D3.2 – UPDATED SYSTEM ARCHITECTURE & DESIGN SPECIFICATION

Deliverable ID	**D3.2**
Deliverable Title	**Updated System Architecture & Design Specification**
Work Package	**WP3 – Architecture design and Component Integration**
Dissemination Level	**PUBLIC**
Version	**1.0**
Date	**2018-06-30**
Status	**Final**
Lead Editor	**Junhong Liang (FRAUNHOFER)**
Main Contributors	**Farshid Tavakolizadeh (FIT), Etienne Brosse (SOFTEAM), Melanie Schranz (LAKE), Regina Bíró (SLAB), Edin Arnautovic (TTTECH), Angel Soriano (ROBOTNIK), Omar Morando (DGSKY), Davide Conzon (ISMB), Gianluca Prato (ISMB)**

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
0.01	2018-04-16	Junhong Liang (FIT)	First Draft with TOC
0.02	2018-06-11	Junhong Liang (FIT)	Incorporated contribution from FIT
0.1	2018-06-15	Junhong Liang (FIT)	Incorporated contributions from all partners
0.2	2018-06-21	Farshid Tavakolizadeh (FIT)	Added the parts related to Deployment Tool; Fixed errors in document; released for internal review
0.3	2018-06-28	Junhong Liang (FIT)	Modified according to review from ISMB
0.4	2018-06-29	Junhong Liang (FIT)	Modified according to review from SOFTEAM
1.0	2018-06-30	Junhong Liang (FIT)	Released for final review

Internal Review History

Review Date	Reviewer	Summary of Comments
2018-06-28	Davide Conzon (ISMB), Gianluca Prato (ISMB)	Version 0.2 document version approved with minor comments.
2018-06-28	Etienne Brosse (SOFTEAM)	Version 0.2 document version approved with comments on sequence diagrams and deployment diagram shown in section 4.2.

Table of Contents

1 Executive Summary

This document is a deliverable of the CPSwarm project, funded by the European Commission's Directorate-General for Research and Innovation (DG RTD), under the Horizon 2020 Research and innovation Program (H2020). It is an updated version of the deliverable 3.1, Initial System Architecture & Design Specification. Similar to D3.1, this document contains two main parts respectively introducing further analysis of relevant engineering methods, tools and standards applicable in CPSwarm and the updated design of the CPSwarm system architecture.

The *Analysis of relevant engineering methods, tools and standards* part provides an up-to-date snapshot of available technologies that can be readily exploited in the project. Since the submission of D3.1, new analysis over such aspects has been carried out across all relevant domains. The new results will be reported in this part.

The *Architecture Design* part gives a brief recap of the initial architecture defined in D3.1. Then an analysis of existing limitations and open issues in the initial architecture is presented. Starting from these identified issues, an updated architecture design has been proposed. The updated architecture is then documented with similar methodology used in D3.1, complying with the ISO/IEC/IEEE 42010 "System and software engineering – Architecture description" [1] standard. Relevant viewpoints of the system are presented as documentation for different architectural aspects of the CPSwarm system.

Besides the main functional and component-based descriptions, cross-cutting concerns such as security, safety, scalability are also reviewed and updated. It is worth mentioning, that even though the architecture design documented in this deliverable is an updated version of architecture, it is still subject to possible modifications in future iterations. Those changes will be documented in D3.3 "Final System Architecture and Design Specification", due in M30.

2 Introduction

This deliverable reports the new updates in analysis of relevant tools and methodologies as well as CPSwarm architecture design since the last documentation in D3.1. This deliverable follows a structure similar to D3.1, consisting of two major parts. The first part gives the reader a short review of the analysis done until M18. Then it introduces new results in the research for relevant tools and methodologies for CPS swarm design. It includes the following aspects:

- Survey of tools and frameworks currently adopted for CPS design / development;
- Survey of existing modelling standards / patterns;
- Preliminary Analysis of design methodologies applicable to the CPS domain.

In the second part of the document, the updated architecture design for the CPSwarm system is illustrated. Similar to part one, this part first gives a brief recap of the previous architecture design defined in D3.1 to provide the reader the context for further discussion. Moreover, the limitations and open issues identified from the previous architecture are discussed. After that, a new modified architecture design is presented to address the identified issues. Besides the architecture design, some cross-cutting aspects such as security, scalability are also addressed. Compared to D3.1, a new aspect in safety is added in this deliverable to tackle the concern regarding how to ensure the operational safety of the swarm of CPS.

At the end of the document, a plan for future steps and possible exploration directions are presented. It is worth noting that the CPSwarm architecture is expected to be revised, extended and refined in the future, accounting for evidences and lessons learned while accomplishing project activities.

2.1 Related documents

ID	Title	Reference	Version	Date
D2.1	Initial Vision Scenarios and Use Case Definition	D2.1	1.0	M4
D3.1	Initial System Architecture & Design Specification	D3.1	1.0	M6
D5.1	CPSwarm Modelling Language Specification	D5.1	1.0	M12
D5.2	Initial CPSwarm Modelling Tool	D5.2	1.0	M9
D6.1	Initial Simulation Environment	D6.1	1.0	M9
D6.3	Initial CPS system design optimization and Fitness function design guideline	D6.3	1.0	M18
D6.5	Initial integration of external simulators	D6.5	1.0	M18
D7.1	Initial CPSwarm Abstraction Library	D7.1	1.0	M18
D7.3	Initial Bulk Deployment Tool	D7.3	0.1	M21

3 Analysis of relevant engineering methods, tools, technologies and standards

In this chapter, an analysis of relevant engineering methods and tools applicable in CPSwarm is presented. The first part of this chapter gives a recap of the technologies analysed in D3.1. New contents have been added to enrich the previous analysis. The second part of this chapter, Section 3.2, presents the new research carried out within the second phase of the project. The new research covers the analysis of communication protocols used in swarm runtime, as well as the simulators used for swarm behaviour simulation/optimization.

3.1 Recap of Previously Reviewed Technologies

3.1.1 Survey of Tools and Frameworks currently adopted for CPS design / development

3.1.1.1 *Design*

The following sections describe the tools commonly used for developing applications in different domains, such as drones, robotic and automotive.

Drones

Building a drone requires more than just flight controls — it requires at least the following aspects: vision and/or GPS based navigation, obstacle avoidance and path planning. For this reason, PX4[1] has been selected as the base platform upon which the complex coordination and swarm behaviours targeted by the project are built upon. PX4 is one of the leading flight control platforms, its architecture is designed to be ready for complex environments and can empower any vehicle from racing and cargo drones to ground vehicles.

Design and simulation of software functions (e.g. control and attitude algorithms, collision avoidance) is typically accomplished by exploiting Simulink/MATLAB[2], a block diagram environment for multi-domain simulation and Model-Based Design. These tools are also used for generating production-level code, which is subsequently tested using a Software-in-the-loop (SITL) simulation based on Gazebo[3], a powerful 3D simulation environment that is particularly suitable for testing object-avoidance and computer vision. It can also be used for multi-vehicle simulation and is commonly used with Robotic Operating System (ROS[4]), a collection of tools for automating vehicle control. Gazebo allows PX4 flight code to control a computer modelled vehicle in a simulated "world". Developer can interact with this vehicle just as he/she might with a real vehicle, using the mission planner QGroundControl[5] (an open-source mission planner that provides a full flight control and vehicle setup for PX4 powered vehicles), an off-board Application Programming Interface (API), or a radio controller/joystick.

Following the process, after the deployment of the generated code into the flight controller (or in a simulated environment with Gazebo), mission can be planned, monitored and controlled using QGroundControl. It provides easy and straightforward usage for beginners, while still delivering high end feature support for experienced users. QGroundControl runs on Windows, Mac OS X, Linux, iOS and Android. It supports multiple autopilots: PX4, ArduPilot[6] or any vehicle that communicates using the MAVLink[7] protocol. QGroundControl works with all vehicle types supported by PX4: multi-rotor, fixed-wing, vertical take-off and landing (VTOL) and ground rover.

Rovers

[1] http://px4.io/
[2] https://www.mathworks.com/products/simulink.html
[3] http://gazebosim.org/
[4] http://www.ros.org/
[5] http://qgroundcontrol.com/
[6] http://ardupilot.org/
[7] https://mavlink.io/

The software of rovers is fully developed in ROS. Packages are written in C++ or Python. The main features of these packages are to offer the ability to move the robot, check the integrated sensors and obtain the robot position relative to its movements (odometry).
Some standard tools of ROS have been used for the design of the software:

- For the 3D model of the robot, the Rviz[8] application in ROS is used to see if all the frame transforms between the components of robot are well positioned as shown in Figure 1. A frame is a coordinate system assigned to a component (e.g. the hand of a robot) or a robot. Transformations between frames are used to express the spatial relationship between components and robots. The Rviz provides an intuitive visualization for developers to identify such relations.
- Also, to see all the frames, the *rqt_tf_tree*[9] tool of ROS is useful. This shows a tree with the relation between all the frames defined in the robot. Figure 2 shows an example of the output of the *rqt_tf_tree* for the rover.
- The *rqt_graph* [10]tool has been used to check the connections between the nodes and the topics. For more details about ROS nodes and topics, please refer to Section 3.1.1.3. Figure 3 shows the connections created for one rover.

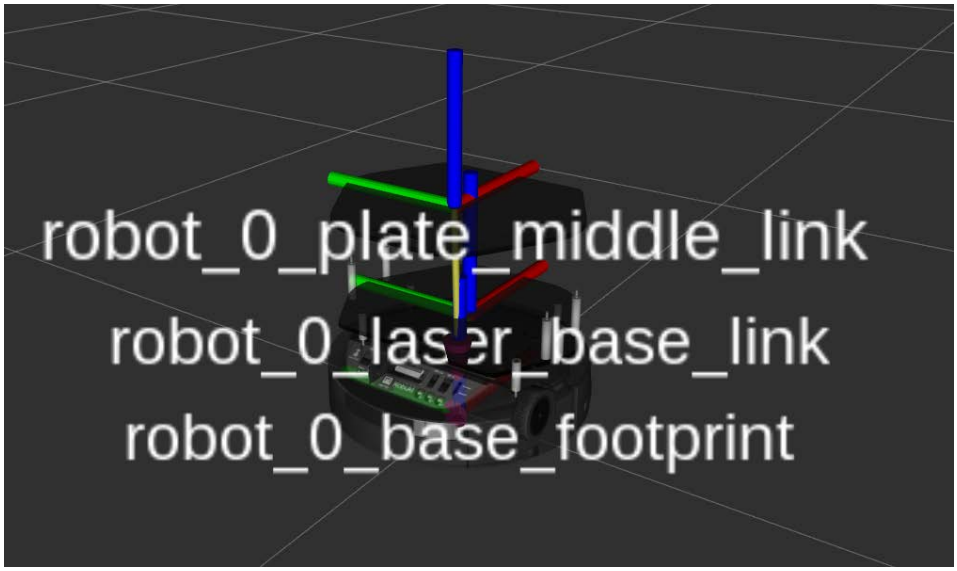

Figure 1. Frame transforms at Rviz visualization

[8] http://wiki.ros.org/rviz
[9] http://wiki.ros.org/rqt_tf_tree
[10] wiki.ros.org/rqt_graph

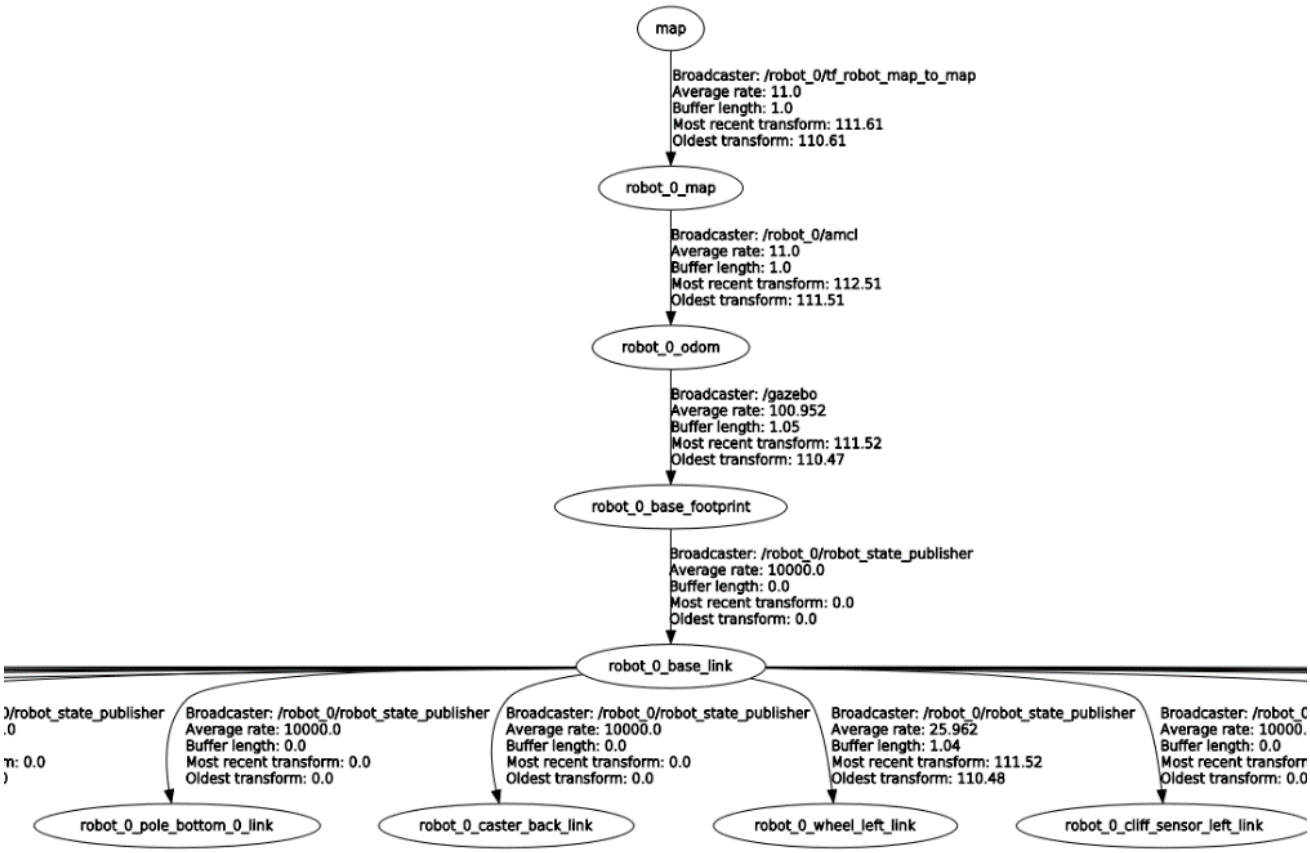

Figure 2. Rqt_tf_tree example of turtlebot robot

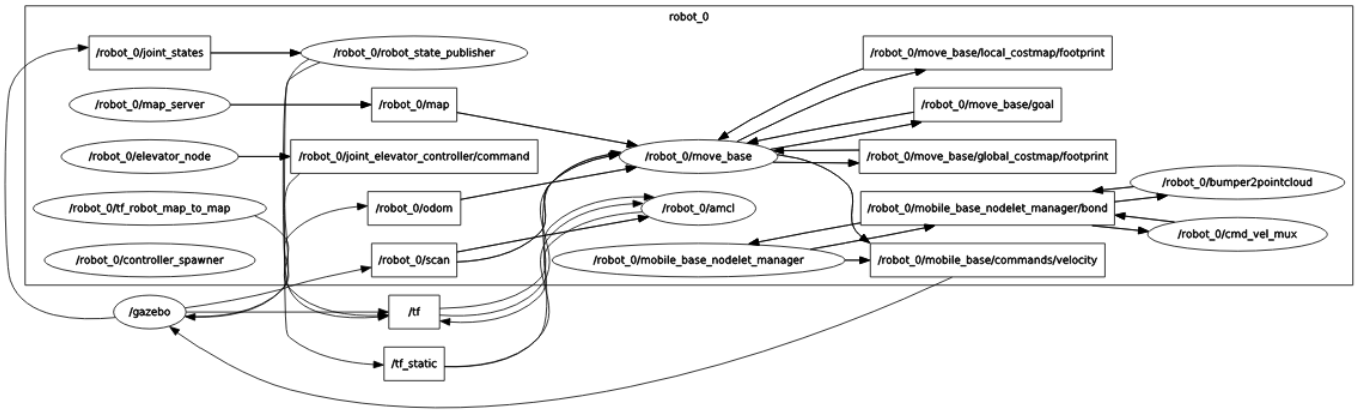

Figure 3. Relation between nodes through topics

Automotive

For the design and management of software complex systems in modern vehicles, tools such as IBM Rational Doors[11] are widely exploited. For high-level software design, general Unified Modelling Language (UML) tools such as IBM Rational Rhapsody[12] or Modelio[13] are used to represent software structures (e.g., UML

[11] https://www.ibm.com/us-en/marketplace/rational-doors
[12] https://www.ibm.com/de-en/marketplace/rational-rhapsody
[13] https://www.modelio.org/

component of class diagrams) and/or behaviors. Since typical automotive software architectures are mainly based on the AUTomotive Open System ARchitecture (AUTOSAR) standard, some specific tools are used for automotive software development. For example, for model-based specification of electronic vehicle systems and their components, as well as for the design of vehicular network architectures, the Vector's PREEvision[14] tool is frequently used. For designing AUTOSAR-compliant (low-level) software structures, including port interfaces, data types, composition of components, or scheduling, tools such as, e.g., DaVinci Developer[15] tool from Vector are used. Whereas, simulation and modelling of software functionality (e.g., control algorithms), is typically accomplished with tools such as Ascet[16] from ETAS or Simulink/MATLAB from MathWorks. These tools are also used for the generation of real-time, production code.
However, in the CPSwarm project, ROS (explained above) has been used for the development of the exemplary CPSwarm automotive application, to better align with the project activities and workbench development.

3.1.1.2 *Simulation*

D3.1 presented the state-of-the-art of simulation engines. That deliverable presents a partial analysis, which has been completed at M9 in D6.1. The results of this analysis are summarized in the following table, where the simulators are subdivided in 2D (see Table 1) and 3D simulators (see Table 2). Simulators under active development are highlighted with yellow background colour. They are the top priority candidates for simulators used in CPSwarm. Simulators no longer under active development are out of scope of consideration. However, they are also listed in the table for reference purpose.

Simulation Environment	License, Cost	Availa-bility	Language / Format	Hardware Models	ROS Inter-face	Hardware Portability	Fidelity (functional, physical)	OS	Active develop ment
MobotSim	All rights reserved, $30	low	Visual Basic					Win.	no
MRSim	All rights reserved	low	Matlab						no
Rossum Playhouse	GPLv2 / MIT, free	high	Java						no
Stage	GPLv2, free	high	C++, config: plain text	some	yes	yes, using ROS or Player	low, low	Linux, Win.	yes
STDR	GPLv3, free	high	C++, config: XML, YAML	some	yes	yes, using ROS	low, low	Linux	yes
Swarm	GPLv2, free	high	Java – Objective-C					Linux Win. MacOS Solaris	no
TeamBots	Free for education / research		Java, config: in source, plain text	some	no	yes	?, low	Linux Win. MacOS	no

Table 1. Overview of two-dimensional simulation environments.

Simulation Environment	License, Cost	Availa-bility	Language / Format	Hardware Models	ROS Inter-face	Hardware Portability	Fidelity (functio-nal, physical)	OS	Active develop ment
ARGoS	MIT, free	high	C++, config: XML	some	yes	No	depending on physics engine	Linux MacOS	yes

[14] https://vector.com/vi_preevision_en.html
[15] https://vector.com/vi_davinci_developer_en.html
[16] https://www.etas.com/en/products/ascet-developer.php

Breve									no
DPRSim									no
Gazebo	ALv2, free	high	C++, config: SDF (XML)	some	yes	yes, using ROS or Player	high, high	Linux Win. MacOS	yes
jMAVSim	BSDv3, free	medium	Java, config: Java, shell script	some	yes	Yes		Linux Win. MacOS	yes
Marilou	All rights reserved, €499							Win.	yes
Mission Lab									no
MORSE	BSDv3, free	high	Python, config: Python	some	yes	yes, using ROS	high, high	Linux	yes
MuRoSimF									no
peekabot									no
Simbad									no
SimSpark									no
Swarmbot3D									no
USARSim									no
v-rep	GPL / commercial , not free	high	Lua, C++, config: binary	many	yes	yes, using ROS	high, high	Linux Win. MacOS	yes
Webots	All rights reserved, €3450	medium	C/C++, Java, Python, MATLAB	some	yes	yes		Linux Win. MacOS	yes

Table 2. Overview of three-dimensional simulation environments.

The results of this analysis have led to definition of some requirements for the CPSwarm Simulation and Optimization Environment. Two example requirements are:

- Support to different simulators shall be provided.
- Co-simulation shall be included in the workbench design.

The conclusion of the analysis and the considerations done on the Runtime environment (described in the next section), has led the partners to focus the study on the ROS based simulators. The study of these simulators done in Task 6.4 and described in the deliverable D6.5 due in M18 are summarized in Section 3.2.2.

3.1.1.3 *Runtime*

ROS

In order to simplify the integration between the various software modules in the companion computer, in the CPSwarm Consortium has decided to adopt ROS, one of the most widespread and reliable integration systems in the industrial field. ROS is robotics middleware (i.e. collection of software frameworks for robot software development). Although ROS is not an operating system, it provides services designed for heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. The main ROS client libraries (C++, Python, and Lisp) are geared toward a Unix-like system, primarily because of their dependence on large collections of open-source software dependencies.
ROS package application areas will include: perception, object identification, face and gesture recognition, motion tracking, stereo vision via two cameras, mobile robotics, navigation, planning and more. ROS contains many open source implementations of common robotics functionality and algorithms. These open source

implementations are organized into "packages". Many packages are included as part of ROS distributions, while others may be developed by individuals and distributed through code sharing sites such as Github.

The basic concepts of ROS runtime are nodes, Master[17], Parameter Server, Messages, Services, Topics, and Bags.

- **Nodes**: Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one Node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client library[18], such as *roscpp*[19] or *rospy*[20].
- **Master**: The ROS Master provides name registration and lookup to the rest of the nodes. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- **Parameter Server**: The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.
- **Messages**: Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).
- **Topics**: Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by *publishing* it to a given topic[21]. The topic is a name[22] that is used to identify the content of the message. A node that is interested in a certain kind of data will *subscribe* to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each other's existence. The idea is to decouple the production of information from its consumption. A topic can be considered as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.
- **Services**: The publish/subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request / reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply (see Figure 4). ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call.
- **Bags**: Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

The ROS Master acts as a name-service in the ROS Computation Graph. It stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The Master will also make callbacks to these nodes

[17] http://wiki.ros.org/Master
[18] http://wiki.ros.org/Client Libraries
[19] http://wiki.ros.org/roscpp
[20] http://wiki.ros.org/rospy
[21] http://wiki.ros.org/Topics
[22] http://wiki.ros.org/Names

when this registration information changes, allowing nodes to dynamically create connections as new nodes are run.

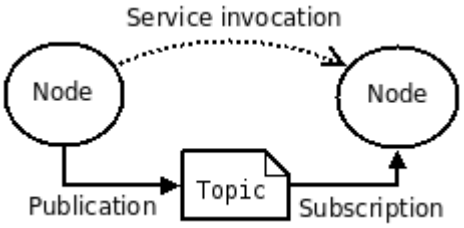

Figure 4. ROS service invocation example

Nodes connect to other nodes directly; the Master only provides lookup information, much like a DNS server. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol. The most common protocol used in a ROS is called TCPROS[23], which uses standard TCP/IP sockets.

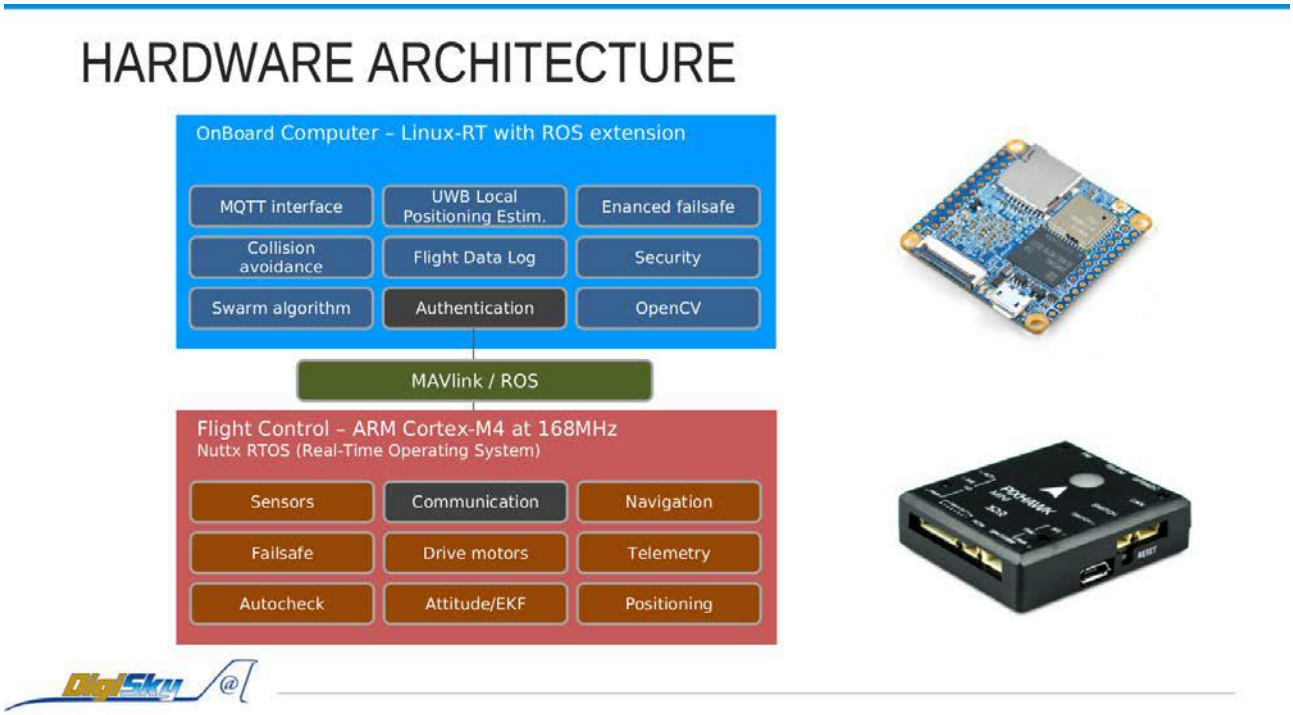

Figure 5. Drones Hardware Architecture

PX4

The architecture of CPSwarm drones is based on a PX4 flight controller and a companion computer, which is in charge of running the following features/modules:

- Off-board drone controller for autonomous flight.
- Swarm algorithms.
- Obstacle detection using proximity sensor.
- Camera streaming.
- Wireless communication.

[23] http://wiki.ros.org/ROS/TCPROS

- Obstacle avoidance based on ROS module.
- Visual target detection based on OpenCV

Communication between flight controller and companion computer is ensured by MAVROS[24], an implementation of the MAVlink protocol on ROS.
All the software modules in PX4 and in the companion computer are built using C++ language and are cross-compiled on a Linux computer using a standard ARM GCC/G++ compiler and CMake.

3.1.2 Survey of existing modelling standards / patterns

3.1.2.1 *Unified Modelling Language (UML)*

The Unified Modelling Language (UML)) is a standard language for Object Oriented Modelling (OOM) mainly used for Software engineering. The first version of UML (0.9) has been defined in 1996 by the Object Management Group (OMG). This first version has been specified by reusing concepts existing in several methods including Booch, Object-Modeling Technique (OMT) . The UML 2.5 is the latest version and has been release in 2015.
UML defines an object-oriented approach, which is massively used for software development. UML defines 13 diagrams, which can be sort in two categories:

1. Static view, which includes:
    a. Class diagram, describing objects and their relations;
    b. Object diagram, used for describing object instances and their links;
    c. Package diagram, describing the logical organisation of the system;
    d. Component diagram, describing components and their interfaces;
    e. Composite Structure diagram, depicting the internal structure of a component;
    f. Deployment diagram, showing the physical deployment of objects.
2. Dynamical view, which includes:
    a. Use Case diagram, describing external functionalities of the described system;
    b. Activity diagram, showing the action of a process;
    c. State Machine diagram, for modelling the possible states and transitions of a given object;
    d. Sequence diagram, to describe the sequential interactions between instances of objects.
    e. Communication diagram describing, like the sequence diagram, the interactions between instances of objects.
    f. Interaction Overview diagram combining activity and sequence diagrams to show the flow between sequences.
    g. Timing diagram to show the object properties evolution over time.

Table 3 UML and its usage in different modelling kinds

Modelling Kind	Support
Structural Design	Several diagrams allow structural view, as Class or Component diagram for example. These diagrams allow the representation of objects with their properties and structural relations.
Discrete Behavioral	Time in milliseconds or hours can be included in sequence diagrams for example. However, UML does not provide any formal specification of timing notations.
Event Based Behavioral	State machine diagram represent the states and transition between these states. These transitions are triggered by occurrence of events.

[24] http://wiki.ros.org/mavros

3.1.2.2 *System Modelling Language (SysML)*

System Modelling Language (SysML) is a standard language in system engineering. Originally defined by the INCOSE in 2001, the latest version (SysML 1.5) has been defined by the OMG in 2017.
SysML defines a component- (block-) oriented approach that fits with software development needs. SysML defines 9 diagrams, which can be sort in three categories:

1. Static view, which includes:
    a. Block diagram, describing the block (component) composing the system and their relation;
    b. Internal Block diagram, depicting the internal structure of a given block;
    c. Package diagram, describing logical organization;
    d. Parametric diagram, specifying the mathematical equations inside the system.
2. Dynamical view, mostly identical to UML, which includes:
    a. Use Case diagram, describing external functionalities of the described system;
    b. Activity diagram, showing the actions of a process;
    c. State Machine diagram, for modelling the possible states and transitions of a given object;
    d. Sequence diagram, to describe the sequential interactions between instances of objects.
3. Requirement view, which includes:
    a. Requirement diagram, to specify the system requirements and their relations between them or with the system.

Table 4 SysML and its usage in different modellling kinds

Modelling Kind	Support
Structural Design	By using Block Definition Diagram and Internal Block Definition Diagram, SysML allows the system decomposition in terms of block or sub-block, including their properties and the relations between them.
Discrete Behavioral	Behavioural definitions are possible using behavioural diagrams such as state machine diagram.
Event Based Behavioral	As for UML, State machine can be used for this kind of modelling.

3.1.2.3 *MARTE*

The UML profile for Modelling and Analysis of Real-Time Embedded Systems (MARTE) expands upon both UML and SysML; and provides extensions (e.g., for performance and scheduling analysis). The latest version of the MARTE specification (v1.1) was finalized in June 2016.
The MARTE profile enables to model software and hardware aspects of a real-time embedded system along with their relations. It also allows taking into consideration platform services (such as the services offered by an OS). MARTE does not provide any additional diagrams as compared to UML. However, MARTE concepts can be used in all UML static and dynamic views. The major concepts in MARTE are:

- Non- Functional Properties (NFPs): MARTE allows describing properties that are not related to functional aspects: e.g. related to energy consumption, memory utilization, consumed resources, etc.
- Time Modelling: MARTE advocates concepts mainly used in synchronous domain as well as in discrete real-time systems. It enables usage of time and clock constraints on UML behavioural models such as sequence diagrams and state machines.

- Allocation: The allocation mechanism in MARTE enriches the SysML Allocation concept and allows mapping application tasks onto the architecture resources.
- Generic Quantitative Analysis Modelling: These MARTE concepts allow designers to focus on high-level analysis, which can be for the software behaviour (such as schedulability and performance) as well as other aspects such as power, energy, fault tolerance, etc.
- Schedulability Analysis Modelling: MARTE also has the capability to carry out schedulability analysis of either the global system or a subsystem, to meet certain constraints such as related to time (e.g., deadlines). Schedulability analysis also helps in the optimization of the system. A system can be analysed under different scenarios or input values in order to observe the differences.
- Performance Analysis Modelling: Finally, MARTE provides designers the feature to carry out analysis of temporal properties of real-time embedded systems.

Table 5 MARTE and its usage in different modelling kinds

Modelling Kind	Support
Structural Design	MARTE uses UML structural design views: As such, MARTE concepts can be used, for example, in Class or Component diagrams.
Discrete Behavioural	MARTE timing and value specification features allow to annotate models with timing notations.
Event Based Behavioural	Similar to that for UML structural design, MARTE concepts can be applied for e.g. on UML state machines, sequence and activity diagrams.

3.1.3 Analysis of design methodologies applicable to the CPS domain

3.1.3.1 *Methodologies for Swarm & Self-Organizing Behaviour Design*

To handle the complex characteristics of swarms of CPSs, natural systems can serve as inspiration: they have evolved over millennia to master NP-hard problems [2]. Swarms can be considered as a kind of quasi-organism that can adapt to changes in the environment by following specific behaviours [3], e.g.:

- Pursuing a specific goal
- Aggregating or dispersing in the environment
- Communicating (direct, indirect)
- Memorizing (local states, morphologies)

The state-of-the art process of designing a swarm model follows the steps described in Figure 6. Engineers use models found in nature in the swarm behaviour of, e.g., insects, bacteria, or fish [4] as an inspiration for solving complex real-world problems. Based on the observations of these natural behaviours a model is created. The model is then simulated to assess how well the intended result can be achieved with a given behaviour. This assessment is usually characterized by a fitness value. Finally, the model is extracted to design a nature-inspired swarm algorithm.

The theoretical and mathematical foundations of traditional swarm algorithms are described in Bonabeau et al. [5], Camazine et al. [6], Garnier et al. [7], Blum and Li [8], Floreano and Mattiussi [9], Parpinelli and Lopes [10], Binitha and Sathya [11], Yang et al. [12], Krause et al. [13], Hassanien and Emary [14], and Yang et al. [15], to name but a few.

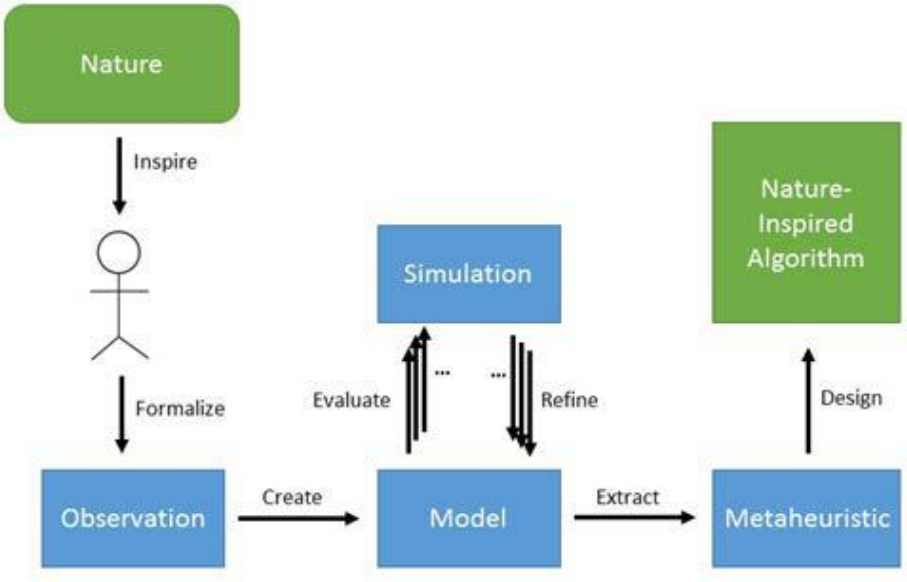

Figure 6. Process of designing a swarm model and the corresponding algorithm (adapted from [11]).

The behaviour of an individual CPS within a swarm consists of a sequence of interactions with the physical world, including the environment and other CPSs. Each interaction consists of reading and interpreting the sensory data, processing this data, and driving the actuators accordingly. Such an interaction is defined by a basic behaviour that is repeatedly executed until a desired state is reached. The behaviours are describing high-level tasks of an individual CPS and the resulting swarm behaviour, which are adapted and expanded from [16]. The authors do not detail on the sensing and acting part, which is specific for each CPS type. Many examples for the observation of the behaviours in nature can be found in [6], [9], and [14]. During the project, the algorithms are selected according to their behaviour (full version with explanations can be found in [17]):

1. Spatial Organization
    a. Aggregation: individual CPSs move to congregate spatially in a specific region of the environment.
    b. Pattern & chain formation: swarm of CPSs arranges in a specific order.
    c. Self-assembly: CPSs connect physically.
    d. Object clustering and object assembly: swarm of CPSs manipulates spatially distributed objects.
2. Navigation
    a. Collective exploration: swarm of CPSs cooperatively explores an environment and navigates through it.
    b. Coordinated motion: a swarm of CPSs moves together.
    c. Collective transportation: a swarm of CPSs is applied to move an object which is too heavy for a single CPS.
    d. Localization: the CPSs of a swarm are located by i) using GPS or another localization system in the infrastructure, ii) identifying their own locations by comparing the surrounding environment with a given map, iii) applying self-localization via establishment of a local coordinate system throughout the swarm.
3. Collective-Decision Making
    a. Consensus: typical example for agreement in a swarm of CPSs.
    b. Task allocation: arising tasks are dynamically assigned.

c. Fault detection: collective behaviour allows the swarm to detect physical faults or faulty behaviour of CPSs in the swarm.
d. Object recognition: goal is to collectively recognize objects in the environment.
e. Synchronization: time of individual CPSs is synchronized.

4. Miscellaneous
   a. Group size regulation: goal is to create a group of CPSs by selecting a certain number of CPSs from the swarm.
   b. Human-swarm interaction: describes how humans control or manipulate the swarm, parts of the swarm, or single CPSs in order to reach a desired goal.
   c. Self-healing: minimize the impact of CPS failure on the rest of the swarm.

3.1.3.2 *Methodologies for user-centred design applied to Human-CPS interaction*

The operator interacts with the swarm of CPSs through a communication interface as in Figure 7. The communication flow Human2Swarm can be in three modes: (one) human to one swarm member, (one) human to multiple (selected) swarm members, or (one) human to swarm. Independent of the mode, the operator is able to

- get outputs from the swarm(s) via visualization
- send inputs to the swarm(s) via control inputs

Getting and setting inputs and outputs needs to be defined already in the modelling stage of the swarm. Therefore, the persons dealing with the modelling/design of the swarm of CPSs need to be aware of the information needed by the operators at the end of the information flow.
More information to the human-swarm/human-CPS interaction can be found in Deliverable D4.2.

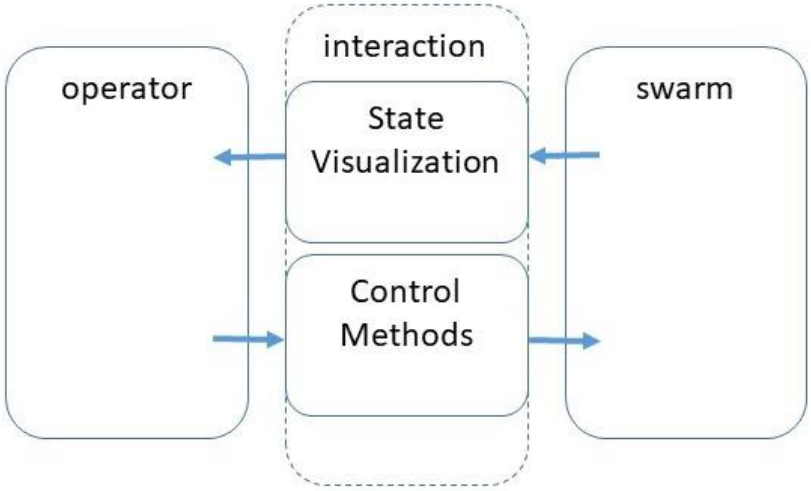

Figure 7. Human-Swarm Interaction Model (adapted from [20])

3.2 Updated Analysis

3.2.1 Communications Architecture and Protocols

Quite a few protocols have been identified and analysed as candidates to build our solution on in the CPSwarm Runtime Environment. We have considered both centralized and decentralized approaches and studied them according to their relevance to our use cases. Below the analysis of such protocols and libraries describes them divided into two groups –centralized and decentralized.

3.2.1.1 *Centralized approach*

In the centralized case, swarm members communicate with and through a central server (Figure 8). Centralized solutions work well with operations where the swarm members are spread far apart and maintain connections through LTE[25] or similar cellular technologies. The main candidate protocols we identified that support this are:

- MQTT[26]
  Message Queuing Telemetry Transport (MQTT) is an open standard defined by OASIS, better suited to constrained environments than HTTP. It is widely used to implement APIs for several open source solutions in the IoT scenario and can be secured by Transport Layer Security (TLS/SSL). MQTT supports only asynchronous communication through publish-subscribe paradigm and is based on TCP.
- AMQP[27]
  The Advanced Message Queuing Protocol (AMQP) is an open standard application layer protocol designed to support a variety of messaging applications and communication patterns. AMQP provides authentications and/or encryption based on SASL and/or TLS and is based on TCP.
- XMPP[28]
  eXtensible Messaging and Presence Protocol (XMPP) is a scalable, standard technology that supports publish-subscribe paradigm as one of its communication patterns based on TCP. It can be secured with TLS and SASL, however the security level required can pose a limit in performance.

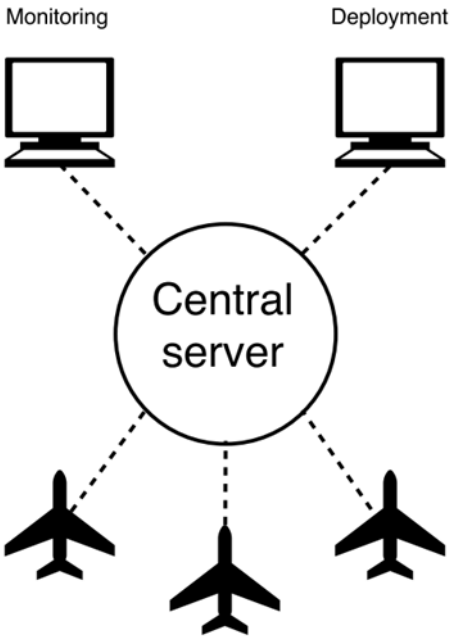

Figure 8. Centralized approach configuration

3.2.1.2 *Decentralized approach*

In the decentralized case, swarm members are on the same network, and the members communicate with each other (Figure 9). This approach works well with deployments where such a network can be established, and connections are maintained through Wi-Fi or other similar local wireless technologies. The main candidate protocols we identified that support this are:

[25] Stands for Long-Term Evolution, a 4G telecommunication technology
[26] http://mqtt.org/
[27] https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol
[28] https://xmpp.org/

- DDS[29]
  The Data Distribution Service (DDS) is an OMG standard enabling scalable, real-time, high performance data exchanges using publish-subscribe pattern. DDS was promoted for use in the Internet of Things, and the standard is used in applications such as smartphone operating systems, transportation systems and vehicles and by healthcare providers.
- ZMQ[30]
  ZeroMQ (ZMQ) is a high-performance asynchronous messaging library, aimed to be used in distributed applications. Although it provides a message queue, a ZMQ system can run without a dedicated message broker. The basic ZeroMQ patterns are request-reply, publish-subscribe, push-pull and exclusive pair.

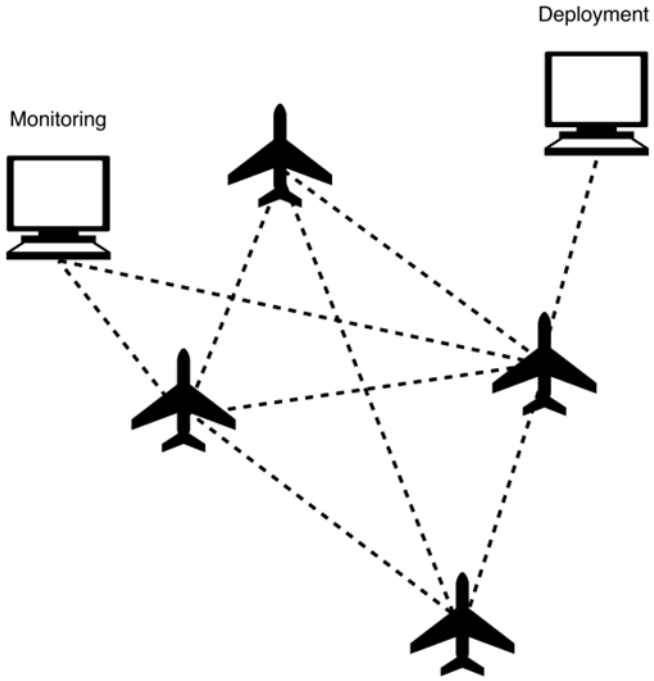

Figure 9. Decentralized approach configuration

To compare the two approaches, the authors analysed the pros and cons that are relevant to CPSwarm use case scenarios.

Decentralized	Centralized
Better fault tolerance	Simple to implement
Better mapping to the swarm concept	Better fit for centralized monitoring
May require custom development	Very mature protocols and libraries
Works well with mesh networks	Works well with cellular networks
Security is harder to get right	Very easy to secure

[29] https://en.wikipedia.org/wiki/Data_Distribution_Service
[30] http://zeromq.org/

Needs discovery component	Discovery is a non-issue

Table 6. Comparison of the two approaches

Table 6 describes the comparison between the two approaches. Despite the fact that centralized solutions may require less effort in development, the CPSwarm Consortium decided to build on a decentralized protocol - using ZMQ - since it fits the concept of swarms better. The specifications of the CPSwarm communications library are described in deliverable D7.1 - Initial CPSwarm Abstraction Library.

3.2.2 Robotic Simulation Tools Analysis

Task 6.4 has focused the study on simulation engines enabled to be used with ROS framework, specifically the simulators analyzed are Stage[31], Gazebo[32], Virtual Robot Experimentation Platform (V-REP)[33] , Autonomous Robots GO Swarming (ARGoS)[34], jMAVSim[35] and Simple Two-Dimensional Robot Simulator (STDR)[36].
Specifically, D6.5 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 CPS behaviour.
The first objective of this analysis is to understand which of these simulators are suitable to be integrated in the CPSwarm workbench, fulfilling these requirements:

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

The results of this analysis have been exploited in several ways:

- To choose the first simulation environment to integrate, which is Gazebo.
- To build the first simulation environment.
- To choose the data format to be used for the integration of the simulators.
- To guide the development of the software used to integrate the simulators in the CPS workbench.

The next few sections summarize the results of the analysis done on the simulators considered.

3.2.2.1 *Stage*

Stage is a low-fidelity two-dimensional simulator that can be used with a variety of robotic platforms and sensors, which can be used as a standalone application that loads a robot control program from a user-defined library or as a ROS-based simulation tool. It uses world files to define the environment and the CPS behaviour can be implemented by writing a C++ or Python code, or using another ROS node.

[31] http://rtv.github.io/Stage/
[32] http://gazebosim.org
[33] http://www.coppeliarobotics.com/
[34] http://www.argos-sim.info/
[35] https://pixhawk.org/dev/hil/jmavsim
[36] http://wiki.ros.org/stdr_simulator

3.2.2.2 *Gazebo*

Gazebo is one of the most used simulators on the ROS architecture, since there are many packages developed for the integration of Gazebo in ROS. It is open-source and it provides a complete 3D environment robotics oriented. 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. These topics usually are the same that the real robot has as already mentioned. To set up the entire environment of the simulation, Gazebo uses a eXtensible Markup Language (XML) file named world. Also, the model uses an XML like standard called Simulation Description Format (SDF)[37]. The CPS behaviour will reside in C++ or python code on the side of ROS, so the implementation will be external to Gazebo.

3.2.2.3 *V-REP*

The V-REP is a 3D robot simulator. It provides also a development environment, which can be used to configure and simulate any robot. Its functionalities can be integrated using the API and script functionality provided. V-REP provides ROS interfaces, which enable to use V-REP as a ROS node. The configuration files for the CPS models and environment use a binary format. The CPS behavior is handled using a combination of embedded scripts and plugins.

3.2.2.4 *ARGoS*

ARGoS is a multi-physics robot simulator designed to efficiently simulate complex experiments involving large swarms of heterogeneous robots. The simulator can be controlled via the command line. The settings of a simulation are set using an XML file and the CPS behaviour is integrated using user code compiled into one or more libraries.

3.2.2.5 *jMAVSim*

jMAVSim is a simple drone simulator, which allows to simulate vehicles running PX4 autopilot around a simulated space. jMAVSim can be executed as a standard Java application via command line or launched compiling a SITL instance of PX4. The environment is statically modelled using an image file. The CPSs are modeled using the WaveFront OBJ file format[38].

3.2.2.6 *STDR*

STDR implements a distributed, client-server architecture, where the nodes can run in a different machine and communicate using ROS interfaces. STDR can be controlled through command line commands. The models to be used for STDR Simulator are subdivided in two main types: maps, where every map is composed by a couple of files: a static png and a yaml file with the properties of the map (source, location of obstacles, etc.) and YAML/SDF files, which can describe robots, laser sensors, range sensors and footprints. STDR Simulator is designed to be completely ROS compliant. Indeed, robots and sensors sends all the measurements on ROS topics.

[37] http://sdformat.org/

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

4 CPSwarm Architecture Design

In this chapter, a brief review of the initial architecture design is first presented. Following that, the limitations and open issues remained in the initial architecture design are discussed. Such limitations serve as the motivation for the new iterations in architecture design, resulting in the updated version of CPSwarm architecture, which will be documented in detail in the last section of the chapter.

4.1 Review of the Initial Architecture

In D3.1, an initial architecture design was presented (Figure 10).

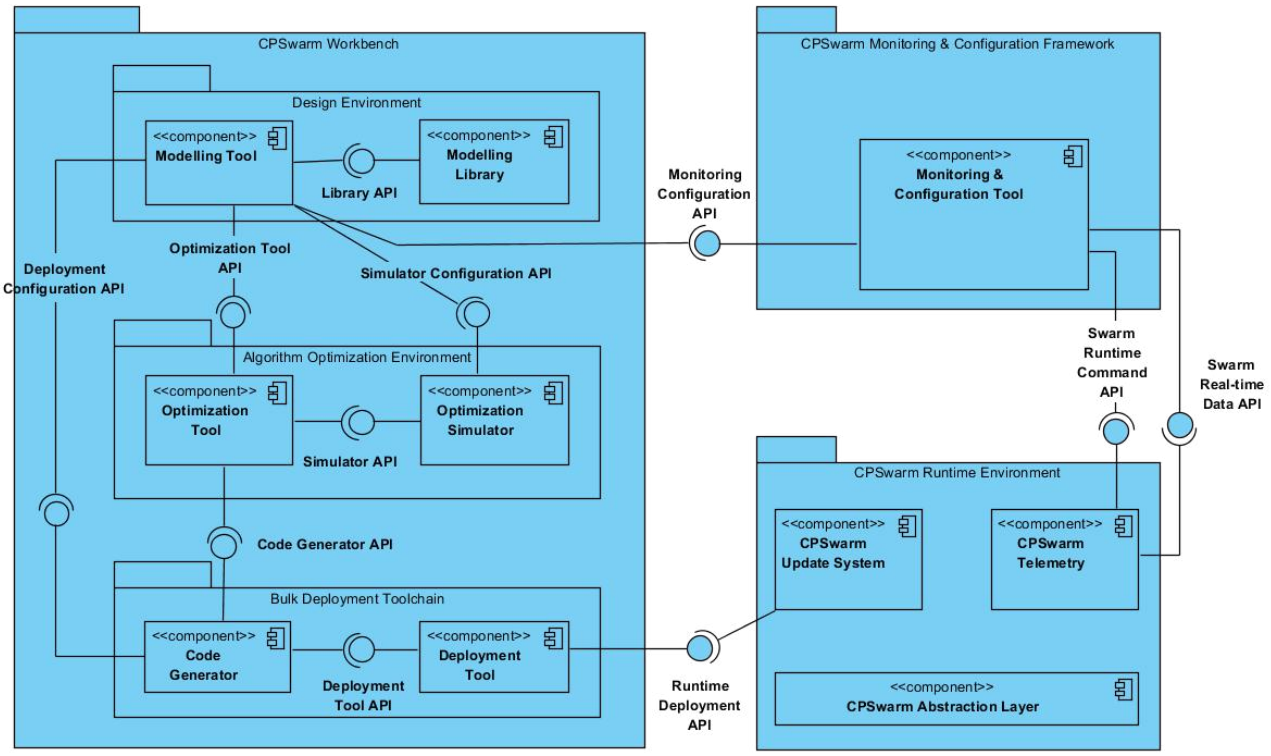

Figure 10. Initial Architecture Design (extracted from deliverable 3.1)

The whole CPSwarm system has been divided into three separate sections: the *CPSwarm workbench*, the *CPSwarm Monitoring and Configuration Framework* and the *CPSwarm Runtime Environment*.

- The *CPSwarm Workbench:* this section includes the design-time tool-chain for swarm modellers, swarm designer and etc. to design a swarm according to a specific workflow, which is defined in D2.1 according to the requirements and stakeholders analysis. The workflow includes the following steps:
  - Modelling: in this step, swarm designers model the swarm via the help of Modelling Tool, which is typically a GUI application. Swarm designers define the fundamental configuration of the swarm, including what types of swarm device are used in the swarm, the goal of the swarm, the algorithm used in the swarm, the operational environment of the swarm, etc. through the GUI application.
  - Optimization: in some complicated scenario, it is very difficult, if it is even possible, to describe the behaviour of the swarm by using pre-existing algorithms. At this point, an optimization step is carried out, in which the evolutionary approach will be used to find the appropriate swarm algorithm.
  - Code Generation: to ease the difficulties in deploying on heterogeneous devices, which operates on different code base, a code generation step is envisioned to generate platform-dependent code according to the algorithm, so that they could be used in different types of CPS.

- Deployment: to avoid the repeated and error-prone work of deploying code on multiple devices, a deployment step is envisioned which broadcasts the work as a new update through an OTA approach. In this way, all subscribing swarm devices can get the update simultaneously without manual work for the developers.

The CPSwarm Workbench provides exactly the tools necessary to fulfil the workflow mentioned above in designing a swarm.

- The *CPSwarm Monitoring & Configuration Framework* is a runtime tool which enables the interaction between the operator and the swarm. This tool helps the operator to monitor the status of the swarm as well as command the swarm to change behaviours in real-time.
- The *CPSwarm Runtime Environment* is the environment deployed on each single swarm device in a target swarm. It includes an update system, which communicates with the Deployment Tool in the workbench to receive OTA-update (Over-The-Air-update), a telemetry module, which is used to communicate with the monitoring and configuration tool, and an Abstraction Library, which abstracts away the hardware details of each CPS, enabling easy deployment on heterogeneous devices.

4.1.1 Limitations and Open Issues

The initial architecture design contained certain limitations and issues that were discovered as the project progressed.

Heavy coupling between components

One of the goals which CPSwarm strives for is to offer high flexibility to the users. This goal is derived from the observation that multiple tools are available to accomplish a task in a specific workflow step. For example, for simulation, there are different tools available such as Gazebo, Stage and etc. The tool to use often depends heavily on the use case as well as the preference of the user. As a result, a conclusion was drawn that the CPSwarm system should offer enough flexibility to the user so that the user could easily replace a component with a preferred external tool, as long as the external tool support the interface defined within the CPSwarm system. With this requirement in mind, the first limitation could be identified from the initial architecture. In Figure 10, the coupling between components is represented as the socket and lollipop symbol. As you can see, for some component, such as the Modelling Tool, there are five coupling with other components. That means, if the user wants to replace the currently in use Modelling Tool with an external one, the external modelling tool will have to implement all the five interfaces. Implementing several interfaces involves extensive work for every added component. Furthermore, in the case of the Code Generator, there is also an over-reaching coupling. That means, the Code Generator not only has coupling with the Optimization Tool, it also has coupling with the Modelling Tool, which controls the Optimization Tool. This kind of coupling often makes the relationship between components more complicated, and hence reduces the flexibility of the system.

Too much responsibility on Modelling Tool

The Modelling Tool was once envisioned to not only provide the interface for modelling, but also provide a user interface for central configuration for all the tools involved in the CPSwarm Workbench, as well as serve as an interface to launch these tools. As a result, there is high coupling between the Modelling Tool and other components in the CPSwarm system, as mentioned in the previous point. In later discussion we discovered the limitations of having all these functionalities built into the Modelling Tool, as well as the difficulty in replacing this component if it has all these responsibilities. Eventually we came to a conclusion that integrating all these functionalities into the Modelling Tool was an overkill and beyond the scope of modelling.

Inflexible workflow

In some later discussion, a scenario is identified, in which sometimes there is no need for optimization (this could be the case, when an already existing algorithm can be found in the Modelling Library). However, in

the initial architecture diagram, the Optimization Tool is coupled with the Modelling Tool directly. As a result, data must be passed through the Optimization Tool, whether the optimization is necessary or not. It is decided that the architecture should reflect the possibility of having different workflow within the CPSwarm system.

4.2 Updated Architecture Design

An updated version of architecture design was proposed to solve the aforementioned issues. Similar to D3.1, the architecture is described according to ISO/IEC/IEEE 42010:2011 "Systems and software engineering - Architecture description". The functional view, the information view and the deployment view are the three views that we will focus on in this deliverable.

4.2.1 Functional View

Figure 11 shows the updated CPSwarm architecture in its functional view, which outlines the relationship between components. The arrows indicate the flows of information between components, which will be further demonstrated in the next chapter. For the functional view, we focus on the coupling between components. Compared to the initial architecture, new components, e.g. the Launcher Service, the Simulation Orchestrator are added. Besides, the coupling between components is also reworked. The new architecture design addresses the above issues in the following way:

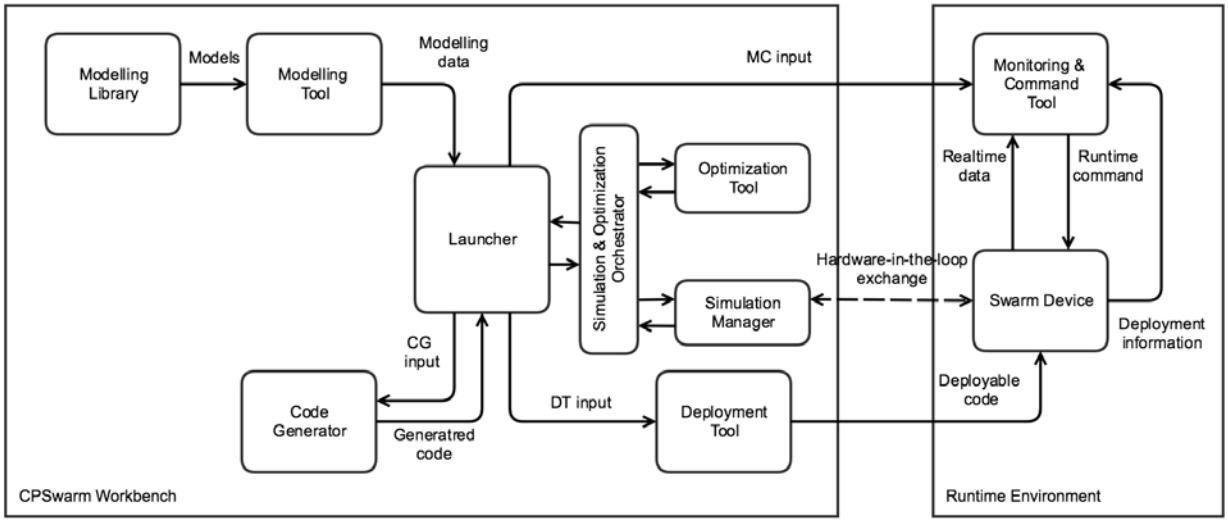

Figure 11. Updated architecture design

Components decoupling with Launcher
One immediately noticeable change in the new architecture is the addition of a component called Launcher. Instead of the Modelling Tool in the initial architecture, the Launcher is now serving as the main interaction point between the user and the CPSwarm system. It provides the user a GUI for configuring different components within the CPSwarm workbench as well as launching them individually. Besides, it offers central storage functionality to group related files into a CPSwarm project to enable easy cooperation among multiple developers. By introducing such a component into the workbench, the system provides low coupling and high cohesion. As shown in Figure 11, only the launcher communicates with all components and this is done only during components launch time. In other words, each component functions with only the knowledge of the existence of the Launcher, but not other components. Further explanation of the Launcher will be presented in the following subchapter.

Relieving the Modelling Tool from overloaded responsibilities
The introduction of the Launcher also solves another issue, which is the overloaded responsibilities in the Modelling Tool. In the new architecture, the Modelling Tool is relieved from the responsibility of managing and launching different components within the CPSwarm workbench, hence is able to focus solely on fulfilling its task of modelling. In this way, it reduces the coupling between the Modelling Tool and other components, which makes replacing it much easier.

Flexible workflow controlled by the user
The Launcher also solves the inflexible workflow issue. As shown in Figure 11, the updated architecture no longer implies an ordered workflow as in the initial architecture. Instead, all components interact solely with the Launcher. As the launcher is the central interaction point for the user, it is up to the user to choose which tool to launch and what the next step for the workflow is. By giving the controlling power to the user, it is possible to enhance the flexibility of the swarm design workflow.

The new architecture also brings some new modifications. One of them is the rework of components related to simulation and optimization. Compared to the initial architecture, an additional component, Simulation Orchestrator, is present in the new architecture. This component manages the optimization/simulation process and hides the details from the launcher, which lowers the coupling between components, and reduces the implementation complexity in the launcher.

The optimization/simulation process is now handled by two components: the Optimization Tool and the Simulation Manager. The Optimization Tool is in charge of generating algorithm via an evolutional approach, while the Simulation Manager control an external simulator to test the generated algorithm. More details will be revealed in the following chapters regarding these two components.

To further illustrate the new architecture, details regarding each of the components shown in Figure 11 are presented in the following subchapters to help the readers have a more complete understanding of the CPSwarm system.

4.2.1.1 *Launcher*

As mentioned in previous chapters, to enhance the flexibility of the CPSwarm system, the consortium has decided to build up the system with highly decoupled components instead of building a piece of monolithic software. To ease the difficulty of managing decoupled components, a launcher is conceived to serve as glue between all these components. The following features have been identified as essential in the launcher:

- The launcher should provide the GUI to launch all components within workbench
- The launcher should provide the storage facility and manage a CPSwarm project, which contains all data generated and needed for all components within the workbench

- The launcher should help the user to navigate through CPSwarm workflow by selectively enabling/disabling certain steps depending on available files
- The launcher should enable the user to select the input files for each component

Figure 12 shows the internal structure of the launcher. As shown in the diagram, the launcher consists of multiple sub-components: the GUI, the Component Launcher and the Project File Manager, each of which serving different purposes.

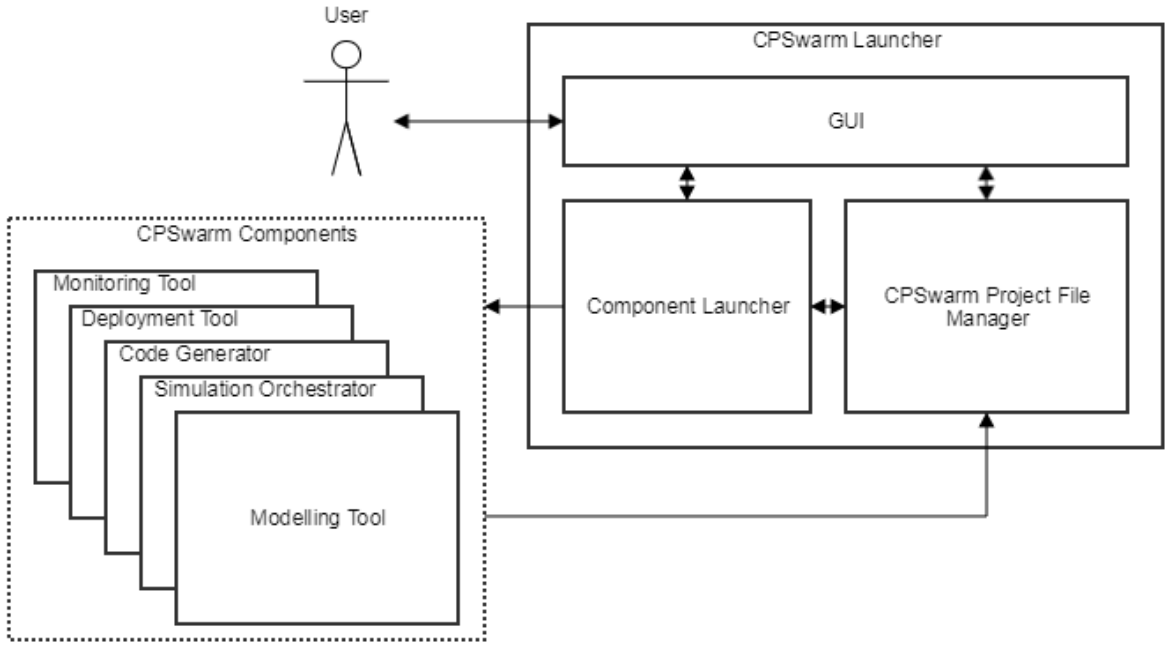

Figure 12. CPSwarm Launcher internal structure

The GUI is the interface which the user interacts with. Figure 13 shows a screenshot of a prototype design of the GUI. On the left side of the launcher, there are different tabs which represent different steps within the workflow of designing a swarm. Once a tab is selected, the detailed content of a specific tab will be shown on the right side. In the detail view of each tab, user can specify the input files as well as the launching parameters for the component to be launched. For example, in the "Sim & Opt" tab (shown in Figure 13), user can specify the modelling files as the input for the Simulation Orchestrator. By clicking the "Launch Simulation Orchestrator" button, the targeted Simulation Orchestrator will be started and the user can proceed to work with the newly started tool.

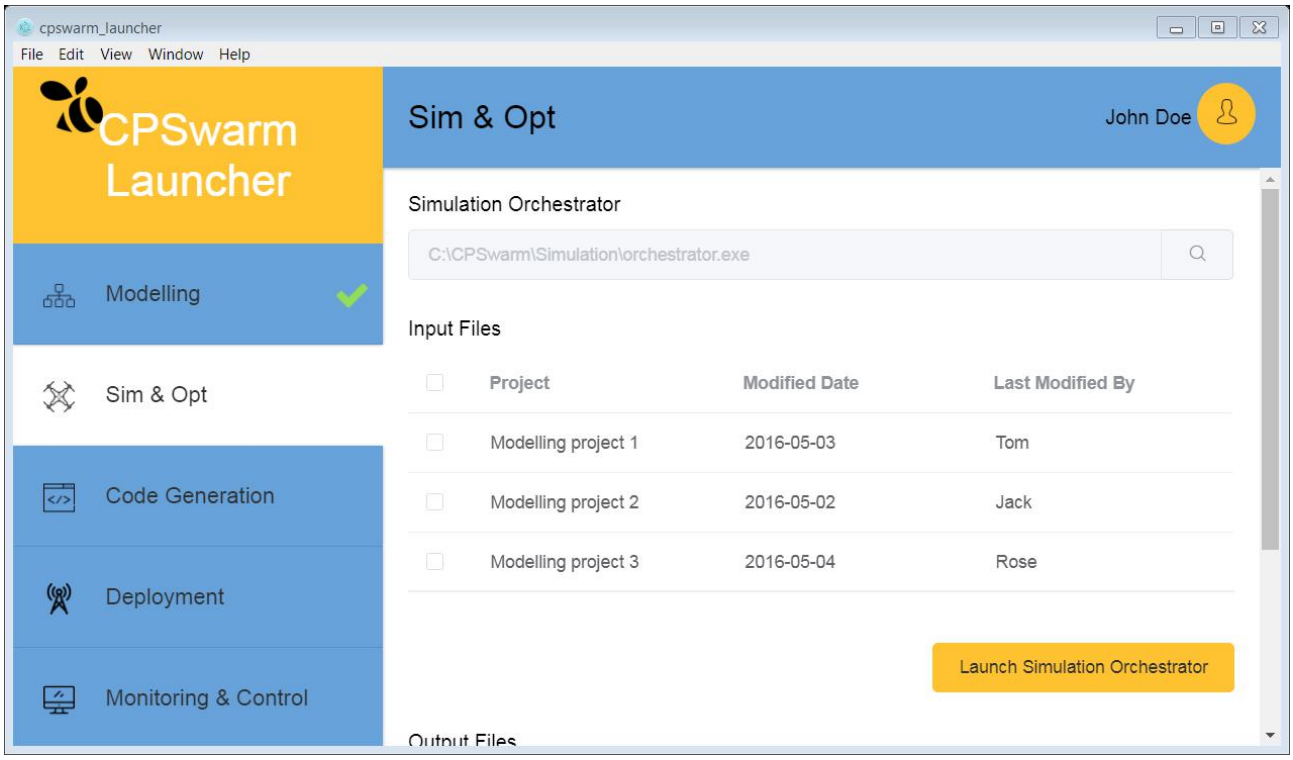

Figure 13. Screenshot of the CPSwarm Launcher

The File Manager inside the launcher provides the facility to manage and store CPSwarm related files. As you may see in the following chapters, within a CPSwarm project, a large variety of files are produced from different components and passed to other components. If all the files are to be manged by the developers by hand, it is not only time-consuming, but also error-prone. Besides, it is difficult to share project files produced by multiple applications among different developers. It is often the case that multiple users are involved in a complete swarm design. To tackle this problem, the launcher is envisioned to help the user to manage all project files. For every swarm to be designed, a CPSwarm project must be created in the launcher. The project provides a pre-defined file structure to store the files generated by different components. When the launcher launches a specific component, the path of the project is also passed into that component. As a result, not only the components could read required files from the project, but also put the generated files into the project without manual interference, which relieves the developers from the tedious task of managing numerous project files. Besides, all the files related to a single CPSwarm project will be grouped under a single directory, which makes project sharing very simple.

The File Manager also provides another functionality, which is file watching. The motivation of this functionality is to detect whether specific files are available within the CPSwarm project. Some actions could only be performed when the prerequisite files are available, for example, the optimization/simulation could only be carried out once the user has finished modelling and necessary modelling files are available. By using the file watcher to detect the existence of these files, the launcher could selectively disable invalid steps in a particular time point, helping the user to navigate through the swarm design workflow in CPSwarm.

4.2.1.2 *Modelling Library*

The Modelling Library's aim is storing and providing reusable parts of models. These reusable parts are sorted by kind inside the library to facilitate their usage. Four kinds of artefacts have currently been identified as part of the CPSwarm modelling library:

- Environment;

- Cost function;
- Swarm Member;
- Hardware Component;
- Behaviour;
- CPSwarm Abstraction Library.

Environment
This part of a CPSwarm model represents the field or environment in which the swarm will be involved. By simulating the swarm behaviour in different kinds of field, it will allow to test its robustness against possible changes of the external conditions.

Cost Function
To evaluate the resulting behaviour of a modelled swarm in a specific environment, one or many criteria related to the result must be defined. Such criteria, called cost-function, are used to optimize the modelled swarm according to one or several environments. A lot of criteria are possible, including:

- Time spent;
- Accuracy;
- Security;
- Robustness;
- Power consumption;
- Etc.

Swarm Member
Swarm Member represent one kind of individuals i.e. CPS involved in a modelled swarm. Composed of multiple Hardware Component and having a specified behaviour, these Swarm Member work together to fulfil – as a Swarm - the Cost Function in an Environment set.

Hardware Component
Hardware Component are the Physical aspect of the Swarm Member. They might be – for example - the controller in which the behaviour is deployed or Sensor responsible of retrieving information from the Environment.

Behavior
The behaviour – modelled as State Machine - is the intelligence of the Swarm Member. It represents how the Swarm Member react to Internal or external event or stimuli.

CPSwarm Abstraction Library
The CPSwarm Abstraction Library model is the model representation of the CPSwarm Abstraction Library – described in detail D7.1. Composed of a set of atomic actions, this library can be used to specify new complex swarm member behaviour in form of Hierarchical Finite State Machines.

4.2.1.3 *Modelling Tool*

The Modelling tool provides a graphical editor (GUI) for the CPSwarm modelling. This GUI allows users to model CPS swarm in term of CPS composition, including their hardware architecture and behaviour, goal – aka fitness function - and environment by using a set of languages specific for the CPS domain, as defined in deliverable D5.1.

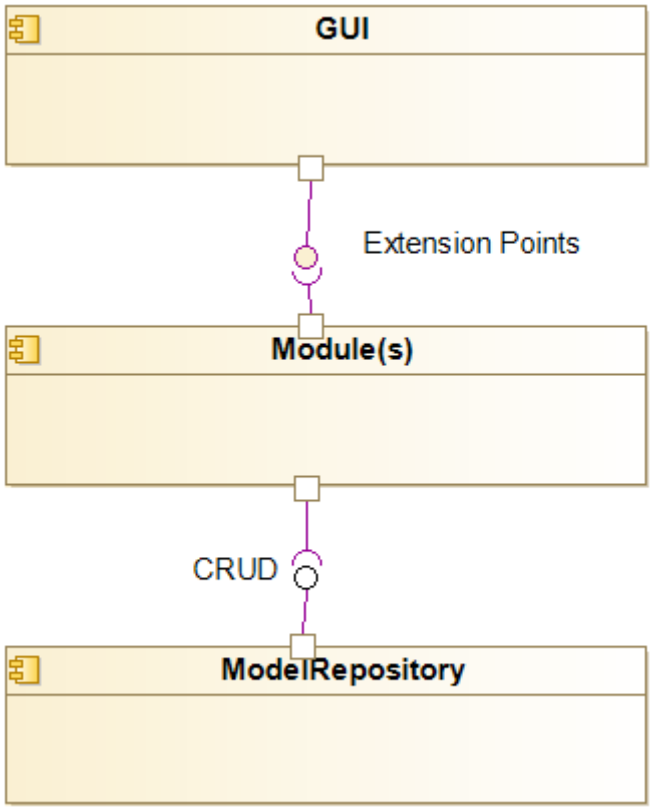

Figure 14. - Functional structure model of Modelling tool

The Modelling tools architecture, depicted in Figure 14, has been already described in D3.1 deliverable. During last period of the project, efforts have been made to detail the internal architecture of the *CPSwarm Module* which is part of the *Module(s)* deployed inside each CPSwarm project. First, required services have been detailed cf. Figure 15. *Extension Points* has been split in *View*, *Diagram*, and *Command Extension Points*. In the same way, distinction have been made in CRUD services between *Read* and *Write* services.

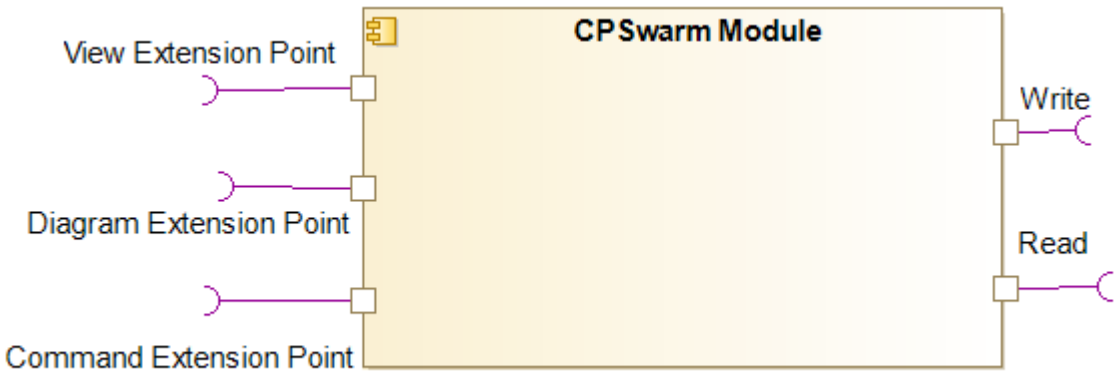

Figure 15. – Detailed services required by CPSwarm Modules

Figure 16 depicts the internal architecture of the CPSwarm Module in term of sub components and services required by each of them. Each sub component is described below.

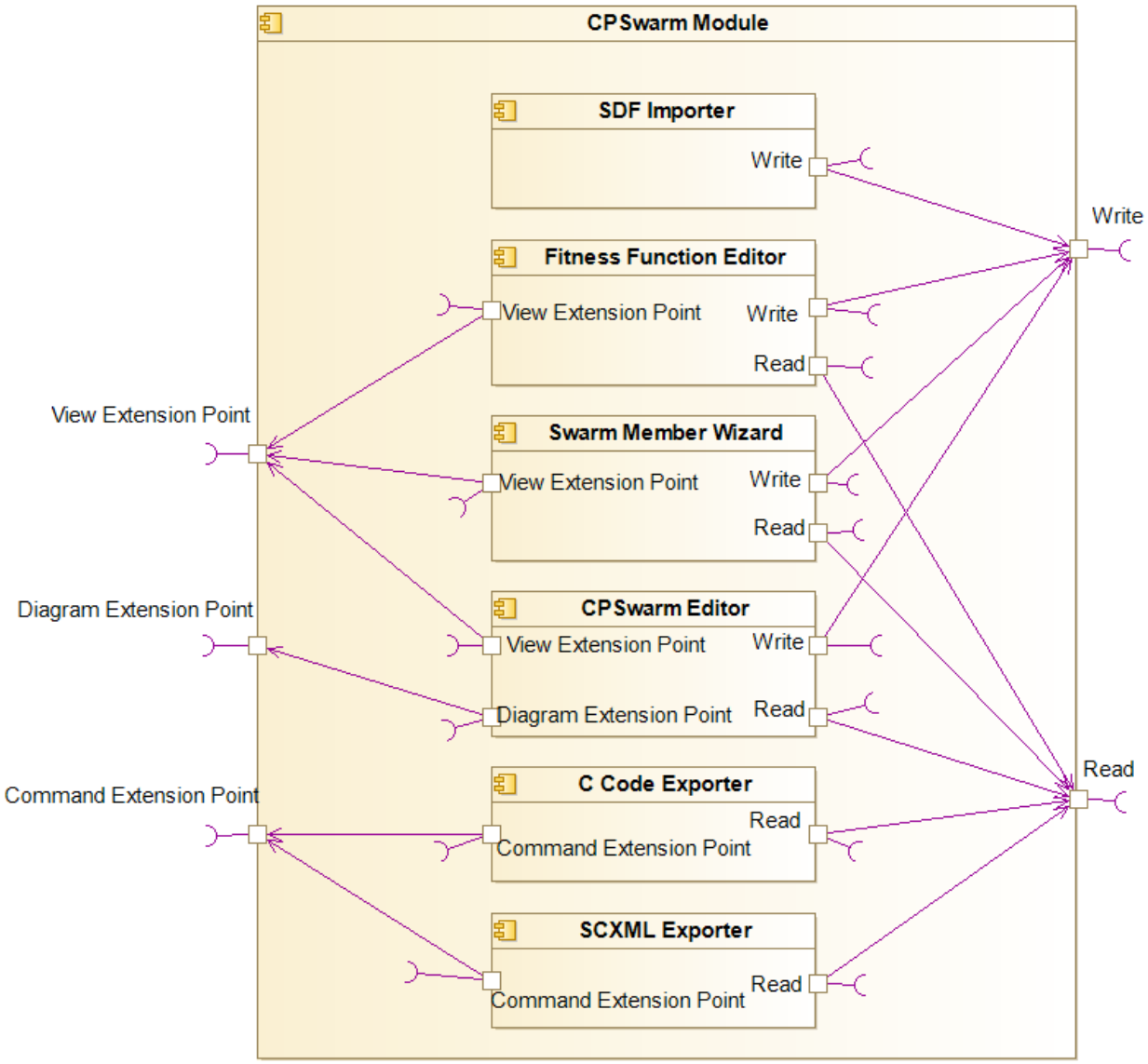

Figure 16. – CPSwarm Module internal architecture

SDF Importer

SDF importer permits to import SDF file inside the modelling tool. SDF format "*is an XML format that describes objects and environments for robot simulators, visualization, and control*". The import result represents the environment in which a Swarm member evolve. It can also be referenced by the *Fitness function* to include environment properties in the CPSwarm goal definition.

Fitness Function Editor

By using the Fitness Function Editor, the user will be able to express the goal of the Swarm as a mathematical expression. More details about the Fitness Function can be found in D6.3.

Swarm Member Wizard

Swarm Member design is the main and more time-consuming activity during the modelling phase. The idea behind this wizard is to reduce the time needed and increase the user friendliness of this activity. Of course, the result of this wizard will still be a valid UML/SysML model but it will be done by using a specific editor instead of UML and SysML diagrams.

CPSwarm Editor
The CPSwarm Editor represents all the diagram and view depicted in D5.. In short, it is composed of all extension made to be able to design CPS swarm but also links all concepts between each other (Swarm, Swarm member, Data/Messages, Behaviour, Environment, Fitness Function, etc.)

C Code Exporter
To achieve independence from the implementation details, *Fitness Function* is expressed as a Mathematical expression. However, the Fitness Function has to be used inside a simulator for optimization purpose to carry out its task of evaluating the performance of algorithm candidates. As a result, this mathematical expression must be translated into code executable by the Simulation environment. The *C Code Exporter* will be the first implementation of such translator. It will take the Fitness function definition as input and translate it to independent C code which can be latter used for Optimization purpose.

SCXML Exporter
The SCXML Exporter is the component responsible of taking swarm device behaviour as entry and export it as SCXML files. These files will later be used as entry by the Code Generator described in section 4.2.1.7.

4.2.1.4 *Simulation and Optimization Orchestrator (SOO)*

This component is in charge of orchestrating the optimization and simulation process. It is the only interface between the Simulation and Optimization environment and the rest of the workbench. The SOO is part of the distributed design for the Simulation and Optimization Environment of the CPSwarm workbench, which has been briefly introduced in the deliverable D6.5 – Initial integration of external simulators, and it is an evolution of the one described in D6.1 – Initial Simulation Environment. The SOO is the centralized component connected, from one side to the Launcher and from the other side to the Optimization Tool and the distributed Simulation Managers using the XMPP protocol[39]. The SOO can be used by the user, both to perform a simulation of an algorithm or to perform an optimization of an evolutionary algorithm. The SOO keeps the updated list of available simulators, so it is able to choose which of them to use for the simulations or optimization processes. In the latter case, when the optimization starts the SOO chooses the list of suitable simulators for that process and indicates them to Optimization Tool, which then uses them for the simulations.
In case of optimization, the SOO receives from the Modelling Tool, through the Launcher, the CPSs models and the environment model, both described using the SDF format[40]. Furthermore, it receives an XML file (based on SDF, with some extensions) that describes the inputs and outputs of the Optimization Tool candidate that correspond to the swarm device sensors and actuators. Finally, it receives also the code of the fitness function to be used to evaluate the optimized behaviour. When the SOO has chosen the suitable simulators, it sends the models and the candidate description to the Simulation Managers that manage these simulators. Furthermore, it sends the configuration file to the Optimization Tool to configure the optimization process, this is done using the XMPP file transfer. Furthermore, a set of XMPP chat messages have been defined to be exchanged among the components (fully described in D6.5). When, the optimization is finished, the SOO receives from the Optimization Tool the optimized code.
In case of simple simulation instead the simulation can receive the algorithm's description and simulate it on the chosen simulator, usually in this case using the GUI of the simulator, to see if the algorithm works properly.
The SOO is currently implemented as a maven-based Java application, which incorporates mainly an XMPP client based on the Smack library[41].

[39] https://xmpp.org/
[40] sdformat.org
[41] https://www.igniterealtime.org/projects/smack/

4.2.1.5 *Optimization Tool*

The Optimization Tool is basically the same described in D3.1 with the addition of an integrated XMPP client, to integrate it with the current version of SOO and Simulation Manager and it uses the API described in D6.5 to communicate with the other components.

4.2.1.6 *Simulation Manager*

The Simulation Manager is the software component that wraps the simulation engine. The broker-based approach designed for the Simulation and Optimization Environment provides distributed simulation engines, each one with one Simulation Manager to handle it. The Managers uses an XMPP client to communicate both with the SOO and the Optimization Tool. From the SOO, the Manager receives the models to be used for the simulation, the description of the inputs and outputs of the candidate and the fitness function code. Instead, from the Optimization Tool the Simulation Manager receives the code of the candidate to be evaluated through simulation. The role of the Simulation Manager is to take all these files and to integrate them in the simulation engine (using its own format), to start the simulation. Once the simulation is finished, in case of optimization, the Simulation Manager uses the fitness function to return the fitness score to the Optimization Tool.
Currently, the Simulation Manager is composed by a set of maven-based Java applications. The generic Simulation Manger project, which contains all the code common to all the managers and then the specific Simulation Managers, one for each simulator, which contain the code specific of the simulator engine. The first Simulation Manager implemented has been the one for Gazebo[42], then other simulators will be integrated starting from the ROS based ones.

4.2.1.7 *Code Generator*

One of the main purposes of the CPSwarm Workbench is to provide a framework able to ease and speed up the development of new CPSs' applications through the use of model-based techniques. Against this background, one of the most common approaches to go from models to deployable code is automatic code generation. The main idea behind this kind of approach is to realize a set of software components that can be reused to produce different outputs according to the varying input they receive. A general know-how of different procedures to realize code generation was depicted in D3.1.

[42] http://gazebosim.org/

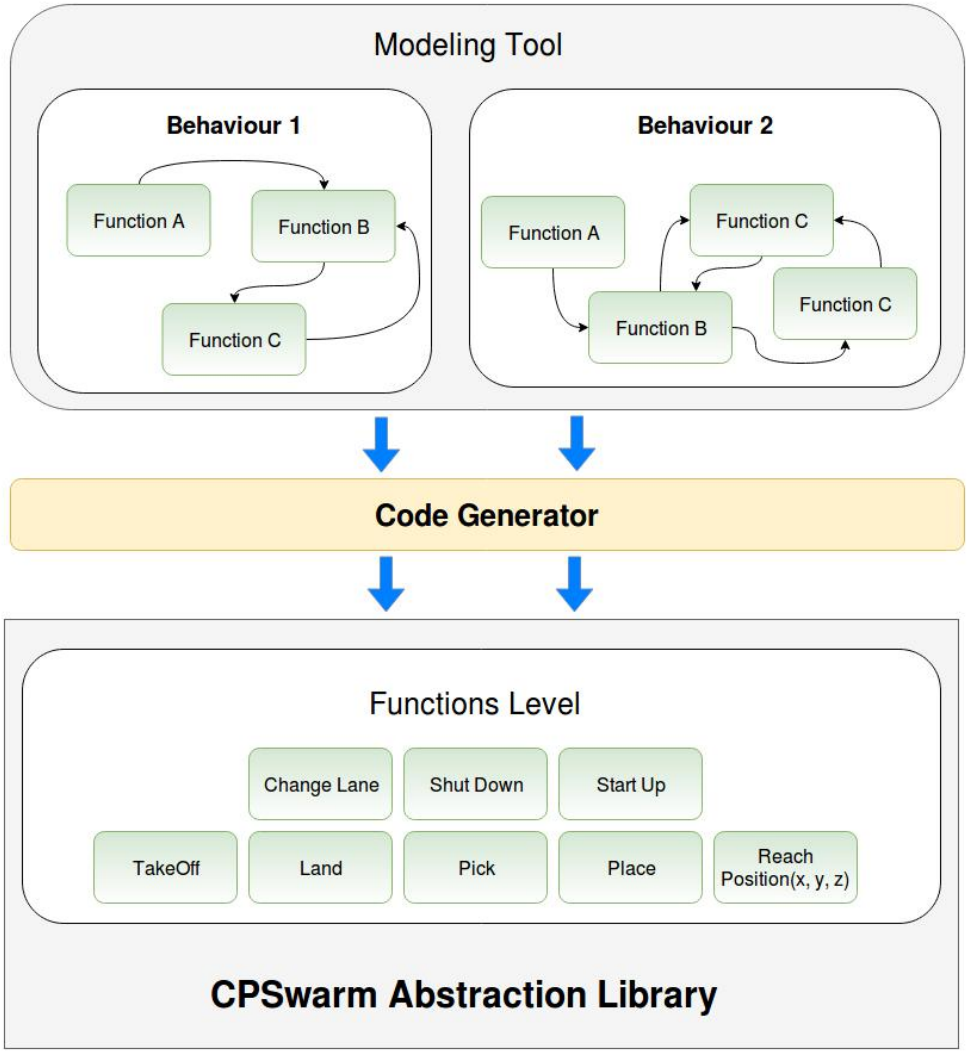

Figure 17. Code Generator role in CPSwarm workbench

In the continuation of tasks 4.3 and 5.3, we decided to focus on the implementation of swarm device behaviour algorithms modelled as Hierarchical Finite State Machines (HFSM). Each state of the state machine can be associated with a high-level functionality provided by the Abstraction Library or with an optimized algorithm coming from the Optimization Tool. In both cases, the role of the Code Generator is to serve as a "glue" level between the platform-independent algorithms realized using the Modelling Tool and the Abstraction Library that provides a set of APIs to access the functionalities of the CPS.

In consequence of that, the template-based generation pattern was identified as the best suited to produce the executable code. Therefore, the main input of the code generator is a description of the state machine using State Chart XML notation with additional information in order to map each active state to a specific function provided by the Abstraction Library. In addition to the state machine description, the Code Generator receive also the target runtime on which the generated code will be executed. For the first implementation, ROS has been selected as only runtime platform, but support to other platforms will be provided through new specific sets of code templates.

4.2.1.8 *Deployment Tool*

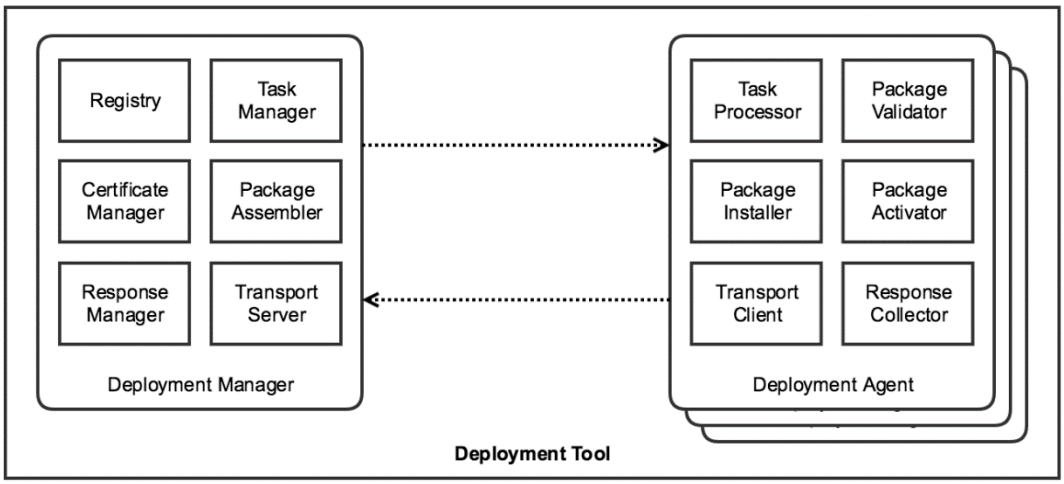

Figure 18. Components of the CPSwarm Deployment Tool

The CPSwarm Deployment Tool is responsible for the over-the-air deployment of generated code on target devices. A deployment task consists of assembly (package preparation), transfer, installation, testing, and activation (execution)[43]. The D3.1 document elaborated initial technical details of the CPSwarm Deployment Tool. While the overall idea remains the same, the design has shifted from a conceptual state closer to realization. The changes address the functional requirements of the system and range from component structure to communication patterns, interfaces, and responsibilities.

Figure 18 shows the latest component structure of Deployment Tool. For better clarity, the Deployment Tool is now broken into the following components:

> **Deployment Manager:** The server-side component responsible for assembly and transfer of packages to designated targets. This component provides an interface to users, enabling task definition, target selection, and status monitoring.
>
> **Deployment Agent:** The client-side component deployed on individual target devices and responsible for installation, testing, and activation. This component communicates with the Deployment Manager using resource-friendly messaging protocols. Each Deployment Agent is also responsible for advertising the target to the Deployment Manager.

Deployment Manager and Deployment Agents communicate with each other over the network using resource-friendly messaging protocols. Section 6.2.2 elaborates the applied techniques to make sure Deployment Manager can coup with large number of Deployment Agents. The components use state-of-the-art security techniques to ensure package integrity, authentication, and authorization. Chapter 5 describes these security techniques in more details.

The detailed description of the Deployment Tool, its requirements, and the first working design will be covered in future deliverable document D7.3 – Initial Bulk Deployment Tool.

[43] Software Deployment: https://en.wikipedia.org/wiki/Software_deployment

4.2.1.9 *Abstraction Library*

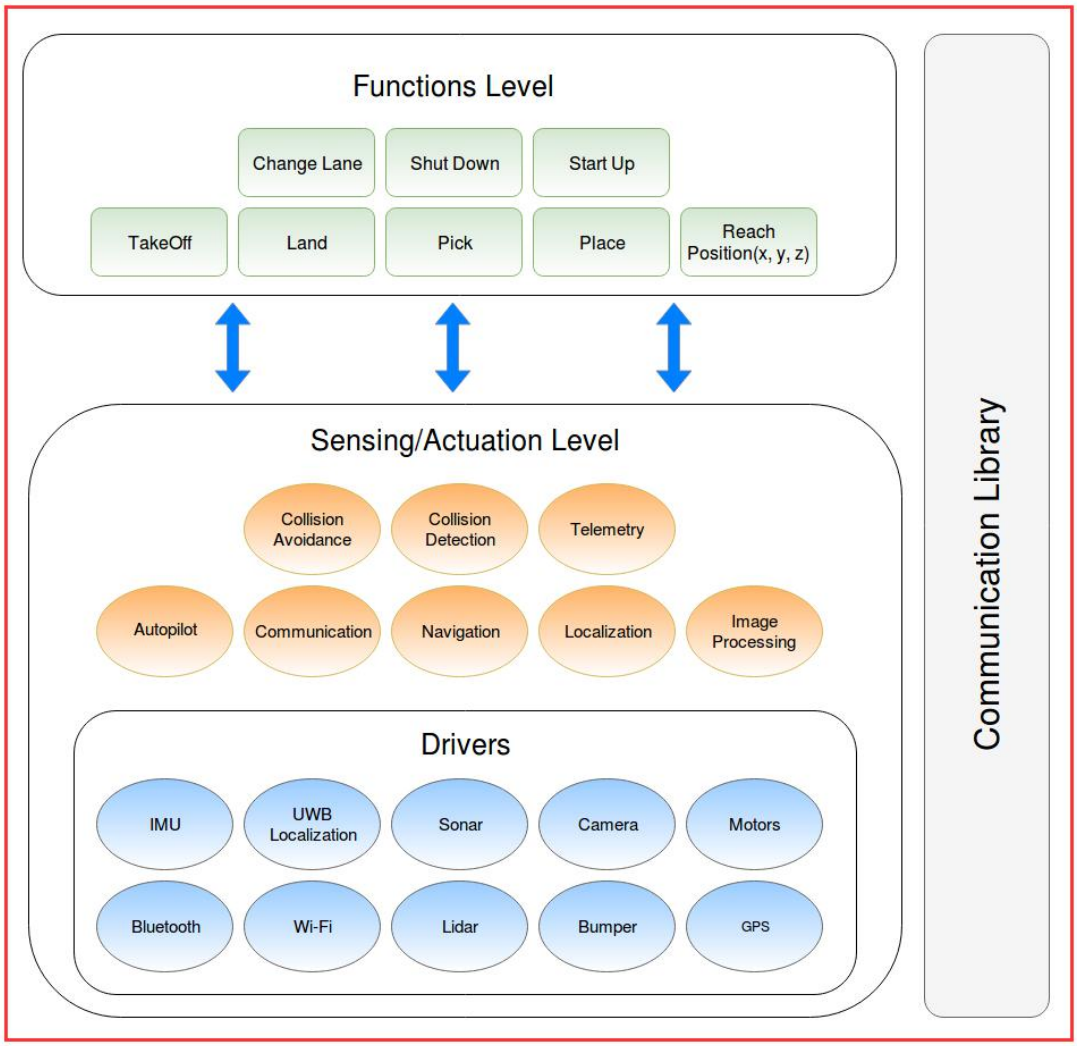

Figure 19. CPSwarm Abstraction Library

As described in the previous architecture deliverable (D3.1), the CPSwarm Abstraction Library cover two different roles inside the project: the former is to provide a set of CPS-specific adaptation libraries in order to access platform-specific information of a robotic system in a standard and coherent way. The latter is to provide support for the development of algorithms using a Model-Driven approach that can be deployed on CPSs using an automated code generation process. Depending on the available computational power, memory and hardware resources of the actual CPSs, the Abstraction Library will be implemented with different levels of complexity and completeness.

The overall structure of the Abstraction Library is depicted in Figure 19. The software has been subdivided into three hierarchical levels: Functions Level, Sensing/Actuation Level and the Drivers Level.

- **Functions Level**: at this stage there is a set of high-level functionalities that constitute the main building blocks used to model new application-dependent's behaviours. Each function interacts with the below level sending specific commands and requesting for specific sensors information.
- **Sensing/Actuation Level**: this level is responsible to provide sensors information and to control the swarm device using his actuators.
- **Drivers Level**: the components of this level are in charge of managing the interaction with the actual hardware mounted on the swarm device.

In addition to these three components, a Communication Library will be developed. This module manages and coordinates all communication of a single swarm device to others swarm devices or to external tools ensuring an appropriate level of security. More details about this component are presented in Section 5.2.1.

4.2.1.10 *Monitoring & Command Tool*

The Monitoring & Command Tool or simply the Monitoring Tool addresses the challenges related to the after-deployment phase, i.e., to the swarm device mission execution. Its main objective is to monitor the swarm members' behaviour by constantly supervising the individual swarm members, the swarm behaviour and performance. Rather than applying local control, it offers the means for continuously checking the performance of real swarm with respect to the mission to reach. In addition to monitoring, the Monitoring Tool also tackles (re-)configuration of swarm members' parameters depending on external factors.

Swarm members can receive commands, e.g., to switch between pre-programmed behaviours, and/or configuration parameters through the channel established by the monitoring tool, exploiting the telemetry core of the runtime environment. Currently, the set of allowed commands as well as the set of envisioned configuration parameters is under development.

The Monitoring Tool runs exclusively in the Runtime Environment. After the deployment phase, the Monitoring Tool is necessary to monitor the actual status of the swarm, as well as to send reconfiguration commands to modify the swarm behaviour, e.g., to abort the mission or to re-purpose part of the swarm members. On one hand, it gathers real-time data from the swarm members and on the other hand, sends out runtime command to the individual swarm members. The information gathered will be presented to the user through the GUI generated in launch time.

Data exchanged between the swarm members and the Monitoring Tool, natively exploits a Publish/Subscribe interaction pattern to account the fact that:

1. Multiple listeners might need to receive telemetry or sensory data, on a dynamic subscription basis. Publish/Subscribe natively support this requirement by decoupling event sources from event consumers.
2. Data may be transferred opportunistically, depending on the actual connectivity and network conditions. This prevents the adoption of any client-server-like interaction paradigm where the CPS acts as server. Cases in which the CPS system plays the client role are possible, however they might not be suited for high-frequency / high-cardinality data streams.

The initial design of the Monitoring Tool will be described in more details in D7.5 "Initial Monitoring and configuration framework".

4.2.2 Information View

In this section, the information flow between components within the CPSwarm system will be examined in detail. The information flow will be further explained in combination with typical swarm design workflow to give the reader better understanding over how the CPSwarm system is supposed to help the developers in rapidly prototyping swarm. Sequence diagrams are used to present the data flows in ordered fashion.

Typically, the workflow of designing a swarm within CPSwarm will involve the following steps: 1) modelling, 2) optimization, 3) simulation, 4) code generation, 5) deployment and 6) runtime monitoring and control. As mentioned above, these steps often involve the usage of different components. To reduce the user's effort in managing all these components, a launcher is envisioned to serve as the glue to connect all the tools. The launcher is the central interaction point for the user. Every time a user needs to execute a certain step, he or she will launch the correspondent component from the launcher, passing necessary data to the component

being launched. When that component has finished its work, it will pass back output files, which are to be stored in a CPSwarm project managed by the launcher.

Since the execution of these steps are highly independent on each other, to improve readability, instead of presenting a complete workflow at once, the data flow in each workflow step will be described individually in the following section.

Modelling

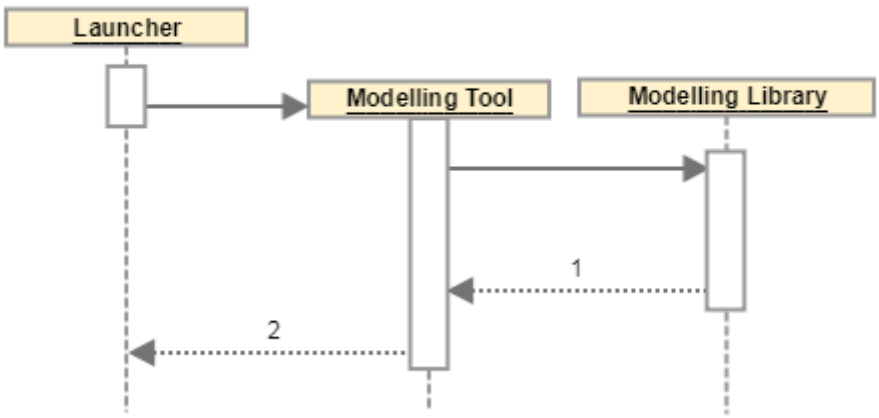

Figure 20. Modelling phase sequence diagram

In the modelling phase (Figure 20), the user launches the Modelling Tool from the launcher. Once the Modelling Tool is launched, the user then switches over to use the Modelling Tool for swarm design. In case pre-existing models are needed, the Modelling Tool will get the data from the Modelling Library, resulting the Message 1, which could include for example, already existing models for different types of CPS, swarm algorithms, etc.

Once the design is finished, data will be passed back to the launcher, resulting the Message 2. The data in Message 2 should provide enough information for executing subsequent tasks such as optimization or code generation. At the moment, the following data are envisioned to be included in this message:

- State machine representation: the description of swarm device behaviour state machine defined in Modelling Tool. The standard SCXML is chosen as the schema for this representation.
- Fitness function: the function used to calculate the fitness score by the Simulation Manager. It is expressed in C-code.
- CPS models: a description of swarm device structure within a swarm. The standard format SDF is chosen to describe such models.
- In-out-description: a description of the input and output of an algorithm candidate.

Optimization/Simulation

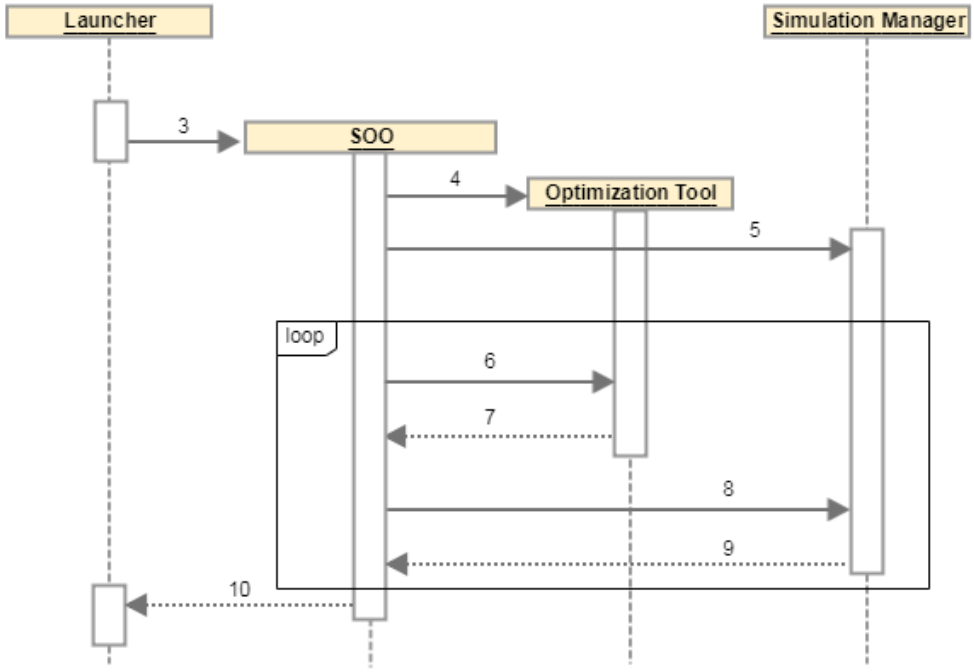

Figure 21. Optimization/simulation phase sequence diagram

As shown in Figure 21, the user starts the optimization/simulation step by launching the Simulation & Optimization Orchestrator (SOO). While launching the SOO, necessary data are passed to it, resulting the Message 3. These data are used to initialize the optimization tool and the simulation manager for subsequent optimization and simulation iterations. It includes the following data:

- Swarm device models: see Message 2 in the Modelling step.
- Environment Description: the description of environment, which will be simulated within simulator. The format SDF is chosen to represent such environment.
- In-out-Description: see Message 2 in the Modelling step.
- Fitness function: see Message 2 in the Modelling step.

The SOO will launch the Optimization Tool. Since the Simulation Manager is a long running service, it doesn't have to be launched by the SOO. In the next step, the SOO initializes the Optimization Tool as well as the Simulation Manager to prepare for the subsequent optimization/simulation iterations. The Message 4 and 5 represent the initial data passed from SOO to Optimization Tool and Simulation Manager respectively. Message 4 contains the configuration parameters for the Optimization Tool. These are parameters used to initialize the Optimization Tool for subsequent optimization iterations. It contains information such as the fitness function, how many iterations to run, the strategy for evolving algorithms, etc. Message 5 represents the data necessary to set up Simulation Manager. It includes the following data:

- Swarm device models: see Message 3.
- Environment SDF: see Message 3.
- In-Out-Description: see Message 3.
- Fitness function: see Message 3.

After these two components have been properly initialized, the optimization/simulation iterations begin. In each iteration, SOO acts as the coordinator between the Optimization Tool and Simulation Manager, passing data between them. In the beginning of each iteration, SOO passes the fitness scores of the algorithm candidates in the previous iteration to the Optimization Simulator (Message 6). The fitness score is a value which indicates how well an algorithm candidate performs in one simulation. It is required by the Optimization Tool to evaluate the correctness of an algorithm candidate and evolves it properly. After that, SOO gets the algorithm candidates from the Optimization Tool, which is expressed in C-code (Message 7). The algorithm candidates are then passed to the Simulation Manager. The Simulation Manager runs the simulation with the algorithm candidates and calculate its fitness scores according to the fitness function. It then returns it back to the SOO, finishing one iteration.

After an optimized algorithm has been found, this algorithm will be returned from the SOO back to the launcher, resulting the Message 10.

Code Generation

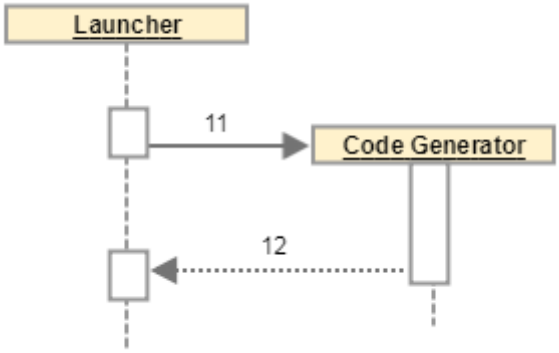

Figure 22. Code generation phase sequence diagram

The task of the code generation step is to generate platform dependent code from the optimized algorithm, so that it could be later deployed on target swarm devices. As shown in Figure 22, the launcher launches the Code Generator and passes data to it (Message 11), which includes the following information:

- Target environment description: the description of the target platform. The Code Generator needs this information to generate platform dependent code.
- State machine description: see Message 2 from the Modelling step.

The Code Generator will then return generated code targeting specific platform, which will be passed to the Deployment Tool later on (Message 12).

Deployment

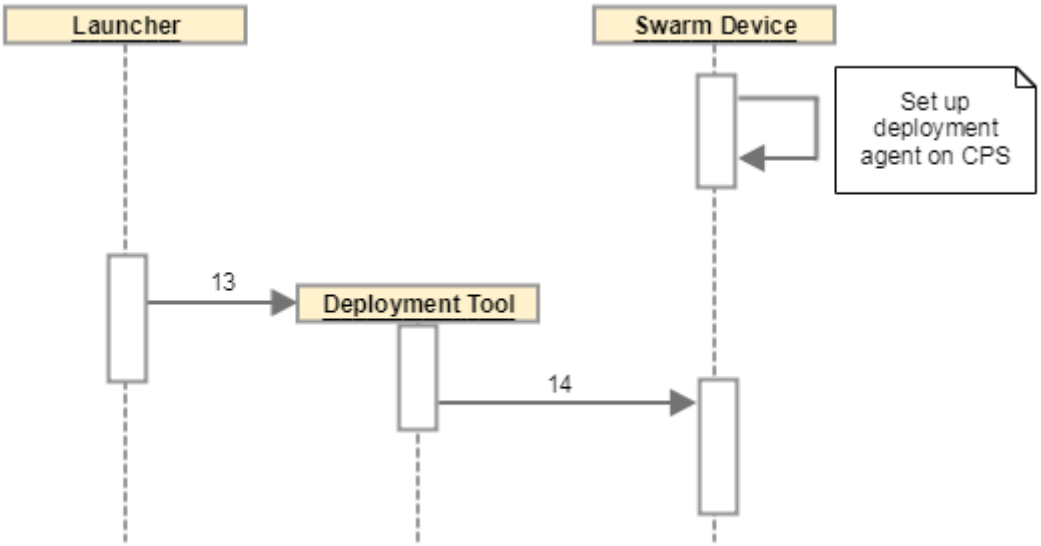

Figure 23. Deployment phase sequence diagram

The task of deployment step is to deploy the generated code on target devices within a swarm (Figure 23). It is important to notice, that in order for the Deployment Tool to work, developers have to set up a deployment agent on each targeted device, as shown in Figure 23. Once that has been done, the Launcher could then invoke the Deployment Tool, passing the following data to it (Message 13):

- Generated Code: see Message 12 in Code Generation step.
- Deployment Configuration: Instructions related to the deployment including the stages (assembly, installation, testing, activation), as well as logging configuration

The output of Deployment Tool will be Message 14, which goes to all targeted swarm devices. It contains the deployable code as well as necessary instructions to be executed on the target.

Monitoring and Controlling

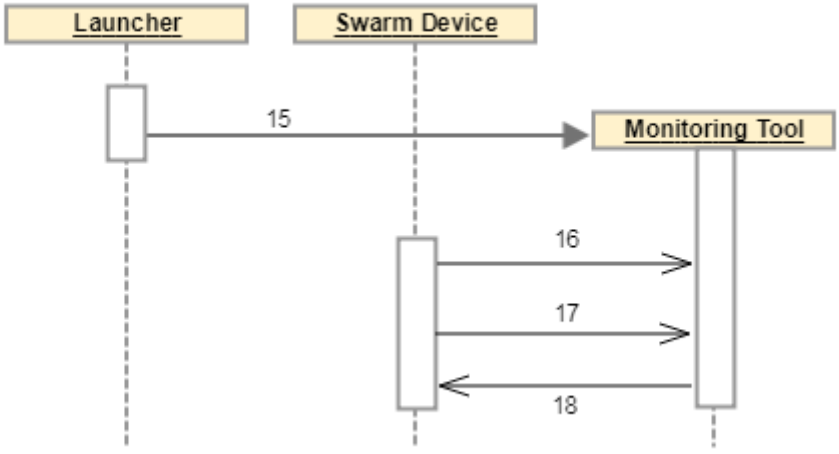

Figure 24. Monitoring and Controlling phase sequence diagram

The task of this step is to enables the user/operator to monitor the real-time status of the swarm as well as change its behaviour during runtime (Figure 24). The launcher starts the monitoring tool, passing necessary

configuration parameters to it (Message 15). After the Monitoring Tool is launched, a swarm device discovery phase will be carried out, in which the swarm device will send data regarding its properties to the Monitoring Tool (Message 16). After that, the Monitoring Tool is ready to monitor and command each member in the swarm. Message 17 represents the data flowing from the swarm device to the Monitoring Tool. It contains the real-time status of the CPS, such as the current location, current speed, current battery life, etc. Message 18 represents the commands sent from the Monitoring Tool to the swarm, such as changing swarm behaviour, shutting down the swarm, etc.

Figure 25 shows all the data flows explained above within the architecture diagram, providing a better overview of the information view in the CPSwarm system. All the data flows are numbered in the same order as illustrated above.

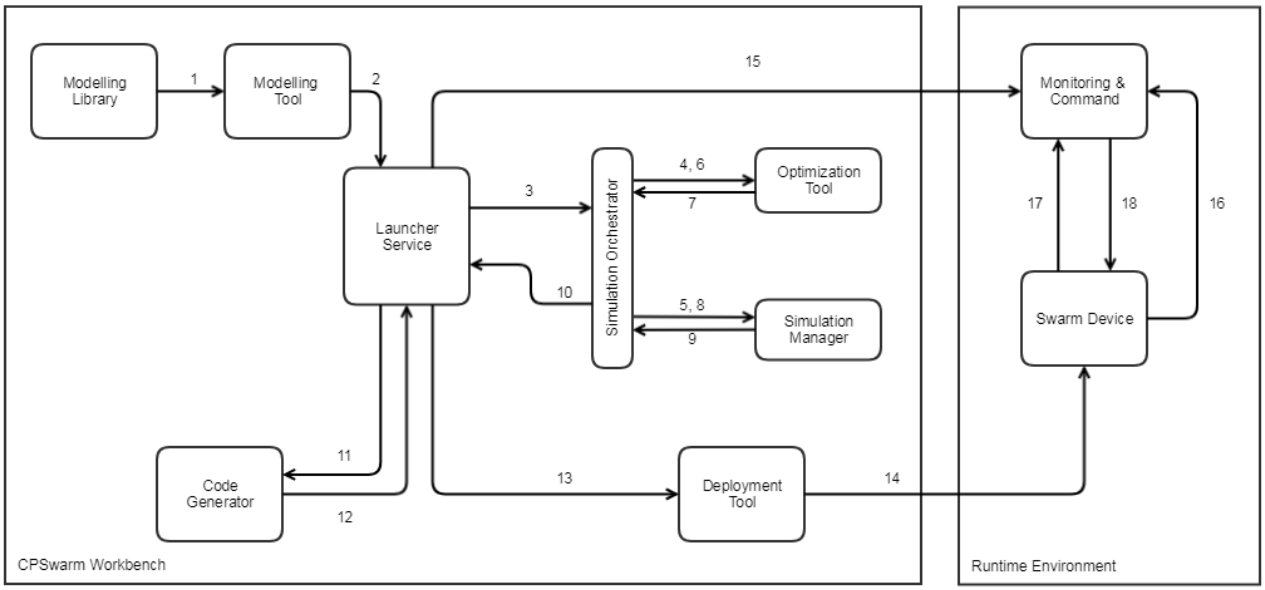

Figure 25. Data flow within CPSwarm system

4.3 Deployment View

Figure 26 shows the deployment view of the CPSwarm system. Notice that Figure 26 is not a strict UML deployment diagram, although it borrows some similar symbols from it. The three-dimensional rectangles represents physical hardware. The rectangles with two decorating small rectangles represent the software components deployed on each hardware. The lines between them highlight the interaction between components.

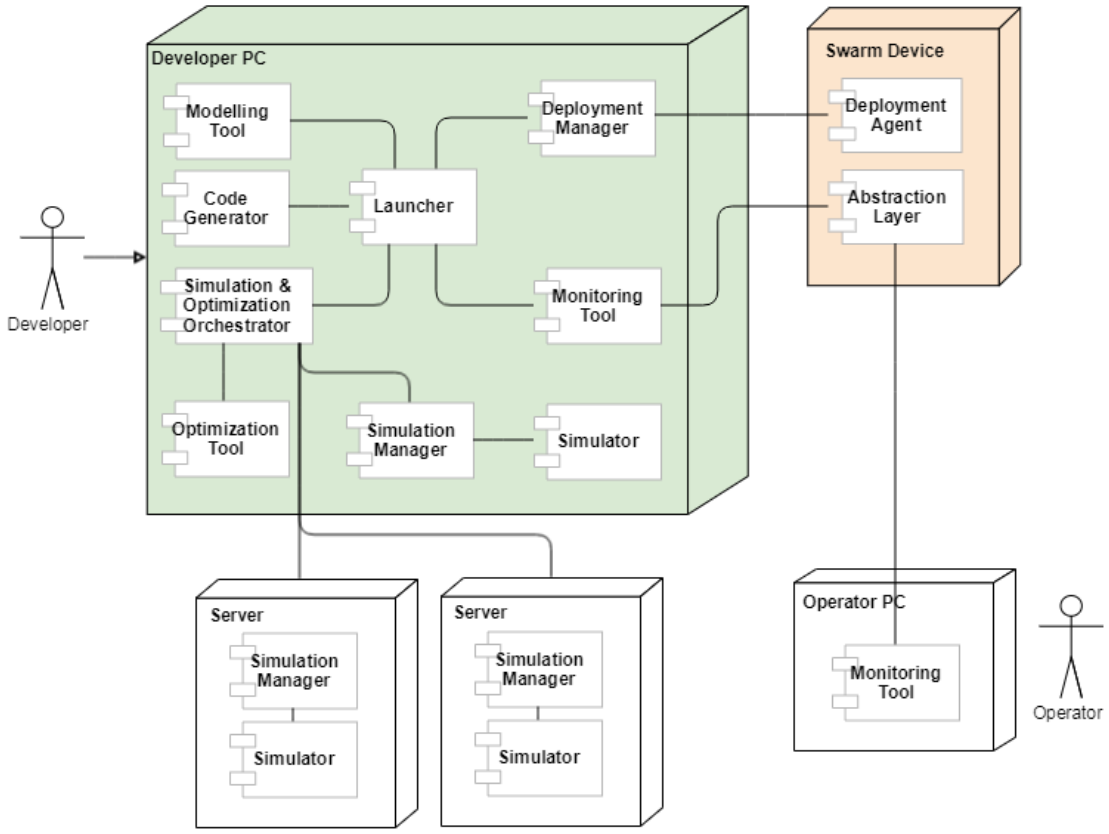

Figure 26. CPSwarm deployment view

Most development-time components, e.g. Modelling Tool, Code Generator, Deployment Tool, etc. are applications which runs locally on the developer's PC. One thing important to notice is that to tackle the scalability issues in simulation and optimization, the CPSwarm system allows the use of distributed PCs to run simulation simultaneously. This feature is shown in the diagram, where the Simulation & Optimization Orchestrator is interacting with three Simulation Managers, one residing locally in the developer's PC, the other two in distributed environment. In simple use cases, where distributed computation is not needed, user can simply spin up the Simulation Manager and the simulator locally to solve the problem. In more complicated situations, distributed simulation servers could be utilized to provide more computing power to speed up the optimization process. All the communication with these Simulation Managers are managed by the Simulation & Optimization Orchestrator.

On each device within a swarm, a Deployment Agent as well as the Abstraction Library is deployed. The Deployment Agent enables the interaction between the CPS swarm device and the Deployment Manager. The Abstraction Library on one hand hides hardware implementation details of the swarm device and provides higher level interface, so that it is easier generate code targeting different platform. On the other hand, it integrates a communication framework, which offers communication functionality to enable real-time communication between swarm members as well as between swarm members and the Monitoring Tool.

The Monitoring Tool provides the user access to the current status of the swarm as well as method to control the swarm during runtime. Since publish-subscribe communication mechanism is used for swarm communication, multiple Monitoring Tools could be present, enabling concurrent monitoring by multiple people.

5 Security and Safety Perspective

5.1 Review of Previous Security Analysis

In the previous architecture deliverable (D3.1), we described a high-level list of possible threats to the CPSwarm System, as well as a collection of possible countermeasures which can be applied both concerning software and hardware security. We collected some observations on how to integrate these countermeasures into the CPSwarm Architecture, from the Design Environment through the workbench components and finally into the Runtime Environment.

5.2 Updated Security Analysis

The Security Analysis conducted at the time of writing of the D3.1 was based on our preliminary Structure Oriented Test and Analysis (SOTA), vision scenarios and the current plan for the CPSwarm Architecture. Since then, the whole concept of the workbench and the use case scenarios has matured, thus in this section we describe the updated security analysis and propose planned countermeasures based on the second version of the CPSwarm Architecture. There have been two security workshops organized by SLAB. First, SLAB proposed security and safety features to be considered for inclusion in the CPSwarm workbench during the plenary meeting in Turin, after which the consortium came to an agreement on the exact features that should be realized. During the second workshop, we focused on implementation details for a subset of these countermeasures to be included in the first demonstration of the CPSwarm Runtime Environment. The subsections below describe the security and safety features we want to implement as part of the CPSwarm project.

5.2.1 Unified framework for secure communications

Our vision was to create a unified solution for all the communications which take place while the swarm is performing its function, including all communications between swarm members and between individual swarm members and the tools included in the CPSwarm Workbench: the Deployment Tool and the Monitoring and Configuration Tool. To make all communications secure, all parties need to be able to authenticate each other and to exchange messages with strong confidentiality and integrity protection. The consortium has agreed on using IP based networking; however, since the project focuses on different vision scenarios, there are multiple network stacks we aim to support:

- Standard, infrastructure mode wireless network (based on IEEE 802.11 a/b/g/n/ac)
- Cellular network (based on 3G/LTE)
- Time triggered wireless network (based on TTTech proprietary technology)
- Low-rate wireless personal area mesh network (based on IEEE 802.15.4)

After considering these requirements (and the requirements derived from our vision scenarios), we decided to build a decentralized solution where swarm members talk to each other directly, without going through a central authority. While this will in turn require more development effort, as there are fewer existing mature solutions, it is an overall better fit for the concept of a swarm – providing increased fault tolerance and more efficient communications over mesh networks.

5.2.2 Platform hardening

As it is used in two of the CPSwarm use cases, ROS (Robot Operating System) is our primary target software platforms. Furthermore, we will select hardware platforms most relevant to the use cases and vision scenarios to target. The platform hardening will start with a security analysis of the target platforms and then ends with testing and deployment. As a final result, for each platform analysed a hardening guide and a target platform optimized image will be delivered.

5.2.3 Fault and tamper detection

This feature is both a security and safety countermeasure – fault detection aims to detect any kind of misbehaviour of the components in order to make it possible to react to them, while tamper detection focuses on the (external) corruption of input values of the components.
Figure 27depicts a vague model of a swarm member, and describes the idea how to handle faults and tampered data by changing behaviour – see Section 5.2.4.

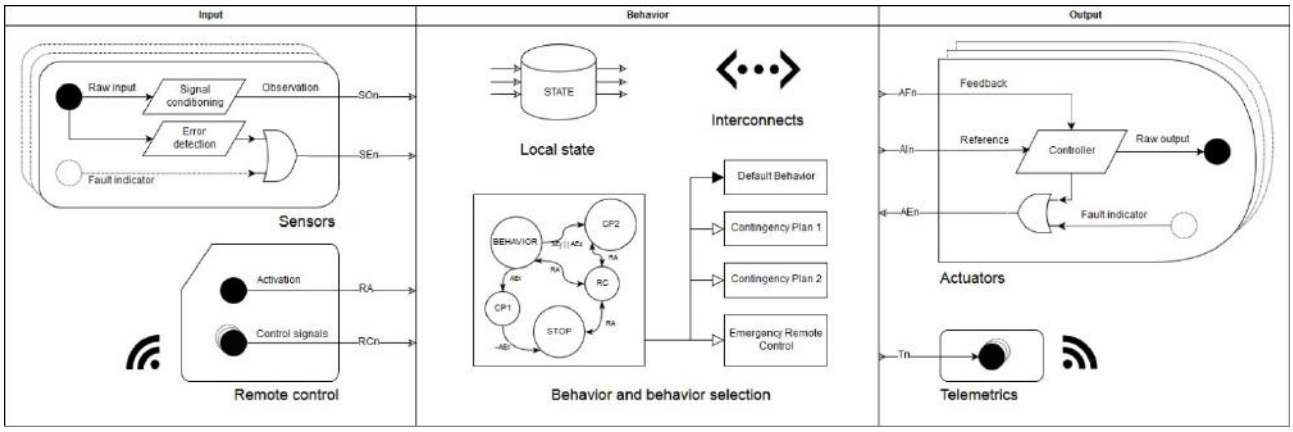

Figure 27. High-level model of a swarm member

5.2.4 Contingency behaviours

Contingency behaviours can be a way to tackle faulty components, such as stopping or going to a safe place when detecting a hardware failure. They can also be triggered by an Operator, through the Monitoring and Configuration Tool or by external event sensed in the environment. This countermeasure both addresses security and safety concerns as different behaviours can be configured to protect tangible and intangible assets. Designing contingency behaviours will be part of the modelling phase, and will be integrated in the design experience of the high-level state machine defining the behaviour of individual swarm members. The Modelling Tool will also support the design process of these behaviour changes by including a feature that makes it possible to model the events that trigger behaviour changes.

5.2.5 Emergency remote control

A different kind of behaviour that can be triggered remotely is the emergency remote control mode, which the Operator will be able to enter using the Monitoring and Configuration Tool. Using this feature, the Operator of the swarm can take control over a swarm member manually in any security or safety critical situation where predefined contingency behaviours might fail, or whenever such control might be required to perform an action the swarm member is incapable of performing on its own.

5.2.6 Emergency shutdown

Emergency shutdown can be viewed as a specific contingency behaviour that can be triggered by either the Operator through the Monitoring and Configuration Tool or by specific events connected to predefined input ranges. This can be an efficient solution to ensure the safety of the swarm, other objects or even humans in a critical situation, for example in extreme weather conditions. We aim to provide two different means of emergency shutdowns:

- Soft-stop – the swarm members return to the base stations
- Hard-stop – the swarm members stop at the next safe opportunity

Apart from being triggered by the Monitoring and Configuration Tool, a physical switch – an IoT device – can also be connected to the swarm and can be used to send the emergency shutdown request, thus providing a safe, emergency shutdown feature for a swarm that is operating autonomously.
Another possible use case for such functionality is to provide a way for authorities to shut down rogue swarm members when the Operator is unavailable or uncooperative – such functionality will require support from the Communications Library in order to identify lawful requests.

5.2.7 Code signing and signature validation

This security countermeasure addresses the Deployment Tool – the code generated by the Code Generator needs to be packaged and signed by the Deployment Manager and then it needs to be validated before execution by the Deployment Agent. The signed package could contain additional restrictions, such as specified target platforms, expiry and downgrade protection.

5.2.8 Rights management

Being able to authorize entities that can affect the operation of the swarm adds an extra layer of security measures to the system. Revoking rights from compromised swarm members can cause limited harm to the whole swarm, and limiting rights to a certain operation can ensure that even if a swarm member is compromised, the damage it can do is limited. Authorization can enforce the separation of different maintenance and monitoring tasks of the operators, such as deployment, monitoring, configuration and remote control.

6 Scalability Perspective

6.1 Review of Previous Scalability Analysis

In D3.1, the importance of scalability within the CPSwarm project was discussed. Furthermore, two aspects which are likely to face scaling bottleneck are analysed: simulation and deployment.

For simulation, in order to find out the proper algorithm for a swarm via the evolutionary approach defined in CPSwarm, a large number of simulation iterations are necessary (often hundreds, thousands or tens of thousands of iterations) to constantly test the performance of a specific candidate. With only a single computer, it could take a very long time before a proper result is found. To tackle this problem, the initial analysis indicated that a solution with distributed computers each running simulation should be envisioned within the CPSwarm system.

Besides simulation, deployment is another aspect which requires careful design in terms of scalability. When dealing with a large number of target devices, the typical approach in which developers deploy newly updated program manually to each single device is extremely repetitive and error-prone. To solve this problem, an update system similar to the Over-The-Air (OTA) update in modern smart phone was envisioned to be implemented in CPSwarm.

6.2 Updated Scalability Analysis

6.2.1 Simulation scalability analysis

The current design of the Simulation and Optimization Environment, briefly presented in D6.5, has the goal to address the main scalability issues seen in the approach described in D6.1 [18]:

1) The first decision taken has been to use the XMPP protocol instead of MQTT for the communication among the components, allowing to use the several mechanisms provided by XMPP. Core XMPP features[44] include unique identifiers, presence mechanism and one-to-one chat messages; or some of the protocol extensions, like file transfer[45] and publish/subscribe[46]. This enables us to leverage the solid scalability features provided the protocol[47, 48].

2) The use of XMPP has allowed improvements to the architecture, refactoring the discovery mechanism, used in the previous approach, for the distributed Simulation Managers. Indeed, that mechanism introduced delays for the time required to the Simulation Managers to handle a large amount of discovery requests (a new discovery starts for every simulation). For this reason, in the new approach, the previous solution has been replaced with a more scalable mechanism, where the Simulation Managers announce themselves to the Simulation & Optimization Orchestrator when they are available, leveraging the XMPP presence mechanism.

3) Finally, in the approach described in D6.1, the simulation of the controller was done keeping it in the Optimization Tool and exchanging messages with the simulation environment. This increased the number of messages and worsened the performances. To address this issue, in the new version, the controller is sent to the Simulation Manager using the file transfer protocol and it is evaluated locally

[44] https://xmpp.org/rfcs/rfc6120.html
[45] https://xmpp.org/extensions/xep-0096.html
[46] https://xmpp.org/extensions/xep-0060.html
[47] https://www.igniterealtime.org/about/OpenfireScalability.pdf
[48] https://www.isode.com/whitepapers/xmpp-performance-constrained.html

in the simulator, sending back only the fitness score, allowing to limit the number of messages exchanged and to increase the performance.

4) The new design will be completely analyzed and evaluated in D6.2 – Final Simulation Environment, due at M28.

6.2.2 Deployment scalability analysis

The CPSwarm Deployment Tool is built on top of the initial OTA update concept. This provides tremendous benefits when it comes to tedious deployment tasks on a number of devices. This relieves users from the burden of direct software deployment but requires an appropriate set of technologies to optimally deploy software on resource-constrained swarm devices.

The Deployment Tool's update system works based on a publish-subscribe messaging pattern tailored for management of a large number of devices using a simple, usable interface provided by the Deployment Manager. The publish-subscribe messaging pattern enables scalable update propagation and monitoring in contrast with a request-reply pattern that relies on frequent polling. Furthermore, it saves network traffic by multicasting packets at the edge of a CPS network using the Pragmatic General Multicast[49] protocol. This way, the Deployment Manager sends a single copy of messages to a remote network, multicast locally to designated targets.

The Deployment Manager is designed with concurrency in mind, however as a centralized instance, it is still subject to host environment limits. To overcome scaling issues when dealing with thousands of devices, the Deployment Manager could benefit from a broker-based architecture with a simple load-balancing scheduler.

[49] https://en.wikipedia.org/wiki/Pragmatic_General_Multicast

7 Future Steps

This deliverable specified an updated architecture design for the CPSwarm system. This new updated architecture addresses most of the open issues identified during phase 1 of the project. However, due to time limits, some questions and details still remain open at the time of writing. As a result, the architecture design iteration will continue in the future. More dedicated discussion about improving the architecture are envisioned. Any future changes and improvements will be documented in the later deliverable 3.3 "Final System Architecture Analysis and Design Specification", which is due on M30.

This deliverable should also serve as a roadmap for upcoming component development and the phase 2 component integration. The integration result will be documented in D3.5, which is due on M22.

8 Conclusion

This deliverable presented an updated analysis over the available methodologies, tools and standards for swarm development as well as an updated version of architecture design for the CPSwarm system from D3.1.

Since the initial phase, as more ideas over the CPSwarm matured, new problems have been identified and new solutions have been discovered. As a result, an updated architecture design was proposed to solve the existing problem with the initial design. The new architecture design shows the direction of development of new components as well as the revision of already existing components.

Now that the updated architecture is defined, this deliverable paves the way for future activities in Task 3.3, *Continuous System Integration*. The second phase of component integration will be carried out and the result of which will be documented in the upcoming deliverable 3.5.

As next steps, the architecture design will still be further revisited, revised and refined as the project evolves. The new modification will be documented in later deliverable D3.3 "*Final System Architecture Analysis and Design Specification*".

Acronyms

Acronym	Explanation
CPS	Cyber Physical System
SITL	Software-in-the-loop
VTOL	Vehicle Take-off and Landing
ROS	Robot Operating System
DNS	Domain Name System
SysML	System Modelling Language
MARTE	Modelling and Analysis of Real-Time Embedded Systems
MQTT	Message Queuing Telemetry Transport
TLS	Transport Layer Security
OASIS	Advancing Open Standards for the Information Society
HTTP	Hypertext Transfer Protocol
AMQP	Advanced Message Queuing Protocol
SASL	Simple Authentication and Security Layer
XMPP	eXtensible Messaging and Presence Protocol
DDS	Data Distribution Service
ZMQ	ZeroMQ
SOO	Simulation and Optimization Orchestrator
OTA	Over-The-Air
V-REP	Virtual Robot Experimentation Platform
ARGoS	Autonomous Robots GO Swarming
STDR	Simple Two-Dimensional Robot Simulator
API	Application Programming Interface
SDF	Simulation Description Format
XML	eXtensible Markup Language
GUI	Graphical User Interface
CRUD	Create, Read, Update, and Delete
SOTA	Structure Oriented Test and Analysis

List of figures

List of tables

Reference

[1] IEEE, *ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description,* 2011.

[2] D. Green, A. Aleti und J. Garcia, *The nature of nature: Why nature-inspired algorithms work,* 2017.

[3] H. Hamann und T. Schmickl, „Modelling the swarm: Analysing biological and engineered swarm systems," *Mathematical and Computer Modelling of Dynamical Systems,* pp. 1-12, 2012.

[4] C. J. L. Lim und S. Dehuri, Innovations in Swarm Intelligence, Springer, 2009.

[5] E. Bonabeau, M. Dorigo und G. Theraulaz, Swarm intelligence: from natural to artificial systems, Oxford University Press, 2008.

[6] S. Camazine, N. Franks, J. Sneyd, E. Bonabeau, J. Deneubourg und G. Theraula, Self-organization in Biological Systems, Princeton University Press, 2001.

[7] S. Garnier, J. Gautrais und G. Theraulaz, „The biologicl principles of swarm intelligence," *Swarm Intelligence,* pp. 3-31, 1 2007.

[8] C. Blum und X. Li, *Swarm intelligence in optimization.*

[9] D. Floreano und C. Mattiussi, Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies, MIT Press, 2008.

[10] R. Parpinelli und H. Lopes, „New inspirations in swarm intelligence: A survey," *International Journal of Bio-Inspired Computation,* 2011.

[11] S. Binitha und S. Sathya, „A survey of bio inspired optimization algorithms," *International Journal of Soft Computing and Engineering,* pp. 137-151, 2 2012.

[12] X. Yang, Z. Cui, R. Xiao, A. Gandomi und M. Karamanoglu, Swarm intelligence and bio-inspired computation: theoriy and application, Elsevier.

[13] J. Krause, J. Cordeiro, R. Parpinelli und H. Lopes, „A survey of swarm algorithms applied to discrete optimization," *Swarm Intelligence and Bio-Inspired Computation,* 2013.

[14] A. Hassanien und E. Alamry, Swarm Intelligence: principles, Advances and Applications, CRC Press, 2015.

[15] X. Yang, S. Deb, Y. Zhao, S. Fong und X. He, „Swarm intelligence: past, present and future," *Soft Computing,* 2017.

[16] M. Brambilla, E. Ferrante, M. Birattari und M. Dorigo, „Swarm robotics: a review from the swarm engineering perspective," *Swarm Intelligence,* pp. 1-41, 7 2013.

[17] M. Shranz, M. Umlauft, M. Rappaport und W. Elmenreich, „A classification of basic swarm behaviors and their application in cyberphysical systems," *Swarm Intelligence,* 2018.

[18] D. C. M. J. M. S. E. F. W. E. Micha Rappaport, „Distributed Simulation for Evolutionary Design

of Swarms of Cyber-Physical Systems," in *ADAPTIVE 2018*, 2018.

[19] H. Ahmed und J. Glasgow, „Swarm intelligence: Concepts, models and applications," School of Computing, Queen' University, Canada, 2012.

[20] A. Kolling, P. Walker, N. Chakraborty, K. Sycara und M. Lewis, „Human interaction with robot swarms: A survey," *IEEE Transactions on Human-Machine Systems,* pp. 9-26, 2016.