

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:

Mixed-reality Simulation of Quadcopter-Swarms

Digital Engineering Project, University of Magdeburg, 2015.

Contents

List of Figures	v
List of Tables	vi
List of Code Listings	vii
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Outline	2
2 Theory	3
2.1 Quadcopter Modelling	3
2.1.1 quadcopter control	4
2.2 Vrep	5
2.3 Communication V-REP-Quadrocopters	6
2.3.1 V-REP Remote API	7
2.3.2 Java API	7
2.3.3 Ivy Bus	8
2.3.4 Communication	10
2.3.4.1 V-REP-Remote API specific requirements	10
2.3.4.2 Ivy Bus specific requirements	11
2.3.4.3 Application specific requirements	12
3 Implementation	15
3.1 Simulation Environment	15
3.1.1 Scene Modeling	15
3.1.2 Simulation Software Structure	17
3.2 Communication V-REP - Quadrocopters	19
3.2.1 Software architecture	19
3.2.2 JavaV-REP	21
3.2.3 JavaIvyBus	22
3.2.4 JavaXmlSax	24
3.2.5 JavaFinken	24
3.3 <i>[fancy name]</i>	24

3.4 Quadcopter	24
4 Evaluation	25
4.1 Speed	25
4.2 Accuracy	25
4.3 	25
4.4 usability	25
5 Conclusion	26
5.1	26
5.2 Future Work	26
Bibliography	27

List of Figures

2.1	Forces and torques of a quadcopter	3
2.2	Structure of the VREP-framework http://www.coppeliarobotics.com/ helpFiles/en/images/writingCode1.jpg 13.10.2015	5
2.3	communication V-REP - Quadrocopter	7
2.4	Ivy-Bus in Paparazzi Ground Station	8
2.5	Topic-based publisher-subscriber	9
2.6	Content-based publisher-subscriber	9
3.1	Parts of the simulation object	16
3.2	Software structure of the FINken Simulation	17
3.3	Java-API architecture	20

List of Tables

List of Code Listings

2.1	Message Xml definition	12
-----	----------------------------------	----

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.

As the copter were developed in the working group and were designed with a focus on modularity, they are under constant change. 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 [?], our focus lies not on hardware development, but on increasing situation complexity by computation power instead of more cost intensive real hardware.

[what are existing solutions, what's different in this approach, what is the improvement]

1.2 Problem Statement

The goal of the project is to provide a realistic, fast and scalable simulation in which single quadcopters can be linked to a real flying FINken 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[?]. The FINken Copter run the Paparazzi[?] 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

[short description of the sections]

2. Theory

2.1 Quadcopter Modelling

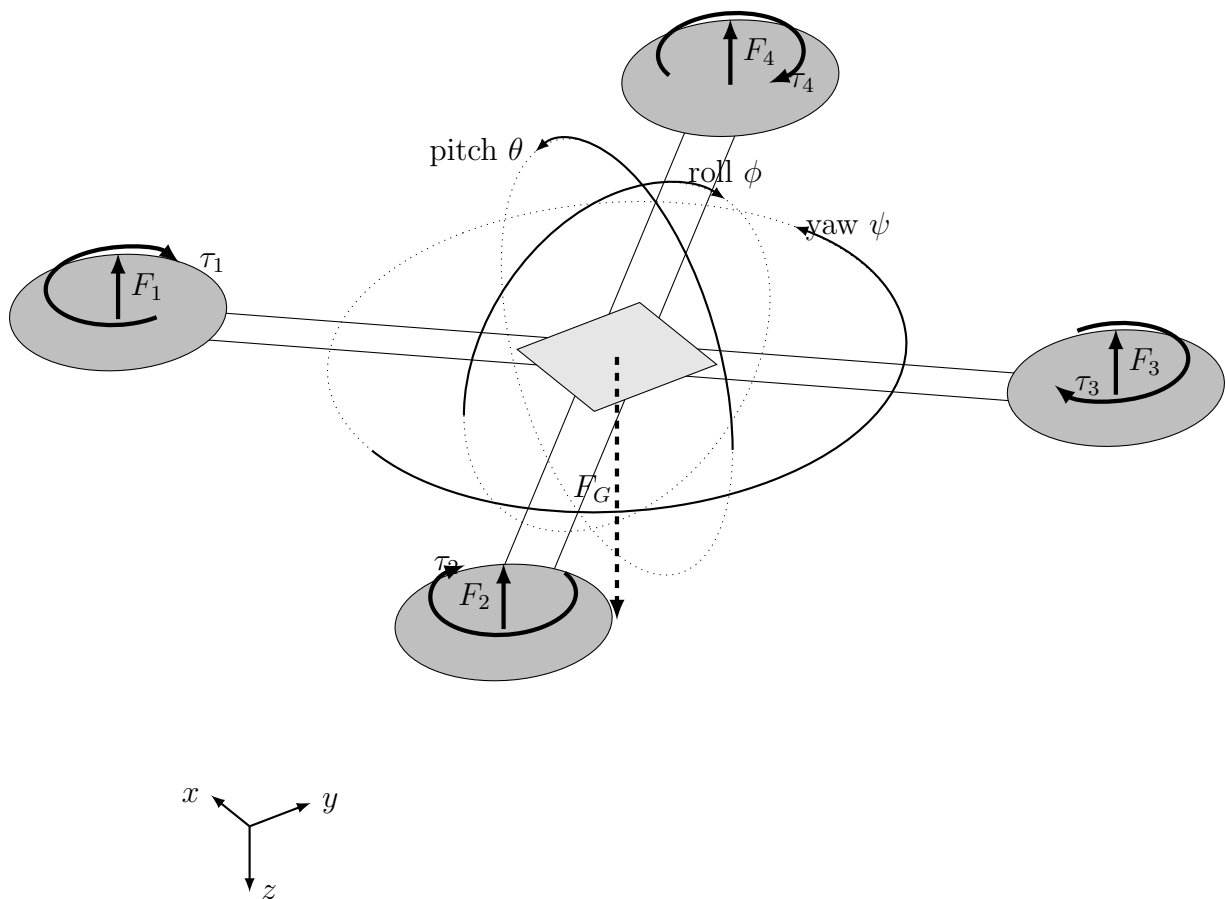


Figure 2.1: Forces and torques of a quadcopter

start of simulating quadcopter: deriving a physical model

to keep a 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.

$$F_G = F_1 + F_2 + F_3 + F_4 \quad (2.1)$$

by turning the rotors, the motors generate the torques $\tau_1.. \tau_4$ which act on the quadcopter and sum up to a final torque around the z -axis

if the $\tau_1 = \tau_2 = \tau_3 = \tau_4$, they sum up to zero, as τ_1 and τ_3 are directed clockwise and τ_2 and τ_4 are directed counter-clockwise

by changing the relation between the clockwise and counter-clockwise spinning rotors, a torque and thereby a rotational movement around the copters z -axis can be generated (yaw)

to keep copter stable, forces and torques have to sum up to zero

force and torque are dependant on each other

due to manufacturing tolerances and wear, the rotors generate a different force for the same torque

this effects will not be modelled

in general, only a thrust is given and the ins? computes the resulting power for the rotors

show thrust -> power -> torque -> rpm -> thrust

when $F_1 .. F_4$ are not equal, they induce an angular movement around the quadcopter

when the copter is tilted in roll or pitch, the thrust vectors of the rotors don't point directly downwards anymore which results in a linear acceleration of the copter

Figure 2.1 blav

2.1.1 quadcopter control

t

t

t

t *[physics of quadcopter movements]*

[theory of quadcopter control]

[physics of the airstream (explain particle simulation)]

2.2 Vrep

V-REP is a versatile, highly customisable 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 visualisation.

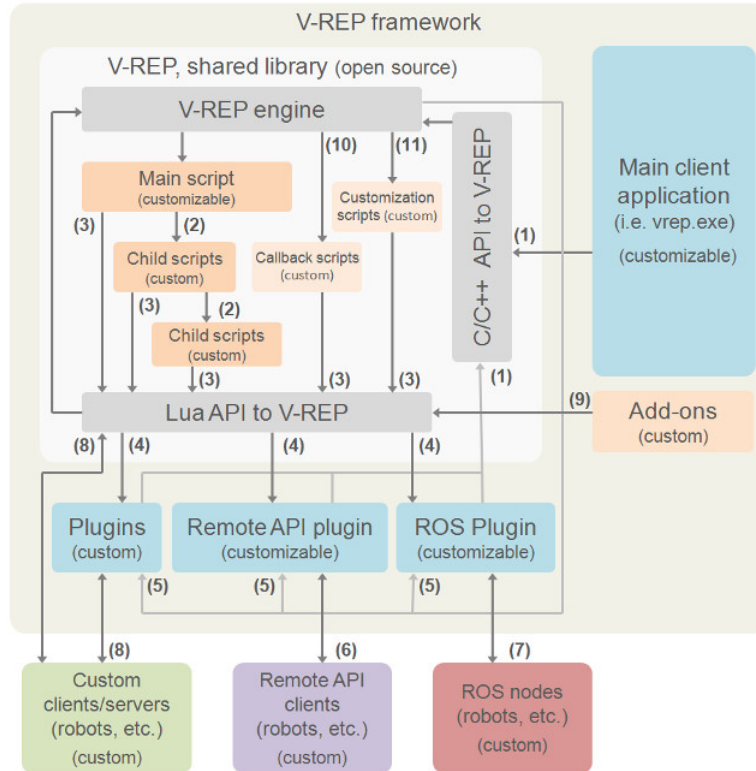


Figure 2.2: Structure of the VREP-framework <http://www.coppeliarobotics.com/helpFiles/en/images/writingCode1.jpg> 13.10.2015

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 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 ?? 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 manoeuvres as those flying in the real environment. In order to make the model replicate this behaviour, the flying quadrocopter must send its linear and angular velocity, its pitch yaw and roll 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 quadrocopter thus providing it with information from a virtual sensor.

The virtual quadrocopters are purely simulation quadrocopter 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.

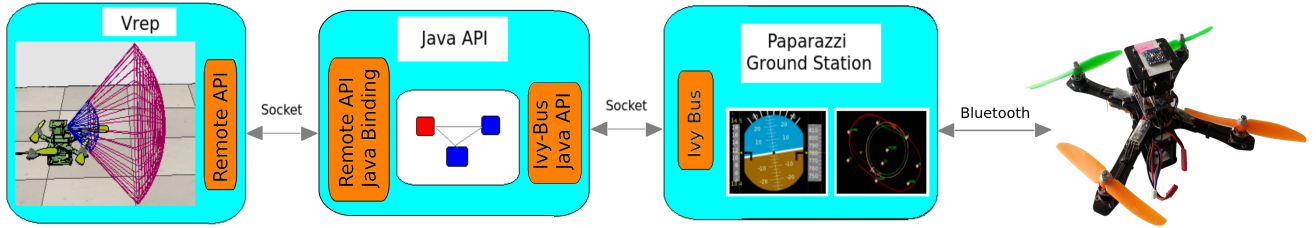


Figure 2.3: communication V-REP - Quadrocopter

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 favours 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.

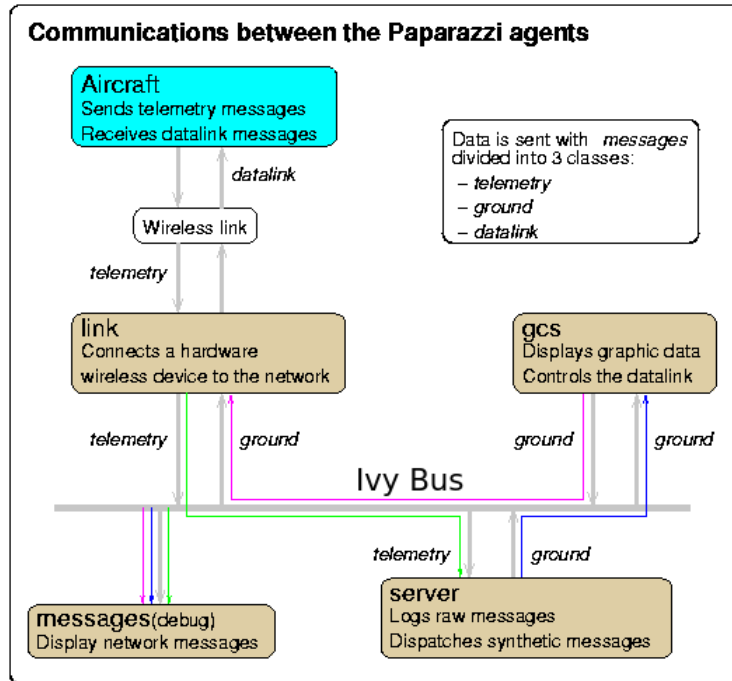


Figure 2.4: Ivy-Bus in Paparazzi Ground Station

The 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.

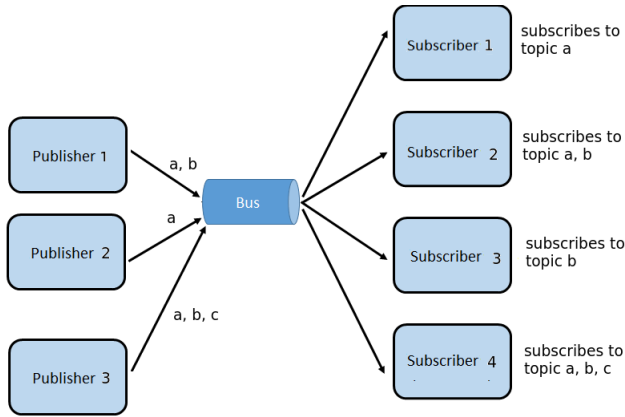


Figure 2.5: Topic-based publisher-subscriber

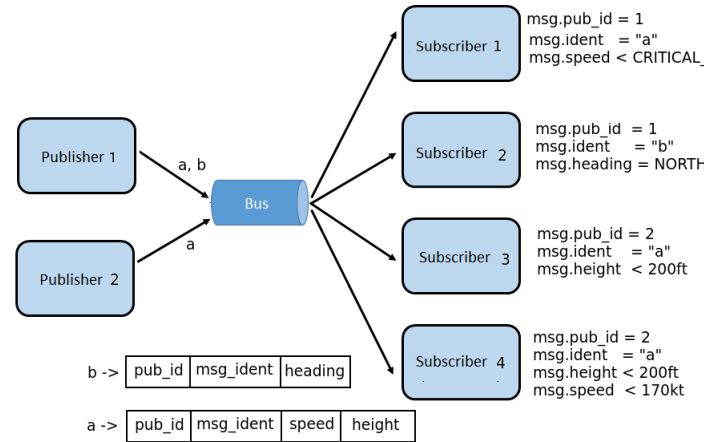


Figure 2.6: Content-based publisher-subscriber

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 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

[integrate into sectionsec:theoryVrep] The main elements in V-REP that are used for building a simulation scene are scene objects (objects in short). Objects are visible in the scene hierarchy and in the scene view. In the scene view, objects have a three dimensional representation. Some of the object types are: shape, joint, proximity sensors, vision sensors and others. The shape object is a rigid mesh that is composed of triangular faces. Our quadrocopters are represented as a shape objects in the V-REP scene. But is also contains four infrared sensors which are of type proximity sensor object. So the quadrocopter is represented as a complex scene object. 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 - liniar 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 be 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 modelled 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 virtual one, implementing their specific behaviour. 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, roll, 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 it 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 retrieves the instances of the real and virtual quadrocopters. 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 objects. 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 VREP, using only simple rectangular shapes. Making the complex shape static, establishing a fixed connection and hiding the simple shape, the result is 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 referring 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. 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 VREP. The motor assembly adds significant weight outside the FINkens center of gravity and therefore has a rather large influence on its behaviour. *[weight table for finken2 and finken3]* 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.

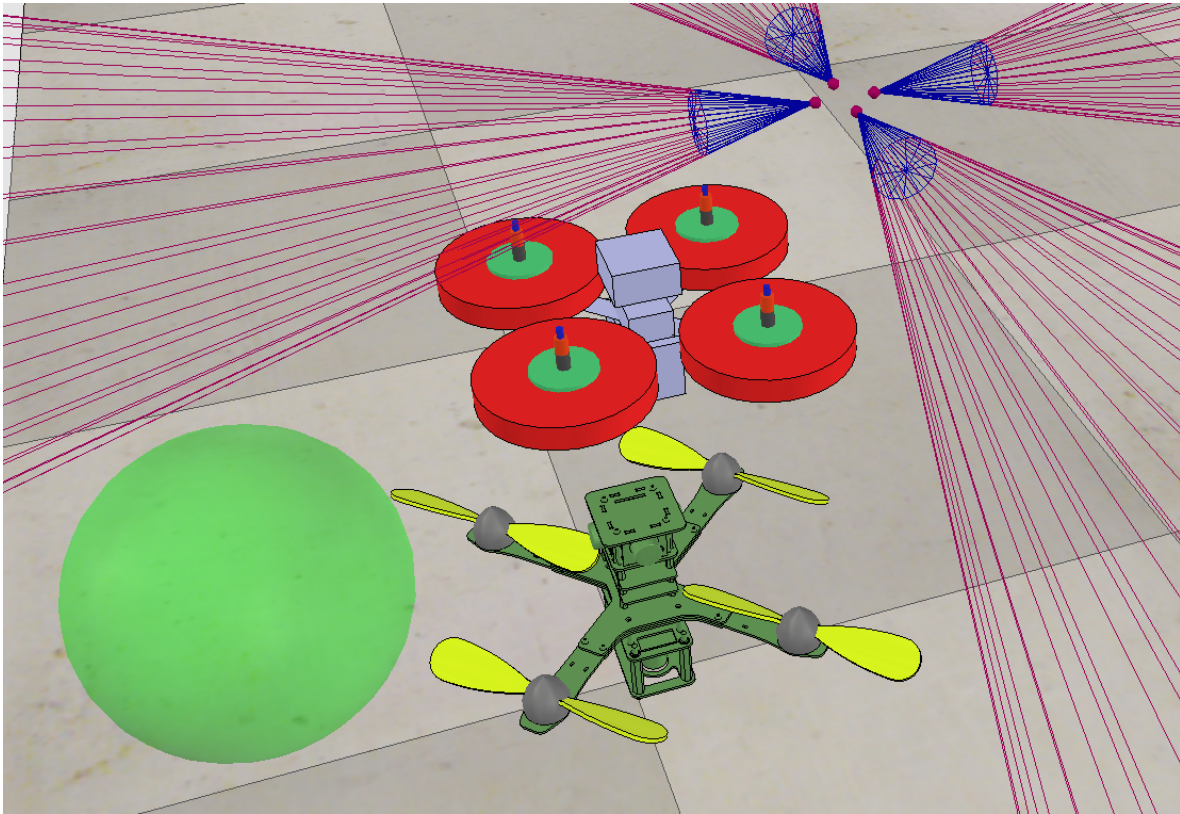


Figure 3.1: Parts of the simulation object

The rotor model in VREP 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. *[add picture of rotor]* 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. *[picture of rotor simulation and rotor visualisation]* 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. The script has several parameters for the particle simulation, *[add complete param list]*. During simulation, the particle velocity is the parameter used to control the finken. This rotor model was already included in VREP as an example and was used without further modifications. *[add formula to show linearity between power and airstream velocity]*

[finken parameter estimation]

[controller tuning]

[simulation parameters]

Vrep provides several physics engines, we use the default bullet engine for our simulation.

3.1.2 Simulation Software Structure

[finish diagram of finken software] The simulation software is written in Lua and runs

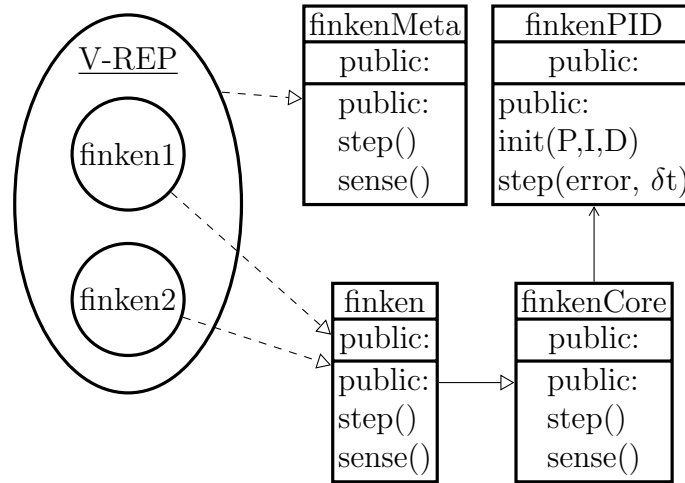


Figure 3.2: Software structure of the FINKen Simulation

inside a non-threaded child script of each FINKen quadcopter in VREP.

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 VREP. Signals inside VREP are the most versatile communication possibility which can be accessed from anywhere inside VREP 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 for each simulated FINKen. It starts the VREP remote API server and creates the API signals. Also, the PID controllers for flight control are initialized. During simulation, calling the `FINKenCore.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 logisitic curve Equation 3.1 to decrease the influence of the throttle close to the hover thrust.

$$throttle_{tuned} = \begin{cases} -\frac{a*|throttle|}{a-|throttle|+50} + 50 & throttle < 0 \\ \frac{b*throttle}{b-throttle+50} + 50 & throttle \geq 0 \end{cases} \quad (3.1)$$

The default values for the throttle tuning function are $a = 1$ and $b = 1$. *[insert plot of function?]*

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, VREP 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.

[FINKEN module as an API for further extensions]

3.2 Communication V-REP - Quadrocopters

This chapter describes the implementation of the requirements on the Java-API, which were 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 3.3](#) 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 were 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](#).

JavaIvyBus is a Java project, that imports the Ivy-Bus library and implements the requirements regarding Ivy-Bus, which were 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

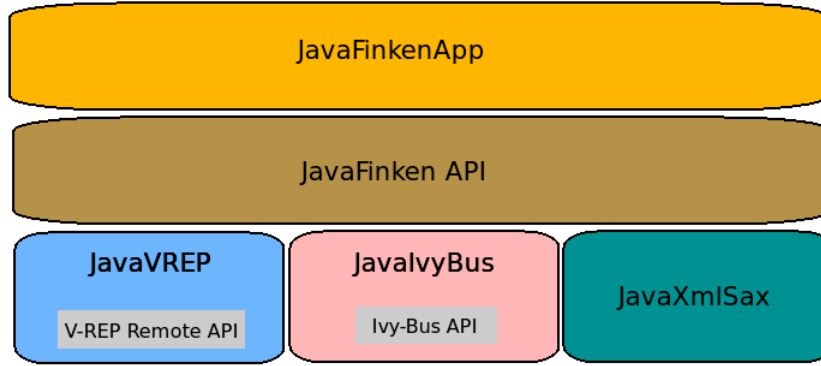


Figure 3.3: Java-API architecture

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 *JavaIvyBus* 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 Graphical User Interface 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 JavaV-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 an instance of the *VrepServer* and open the client:

```
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 *ObjectUpdater* and its abstract implementation *AbsObjectUpdater*, which is used to update the *VrepObjects* with real-time parameters from V-REP, like their *Position*, *Orientation*, *LinearVelocity* and *AngularVelocity*.

3.2.3 JavaIvyBus

The requirements regarding the Ivy-Bus, that were 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 *MessageFields*.

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 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 quadcopter which 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 *MessageFields* initialized with the actual message values. The following listing shows an example how to subscribe to a message and get asynchronous notification when the message is received.

```
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

3.2.5 JavaFinken

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

3.3 *[fancy name]*

3.4 Quadcopter

4. Evaluation

[how realistic is the simulation?] [which properties can be modelled well, which can't?]

4.1 Speed

[communication delay] [vrep simulation speed]

4.2 Accuracy

[error]

4.3 |

Stability

4.4 usability

5. Conclusion

5.1

[do the results show that it works?]

5.2 Future Work

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 29. November, 2013