D6.5 – INITIAL INTEGRATION OF EXTERNAL SIMULATORS

Deliverable ID	**D6.5**
Deliverable Title	**Initial Integration of External Simulators**
Work Package	**WP6 – Simulation and Performance Prediction**
Dissemination Level	**PUBLIC**
Version	**1.0**
Date	**30/06/2018**
Status	**Final**
Lead Editor	Davide Conzon **(ISMB)**
Main Contributors	Arthur Pitman **(UNI-KLU),** Miquel Cantero, Angel Soriano **(ROBOTNIK),** Omar Morando **(DIGISKY)**

Published by the CPSwarm Consortium

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731946.

Document History

Version	Date	Author(s)	Description
0.1	2017-03-26	Davide Conzon (ISMB)	First Draft with TOC
0.2	2018-04-09	Davide Conzon (ISMB)	First sections
0.3	2018-05-25	Davide Conzon (ISMB), Arthur Pitman (UNI-KLU), Miquel Cantero, Angel Soriano (ROBOTNIK)	Integration of the simulation descriptions
0.4	2018-06-09	ISMB	Completed introduction, conclusions and almost completed the implementation section
0.5	2018-06-13	Davide Conzon (ISMB), Arthur Pitman (UNI-KLU), Omar Morando (DIGISKY)	Revised some sections and completed the section about implementation
0.6	2018-06-14	Davide Conzon (ISMB), Arthur Pitman (UNI-KLU)	Some minor revision about the interaction between the components
0.7	2018-06-22	Davide Conzon (ISMB)	Integrated the reviews of FIT and UNI-KLU
1.0	2018-06-30	Davide Conzon (ISMB)	Final version

Internal Review History

Review Date	Reviewer	Summary of Comments
2018-06-22	Sisay Chala (FRAUNHOFER)	Few modifications and comments.
2018-06-26	Arthur Pitman (UNIKLU)	Minor comments.

Table of Contents

Document History 2
Internal Review History 2
Table of Contents 3
1 Executive Summary 6
2 Introduction 7
2.1 Scope 8
2.2 Background 8
2.3 Document Organization 9
2.4 Related documents 9
3 Simulation and Optimization Environment in the CPSwarm Workbench 10
3.1 API 11
3.1.1 SOO and Optimization Tool interaction 11
3.1.2 SOO and Simulation Managers interaction 11
3.1.3 Optimization Tool and Simulation Managers interaction 11
4 ROS-based robotics simulation tools 12
4.1 Stage 12
4.1.1 Control interface 13
4.1.2 Simulation settings 13
4.1.3 Implementation of the CPS behavior 13
4.1.4 Integration with Ground Control Stations 13
4.1.5 Other configuration parameters 13
4.2 Gazebo 14
4.2.1 Control interface 14
4.2.2 Simulation settings 15
4.2.3 Implementation of the CPS behavior 17
4.2.4 Integration with Ground Control Stations 17
4.2.5 Other configurations 18
4.3 V-REP 19
4.3.1 Control interface 19
4.3.2 Simulation settings 20
4.3.3 Implementation of the CPS behavior 21
4.3.4 Integration with Ground Control Stations 21
4.3.5 Other configuration parameters 21
4.4 ARGoS 21
4.4.1 Control interface 22
4.4.2 Simulation settings 22
4.4.3 Implementation of the CPS behavior 23
4.4.4 Integration with Ground Control Stations 23
4.4.5 Other configuration parameters 24
4.5 jMAVSim 24

4.5.1 Control interface 24
4.5.2 Simulation settings 24
4.5.3 Implementation of the CPS behavior 24
4.5.4 Integration with Ground Control Stations 24
4.5.5 Other configuration parameters 24
4.6 STDR 25
4.6.1 Control interface 25
4.6.2 Simulation settings 26
4.6.3 Implementation of the CPS behavior 26
4.6.4 Integration with Ground Control Stations 26
4.6.5 Other configuration parameters 26
5 Simulation Virtual Machine 27
6 Implementation and Integration 28
6.1 Data formats of the simulation interfaces in the CPSwarm Workbench 28
6.1.1 JSON 28
6.1.1.1 StartOptimization 28
6.1.1.2 GetProgress and CancelOptimization 28
6.1.1.3 OptimizationResult 28
6.1.1.4 OptimizationStarted, OptimizationCancelled and OptimizationProgress 29
6.1.1.5 RunSimulation 29
6.1.2 XML 29
6.1.2.1 Optimization Tool configuration 29
6.1.2.2 Optimization Tool result 29
6.1.3 SDF 30
6.1.4 Wrapper 31
6.1.4.1 Main() 31
6.1.4.2 Sync() 31
6.1.4.3 SyncUpdate() 31
6.1.4.4 simStep() 31
6.1.5 Candidate 31
6.1.5.1 getOutput(float netInput[],long inputsize) 31
6.2 Integration of FREVO as an Optimization Tool 32
6.3 SOO 32
6.3.1 Listeners 32
6.3.1.1 ConnectionListenerImpl 32
6.3.1.2 MessageEventCoordinatorImpl 32
6.3.1.3 PacketListenerImpl 33
6.4 Simulation Manager 33
6.4.1 Listeners 33
6.4.1.1 ConnectionListenerImpl 33
6.4.1.2 AbstractFileTransferListener 33
6.4.1.3 PresencePacketListener 34

6.5 Gazebo Simulation Manager 34
6.5.1.1 FileTransferListenerImpl 34
7 Conclusions 35
Acronyms 36
List of figures 36
List of tables 37
References 37
ANNEX A – Optimization Tool Configuration 38
ANNEX B – Optimization Tool Result 39
ANNEX C – SDF world example 40
ANNEX D – SDF environment model example 41
ANNEX E – SDF Object model example 42
ANNEX F – JSON for StartOptimization 45
ANNEX G – JSON for GetProgress and CancelOptimization 46
ANNEX H – JSON OptimizationResult 47
ANNEX I – JSON OptimizationReply 48
ANNEX J – JSON RunSimulation 49

1 Executive Summary

This deliverable, "D6.5 Initial Integration of External Simulators", gives a detailed description of how the external simulators have been initially integrated in the CPSwarm Workbench. This deliverable continues from deliverable D6.1 - *Initial Simulation Environment* in which, a first survey of available simulators was presented. First, this deliverable outlines the architecture designed in the CPSwarm Workbench for the Simulation and Optimization Environment. Then, the authors analyze the features provided by the most interesting simulators among the ones collected in D6.1, focusing on how they can be integrated and controlled. Then, the authors describe the required simulation environment built to do the first tests. Finally, the deliverable describes the software and the data formats designed and implemented to integrate the simulators in the workbench.

This deliverable reports on the results of Task 6.4 - *External simulators integration*.

2 Introduction

The deliverable D6.1, released in M9, presented the state-of-the-art both of the simulators and of the standard data formats usually leveraged to exchange data in this context. Furthermore, it introduced the first version of the distributed Simulation and Optimization Environment, based on the broker-based approach (see Figure 1) designed for the CPSwarm Workbench.

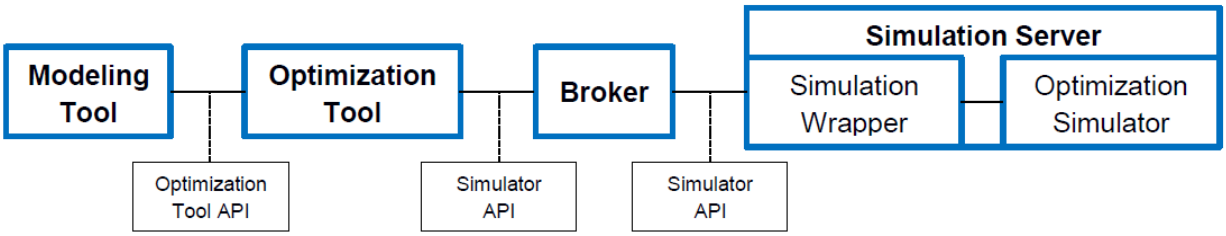

Figure 1 – The broker-based approach of the CPSwarm Worbench

Deliverable D6.1 forms the base of the work described in this document. Indeed, the state-of-the-art of the simulators has been leveraged to choose the more interesting simulators to be used in the CPSwarm Workbench, evaluating them in terms of openness, diffusion and features provided. These considerations have led to the selection of solutions based on the Robotic Operating System (ROS). Another main reason to choose this type of simulator is the ongoing work in Task 7.1 – *CPS Abstraction Library*, where the CPSwarm Consortium has decided to use ROS as the Cyber Physical System (CPS) Abstraction Library that abstracts the main functions of the CPSs, integrated in the CPSwarm Workbench. ROS has been chosen because is an open-source solution, widely supported by several robotics platforms (including the ones used in the CPSwarm scenarios), which provides modularity and interoperability. Indeed, ROS allows to build software solutions where several modules communicate, also if written in different programming languages (please, refer to D7.1, for the details about ROS and its role in the CPSwarm Workbench), Furthermore, the state-of-the-art standards used for the exchange of data in the context of distributed simulation has been used, in combination with the analysis of the formats already leveraged by the ROS simulators, to choose the data format for the CPSwarm Workbench. Finally, the analysis of the issues identified in the broker-based approach proposed in D6.1 has led to the design of the Simulation and Optimization Environment, described in this document, which can be used both to test a swarm algorithm, or to optimize an evolutionary algorithm using an Optimization Tool, e.g. FRamework for Evolutionary Design (FREVO)[1].

The deliverable starts by describing the distributed architecture, based on the eXtensible Messaging and Presence Protocol (XMPP)[2] protocol, designed for the Simulation and Optimization Environment in the CPSwarm Workbench, describing the features introduced to improve the previous approach and the roles of the main components of this approach: the centralized Simulation and Optimization Orchestrator (SOO), the distributed Simulation Managers and the Optimization Tool. Then, the document presents a detailed analysis of the main ROS-based simulators, specifically, focusing on how they can be controlled and configured and the formats they use for the models and the CPS behavior. The results of this analysis have led the CPSwarm Consortium to build a Linux-based virtual machine, with all the software needed for a complete Simulation

[1] http://frevo.sourceforge.net/
[2] https://xmpp.org/

Environment for ROS-based CPSs. Specifically, the Gazebo simulator[3] has been chosen for these first tests, because it provides all the features required and an open Application Programming Interface (API) both for the configuration and the control through ROS. Then, the document outlines the developments done in the task T6.4 to integrate the external simulators, implementing the proposed distributed approach, based on XMPP. Specifically, the document focuses on the interactions among the main components, involved in the simulation and optimization process, describing the API defined, the data format used and the implementation done for the interaction of these components. Finally, the document describes the implementation done to integrate the Gazebo simulator in the CPS Workbench, configuring it and using it to simulate a swarm of CPSs in the proposed scenarios.

2.1 Scope

This deliverable is focused on simulation environments for robotics behavior (specifically rovers and drones). Many other simulators exist, such as traffic simulators, like Simulation of Urban MObility (SUMO)[4], or network simulators (like omnNET++[5]), but they are out of scope in this deliverable. Furthermore, only the software components and interfaces used to integrate the external simulators are completely covered. The other components of the Simulation Environment and their interfaces are only briefly explained, because they are the main focus of D6.1 and of D6.2 - *Final Simulation Environment* due in M28. Finally, this deliverable doesn't describe how the fitness function is used to calculate the fitness value for the simulation, since this is scope of the deliverable D6.3 -*Initial CPS System design optimization and Fitness function design guidelines*.

2.2 Background

CPSwarm is based on the conceptual architecture shown in Figure 2. One of the base components is the Models library, which collects formal representations of CPSs, behavior routines, swarm algorithms and human-to-CPS interaction patterns. This library will be one of the first model libraries for designing swarms of CPS. Building upon these models, the project partners will develop a design, simulation and performance prediction Workbench, namely the CPSwarm Workbench, which will enable CPS engineers to collaborate and assemble model-based descriptions of CPS systems, with humans in the loop.

[3] http://gazebosim.org
[4] http://www.dlr.de/ts/en/desktopdefault.aspx/tabid-9883/16931_read-41000/
[5] https://www.omnetpp.org/

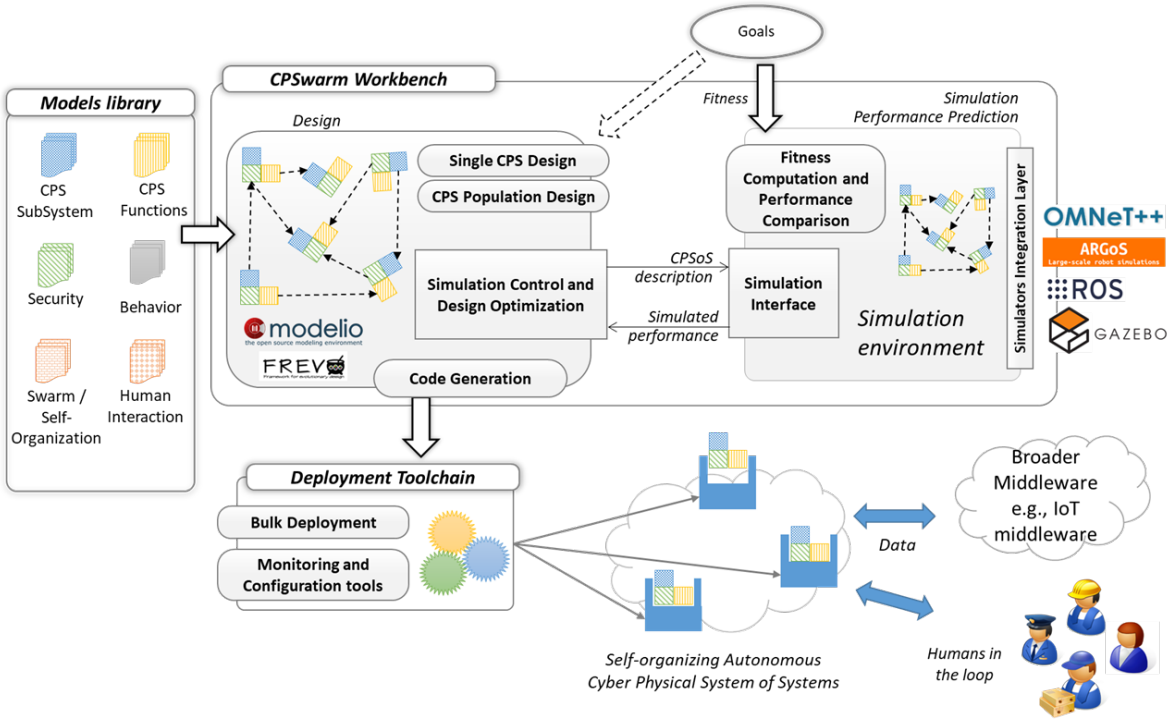

Figure 2 - CPSwarm concept

The outcome of the first, model-based, design phase will be a swarm, potentially heterogeneous, of CPSs and their behavioral models. Then, the CPSwarm Workbench enables simulation of the overall emerging system behavior and evaluation of its performances. The simulation includes complex real data, and/or hardware/human in the loop solutions where needed. Co-simulation is also considered, where needed, by integrating a set of selected simulation engines that allow the simulation of different CPS aspects. A dedicated Simulators Integration Layer is part of the CPSwarm solution to decouple the CPSwarm Simulation and Optimization Environment from engine-specific, control commands and data formats.

2.3 Document Organization

The rest of this document is organized as follows: Firstly Section 3 briefly introduces the Simulation and Optimization Environment designed in task T6.1 - *Simulation Environment* and T6.2 - *Simulation control and CPS System design optimization*. Then, the Section 4 presents the analysis of the simulation engines to be integrated in the CPSwarm Workbench. Section 5 describes the Simulation Environment built to test the chosen simulators. Finally, Section 6 presents the implementations done to integrate the simulators in the workbench, in terms of data format defined and software components developed.

2.4 Related documents

ID	Title	Reference	Version	Date
D6.1	Initial Simulation Environment	D6.1	6.0	M9
D6.2	Final Simulation Environment	D6.2	1.0	M28
D7.1	CPS Abstraction Library	D7.1	1.0	M18

3 Simulation and Optimization Environment in the CPSwarm Workbench

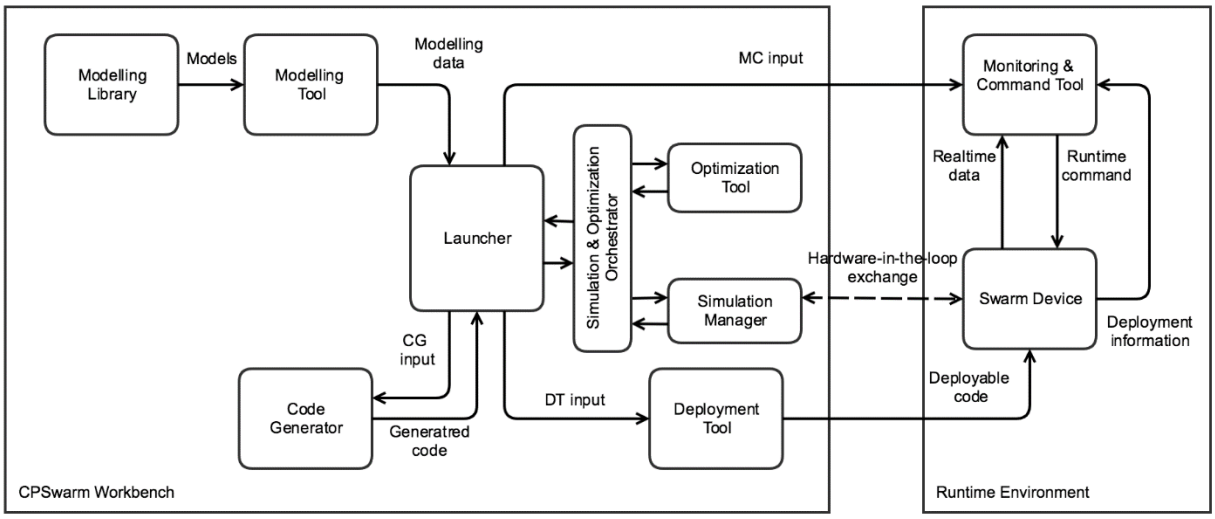

Figure 3 - CPSwarm reference architecture

The external simulators are integrated in the CPSwarm Wokbench architecture (Figure 3), using a broker-based distributed approach, designed in Tasks 6.1 and 6.2. This deliverable will briefly describe this approach, detailing only the interaction between the Optimization Tool and the simulation components. Indeed, the detailed description of the whole approach (which is an evolution of the one proposed in D6.1) will be included in D6.2, to be submitted in M28.

The current approach followed to design the Simulation and Optimization Environment shares, with the approach described in D6.1, the concept to have a distributed architecture, with several simulation engines distributed on different machines. The main differences introduced have the goal to address the issues seen in the previous solution (Micha Rappaport, 2018), which limit the performance of that solution, like the not-efficient discovery mechanism and the high number of messages exchanged between the Optimization Tool and the simulators during the simulation.

In the new architecture, there is a centralized component, called the SOO, which orchestrates the work of the Optimization Tool and the distributed simulators, each one wrapped by a Simulation Manager.

The XMPP protocol will be used instead of Message Queue Telemetry Transport (MQTT), for the communication among these components, allowing some core features[6] to be used, such as unique identifiers, presence mechanism and one-to-one chat messages; or some of the protocol extensions, like file transfer[7] and publish/subscribe[8]. Furthermore, at the same time, it will be possible to leverage all the security features natively provided by the protocol (Conzon, 2012).

One way in which XMPP has been leveraged to improve the architecture has been in the refactoring of the discovery of the distributed Simulation Managers. The discovery mechanism used in the previous approach introduced delays, due the time spent by the Simulation Managers to handle the discovery requests, every time a new simulation was needed. For this reason, in the new approach the previous solution has been

[6] https://xmpp.org/rfcs/rfc6120.html
[7] https://xmpp.org/extensions/xep-0096.html
[8] https://xmpp.org/extensions/xep-0060.html

replaced with a more scalable mechanism, where the Simulation Managers announce themselves to the SOO once they are available, leveraging the XMPP presence mechanism.

Finally, in the approach described in D6.1, the simulation of the controller was done, by exchanging sensor/actuator messages between the simulator and the controller, which was running in the Optimization Tool. This increased the number of messages and decreased performances. To address this issue. In the new version, the complete controller is sent to the Simulation Manager and it is evaluated in the simulator, sending back only the fitness score, limiting the number of messages exchanged.

3.1 API

This section describes the main API among the components of the Simulation and Optimization Environment.

3.1.1 SOO and Optimization Tool interaction

At the start the SOO and the Optimization Tool connect to each other, to exchange presences. Then, the SOO sends a *StartOptimization* message (see Section 6.1.1.1) to the Optimization Tool, when a new optimization process must be started. The Optimization Tool answers this request by sending an *OptimizationStarted* message (see Section 6.1.1.4), indicating if all is ok or if there has been some error. If all is OK, the SOO sends the Optimization Tool a configuration file (see Section 6.1.2.1). During the optimization process, the SOO can send messages to the Optimization Tool, via chat, to control the process. *GetProgress* or *CancelOptimization* (see Section 6.1.1.2), can be used respectively to get the status of the optimization and to stop it. The Optimization Tool answers these messages with an *OptimizationProgress* or *OptimizationCancelled* reply (see Section 6.1.1.4). When an optimization process is finished, the Optimization Tool sends the optimization result to the SOO (see Section 6.1.2.2).

3.1.2 SOO and Simulation Managers interaction

Every time a Simulation Manager starts, it adds the SOO to its roster and then publishes a presence with the server info in the status (the format is the one described in D6.1 as server). When the SOO receives this presence, it adds the Simulation Manager to the available ones and it stores the info of that manager. In the same way, when the SOO receives an unavailable notification from one Simulation Manager, it removers this from the list of the available ones. The SOO sends, using the XMPP file transfer, the files needed to configure the simulation, this can be done when a new simulation must be started or when a new optimization process starts (in this case, it contains the files that will need to be used in all the simulations, excluding the candidate, which change for each simulation run). If the simulation doesn't require optimization, the SOO can send a *RunSimulation* message (see Section 6.1.1.5), to the Simulation Manager also, to directly start a simulation.

3.1.3 Optimization Tool and Simulation Managers interaction

The Optimization Tool sends, using the file transfer, the candidate to be used for that simulation to the Simulation Manager and then a *RunSimulation* message to start it. If the Simulation Manager is not already busy, it starts the simulation and answers when it is finished by sending an *OptimizationResult* message (see Section 6.1.1.3) to the Optimization Tool, indicating the fitness value calculated, otherwise, it indicates that it is not available.

4 ROS-based robotics simulation tools

The deliverable D6.1 contains an analysis of the state-of-the-art of simulation tools. Based on this analysis and according to the activities ongoing in task T7.1 - *CPSwarm Abstraction Library*, where the Consortium has chosen to base the CPSwarm Abstraction Library on ROS, simulation tools, which support the ROS framework are considered in detail in this document, namely Stage[9], Gazebo, V-REP[10], ARGoS[11], jMAVSim[12] and STDR[13].

Specifically, this section presents an analysis of the robotic simulators that can be used in the CPSwarm Workbench, focusing on the way they can be controlled and configured and the formats they use for the models and the behavior.

The first objective of this analysis is to understand, which simulators are suitable for integration in the CPSwarm Workbench and thus the ones, which fulfill requirements such as:

- ROS integration.
- Possibility to control the simulator from an external process, through command line or some sort of API.
- Open format to describe the CPS and environment models.
- Possibility to describe the CPS behavior.
- (Optional) possibility to interact with a Ground Control Station during the simulation.

The results of this analysis, presented in this section will be used in four different ways:

- To choose the first simulators to be integrated in the CPSwarm Workbench, based on the features provided and the ease of integration.
- To build a first Simulation Environment to be used for testing (see Section 5).
- To choose among the data format presented in the deliverable D6.1, for the integration of external simulators in the CPSwarm Workbench (see Section 6.1).
- The information collected, will form the basis for the development of the respective Simulation Managers (see Section 6).

4.1 Stage

Stage is a low-fidelity two-dimensional simulator that can be used with a variety of robotic platforms and sensors. It supports the concurrent simulation of many robots, allowing swarms of potentially hundreds of thousands of robots to be simulated at the same time (Staranowicz, 2011). Stage can be used as a standalone application that loads a robot control program from a user-defined library, a ROS-based simulation tool and as a plugin for Player (libstageplugin). Stage is available for Linux, macOS and Windows.

[9] http://rtv.github.io/Stage/
[10] http://www.coppeliarobotics.com/
[11] http://www.argos-sim.info/
[12] https://pixhawk.org/dev/hil/jmavsim
[13] http://wiki.ros.org/stdr_simulator

4.1.1 Control interface

Stage is a part of two projects the Player Project (Player/Stage/Gazebo) and ROS. In ROS, first a Stage controller need to be compiled using the command:

```
rosmake stage_controllers
```

Then, the simulation can be started by:

```
rosrun stage stageros `rospack find stage_controllers`/world/CONTROlER_NAME.world
```

where, the *stageros* node wraps Stage simulator, via *libstage*, then Stage simulates a world as defined in a *world* file.

4.1.2 Simulation settings

Stage simulates a scenario or a world that is defined in a *world* file, as mentioned above. Many sample *world* files can be found in the /world subdirectory in the *ros_stage* package. Each file contains the complete set of information relevant to a scenario such as obstacles and robots, as well as environment parameters, and details of the robotic platforms and sensors. A top-down black and white *png* image is used to specify the initial location of the obstacles in the environment.

4.1.3 Implementation of the CPS behavior

CPS behavior can be implemented by writing C++ or Python code, or using another ROS node. Starting with Stage V3, integrated controllers can provide behavior without writing any code or using a ROS node. For example:

- The 'lasernoise' controller: simulates noise generated in a robot's laser sensor
- The 'pioneer_ flocking' controller: enabling flocking behavior for a number of pioneer robots

4.1.4 Integration with Ground Control Stations

Stage is not designed for real-time simulation, which is an essential feature for Hardware In The Loop (HITL) or Software In The Loop (SITL) scenarios, for this reason an integration with ground control stations is not supported.

4.1.5 Other configuration parameters

Table 1 contains the argument, which can be passed to Stage to configure it.

Table 1 - Stage parameters

Parameter	Description
q	If set, Stage will run "headless", i.e. no Graphic User Interface (GUI) will be displayed

4.2 Gazebo

Gazebo is one of the most used simulators based on the ROS architecture. It provides a 3D robotics environment that takes into account features like gravity, friction, inertia or collision detection. It is open-source[14].

The ROS community has devoted much effort in developing specific packages for the integration of Gazebo with ROS[15], to the point of integrating the program within the distribution itself. Gazebo is integrated into the ROS distribution since the Hydro version.

In addition to simulation, Gazebo also offers emulation of the sensors and actuators, which are totally integrated within ROS. This integration is done through plugins that tie in ROS messages and services calls for sensors outputs and motors inputs.

The main idea of these features is to respond like real robots without changes to the robot code. In this way the developers can test algorithms, design robots or train AI systems in a simulation environment, in the same way they would within real robots.

4.2.1 Control interface

Gazebo uses a distributed architecture with separate libraries for physics simulation, rendering, user interface, communication, and sensor generation. Additionally, it provides two executable programs for running simulations:

- A server *gzserver* for simulating the physics, rendering, and sensors.
- A client *gzclient* that provides a graphical interface to visualize and interact with the simulation.

Gazebo can also be launched form the command line by running

```
Gazebo.exe
```

in Windows, or

```
gazebo
```

in Linux and macOS. In this way the server and the client are launched simultaneously.

By modifying a set of variables, the user can add new models or worlds. These variables are listed in Table 2.

Table 2 - Gazebo environment variables

Variable	Description
GAZEBO_MODEL_PATH	Colon-separated set of directories where Gazebo will search for models
GAZEBO_RESOURCE_PATH	Colon-separated set of directories where Gazebo will search for other resources such as world and media files.

[14] https://bitbucket.org/osrf/gazebo/.
[15] http://wiki.ros.org/gazebo_ros_pkgs

GAZEBO_MASTER_URI	Uniform Resource Identifier (URI) of the Gazebo master. This specifies the IP and port where the server will be started and tells the clients where to connect to.
GAZEBO_PLUGIN_PATH	Colon-separated set of directories where Gazebo will search for the plugin shared libraries at runtime.
GAZEBO_MODEL_DATABASE_URI	URI of the online model database where Gazebo will download models from.

Gazebo offers two kinds of API, one related to the source code of Gazebo and one related to Gazebo Messages[16]. These APIs are designed for the development rather than for the interaction with Gazebo.

The interaction with Gazebo simulation happens through ROS messages. The most common way is use ROS topics and services that the '*gazebo control plugin*' offers. As already mentioned, the topics correspond to the those of the real robot.

Gazebo can also be executed directly as a ROS node, by using the following command

```
roslaunch gazebo_ros my_world.launch
```

In this case, it needs a *roscore* node launched before. From the ROS side, Gazebo starts a node called '/gazebo' which provides all the topics and services of the simulation.

4.2.2 Simulation settings

To set up the entire environment of the simulation, Gazebo uses an eXtensible Markup Language (XML) file named world. This term is used to describe the collection of robots and objects (buildings, tables, lights, etc) and global parameters like the sky, the ambient light and the physics properties.

There are two options to define a world:

1. From the Interface: The user can add some primitives like spheres or cubes to the world by clicking icons (see Figure 4) and then setting their poses in the scenario (see Figure 5). Also, by selecting the 'Insert' tab in the upper left-hand corner, the user can access the model database of Gazebo where there are more complex models.

[16] http://gazebosim.org/api.html

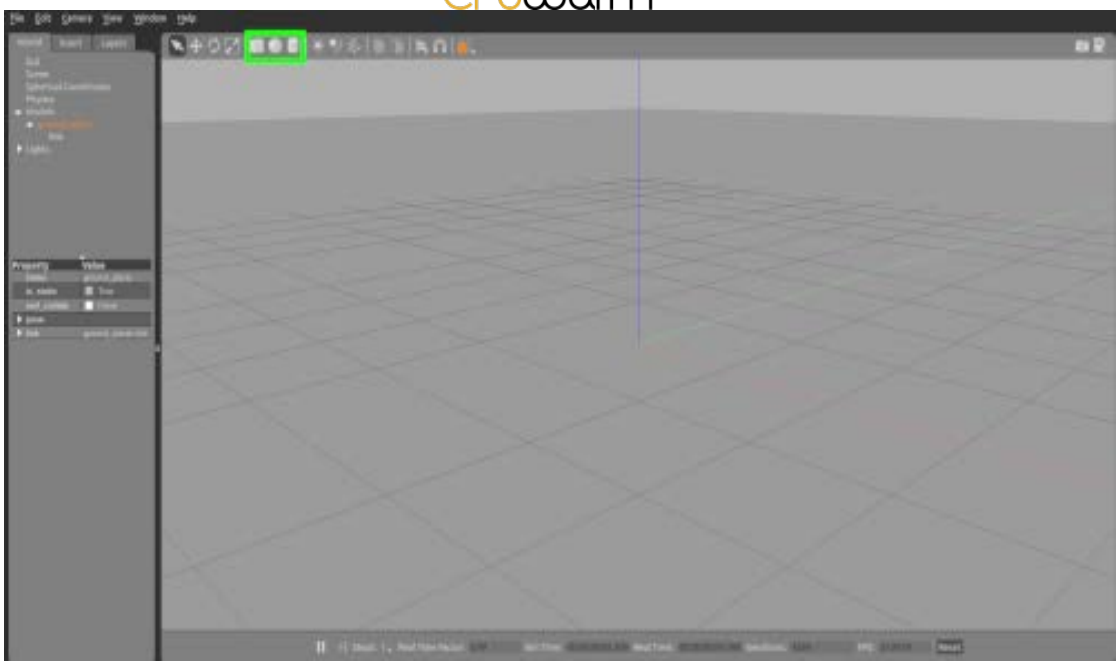

Figure 4 –Icons in Gazebo interface

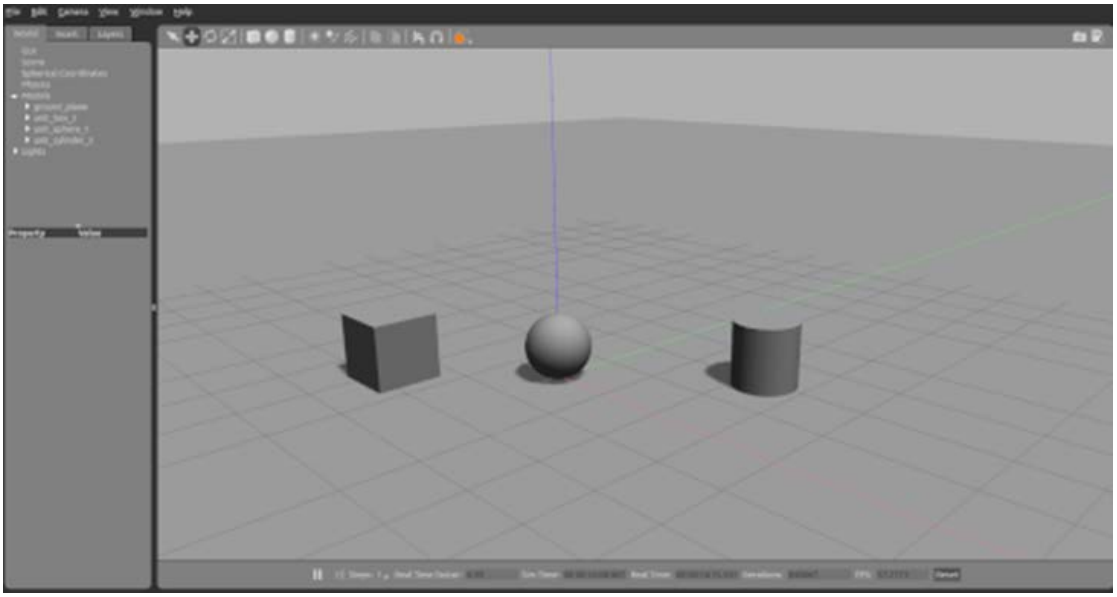

Figure 5 - Objects posed in Gazebo interface

2. From an XML file: The user can write a plain text XML file adding as a plain text the models and the properties needed, using the Simulation Description Format (SDF) (see Section 6.1.3).

New 3D models can also be integrated into the world. They can be designed using the Model Editor of Gazebo by combining primitives or imported from a STereo Lithography (STL) standard file[17].

By default, Gazebo launches the 'ODE' physics engine[18], but this can be changed in the world XML file to other physics engine like 'Bullet'[19], 'Simbody'[20] or 'DART'[21].

[17] http://gazebosim.org/tutorials?tut=guided_b3

Once the world is ready, it can be saved by selecting 'Save World As' in the 'File' Menu. A saved world may be loaded on the command line:

```
gazebo my_world.world
```

or setting the launch file as mentioned above.

The last point in configuring the simulation settings is to spawn the robots. There are two ways to do it:

1. ROS Service Call Spawn Method: The first method keeps the robot's ROS packages more portable between computers and repository check outs. It allows you to keep the robot's location relative to a ROS package path, but also requires making a ROS service call using a small (Python) script.

2. Model Database Method: The second method includes the robots within the world file, which seems cleaner and more convenient but requires adding the robots to the Gazebo model database by setting an environment variable.

4.2.3 Implementation of the CPS behavior

While using Gazebo, the CPS behavior will reside in C++ or Python code on the side of ROS, so the implementation remains external to Gazebo.

4.2.4 Integration with Ground Control Stations

A typical Simulation Environment that integrates the Control Ground Station and Gazebo is shown in Figure 6. It uses PX4[22] in SITL mode. The components of the system connect to each other using User Datagram Protocol (UDP), and may be located on the same machine or distributed.

PX4 listens on UDP port 14560. Gazebo connects itself to this port, then exchange information using the Simulator Micro Air Vehicle Link (MAVLink)[23] API, which defines a set of MAVLink messages that can be used to supply sensor data received by the simulator environment and return motor and actuator values. Then, PX4 uses the normal MAVLink module to communicate with the Ground Stations (which listen on port 14550) and external developer APIs like ROS (which listen on port 14540).

This can also be done using the HITL approach. where the PX4 software is running on an actual device.

[18] www.ode.org
[19] https://pybullet.org/wordpress/
[20] https://simtk.org/projects/simbody/
[21] http://dartsim.github.io/
[22] http://px4.io/
[23] https://mavlink.io/en/

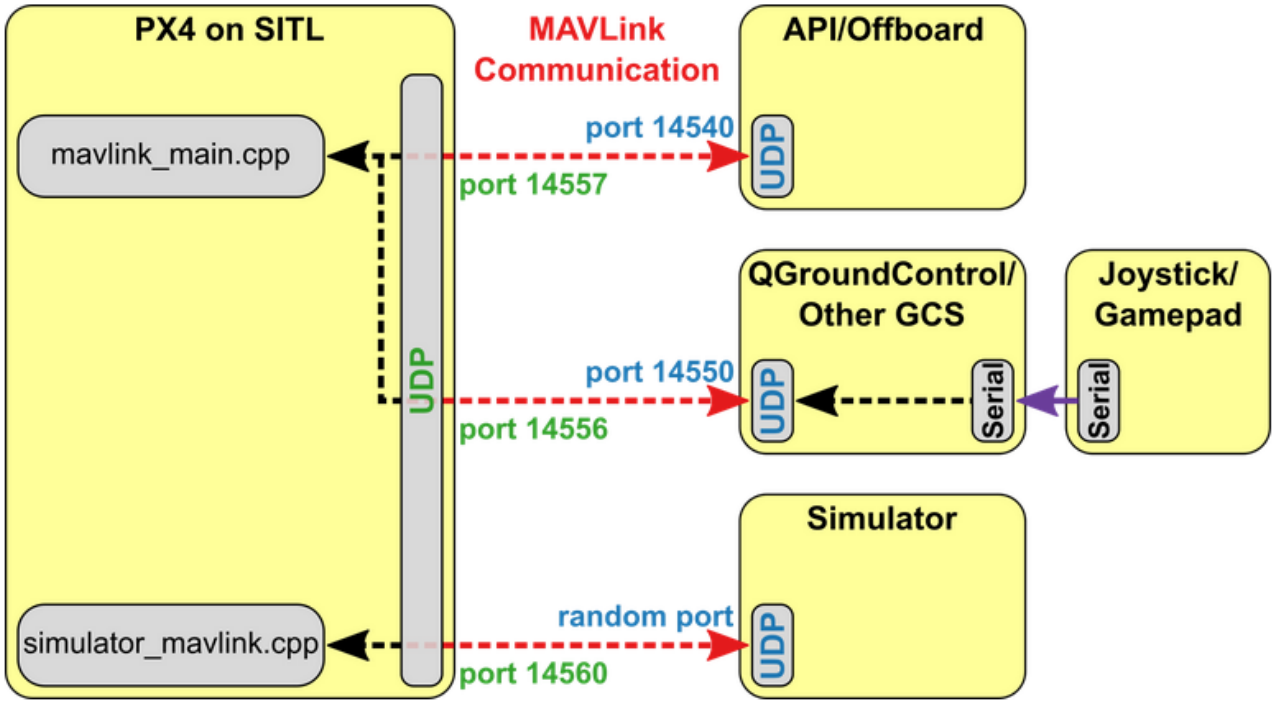

Figure 6 - Configuration to use Ground Control Station with Gazebo

4.2.5 Other configurations

Table 3 lists the arguments that can be used to configure the Gazebo simulation when starting it from ROS via the command line:

Table 3 - Gazebo parameters

Parameter	Description
paused	Starts Gazebo in paused state (default false)
use_sim_time	Tells ROS nodes asking for the time to get the Gazebo-published simulation time, published over the ROS topic/clock (default true).
gui	Launches the user interface window of Gazebo (default true).
recording	Enables Gazebo state log recording
debug	Starts gzserver in debug mode using gdb (default false)

For example, to launch the simulator with all the parameters:

```
roslaunch gazebo_ros my_world.launch paused:=true use_sim_time:=false gui:=true throt
tled:=false recording:=false debug:=true
```

Also, these parameters can be defined in a launch XML file, as shown in Figure 7.

```
<launch>
  <!-- We resume the logic in empty_world.launch, changing only the name of the world to be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="worlds/mud.world"/> <!-- Note: the world_name is with respect to GAZEBO
e -->
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="recording" value="false"/>
    <arg name="debug" value="false"/>
  </include>
</launch>
```

Figure 7 - Gazebo launch file

4.3 V-REP

The Virtual Robot Experimentation Platform (V-REP) is a 3D robot simulator. It also provides a development environment, which can be used to configure and simulate any robot.

Its functionalities can be integrated and combined smoothly, leveraging the API and script functionality provided. V-REP supports two physics engines: Bullet[24] and Open Dynamic Engine (ODE)[25], handling advanced features, like camera sensor simulation, graphing, etc.

4.3.1 Control interface

V-REP can be run from the command line using

```
vep.exe
```

on Windows, or

```
/vrep.app/Contents/MacOS/vrep'
```

on macOS or

```
./vrep.sh
```

on Linux.

V-REP provides an API framework, which may be used to control a simulator instance in several ways, as shown in Figure 8. The regular API is used to control the framework from V-REP internal components. While different APIs may be used, from external entities. The two most interesting are:

- The remote API (based on socket), which can be used to control the simulator from applications written in several languages (Java, Python and C++, among others).
- The ROS interfaces, which can be used to control the simulator using ROS.

The remote APIs are subdivided in two parts: the client side and the server side. The client side is composed of software libraries, written in several languages, which can be integrated into the application to control the

[24] http://bulletphysics.org/wordpress/
[25] http://www.ode.org/

simulator remotely; the server side instead is a plugin that can be automatically loaded into the simulator to enable the API.

V-REP also provides ROS interfaces, which duplicate the Remote API with a good fidelity, enabling V-REP to be used as a ROS node, which can communicate with other nodes via ROS services, ROS publishers and ROS subscribers. This feature is enabled in V-REP using a specific plugin, which can be loaded automatically when V-REP is launched, but that requires *roscore* running concurrently.

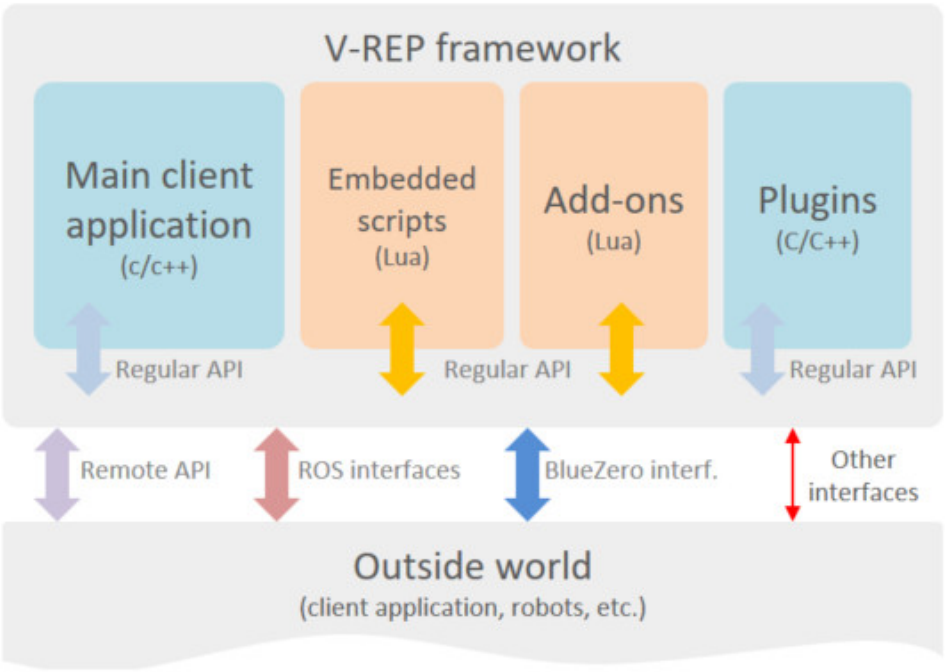

Figure 8 - V-REP framework

4.3.2 Simulation settings

The simulation settings are handled using scenes and models files, where a scene can contain one or more models. The scenes files are saved as *ttt* files, while the models files are saved as *tttm* files. Both scenes and models can contain several elements, such as:

- Objects: main elements used to compose a simulation scene, such as types: shapes, cameras, lights and so on.
- Child scripts: a set of routines written in Lua[26], which handle a specific function in a simulation. Every child script can be attached or associated with a scene object. Every child script can have a list of simulation parameters attached to them, which can be used to adjust values of a specific simulation model (like maximum velocity of a robot or resolution of a sensor).

Furthermore, only the scenes can contain:

- The environment that defines properties and parameters like background colors, ambient light and additional settings.

[26] https://www.lua.org/

- The main script, which Is the basic code that allows a simulation to run. It contains all the functions, which are called by the system to run the simulation. Default main scripts are provided by V-REP, it is better to avoid modifying them, because it is a risky operation that can break the entire simulation and because if not modified the main scripts are updated automatically when new functionality is introduced.

Unfortunately, currently the files are saved in binary format and can only be modified using V-REP. The V-REP developers[27] currently suggest using a workaround that involves creating a script, using the V-REP API to modify an existing scene. The development of a text-based file format has been declared in the same discussion as ongoing, but seems to have been postponed.

4.3.3 Implementation of the CPS behavior

The behavior of the objects is handled through embedded scripts, written using the Lua language attached to the objects of the scene. More dynamic behaviors of the CPSs, can exploit the plugins mechanism. These plugins can be written in any language that can generate a shared library and is able to call exported C functions. They are automatically loaded by the application at startup (if contained in a specific folder), or they can be loaded/unloaded dynamically at run-time calling two specific APIs. The plugins are usually used in combination with scripts, in order to make them more dynamic. Instead of embedding the behavior in the script, it contains only the calls to the plugin, which contains the actual implementation of the functions.

4.3.4 Integration with Ground Control Stations

Currently, V-REP doesn't support a MAVLink plugin. For this reason, the simulator is not compatible with the more commonly used ground control stations. Anyway, a ground control station can be interfaced using the ROS interface provided by V-REP.

4.3.5 Other configuration parameters

Table 4 shows some of the parameters that can be specified while launching the V-REP instance.

Table 4 - V-REP parameters

Parameter	Description
h	Runs V-REP in headless mode (i.e. without any GUI).
sXXX	Automatically starts the simulation. XXX represents an optional simulation time in milliseconds after which simulation should stop again.
q	Automatically quits V-REP after the first simulation run has ended.
DEBUG	Enables or disables the debug mode (set to *TRUE* or *FALSE*).

4.4 ARGoS

Autonomous Robots GO Swarming (ARGoS) is a multi-physics robot simulator designed to efficiently simulate complex experiments involving large swarms of heterogeneous robots. Plugins can be added to

[27] http://www.forum.coppeliarobotics.com/viewtopic.php?t=2319

customize ARGoS. For example, multiple physics engines may be used concurrently and assigned to different regions of the physical space. As robots navigate the environment, they are automatically transferred from engine to engine.

ARGoS can be installed on Linux using binary packages. For macOS, it can be done via the command line using a third-party package manager. It may work on Windows 10 using the Linux sub-system, however this has not been confirmed. Further documentation about installing ARGoS is provided on the ARGoS webpage and Github.

4.4.1 Control interface

A simulation, such as the diffusion 1 simulation, can be launched by entering the command:

```
launch argos3 -c xml/diffusion 1.xml
```

4.4.2 Simulation settings

The settings for an experiment in ARGoS are described in an XML configuration file. As illustrated in Figure 9, it includes the information about:

a. Framework: ARGoS parameters like the number of threads and the base random seed.

b. Controllers: configuration of user-defined controllers.

c. Arena: how objects are distributed in the simulated environment.

d. Physical_engines: the configuration of how the engines are connected to each other.

e. Media: configures the communication media to be used.

f. Visualization: an optional feature to add visualization.

g. Loop_function: an optional feature with user-defined conditions.

```
<?xml version="1.0" ?>
<argos-configuration>
  <framework>
  ...
  </framework>
  <controllers>
  ...
  </controllers>
  <arena size="2, 2, 1">
  ...
  </arena>
  <physics_engines>
  ...
  </physics_engines>
  <media>
  ...
  </media>
  <visualization>
  ...
  </visualization>
  <loop_functions library="/path/to/libmy_loop_functions.so" label="my_loop_functions" />
</argos-configuration>
```

Figure 9 - A general configuration file in ARGoS

4.4.3 Implementation of the CPS behavior

As described in Figure 10, ARGoS requires two kinds of input[28]:

1- The XML configuration *argos* file, introduced in Section 4.4.2.

2. Robot controllers and loop functions files, which are basically user code compiled into one or more libraries. Both files are C++ implementations of robot swarm behavior.

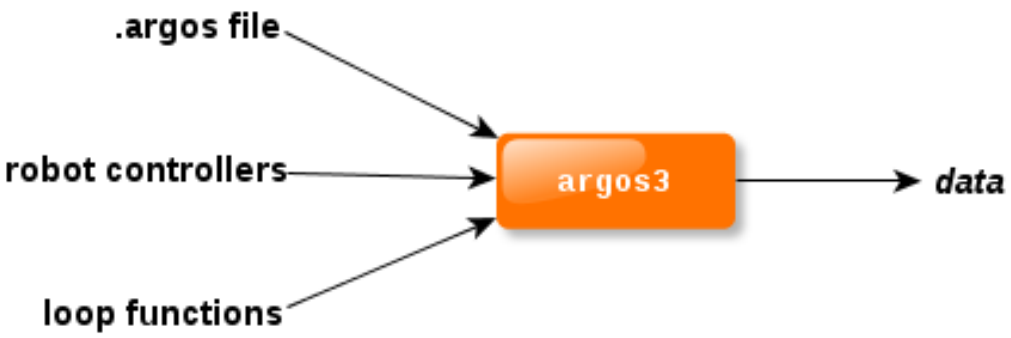

Figure 10 - Input files for ARGoS

4.4.4 Integration with Ground Control Stations

Since simulations with ARGoS are not specialized for real-time simulation, which is an essential feature for hardware-in-the-loop scenarios (Lächele, 2013), an integration with ground control stations is not supported.

[28] http://www.argos-sim.info/user_manual.php

4.4.5 Other configuration parameters

- Optionally, the POV-Ray package can be installed on Ubuntu and KUbuntu to provide high-quality visualizations.
- In the XML file, Framework section, there are two attributes, both optional, shown in Table 5.

Table 5 - ARGoS parameters

Parameter	Description
threads	Number of threads to execute each simulation step in parallel.
method	Indicate task assignment method for threads.

4.5 jMAVSim

jMAVSim is a simple drone simulator, which allows vehicles running PX4 autopilot to be simulated within a simulated space. It is easy to set up and can be used to test both the drone's features and its capability to react to failures. It can work both in SITL and HITL mode. In the first case the PX4 code is installed and run locally on the simulating machine, while in the latter case, the code runs on the drone.

4.5.1 Control interface

jMAVSim is a Java application that can be executed as a standard Java application via the command line or launched by compiling a SITL instance of PX4. jMAVSim doesn't provide an API but, can be controlled via the command line also during the simulation.

4.5.2 Simulation settings

The environment is statically modelled using an image file.

The CPSs are modeled using a the WaveFront OBJ file format[29], which contains the list of polygons to be shown, described listing their points and their faces. There is no description of the colors of the faces, but the OBJ files can contain a link to material files (*mtl*) that contain the definitions for the RGB values for the material's diffuse, ambient, and specular colors, along with other characteristics such as specularity, refraction, transparency, etc.

4.5.3 Implementation of the CPS behavior

jMAVSim can integrate the CPS behavior into a ROS node, which communicates with the jMAVSim instance.

4.5.4 Integration with Ground Control Stations

The integration with ground control stations is done in the same way described in Section 4.2.4 for the Gazebo simulator and also in this case, it works with both SITL and HITL approaches.

4.5.5 Other configuration parameters

Table 6 shows some parameters, which can be indicated launching the jMAVsim instance.

[29] http://www.fileformat.info/format/wavefrontobj/egff.htm

Table 6 - jMAVSim parameters

Parameter	Description
udp - <mav ip>:<mav port>	Opens a TCP/IP UDP connection to the MAV (default: 127.0.0.1:14560).
serial [<path> <baudRate>]	Opens a serial connection to the MAV instead of UDP.
r <Hz>	Refreshes rate at which jMAVSim runs. This dictates the frequency of the HIL_SENSOR messages. Default is 500 Hz.
ap <autopilot_type>	Specifies the MAV type. E.g. 'px4' or 'aq'. Default is: generic
qgc <qgc ip address>:<qgc peer port>	Forwards message packets to QGC via UDP at 127.0.0.1:14550
Rep	Starts with data report visible.

4.6 STDR

Simple Two Dimensional Robot Simulator (STDR) implements a distributed, client-server architecture, where the nodes can run in a different machines and communicate using ROS interfaces. STDR Simulator, also provides a GUI developed in QT, for visualization purposes and more. Furthermore, STDR provides also a GUI, which can be optionally used.

The STDR simulator is composed by a set of ros packages. The main ones are: *stdr_server* that implements synchronization and coordination functionalities of STDR Simulator. *stdr_gui,* which is a GUI for visualization purposes in STDR Simulator. *stdr_msgs* that provides messages, services and actions for STDR Simulator. *stdr_launchers*, which launches files, to easily bring up server, robots and the GUI. *stdr_resources that p*rovides robot and sensor description files for STDR Simulator.

4.6.1 Control interface

STDR can be controlled through command line commands. The package *stdr_launchers* provides several default ways to launch a server. The basic way to launch the server is:

```
roslaunch stdr_launchers server_no_map.launch
```

which launches the STDR simulator without map and robots preloaded. Furthermore, there are other launchers that add default features to STDR: a random map (*server_with_map*), the random map and the GUI (*server_with_map_and_gui*) and with map, gui and a robot (server_with_map_gui

STDR provides also several other commands that can be used to execute several actions:

- To load a map stored in a *yaml* file, the command is

```
rosrun stdr_server load_map maps/sparse_obstacles.yaml
```

- To add a robot (described by a *yaml* file) in a precise position the command is

```
rosrun stdr_robot robot_handler add resources/robots/pandora_robot.yaml 9 7
```

- To launch the GUI, the command is

```
roslaunch stdr_gui stdr_gui.launch
```

4.6.2 Simulation settings

The models to be used for STDR Simulator are contained in the *stdr_resources* package. Specifically, this is subdivided into two main types:

- Maps: every map consists of a couple of files: a static *png* and a *yaml* file with the properties of the map (source, location of obstacles, etc.).
- YAML/SDF files: descriptions of robots, laser sensors, range sensors and footprints.

Both the maps and the models of the robots can be customized by hand, writing the *yaml* file, following the examples available online, or using a software: the maps can be created using gmapping[30] and the robots can be created using directly the STDR GUI[31].

4.6.3 Implementation of the CPS behavior

STDR Simulator is designed to be completely ROS compliant. Indeed, robots and sensors sends all the measurements on ROS topics. In that way, STDR uses all ROS advantage provided by the robotic framework.

4.6.4 Integration with Ground Control Stations

STDR is a simple simulator, which is not designed to support advanced features like real-time simulation with HITL or SITL, for this reason an integration with ground control stations is not supported.

4.6.5 Other configuration parameters

STDR doesn't provide the possibility to specify additional arguments to configurate the simulation.

[30] http://wiki.ros.org/gmapping
[31] http://wiki.ros.org/stdr_simulator/Tutorials/Robot%20creator

5 Simulation Virtual Machine

After the analysis of ROS based simulator, the partners have decided to start the integration of external simulators, by integrating Gazebo. This simulator has been chosen because it provides all the features needed, allows the models of the environment and of the CPS to be described using an XML-like format, and to control the simulation via the command line. Furthermore, it also provides a GUI, which can be used for visualization of the simulation.

For this reason, a virtual machine has been built to provide a complete environment available to test the Gazebo integration. Since Gazebo is natively a Linux application, the virtual machine has been based on Ubuntu 16.04 LTS. The following packages have been installed to build a complete simulation environment:

- PX4 flight control stack version 1.8, consisting of two main layers: the flight stack that is an estimation and flight control system, and the middleware that consists primarily of device drivers for embedded sensors, communication with the external world (companion computer, Ground Control Station, etc.) and the uORB[32] publish-subscribe message bus. In addition, the middleware includes a simulation layer that allows PX4 flight code to run on a desktop operating system and control a computer modeled vehicle in a simulated "world".
- ROS Kinetic[33]: Kinetic Kame is the 10th official ROS release. It is supported on Ubuntu Wily and Xenial.
- MAVROS[34]: this package enables MAVLink extendable communication between computers running ROS and MAVLink enabled autopilots like PX4. Additional it provides UDP MAVLink bridge for ground control stations (e.g. QGroundControl)
- Gazebo 7.0: the simulator as presented in Section 4.2.
- jMAVSim: the simulator described in Section 4.5.
- OpenCV[35]: Open Source Computer Vision Library is an open source library that provides more than 2500 optimized algorithms, spawning from a set of both classic and state-of-the-art computer vision and machine learning algorithms.
- QGroundControl[36]: open-source software that provides full flight control and mission planning for any MAVLink-enabled drone.

Whit this software, the Virtual Machine can be used to create an environment similar to the one showed in Figure 6, which allows simulation with SITL or HITL, both with Gazebo or jMAVSim.

[32] https://dev.px4.io/en/middleware/uorb.html
[33] http://wiki.ros.org/kinetic
[34] http://wiki.ros.org/mavros#mavros.2BAC8-Plugins.sys_status
[35] https://opencv.org/
[36] http://qgroundcontrol.com/

6 Implementation and Integration

This section describes the integration of the initial simulation environments into the CPSwarm Workbench, firstly describing the custom and standard data formats chosen and then describing the implementations themselves. The latter has been subdivided in two parts: the first describes the implementation of the API linking the Optimization Tool, SOO and Simulation Managers; the second describes the Simulation Manager code shared among all the Simulation Managers, and the implementation of the Gazebo Simulation Manager.

Please note that, as indicated in Section 2.1 the description of the whole Simulation and Optimization Environment architecture and API is material of the deliverable D6.2 – Final Simulation Environment (due at month 28); for this reason, this section describes only the implementation of the parts relative to the simulator integration and not the interaction of the components with the rest of the CPSwarm Workbench.

6.1 Data formats of the simulation interfaces in the CPSwarm Workbench

Studying the data formats collected in the state-of-the-art described in the deliverable D6.1 and the formats used by the simulation engines described in Section 3, the authors have decided to adopt mainly two solutions: the SDF standard[37], where it is suitable, specifically for the description of the CPS and environmental model; and XML and JavaScript Object Notation (JSON) documents in the other cases. Where it is not possible to use plain text communication, the components exchange files. For example, the CPS behavior is described using a candidate (C code, generated by the Optimization Tool, which describes the actual CPS behavior, returning the outputs from the inputs received) and a wrapper (Java code, which is used to collect the inputs from the sensors, using ROS).

6.1.1 JSON

Several custom JSON formats have been defined for the messages exchanged among the different components. These formats are described in the following subsections.

6.1.1.1 StartOptimization

This message is sent from the SOO to the Optimization Tool to start an optimization process. ANNEX F contains the schema of the *StartOptimization* message. This message basically instructs the Optimization Tool about how to start the optimization. It indicates the number of threads to be used, the parameters to be used for the simulations and finally the list of the accounts of the Simulation Managers to be used by the Optimization Tool.

6.1.1.2 GetProgress and CancelOptimization

These two messages sent from the SOO to the Optimization Tool have a common format (ANNEX G), which contains a title (*GetProgress* or *CancelOptimization*) and the ID of the optimization to query.

6.1.1.3 OptimizationResult

This message is used by the Simulation Managers to return the simulation results to the Optimization Tool, using the format shown in ANNEX H that contains the ID of the simulation and the fitness value calculated using the fitness function.

[37] http://sdformat.org/

6.1.1.4 OptimizationStarted, OptimizationCancelled and OptimizationProgress

These three messages are sent by the Optimization Tool to the SOO as a reply to *StartOptimization, CancelOptmization* and *GetProgress*. They share the same schema (shown in ANNEX I) with the ID of the optimization and a message, which can be:

- "OK": if everything has gone well, as answer of *StartOptimization and CancelOptmization* messages.
- A percentage: as positive answer to a *GetProgress* message.
- or an error message if something has gone wrong for all the messages.

6.1.1.5 RunSimulation

This message is sent by the Optimization Tool to the Simulation Manager to run a simulation and consists of an ID and the parameters to be used for the simulation (such a flag indicating if the GUI should be displayed or not), The format of the message is given in ANNEX J.

6.1.2 XML

For the configuration of the Optimization Tool, a custom XML format has been used. It is described in the following subsection.

6.1.2.1 Optimization Tool configuration

The Optimization tool is configured using an XML format (see the *dtd* in ANNEX A). This formalism contains the following tags:

- *SessionConfig*: it contains all the generic parameters used to configure the Optimization Tool, each one is described by a <configurationEntry>, which contains key, type and value.
- *Problem*: it contains the parameters for the definition of the problem to be optimized by the Optimization Tool, each one is described by a <problemEntry>, which contains key, type and value.
- *Method*: it contains the parameters for the definition of the method to be used to evolve the algorithm in the Optimization Tool, each one is described by a <methodEntry>, which contains key, type and value.
- *Representation*: it contains the parameters for the creation of the candidates to be evaluated by the Optimization Tool, each one is described by a <representationEntry>, which contains key, type and value.
- *Ranking*: it contains the parameters for the definition of the ranking to be used by the Optimization Tool to evaluate the results of the simulation, each one is described by a <rankingEntry>, which contains key, type and value.

6.1.2.2 Optimization Tool result

The result of one optimization process uses an XML format like the one used for its configuration ((see the *dtd* in ANNEX B). Indeed, many fields are identical to those described in Section 6.1.2.1; the only difference is the *populations* tag, which describes the populations created, each one described with three parameters: *count, generation* and *randomseed*.

6.1.3 SDF

SDF is an XML format for describing CPSs and environments for robot simulators. This format was designed as part of Gazebo, with scientific robot applications in mind and due to its flexibility has become a de-facto standard, for describing all aspects of a simulation, like robots, static and dynamic objects, lighting, terrain, and even physics.

SDF may be used to describe all aspects of a robot, including complex ones, like a humanoid. Many properties can be defined, such as kinematic and dynamic attributes, sensors and textures among others. For this reason, SDF can be used for both simulation, visualization, motion planning, and robot control.

In addition, SDF provides a format to define environments, which can include multiple lights, terrain, streets from OpenStreetMaps[38], and any model provided from an online repository of 3D models[39].

Among others, SDF is composed of the following modules:

- World: a container for all the other modules.
- Scene: generic features of the world, like ambient light or color of the background.
- State: a description of the current state of the simulation.
- Light: a description of a light source.
- Model: a description of a complete CPS or any physical object.

In addition to the XML format, the SDF website provides also a C++ library that can be used to parse SDF documents.

In CPSwarm this format is used to pass the environment and CPS models from the Modelling Tool to the SOO and to the Simulation Managers. This document provides some examples of SDF files:

- ANNEX C contains an example of a *world* file, where the main tags are: <include>, which are the environment models included in this world; and <model>, which contains the CPS to be integrated in the world, eventually with the associated plugins.
- ANNEX D contains an example of an environment model, where the main tags are <link>, which describes a physical link with the inertia, collision and visual properties; <visual>, which specifies the shapes of the environment for visualization properties; and <collision>, which contains the physical properties of the environment and its position.
- ANNEX E contains an example of an object model, in this case a model of Velodyne LIDAR HDL-32E[40]. The model is composed by the following components:
    - Two <link> tags, which describe the base part and the top part of the sensor and its physical and visualization properties.

[38] https://www.openstreetmap.org/
[39] http://thepropshop.org/
[40] http://velodynelidar.com/hdl-32e.html

- A <sensor> tag, which describes the laser sensor properties (including a model of the noise that influence the sensor, contained in the <noise> tag).
- A <joint> tag, which describes the joint between the two links, in this case the joint that allows the top part of the model to rotate around the base part.

6.1.4 Wrapper

The wrapper contains methods useful for coordinating the swarms behavior, such as synching several agents, through ROS messages. The main methods are described in the following subsections.

6.1.4.1 Main()

Main subscribes the agent to the ROS topics needed to receive messages (for example to information about the map used). Then the method calls the *sync* method to wait that all the agents in the swarm are able to start the simulation. Then the method starts the main loop, which iterates until the agent reaches the exit or the maximum number of allowed simulation steps is reached.

6.1.4.2 Sync()

Publishes a *sync* message, advising the other agents that this one is ready to move.

6.1.4.3 SyncUpdate()

This callback is used to receive the *sync* messages from the other agents. It collects the messages and keeps count of ready agents. When all other agents are ready, it calls *simStep* method.

6.1.4.4 simStep()

Firstly, it resets the number of ready agents, then it collects the input to be passed to the candidate, using a combination of the info already available and info obtained from the sensors, through ROS messages. For example, in the emergency exit algorithm, it uses the positions of the exits to calculate the closest exit, to the CPS position and uses the laser sensors to detect if there is another CPS in the cells nearby. These inputs are passed to the candidate that returns the outputs, which are used to call the appropriate ROS actions, and provide the actuations needed on the simulated CPSs. Finally, it publishes again a *sync* message, to advertise the other agents that it is ready for another step, and then it waits again until all other agents are ready.

6.1.5 Candidate

The Candidate is the actual code that controls the CPS behavior. It is generated by the Optimization Tool, based on the underlying representation being optimized, such as an Artificial Neural Network (ANN). The main member of one candidate is the one described in the following subsection.

6.1.5.1 getOutput(float netInput[],long inputsize)

The *getOutput* method is called by the wrapper to get the output of the representation from the given inputs. The parameters of the method are *newInput* that contains the vector of the inputs and *inputSize* that contains the number of inputs passed. For the emergency exit example, the candidate receives the following inputs: x and y distance from the closest exit and, an indication of the state of the four neighboring cells (with 0/1 values). The candidate passes the result back to wrapper. In the proposed example, the result is the direction in which to move, to reach the closest exit avoiding the other CPSs of the swarm.

6.2 Integration of FREVO as an Optimization Tool

FREVO is utilized as an example Optimization Tool within the CPSwarm project, providing evolutionary optimization as outlined in D6.3 - *Initial CPS System design optimization and fitness function design guidelines*. Rather than offering its usual user interface, FREVO runs headless and handles requests for optimization via XMPP, exchanging data as described in Section 6.1.

While in this mode, FREVO is capable of executing any optimization process that it would normally be able to carry out with the restriction that the Problem tag must be set to *XmppProblem*. This generic XMPP problem specification allows FREVO to forward candidate controllers to the various Simulation Manager instances to be embedded directly in the simulators. The Method, Representation and Ranking may be freely specified and, due to FREVO's unique plugin architecture, may be extended with custom components, e.g. additional representations.

6.3 SOO

As indicated before, this section doesn't contain the complete description of the SOO, which will be fully detailed in D6.2, but only the description of the implementation of the interfaces with the Optimization Tool and the Simulation Managers.

The SOO contains the main class *SimulationOrchestrator*, which has the following parameters.

- serverIP: IP of the XMPP server to be used.
- serverName: name of the XMPP server to be used**.**
- serverPassword: password to be used to connect to the server.
- dataFolder: folder to be used to store the data.
- optimizationUser: XMPP username of the Optimization Tool.

Firstly, the *SimulationOrchestrator* starts an XMPP client, attaching to the connection all the useful listeners (described in the next section). Then, it adds to its roster Optimization Tool, which JID is known thanks to the data passed as parameters. Finally, it waits to receive the request to start a simulation / optimization and the availability from the Simulation Managers.

6.3.1 Listeners

6.3.1.1 ConnectionListenerImpl

This listener is used to receive the notifications of the status of the connection, in order to be able eventually to react when the connection goes down (or, at least, to notify the user).

6.3.1.2 MessageEventCoordinatorImpl

This listener is used to receive the chat messages sent by the Optimization Tool (described in Section 3.1.1). It handles the messages, taking the decisions based on this. If it is a positive *OptimizaitonStarted* message, it sends the configuration file to the Optimization Tool; if it is a positive *OptimizationCanelled*, it reset the status of the current optimization; if it is positive *optimizationProgress,* it forwards the message to who is required it. Finally, in case of errors, they are logged and handled accordingly.

6.3.1.3 PacketListenerImpl

This listener is used to receive the presences from the Optimization Tool and the Simulation Managers, to accept their requests of subscription of the presences and to collect the info of the Simulation Managers when they are received as status of the presence.

6.4 Simulation Manager

The Simulation Manager is the module, which contains all the parts the code, common to all the managers. The main class of the Simulation Manager is *SimulationManager*. This is an abstract class implemented by all the managers. It has the following parameters:

- *serverIP*: IP of the XMPP server to be used.
- *serverName:* name of the XMPP server to be used.
- *serverPassword:* password to be used to connect to the server.
- *dataFolder:* folder to be used to store the data.
- *serverInfo:* info of the simulator wrapped by this manager.
- *optimizationUser:* XMPP username of the Optimization Tool.
- *orchestratorUser:* XMPP username of the SOO.

The *SimulationManager* encapsulates an XMPP client (based on the open-source XMPP library smack[41]); firstly, it connects itself to the XMPP server and it creates the account with the ID of the manager that is constituted by the term "manager_" followed by a random UUID. It adds to the connection a set of listeners (detailed in the following subsection) and then, it publishes a presence, which contains in the status the info of the simulator, wrapped by this manager (i.e. dimensions supported, maximum number of agents, etc.). Finally, the manager adds to its roster[42] the Optimization Tool and the SOO. In this way, the manager subscribes itself to the presences of Optimization Tool and SOO and vice versa.

The following subsection details the listener classes used by the *SimulationManager* to receive the messages and presences from the other component.

6.4.1 Listeners

6.4.1.1 ConnectionListenerImpl

The same listener described in Section 6.3.1.1.

6.4.1.2 AbstractFileTransferListener

This listener is fired when a request to transfer files is received. The manager checks who is the sender: if it is the SOO, the files to be received are the ones to be used to setup the simulation; else, if it is the Optimization Tool, the file received it is the candidate which has to be evaluated. The class is abstract because the implementation of the methods used to handle the configuration files and the candidate is delegated to the

[41] https://www.igniterealtime.org/projects/smack/
[42] https://xmpp.org/rfcs/rfc6121.html#roster

specific Simulation Managers, since every Simulation Manager will need to do different operations or conversions.

6.4.1.3 PresencePacketListener

This listener is used to receive the subscription requests from Optimization Tool and SOO. It accepts the request, authorizing the exchange of the presences with the other components.

After the description of the parts common to all the Simulation Managers, the next section will introduce the first implementation of the one used to integrate the Gazebo simulator.

6.5 Gazebo Simulation Manager

The main class of this Simulation Manager is *GazeboSimulationManager* that instantiates its superclass indicating the *AbstractFileTransferListener* implementation to use.

6.5.1.1 FileTransferListenerImpl

It is the implementation of *AbstractFileTransferListener* for the Gazebo Simulation Manager. Since Gazebo uses natively SDF files both for the environment and CPS description and this eases a lot the integration of the simulator with the proposed solution, since this format is the same chosen to exchange the models as described in section 6.1.3. When the configuration files are received, for each SDF model file, the Simulation Manager creates a directory named as the name of the file and stores in the model file and the corresponding configuration file (which is an XML file with some metadata to associate to the model, like the name of the model and the SDF version used to describe it). Instead, the wrapper to be used to wrap the candidate and the *world* file that describes the simulation are stored in a local folder, named with the ID of the optimization process. Then, every time the Simulation Manager receives a new candidate, it uses the wrapper to compile the new plugin (using the command available to compile ROS packages[43]) and then it starts the simulation in Gazebo using the parameters indicated by the user, using a launch file, which allows to start all the nodes needed for the simulation.

[43] http://catkin-tools.readthedocs.io/en/latest/index.html

7 Conclusions

This deliverable has presented the work done in Task 6.4 for the integration of the external simulators in the CPSwarm Workbench. The deliverable firstly has briefly introduced the Simulation and Optimization Environment designed in CPSwarm. Then, it has described the analysis done on the more interesting simulation engines, among the ones collected as state-of-the-art in D6.1. Finally, the deliverable has presented the implementations done, based on the result of this analysis: firstly, the Virtual Machine built with all the software needed to test the Simulation Environment; then the data formats used to integrate the simulators and the software developed for the integration of the first simulation engine (Gazebo).

The deliverable will have two other iterations: deliverable D6.6 - *Updated Integration of external simulators* due at month 28 and D6.7 - *Final Integration of external simulators* due at month 36. These deliverables will contain the further developments that will be done in the next months in the Task 6.4. Specifically:

- Integration of other simulation engines, beside Gazebo (i.e. the other simulator analyzed in this deliverable).
- Continue refactoring of the implementations and the data formats to make it compatible with a large set of simulation engines.
- Integration of the work done in the Task 6.4, with the outcomes of the other tasks of WP6, related to the architecture of the Simulation and Optimization Environment and to the fitness function design.

Acronyms

Acronym	Explanation
API	Application Programming Interface
CPS	Cyber Physical System
ROS	Robot Operating System
CPU	Central Processing Unit
JSON	JavaScript Object Notation
ANN	Artificial Neural Network
SOO	Simulation and Optimization Orchestrator
UUID	Univers Unique IDentifier
ROS	Robot Operating System
SITL	Software In The Loop
HITL	Hardware In The Loop
MAVLink	Micro Air Vehicle Link
V-REP	Virtual Robot Experimentation Platform
ODE	Open Dynamic Engine
ARGoS	Autonomous Robots GO Swarming
STDR	Simple Two Dimensional Robot Simulator
XMPP	eXstensible Messaging and Presence Protocol
XML	eXtensible Markup Language
SDF	Simulation Description Format
STL	STereo Lithography
UDP	User Datagram Protocol
JSON	JavaScript Object Notation
URI	Uniform Resource Identifier
MQTT	Message Queue Telemetry Transport
SUMO	Simulation of Urban MObility
FREVO	FRamework for Evolutionary Design
GUI	Graphic User Interface
openCV	Open Source Computer Vision Library

List of figures

Figure 1 – The broker-based approach of the CPSwarm Worbench 7
Figure 2 - CPSwarm concept 9
Figure 3 - CPSwarm reference architecture 10
Figure 4 –Icons in Gazebo interface 16
Figure 5 - Objects posed in Gazebo interface 16
Figure 6 - Configuration to use Ground Control Station with Gazebo 18
Figure 7 - Gazebo launch file 19
Figure 8 - V-REP framework 20
Figure 9 - A general configuration file in ARGoS 23
Figure 10 - Input files for ARGoS 23

List of tables

Table 1 - Stage parameters 13
Table 2 - Gazebo environment variables 14
Table 3 - Gazebo parameters 18
Table 4 - V-REP parameters 21
Table 5 - ARGoS parameters 24
Table 6 - jMAVSim parameters 25

References

Conzon, D. T. (2012). The VIRTUS Middleware: An XMPP Based Architecture for Secure IoT Communications. *21st International Conference on Computer Communications and Networks (ICCCN)*, (pp. 1-6).

Lächele, J. M. (2013). "Swarm-simx and telekyb: Two ros-integrated software frameworks for single-and multi-robot applications.". *Int. Work. on Towards Fully Decentralized Multi-Robot Systems: Hardware, Software and Integration at 2013 IEEEInt. Conf on Robotics and Automation.*

Micha Rappaport, D. C. (2018). Distributed Simulation for Evolutionary Design of Swarms of Cyber-Physical Systems. *ADAPTIVE 2018*, (p. 6).

Staranowicz, A. a. (2011). "A survey and comparison of commercial and open-source robotic simulator software.". *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments,*, (p. p. 56).

ANNEX A – Optimization Tool Configuration

```
<!ELEMENT frevo (sessionconfig, problem, method, representation, ranking) >

<!ELEMENT sessionconfig (configentry)+ >

<!ELEMENT configentry EMPTY >

<!ATTLIST configentry
              key CDATA #REQUIRED
              type CDATA #REQUIRED
              value CDATA #REQUIRED >

<!ELEMENT problem (problementry)+ >

<!ATTLIST problem
              class CDATA #REQUIRED >

<!ELEMENT problementry EMPTY >

<!ATTLIST problementry
              key CDATA #REQUIRED
              type CDATA #REQUIRED
              value CDATA #REQUIRED >

<!ELEMENT method (methodentry)+ >

<!ATTLIST method
              class CDATA #REQUIRED >

<!ELEMENT methodentry EMPTY >

<!ATTLIST methodentry
              key CDATA #REQUIRED
              type CDATA #REQUIRED
              value CDATA #REQUIRED >

<!ELEMENT representation (representationentry)+ >

<!ATTLIST representation
              class CDATA #REQUIRED >

<!ELEMENT representationentry EMPTY >

<!ATTLIST representationentry
              key CDATA #REQUIRED
              type CDATA #REQUIRED
              value CDATA #REQUIRED >

<!ELEMENT ranking (rankingentry)+ >

<!ATTLIST ranking
              class CDATA #REQUIRED >

<!ELEMENT rankingentry EMPTY >
```

```
<!ATTLIST rankingentry
                key CDATA #REQUIRED
                type CDATA #REQUIRED
                value CDATA #REQUIRED >
```

ANNEX B – Optimization Tool Result

```
<!ELEMENT frevo (problem, method, representation, ranking, populations) >

<!ELEMENT problem (problementry)+ >

<!ATTLIST problem
                class CDATA #REQUIRED >

<!ELEMENT problementry EMPTY >

<!ATTLIST problementry
                key CDATA #REQUIRED
                type CDATA #REQUIRED
                value CDATA #REQUIRED >

<!ELEMENT method (methodentry)+ >

<!ATTLIST method
                class CDATA #REQUIRED >

<!ELEMENT methodentry EMPTY >

<!ATTLIST methodentry
                key CDATA #REQUIRED
                type CDATA #REQUIRED
                value CDATA #REQUIRED >

<!ELEMENT representation (representationentry)+ >

<!ATTLIST representation
                class CDATA #REQUIRED >

<!ELEMENT representationentry EMPTY >

<!ATTLIST representationentry
                key CDATA #REQUIRED
                type CDATA #REQUIRED
                value CDATA #REQUIRED >

<!ELEMENT ranking (rankingentry)+ >

<!ATTLIST ranking
                class CDATA #REQUIRED >

<!ELEMENT rankingentry EMPTY >

<!ATTLIST rankingentry
                key CDATA #REQUIRED
                type CDATA #REQUIRED
                value CDATA #REQUIRED >

<!ELEMENT populations (population)+ >

<!ATTLIST populations
                count CDATA #REQUIRED
```

```
              generation CDATA #REQUIRED
              randomseed CDATA #REQUIRED >

<!ELEMENT population ANY >
```

ANNEX C – SDF world example

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <!-- A testing model that includes the Velodyne sensor model -->
    <model name="my_velodyne">
      <include>
        <uri>model://velodyne_hdl32</uri>
      </include>

      <!-- Attach the plugin to this model -->
      <plugin name="velodyne_control" filename="../libvelodyne_plugin_ros.so">
        <velocity>2000</velocity>
      </plugin>
    </model>

  </world>
</sdf>
```

ANNEX D – SDF environment model example

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <model name="ground_plane">
    <static>true</static>
    <link name="link">
      <collision name="collision">
        <geometry>
          <plane>
            <normal>0 0 1</normal>
            <size>100 100</size>
          </plane>
        </geometry>
        <surface>
          <friction>
            <ode>
              <mu>100</mu>
              <mu2>50</mu2>
            </ode>
          </friction>
        </surface>
      </collision>
      <visual name="visual">
        <cast_shadows>false</cast_shadows>
        <geometry>
          <plane>
            <normal>0 0 1</normal>
            <size>100 100</size>
          </plane>
        </geometry>
        <material>
          <script>
            <uri>file://media/materials/scripts/gazebo.material</uri>
            <name>Gazebo/Grey</name>
          </script>
        </material>
      </visual>
    </link>
  </model>
</sdf>
```

ANNEX E – SDF Object model example

```
<?xml version="1.0" ?>
<sdf version="1.5">

  <model name="velodyne_hdl-32">
  <!-- Give the base link a unique name -->
  <link name="base">

    <!-- Offset the base by half the lenght of the cylinder -->
    <pose>0 0 0.029335 0 0 0</pose>
    <inertial>
      <mass>1.2</mass>
      <inertia>
        <ixx>0.001087473</ixx>
        <iyy>0.001087473</iyy>
        <izz>0.001092437</izz>
        <ixy>0</ixy>
        <ixz>0</ixz>
        <iyz>0</iyz>
      </inertia>
    </inertial>


    <collision name="base_collision">
      <geometry>
        <cylinder>
          <!-- Radius and length provided by Velodyne -->
          <radius>.04267</radius>
          <length>.05867</length>
        </cylinder>
      </geometry>
    </collision>

<visual name="base_visual">
  <!-- Offset the visual by have the base's height. We are not rotating
       mesh since symmetrical -->
  <pose>0 0 -0.029335 0 0 0</pose>
  <geometry>
    <mesh>
      <uri>model://velodyne_hdl32/meshes/velodyne_base.dae</uri>
    </mesh>
  </geometry>
</visual>
</link>

  <!-- Give the base link a unique name -->
  <link name="top">

    <!-- Vertically offset the top cylinder by the length of the bottom
        cylinder and half the length of this cylinder. -->
    <pose>0 0 0.095455 0 0 0</pose>
    <inertial>
     <mass>0.1</mass>
     <inertia>
       <ixx>0.000090623</ixx>
       <iyy>0.000090623</iyy>
```

```
      <izz>0.000091036</izz>
      <ixy>0</ixy>
      <ixz>0</ixz>
      <iyz>0</iyz>
    </inertia>
  </inertial>

  <collision name="top_collision">
    <geometry>
      <cylinder>
        <!-- Radius and length provided by Velodyne -->
        <radius>0.04267</radius>
        <length>0.07357</length>
      </cylinder>
    </geometry>
  </collision>

  <!-- The visual is mostly a copy of the collision -->
  <visual name="top_visual">
    <pose>0 0 -0.0376785 0 0 1.5707</pose>
    <geometry>
    <mesh>
     <!-- The URI should refer to the 3D mesh. The "model:"
          URI scheme indicates that the we are referencing a Gazebo
          model. -->
     <uri>model://velodyne_hdl32/meshes/velodyne_top.dae</uri>
   </mesh>
    </geometry>
  </visual>


<!-- Add a ray sensor, and give it a name -->
<sensor type="ray" name="sensor">

  <!-- Position the ray sensor based on the specification. Also rotate
       it by 90 degrees around the X-axis so that the <horizontal> rays
       become vertical -->
  <pose>0 0 -0.004645 1.5707 0 0</pose>

  <!-- Enable visualization to see the rays in the GUI -->
  <visualize>true</visualize>

  <!-- Set the update rate of the sensor -->
  <update_rate>30</update_rate>

<ray>
  <noise>
      <!-- Use gaussian noise -->
      <type>gaussian</type>
      <mean>0.0</mean>
      <stddev>0.02</stddev>
    </noise>

  <!-- The scan element contains the horizontal and vertical beams.
       We are leaving out the vertical beams for this tutorial. -->
  <scan>

    <!-- The horizontal beams -->
    <horizontal>
      <!-- The velodyne has 32 beams(samples) -->
      <samples>32</samples>
```

```
      <!-- Resolution is multiplied by samples to determine number of
           simulated beams vs interpolated beams. See:
           http://sdformat.org/spec?ver=1.6&elem=sensor#horizontal_resolution
           -->
      <resolution>1</resolution>

      <!-- Minimum angle in radians -->
      <min_angle>-0.53529248</min_angle>

      <!-- Maximum angle in radians -->
      <max_angle>0.18622663</max_angle>
    </horizontal>
  </scan>

  <!-- Range defines characteristics of an individual beam -->
  <range>

    <!-- Minimum distance of the beam -->
    <min>0.05</min>

    <!-- Maximum distance of the beam -->
    <max>70</max>

    <!-- Linear resolution of the beam -->
    <resolution>0.02</resolution>
  </range>
</ray>
</sensor>
  </link>

<!-- Each joint must have a unique name -->
<joint type="revolute" name="joint">

  <!-- Position the joint at the bottom of the top link -->
  <pose>0 0 -0.036785 0 0 0</pose>

  <!-- Use the base link as the parent of the joint -->
  <parent>base</parent>

  <!-- Use the top link as the child of the joint -->
  <child>top</child>

  <!-- The axis defines the joint's degree of freedom -->
  <axis>

    <!-- Revolve around the z-axis -->
    <xyz>0 0 1</xyz>

    <!-- Limit refers to the range of motion of the joint -->
    <limit>

      <!-- Use a very large number to indicate a continuous revolution -->
      <lower>-10000000000000000</lower>
      <upper>10000000000000000</upper>
    </limit>
  </axis>
</joint>

</model>
```

```
</sdf>
```

ANNEX F – JSON for StartOptimization

```
{
  "$id": "http://example.com/example.json",
  "type": "object",
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "title": {
      "$id": "/properties/title",
      "type": "string",
      "title": "The Title Schema ",
      "default": "",
      "examples": [
        "StartOptimization"
      ]
    },
    "ID": {
      "$id": "/properties/ID",
      "type": "string",
      "title": "The Id Schema ",
      "default": "",
      "examples": [
        "abcd23412"
      ]
    },
    "threads": {
      "$id": "/properties/threads",
      "type": "integer",
      "title": "The Threads Schema ",
      "default": 0,
      "examples": [
        2
      ]
    },
   "gui": {
      "$id": "/properties/gui",
      "type": "boolean",
      "title": "The Gui Schema ",
      "default": false,
      "examples": [
        false
      ]
    },
    "params": {
      "$id": "/properties/params",
      "type": "string",
      "title": "The Params Schema ",
      "default": "",
      "examples": [
        "-d -p test"
      ]
    },
    "SimulationManagers": {
      "$id": "/properties/SimulationManagers",
      "type": "array",
      "items": {
        "$id": "/properties/SimulationManagers/items",
        "type": "string",
        "title": "The 0th Schema ",
        "default": "",
        "examples": [
          "test@desktop-akqg1ea/cpswarm",
          "test2@desktop-akqg1ea/cpswarm"
        ]
      }
    }
  }
}
```

ANNEX G – JSON for GetProgress and CancelOptimization

```
{
  "$id": "http://example.com/example.json",
  "type": "object",
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "title": {
      "$id": "/properties/title",
      "type": "string",
      "title": "The Title Schema ",
      "default": "",
      "examples": [
        "GetProgress"
      ]
    },
    "ID": {
      "$id": "/properties/ID",
      "type": "string",
      "title": "The Id Schema ",
      "default": "",
      "examples": [
        "abcd23412"
      ]
    }
  }
}
```

ANNEX H – JSON OptimizationResult

```
{
  "$id": "http://example.com/example.json",
  "type": "object",
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "title": {
      "$id": "/properties/title",
      "type": "string",
      "title": "The Title Schema ",
      "default": "",
      "examples": [
        "OptimiationResult"
      ]
    },
    "ID": {
      "$id": "/properties/ID",
      "type": "string",
      "title": "The Id Schema ",
      "default": "",
      "examples": [
        "abcd23412"
      ]
    },
    "fitnessValue": {
      "$id": "/properties/fitnessValue",
      "type": "number",
      "title": "The Fitnessvalue Schema ",
      "default": 0,
      "examples": [
        -2.85
      ]
    }
  }
}
```

ANNEX I – JSON OptimizationReply

```
{
  "$id": "http://example.com/example.json",
  "type": "object",
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "title": {
      "$id": "/properties/title",
      "type": "string",
      "title": "The Title Schema ",
      "default": "",
      "examples": [
        "OptimizationReply"
      ]
    },
    "ID": {
      "$id": "/properties/ID",
      "type": "string",
      "title": "The Id Schema ",
      "default": "",
      "examples": [
        "abcd23412"
      ]
    },
    "operationStatus": {
      "$id": "/properties/operationStatus",
      "type": "string",
      "title": "The Operationstatus Schema ",
      "default": "",
      "examples": [
        "OK"
      ]
    }
  }
}
```

ANNEX J – JSON RunSimulation

```
{
  "$id": "http://example.com/example.json",
  "type": "object",
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "title": {
      "$id": "/properties/title",
      "type": "string",
      "title": "The Title Schema ",
      "default": "",
      "examples": [
        "RunSimulation"
      ]
    },
    "ID": {
      "$id": "/properties/ID",
      "type": "string",
      "title": "The Id Schema ",
      "default": "",
      "examples": [
        "sdafsad24312"
      ]
    },
    "gui": {
      "$id": "/properties/gui",
      "type": "boolean",
      "title": "The Gui Schema ",
      "default": false,
      "examples": [
        false
      ]
    },
    "params": {
      "$id": "/properties/params",
      "type": "string",
      "title": "The Params Schema ",
      "default": "",
      "examples": [
        "-d -p test"
      ]
    }
  }
}
```