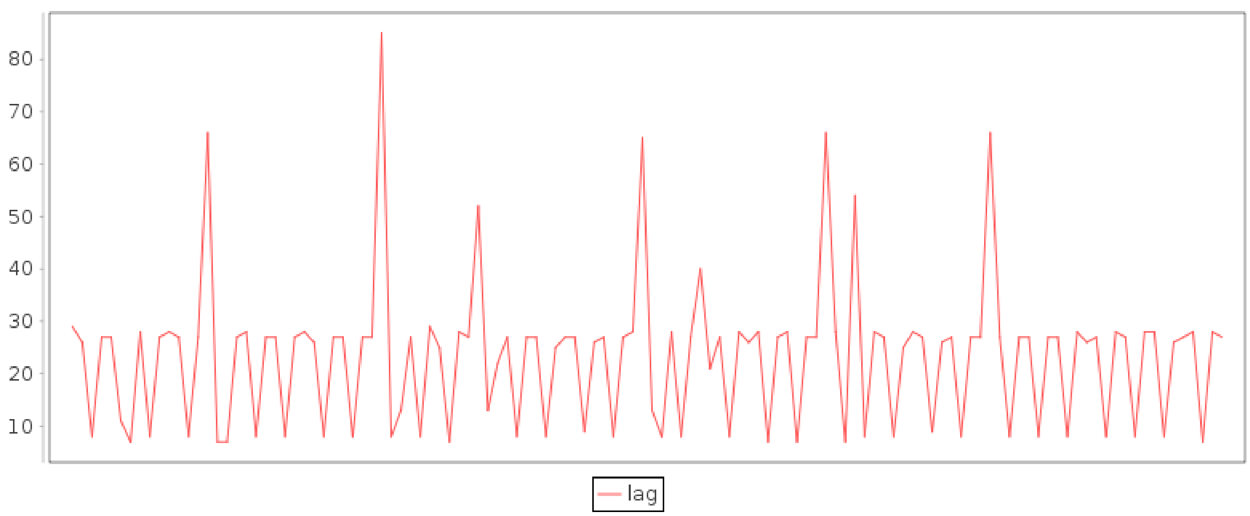

Figure 4.3: message lag with two quadrocopter

4.3 Accuracy

To evaluate the accuracy of the simulation, we compared the orientation of the real FINken and the simulated model. We let the real FINken fly freely in the arena using the wall avoid control and linked the simulated FINken. V-REP, paparazzi and the Java bridge were all run on the same machine. The test flight which is referred to in the following was filmed. The video, a screen recording of the simulation and the corresponding log files can be found online The real quadcopter has some issues with the yaw angle. A cause for this is that the magnetometer does not provide reliable values indoors, due to interferences of electro-magnetic fields. The sensors of the FINken are

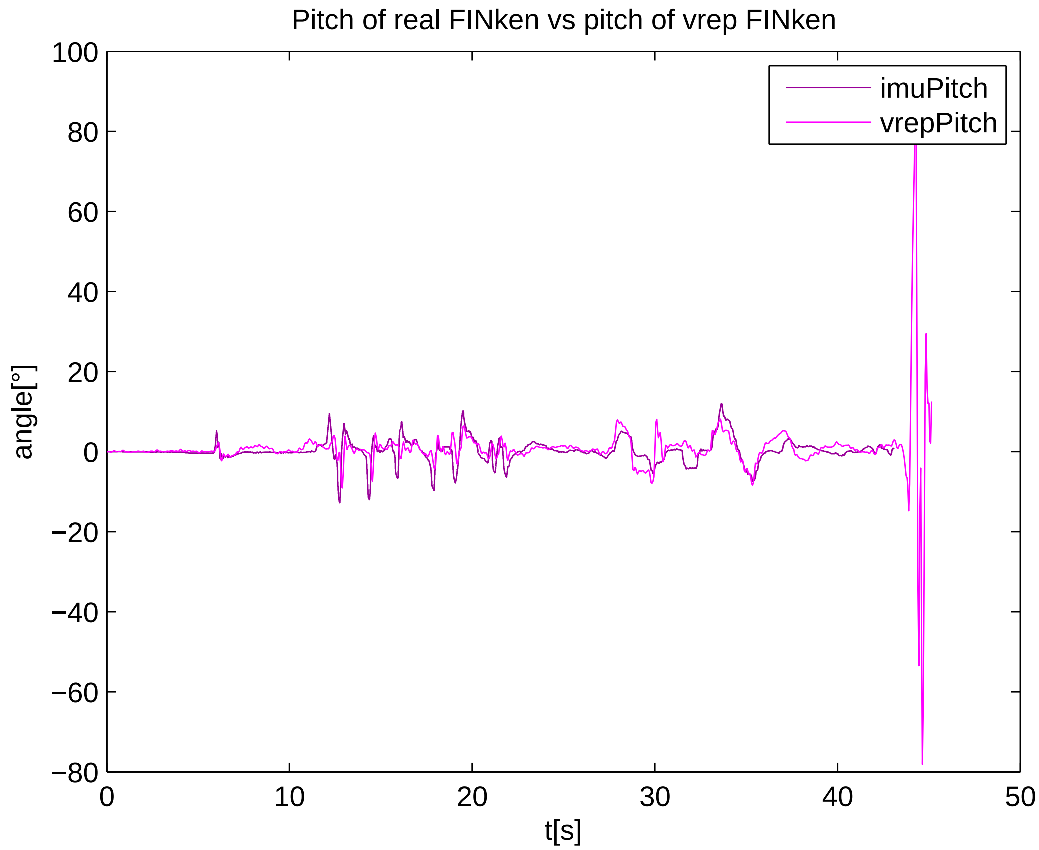

Figure 4.4: Pitch angles of simulated FINken and real FINken

symmetrical, so yaw is not needed for regular movements. Thus, it is tried to keep the copters orientation around the z axis more or less stable. The evaluation is described exemplary for one flight. Of course, this was not the only test that was run, but other experiments fortified the following results.

The pitch and roll angles of the simulated FINken are more interesting. When we logged the values via the telemetry link of paparazzi, and compared them to the values exported from the simulation, we noticed a time offset. This is not surprising, as we don't use an absolute reference time, but it made it impossible to directly compare the orientation logs. To get a meaningful graphic, we manually resampled the logs and put them on a common time base. Characteristic patterns were used to achieve a common time base. In Figure 4.4 can be seen that the simulated FINken nicely adopts the real FINken's movements. The first spike at about $7s$ shows the takeoff. From $12s$ to $22s$ it can be seen how the real FINken got close to a wall and started to avoid it. The flight was stopped after about $43s$, the huge spikes at the end of the graph show how the response when the copter fell to the ground.

The virtual copter started to drift away approximately at $30s$ as can be observed in the video. In Figure 4.4 can be seen, that the virtual and real pitch deviate around $30s$ for a short time, which could be the cause for the drift.

A more detailed plot of the pitch comparison is shown in Figure 4.5. Every spike of the real FINken's movement is directly followed by a spike in the same direction of the simulated FINken. The smaller movements do not correspond exactly, as the simulated copter contains its own attitude controller. Thus, keeping the simulated FINken stable in air has a higher priority than following the real copter's movements. The graph shows some points, e.g. before $17s$ where movement of the simulated FINken appears to precede the real FINken's movement. This can have two possible explanations. Firstly, as mentioned before, the time base was readjusted manually. Thus, it might be possible, that as explained in Section 4.2.2, the values were sent with a wrong timestamp because of paparazzi's internal buffering. If the wrong timestamps weren't corrected during the adjustment, or even made worse by the resampling ,they could shift the logged values. Secondly, the controller of the simulated FINken's controller has the same goal as the real FINken, namely to keep the copter stable. In the following, it could be possible, that both controllers are going to apply the same changes and that the virtual one is slightly faster.

Interestingly, the roll of the FINkens as shown in Figure 4.6 doesn't fit as well as the pitch, despite having identical controllers. In this flight, we observed some logging error in V-REP. The huge spikes in the virtual FINken's roll angle could not be observed in the screen capture of the simulation. When comparing the spikes in Figure 4.6 with the plot of the pitch in Figure 4.4, one can notice that the erroneous spikes in the roll correspond to the valid spikes in the pitch. An explanation could be, that the rotation matrix for the dummy object, which is linked to the simulated FINken's body, was not computed correctly. Unfortunately, we could not reproduce this error, but it shows that one should be careful when using sensor data, be it from hardware sensors or from software.

The plots for the angular responses of the simulation show, that it is possible to let a virtual FINken mimic a real flying one by transmitting the AHRS values. The response of the model does not exactly match, but to achieve this, the simulation model needs to be excessively tuned, which is out of scope of this work.

Eventually we wanted to evaluate to what extend the path of the flying quadrocopter matches the path of the simulated quadcopter. V-REP provides an easy way to plot the two dimensional position of the quadrocopter and draw the flying path, but our real quadrocopter does not have a positioning system and its position estimation is not possible.

In order to estimate the position we have used the MEDUSA localization system developed in our faculty by Zug, Steup, Dietrich and Brezhnyev[ZSDB11]. The system calculates the two dimensional position of a mobile robot moving in a rectangular arena, using eight proximity sensors and a gyroscope. It uses the yaw angle of the robot and

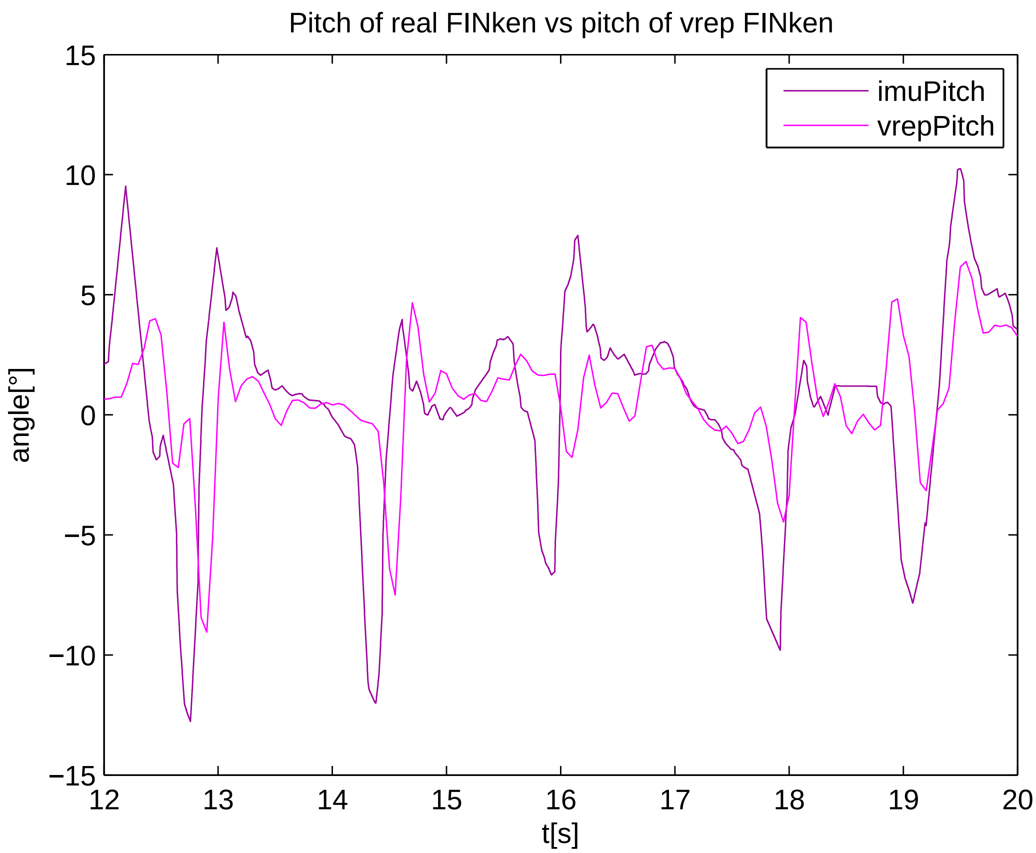

Figure 4.5: Detailed pitch angles of simulated FINken and real FINken

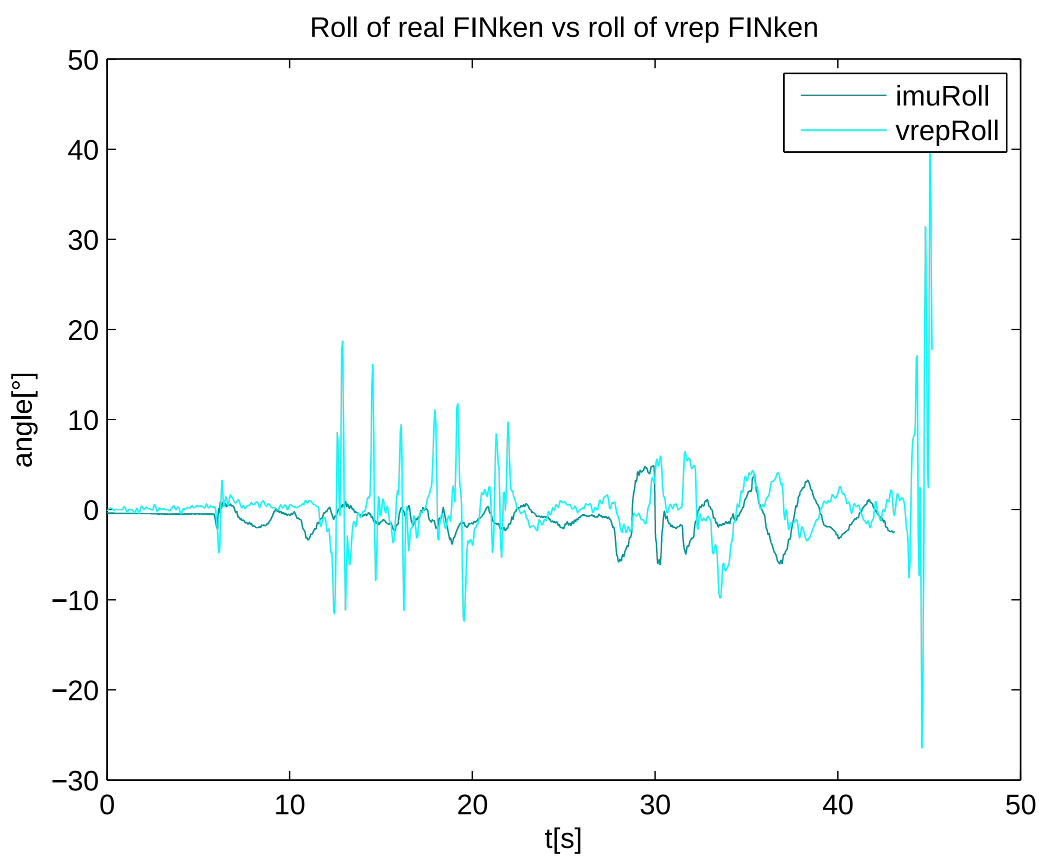

Figure 4.6: Roll angles of simulated FINken and real FINken

the distances to the walls from the proximity sensors to calculate a set of possible positions, where the quadrocopter could be. Then it moves through all possible points and calculates the distances, that the proximity sensors should read at that particular point and angle of rotation. Comparing the real readings from the quadrocopter proximity sensors and the calculated distances, a probability function is calculated, that indicates the probability that the real quadrocopter is situated at the particular point. The point with the highest probability is chosen for the quadcopter position.

The MEDUSA positioning system turned out to be an easy and fast way to get an estimation of the position, since our quadcopter has four ultrasound sensors, positioned at 90 degrees from each other, and a gyroscope, which provides the angle of rotation. The fact, that the original positioning system uses eight distance sensors was not disturbing, since it is possible to estimate the position even with two sensors. But a higher number of sensors provides more fault-tolerance to the system and more accuracy. It was also not necessary to implement the algorithm in the firmware of the quadcopter. We used the log file, that the Paparazzi software uses to write the sensor data and fetch the sensor readings from there using a Matlab script. Only a minor changes ware required in the configuration of the sensor numbers, the sensors maximum range and the size of the arena.

On Figure 4.7 you can see the calculated possible positions and the quadcopter positioned at the point with the highest possibility value. The lines represent the ultrasound sensors and the length of the lines depict the maximum sensor range. The red fractions of the lines show the actual distance measured by the sensor.

The positioning system relies on accurate sensor readings to provide a precise position estimation. Our gyroscope provides noisy estimation of the yaw angle and the distance sensors are also quite noisy. Since our quadcopter is equipped with just four distance sensors, the localization system cannot tolerate the noise as good as with eight sensors. For this reasons an accurate position estimation could not be expected, but it should provide some reference estimation of the flying path. On Figure 4.8 you can see the results of the position estimation. Each position has been marked with a blue star, so that we can see where the quadcopter has been. It is not possible to show the actual flying path as a function of time, but it shows which parts of the arena had been occupied by the quadcopter during the flight. Figure 4.9 shows the flight path of the simulated quadcopter in the V-REP. As in Figure 4.8 it does not shows the trajectory as a function of time, but just the occupied positions during the flight. The x and y axes of the plot show the V-REP scene arena and the small rectangle represents the flight arena.

The estimated positions on Figure 4.8 are very scattered, due to the fact that the localization system cannot estimate the position with a big accuracy. In the worst case, as a correct position was assumed one of the calculated possible points, which was far away from the previous estimated position. This creates a big jumps in the

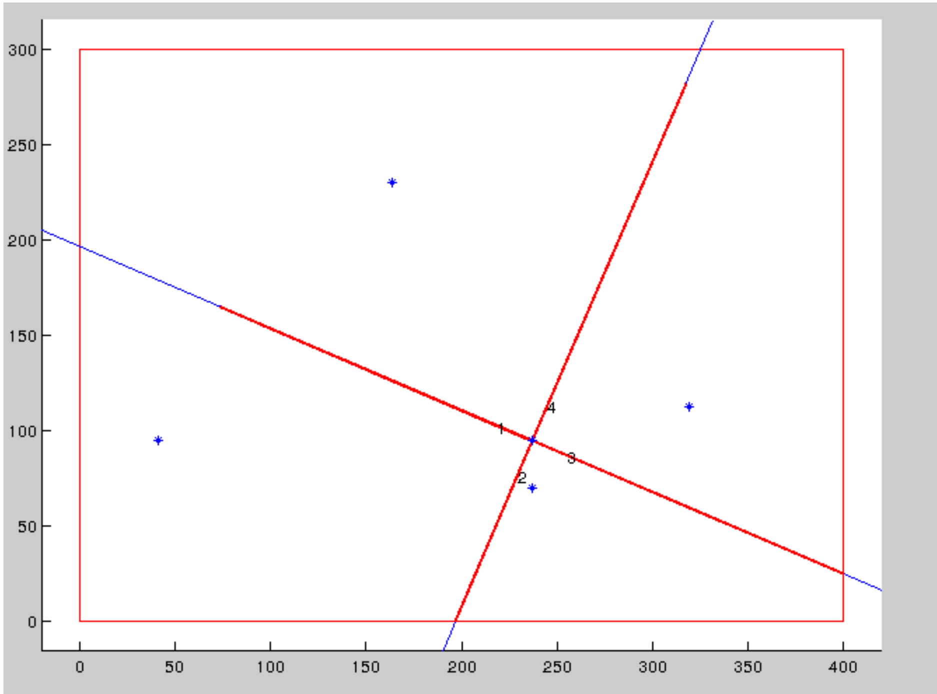

Figure 4.7: position estimation

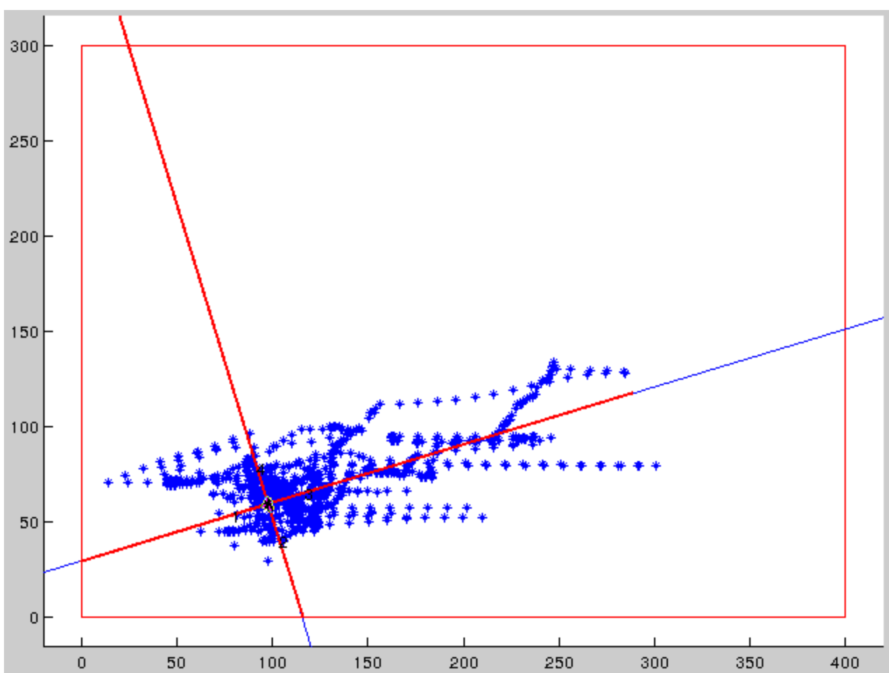

Figure 4.8: Estimated flight path

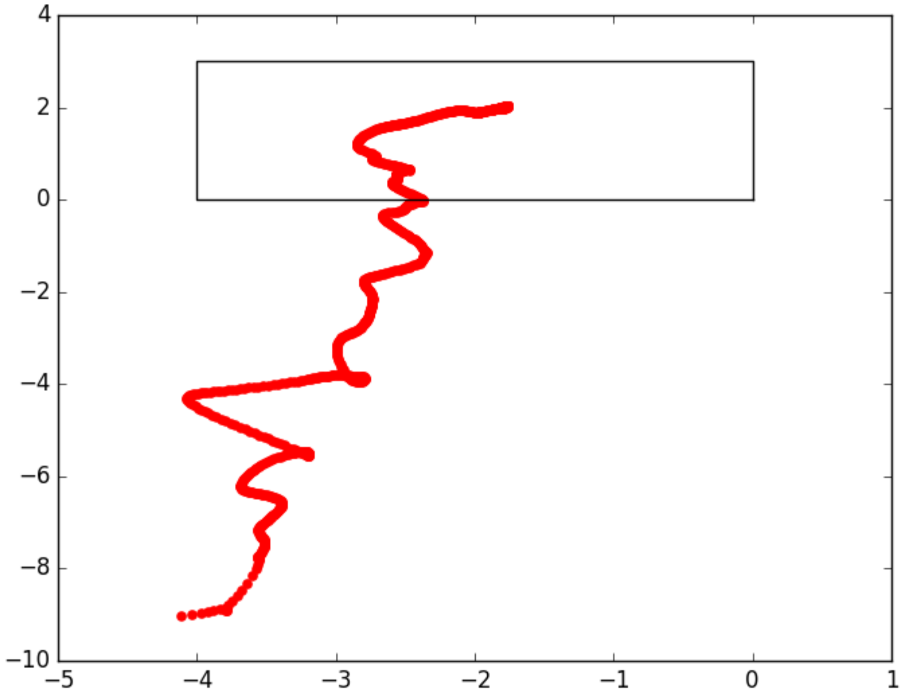

Figure 4.9: V-REP flight path

path, which does not coincide with the real flight path. The scattered plot and the jumps are caused by the noisy sensor measurements. A couple of attempts to filter the sensor readings ware made, but the plot still looked scattered to a significant extend. Thereupon, to eliminate the jumps, we limited the distance the copter could move in a single time step. This resembles that the real FINken can only move with a limited velocity. However, this improved the results slightly but still isn't satisfying for a perfectly accurate positioning system.

On the other hand Figure 4.9 shows a clear flying path, due to the fact that the position is not estimated, but taken directly from the V-REP environment. The graph shows how the quadcopter drifts and to what extend it flies away from the arena.

Eventually we came to the conclusion, that the used approach for position estimation was not good enough for our needs and a detailed comparison between the flying paths of the real and simulated quadcopter could not be done at this stage. However the working group has the intention to build a camera positioning system and track the quadcopter position with a camera mounted on the top of the arena.

5. Conclusion

5.1 Reached goals

During the project, a stable simulation of the FINken quadcopter was built. The performance of the simulation needs to be improved for bigger quadcopter swarms, but one real FINken can be connected to one simulated FINken running in real time on any modern computer. The simulation reproduces the real FINken's movements sufficiently as shown in Section 4.3. The simulation is based on un-threaded Lua scripts attached to the simulation objects. The scripts are stored outside V-REP and have a modular structure. A base class provides attitude control and a flight control API. Additionally, custom functionalities can be implemented in Lua in a custom class for each virtual FINken.

A Java API was build, that manages the V-REP scene and communication to the quadcopters. The API is very modular and comprises of several stand alone projects, each of which can be reused for other projects and mixed-reality simulations. The software architecture was build according to the SOLID [sol16] principles. The design uses abstraction layers between the modules, which make it possible to benefit from the open-closed principle and add new functionality without changing existing code.

The communication between V-REP and the real FINken is fast enough to send new data for every simulation timestep. The controllers of the virtual FINken adapt the new target values quickly and the movements of the real quadcopter can be observed in the virtual one. Internal sensor noise and inaccuracies and a missing external reference let the virtual FINken drift away. Correctly calibrating the FINken's internal sensor vastly improved the drift by removing sensor offsets. However, stable flight is only possible for typically half a minute, then, the cumulated errors let the virtual FINken drift out of the arena.

A major problem for the simulation is the drift around the z axis. When starting the FINken, often it starts yawing heavily right after take off. This yaw occurs due to sensor errors and resulting wrong correction commands to the motor controllers. Hence this error can't be detected and falsifies the simulation. With a different yaw angle, the coordinate systems of the real and virtual copter aren't aligned anymore, so pitch and roll commands result in movement into different directions. When this behaviour was observed, the experiment was aborted, as no useful results could be obtained. Until now, the cause of this behaviour could not be found.

5.2 Future Work

The fundamentals for a mixed reality simulation were set during this work. However, it would be desirable to get a stable simulation for the whole quadcopter flight time to use the mixed reality simulation for longer scenarios. A first step towards this goal could be to include data form the ultrasound sensors and the optical flows sensor which was set up parallel to this project. These sensors would provide a reference to the environment and a sensor fusion would have a major positive benefit on the accuracy of the virtual copters position. For evaluation purposes, an external reference system by optical tracking the copters could be used to get better internal sensor models by comparing the internal sensor data with the results from the external system.

An other idea would be to use the simulation environment to test the software that runs on the copter, by loading the C-code running on the copter into the simulation. This would most likely need a more detailed physical model, but would have the benefit of detecting problem completely without risking the expensive hardware.

Additionally, in the case described above, the API of the real and virtual copters would be identical. However, this could be achieved more easily, and would make it easier to test scenarios with multiple FINkens without the need do differentiate between real world and simulation.

The simulation provides the possibility to implement swarm algorithms that should later be handled by the real FINken. This could be started right away, but defining a API first that can be handled by real FINkens and simulation would increase the reuse of this work.

To scale the simulation, it is desireable to split the computation tasks into multiple threads, increasing simulation performance on multi-core computers.

Bibliography

[BSH11] Burgbacher, Ulrich ; Steinicke, Frank ; Hinrichs, Klaus H.: Mixed Reality Simulation Framework for Multimodal Remote Sensing. In: *IUI 2011 workshop on location awareness for mixed and dual reality (LAMDa)*, DFKI, 2011 (cited on Page 2)

[bul] *Bullet Physics Library*. http://bulletphysics.org/wordpress/ (cited on Page 4)

[CMW11] Chen, Ian Yen-Hung ; MacDonald, Bruce A. ; Wünsche, Burkhard C.: A flexible mixed reality simulation framework for software development in robotics. In: *Journal of Software Engineering for Robotics* 2 (2011), Nr. 1, S. 40–54 (cited on Page 1)

[CR] Coppelia-Robotics: *V-REP*. http://www.coppeliarobotics.com (cited on Page 2)

[Dee06] Deeg, Carsten: *Modeling, Simulation, and Implementation of an Autonomously Flying Robot*, TU Berlin, Diss., 2006. http://www.carstendeeg.de/marvin/dissertation/aerodynamik.html (cited on Page 7)

[HMS+15] Hönig, Wolfgang ; Milanes, Christina ; Scaria, Lisa ; Phan, Thai ; Bolas, Mark ; Ayanian, Nora: Mixed reality in robotics. In: *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, 2015 (cited on Page 1)

[Lau11] Lautrup, Benny: *Physics of Continuous Matter, Second Edition: Exotic and Everyday Phenomena in the Macroscopic World*. CRC Press, 2011. – 233 S. (cited on Page 6)

[Luu11] Luukkonen, Teppo: Modelling and control of quadcopter. In: *Independent research project in applied mathematics, Espoo* (2011) (cited on Page 5)

[ppr] *Paparazzi*. http://wiki.paparazziuav.org/wiki/Main_Page (cited on Page 2)

[sol16] *SOLID (object-oriented design)*. Wikipedia. https://en.wikipedia.org/wiki/SOLID_(object-oriented_design). Version: 2 2016 (cited on Page 50)

[ZSDB11] Zug, S. ; Steup, C. ; Dietrich, A. ; Brezhnyev, K.: Design and implementation of a small size robot localization system. In: *Robotic and Sensors Environments (ROSE), 2011 IEEE International Symposium on*, 2011, S. 25–30 (cited on Page 44)