# D3.3 – FINAL SYSTEM ARCHITECTURE & DESIGN SPECIFICATION

| | |
|---|---|
| Deliverable ID | **D3.3** |
| Deliverable Title | **Final System Architecture & Design Specification** |
| Work Package | **WP3 – Architecture design and Component Integration** |
| Dissemination Level | **PUBLIC** |
| Version | **1.0** |
| Date | **2019-07-10** |
| Status | **Final** |
| Lead Editor | **Farshid Tavakolizadeh (FRAUNHOFER)** |
| Main Contributors | **Junhong Liang (FRAUNHOFER), Etienne Brosse (SOFTEAM), Melanie Schranz, Micha Sende (LAKE), Ákos Milánkovich, Judit Torma (SLAB), Andreas Eckel (TTTECH), Angel Soriano (ROBOTNIK), Omar Morando (DGSKY), Davide Conzon, Gianluca Prato (LINKS), Arthur Pitman (UNI-KLU)** |

**Published by the CPSwarm Consortium**

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

## Document History

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 0.1 | 2019-05-17 | Junhong Liang (FRAUNHOFER) | Initial draft |
| | 2019-06-04 | Andreas Eckel (TTTECH) | Minor revision |
| | 2019-06-04 | Etienne Brosse (SOFTTEAM) | Revision |
| | 2019-06-05 | Davide Conzon, Gianluca Prato (LINKS), Arthur Pitman (UNI-KLU) | Updates to Related documents, Simulation and Optimization Orchestrator, Optimization Tool, Simulation Manager, Code Generator |
| 0.2 | 2019-06-07 | Farshid Tavakolizadeh (FRAUNHOFER) | Merged changes |
| 0.3 | 2019-06-13 | Farshid Tavakolizadeh (FRAUNHOFER) | Updated functional view chapter |
| | 2019-06-13 | Ákos Milánkovich, Judit Torma (SLAB) | Updated security analysis |
| 0.4 | 2019-07-01 | Farshid Tavakolizadeh (FRAUNHOFER) | Polished the final architecture and overall content |
| 1.0 | 2019-07-10 | Farshid Tavakolizadeh (FRAUNHOFER) | Addressed interval review comments. Finalized for submission. |

## Internal Review History

| Review Date | Reviewer | Summary of Comments |
|---|---|---|
| 2019-07-05 | Davide Conzon, Gianluca Prato (LINKS) | Approved with minor comments |
| 2019-07-09 | Arthur Pitman (UNI-KLU) | Approved with comments |

## 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 the final version of a series of deliverables (D3.1-D3.3), documenting the final architecture design of the CPSwarm system. The architecture design documented in this deliverable is the final result of multiple discussion and design iterations within the consortium. It serves as the high-level blueprint for the development of individual software components within the project.

The final architecture design is heavily based on the second phase documented in D3.2 - Updated System Architecture Analysis and Design Specification. To enhance the readability, content from D3.2 is reused in this document to make it a complete and standalone document. The final architecture is then documented with similar methodology used in previous series of deliverables, complying with the ISO/IEC/IEEE 42010 System and software engineering (IEEE, 2011) standard. Relevant viewpoints of the system are presented as documentation for different architectural aspects of the CPSwarm system.

Besides the main functional design, cross-cutting aspects such as security, safety and scalability are also important topics in architecture design. The consortium has realized that these aspects must be addressed not only in the implementation of single component, but also on a system level. As a result, the consortium's consideration regarding these aspects in separate sections at the end of this document.

# 2 Introduction

The CPSwarm architecture specifies the structure of a system enabling model-based realization of collaborative, autonomous CPSs. It consists of several cohesive, standalone components addressing different aspects of swarm modelling, simulation, optimization, code generation, deployment, runtime, and monitoring.

This document presents the architecture according to ISO/IEC/IEEE 42010:2011 (IEEE, 2011), an international standard for architecture descriptions of systems and software. The document focuses on functional view, information view, and deployment view presented in chapters 3.1, 3.2, and 3.3 respectively.

In the second part of the document, design consideration regarding important cross-cutting aspects, such as security, safety and scalability are elaborated. These aspects require efforts not only for the implementation of a single component, but also on the system level. As a result, this document briefly discusses them in the scope of system architecture.

## 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 |
| D3.2 | Updated System Architecture & Design Specification | D3.2 | 1.0 | M18 |
| 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 |
| D6.2 | Final Simulation Environment | D6.2 | 1.0 | M28 |
| D6.3 | Initial CPS System Design Optimization and Fitness Function Design Guideline | D6.3 | 1.0 | M18 |
| D6.4 | Final CPS System Design Optimization and Fitness Function Design Guideline | D6.4 | 1.0 | M30 |
| D6.5 | Initial Integration of External Simulators | D6.5 | 1.0 | M18 |
| D6.6 | Updated Integration of External Simulators | D6.6 | 1.0 | M28 |
| D7.1 | Initial CPSwarm Abstraction Library | D7.1 | 1.0 | M18 |
| D7.3 | Initial Bulk Deployment Tool | D7.3 | 1.0 | M21 |

# 3 Final Architecture Design

In this chapter, the final architecture design is documented in detail. In accordance to ISO/IEC/IEEE 42010:2011 architecture description, this chapter describes the functional, information and deployment aspects of the final architecture.

## 3.1 Functional View

Figure 1 shows a functional view of the final CPSwarm architecture, which outlines the boundaries between components as well as their relationships. The following sub-chapters focus on explaining the functionality of each component. The arrows indicate the flow of information between components, which are described in Section 3.2.

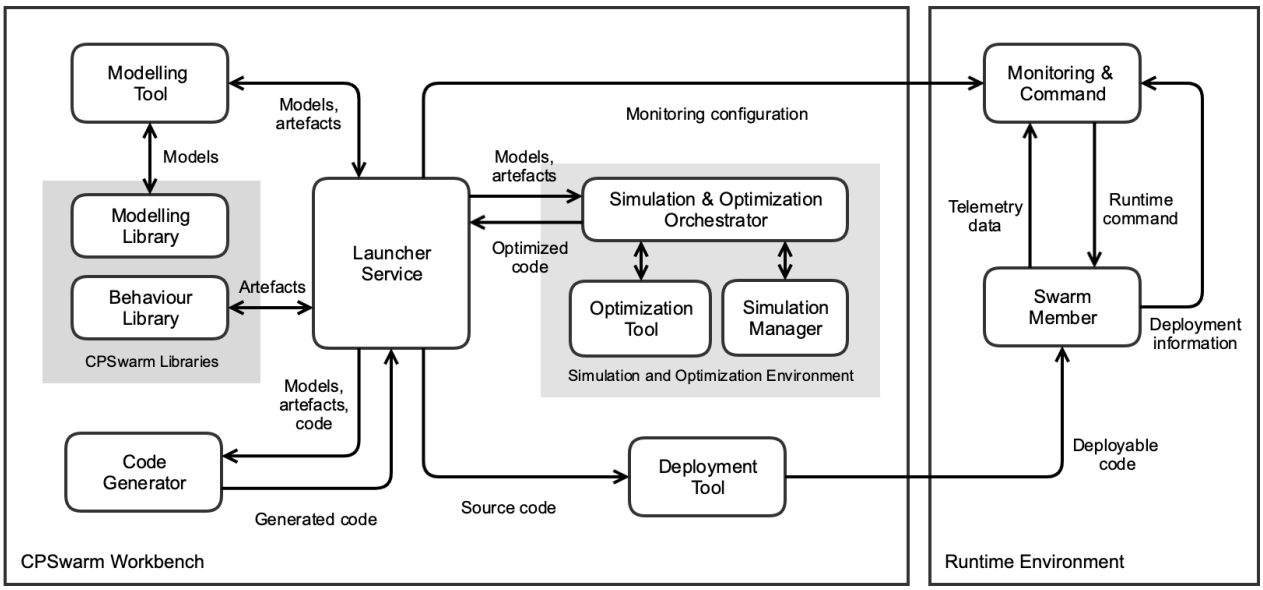


**Figure 1. Final architecture design**

### 3.1.1 Launcher

One goal of CPSwarm is to build a flexible and extensible swarm design system. To enhance the flexibility of the system, the consortium decided to build the system with highly decoupled components instead of monolithic software. To ease the difficulty of managing decoupled components, a launcher with a graphical user interface (GUI) and backend service is developed to interconnect the different APIs of components. The launcher service offers the following functionalities:

- A GUI to launch all components within the Workbench, helping the user to navigate through the CPSwarm workflow by selectively enabling/disabling certain steps depending on available files
- A storage facility to manage a CPSwarm project which contains all data generated and needed for all components within the Workbench
- Orchestration of configurations and input files between components

Figure 2 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.

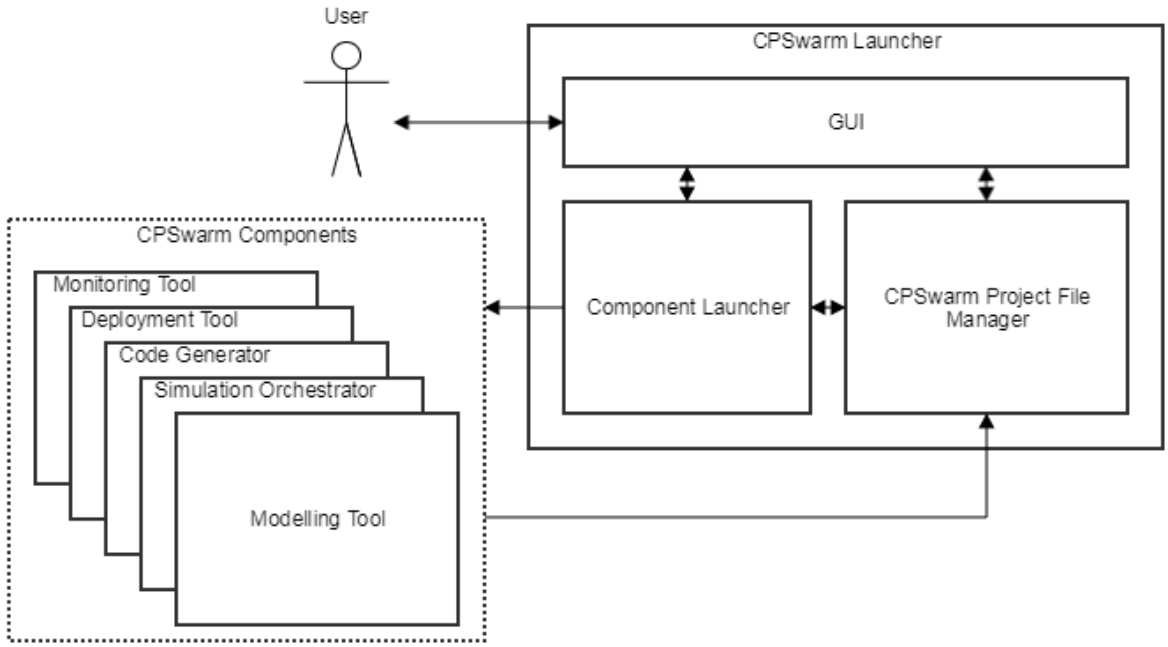


**Figure 2. CPSwarm Launcher internal structure**

The GUI is the interface which the user interacts with. Figure 3 shows a screenshot of a prototype design of the GUI. On the left side, the Launcher includes tabs which represent different steps within the workflow of realizing a swarm system. 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, the user can specify the input files as well as the launching parameters for the component to be launched. For example, in the "Swarm Modelling" tab (shown in Figure 3), the user can specify the modelling files as the input for the Simulation Orchestrator. Selecting the "Launch Simulation Orchestrator" button starts the target Simulation Orchestrator such that the user can proceed to work with the newly started tool.

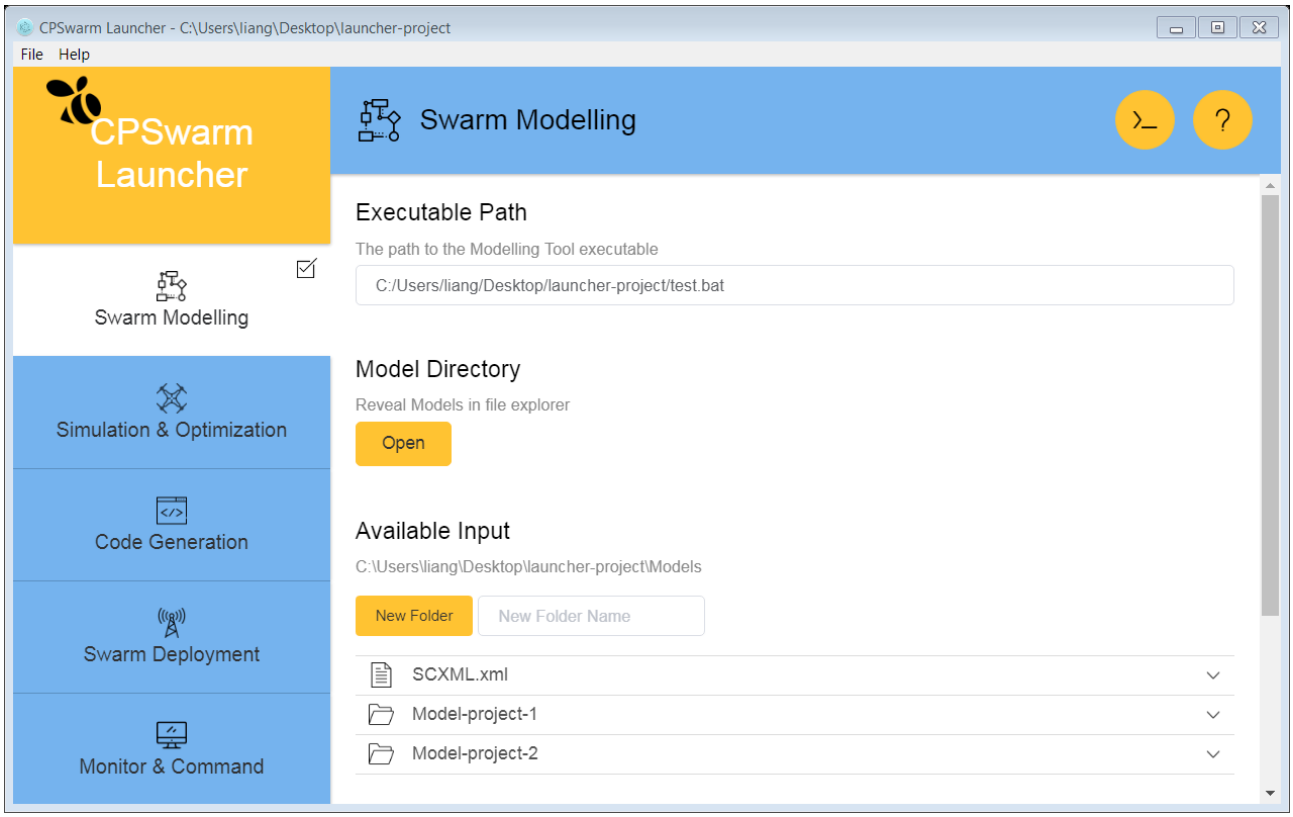


**Figure 3. Screenshot of the CPSwarm Launcher prototype**

The File Manager manages CPSwarm related project files. As described in the following chapters, a large variety of files are passed between different components within a single project. Managing these files manually would be a tedious and error-prone task. Besides, it is difficult to share project files produced by multiple applications among different developers, which is often the case when multiple users are involved in a complete swarm design. The Launcher tackles this by providing a structured directory model. For every swarm design, a CPSwarm project is created in the Launcher. The project provides a pre-defined file structure to store the files generated by different components. When the Launcher starts a specific component, the path of the project is also passed into that component. As a result, the components can read required files from and place generated files into the project without manual interference. Moreover, the Launcher detects generated files and automatically enables/disables available functions. This helps users follow a correct set of operations without missing mandatory steps. Lastly, the uniform file structure allows project collaboration using off-the-shelf file sharing and version control systems.

The CPSwarm Launcher will be thoroughly described in D3.6 – Final CPSwarm Workbench and associated tools.

### 3.1.2 Modelling Tool

The Modelling Tool provides the GUI interface for the user to model different aspects of a swarm. Compared to previous designs, the Modelling Tool focuses on modelling only and performs a smaller set of tasks. The reason for this change was to reduce dependencies on any single component and create a more modular, extensible solution.

The Modelling Tool now focuses on modelling swarm behaviour related aspects. Other functionalities, such as the modelling of hardware specifications, such as a swarm member's 3D model, equipped sensors and the properties of each sensor (e.g. accuracy, noise, etc.) are out of the scope of the Modelling Tool. Instead, such

hardware specifications are expected to be provided by the manufacturers or open source communities. In addition, the modelling of environment is now expected to be carried out by third-party tools.

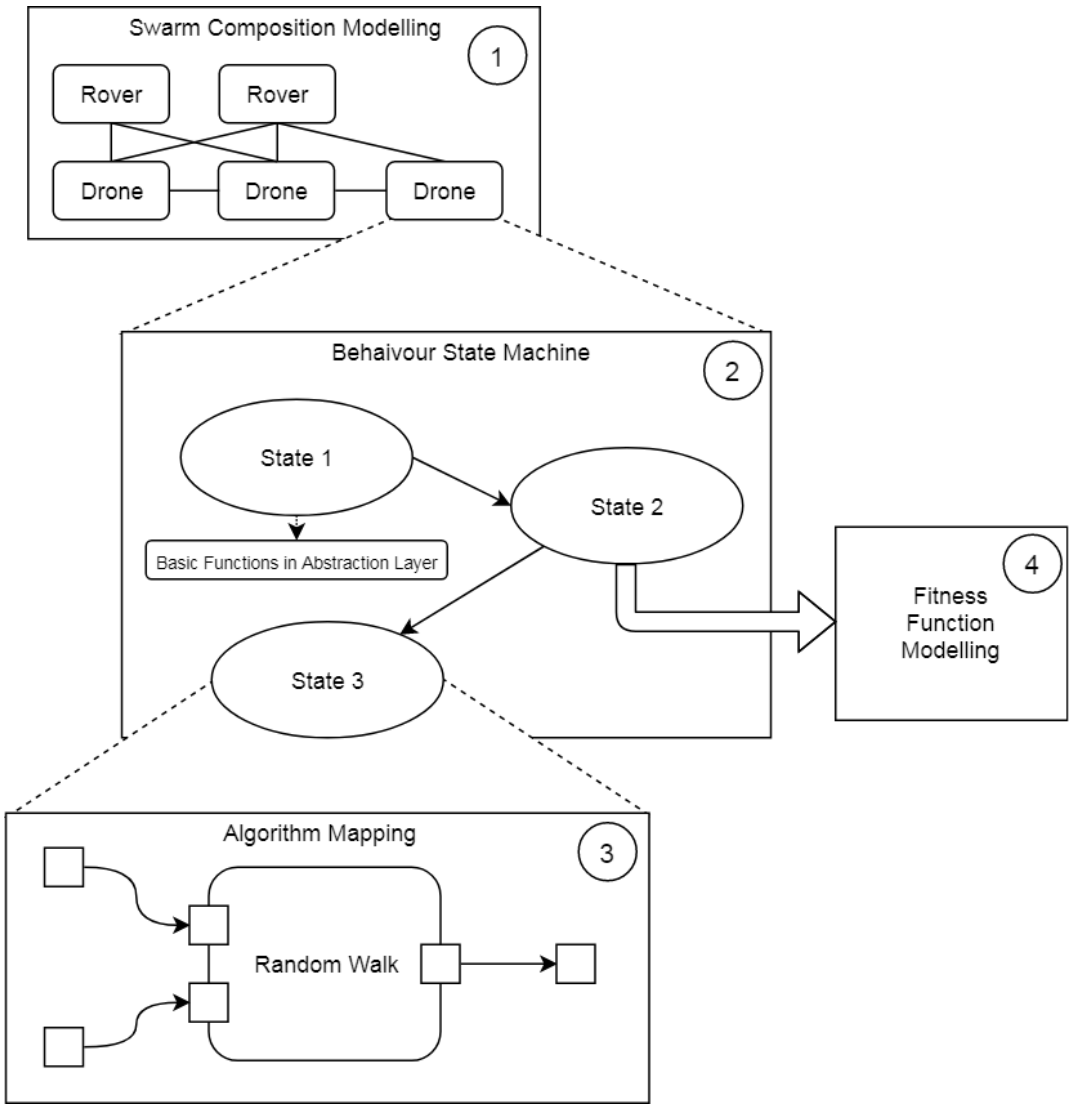


**Figure 4. Hierarchical illustration of the core tasks of the Modelling Tool**

In the final architecture, the following functionalities have been identified as the core tasks of the Modelling Tool:

1) Modelling of the swarm composition
2) Modelling of behaviour state machines
3) Mapping of algorithm input and output
4) Modelling of fitness functions

Before explaining each core task in detail, it is beneficial to have an overview of the relationships between these tasks. Figure 4 illustrates such relationships in a hierarchical way. The user will typically start the swarm design process by modelling the swarm composition for the mission (marked as *1* in Figure 4). For each of the devices in the swarm, the user will model a behaviour state machine (marked as *2* in Figure 4), which serves as the backbone for each device's behaviour in different situations. For a certain state, the user may want to use an existing algorithm to control the device in that state. In this case, a mapping between the input/output of the algorithm and the input/output of the device's Abstraction Layer has to be done (marked as *3* in Figure 4). Alternatively, the user may decide to use the optimization process to generate a new algorithm for a certain

state. In that case, the user will model a fitness function for that state (marked as *4* in Figure 4), which will later be used by the Optimization Tool to find the best algorithm candidate.

In the following paragraphs, each core task is presented in detail.

**Modelling of Swarm Composition**
The first thing a user needs to do when designing a swarm is to determine the composition of the swarm. That means, the user needs to model e.g. how many rovers or drones are used in the mission, what their models are, and how they communicate and share information with each other. These pieces of information are needed during the simulation and optimization phase to simulate the swarm.

**Modelling of Behaviour State Machine**
After the composition of the swarm has been defined, the next step would be to model the behaviour for each swarm member. In CPSwarm, the consortium has decided to use a finite state machine to represent the high-level behaviour of a swarm member. Each state within the state machine represents a certain algorithm that controls the behaviour of the swarm member in a specific situation. For example, the drones in the search and rescue demo may have one state in which they cover the infrastructure to inspect for anomalies, while another state may guide trapped victims to the nearest exit. One important task of the Modelling Tool is to provide an interface to model such state machines. During this process, the user defines the algorithm to be used in a specific state, as well as the trigger conditions between states. According to the use cases, a user may want to use a pre-defined algorithm or generate an algorithm using the Optimization Tool for a certain state. Different modelling activities have to be carried out based on the chosen algorithm, as outlined in the following paragraphs. The modelled state machine is later converted into runnable code by the Code Generator.

**Mapping of Algorithm Input and Output**
The user may choose to use a pre-defined algorithm for a certain state (e.g. *State 3* in Figure 4). A pre-defined algorithm would typically require certain inputs (such as input values of an infrared distance sensor) and produces certain output (such as the twist vector for the wheel actuator). In order to use the algorithm on a swarm device, the user needs to map the required input of the algorithm to the proper sensor APIs in the Abstraction Layer and map the generated output to the actuator APIs in the Abstraction Layer. The Modelling Tool should provide a graphical interface for the user to make such a mapping, which is similar to the *Algorithm Mapping* box marked as *3* in Figure 4.

**Modelling of Fitness Functions**
Alternatively, user may choose to use the Optimization Tool to generate an optimized algorithm for a certain state (e.g. *State 2* in Figure 4). In order for the optimization process to work, the user needs to model the so-called fitness function (marked as *4* in Figure 4). The fitness function is a function which takes some KPIs of a simulation as input and outputs a score. The output score indicates how well the algorithm behaves within the simulation. The fitness function is needed by the Optimization Tool during optimization phase, so that it can use the fitness function to evaluate the performance of each candidate algorithm, hence selecting the best and eliminating the worst.

The final specifications of the Modelling Tool will be presented in D5.4 – Final CPSwarm Modelling Tool.

### 3.1.3 Modelling Library

The Modelling Library stores and provides reusable parts of models for the Modelling Tool. It is an archive of SysML 1.2 models serialized using the XMI 2.1 standard.

The Modelling Library contains the following types of models:

**CPS Hardware Specifications**

The CPS hardware specifications include information related to the hardware aspects of a specific CPS. For example, the 3D model of the CPS, the specifications of its sensors and actuators (such as the accuracy, bias, noise). Such information is required to carry out simulations of the CPS.

**Environment**
This part of a CPSwarm model represents the field or environment in which the swarm will operate. It describes the environment, such as how large it is and what objects exist within it. This is needed to simulate the swarm behaviour in different kinds of fields, testing 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, known as cost functions, are used to optimize the modelled swarm according to an environment. A lot of criteria are possible, including time spent, accuracy, security, robustness, and power consumption.

The final design of this component will be described in D4.3 – Final CPS modelling library.

### 3.1.4 Behaviour Library

The Behaviour Library is a collection of software libraries that enable the operation of swarm devices at different layers. These libraries are open source and available as public Git repositories. The initial set of libraries are being developed as part of CPSwarm will be available on CPSwarm GitHub organization[1]. The libraries can be extended by external contributors via Git forking. An extended library may remain and be used as a standalone repository or submitted to the master repository via a Git pull request. Merging external changes into a master library is subject to the approval of the library owner.

The Behaviour Library included two set of libraries:

**Abstraction Library**
The CPS Abstraction Library includes libraries that implement communication functionalities as well as abstraction of heterogeneous hardware and native CPS functionalities. These libraries are the building blocks of an Abstraction Layer for a specific CPS device. It is expected that the Abstraction Layer is deployed on target devices, before they can interact with other components within the CPSwarm system. Besides the source code implementation, the Abstraction Library contains also provide an abstraction description file (ADF), which describe the exposed APIs of the Abstraction Layer implementation. For more detailed explanation, please refer to chapter 3.1.10.

**Swarm Library**
The Swarm Library includes implementations of specific behaviour algorithms that control the CPS in a specific state. These algorithms do not require optimization and will be deployed to the swarm device by the Deployment Tool. Besides the source code implementation, each swarm algorithm should also provide a so-called algorithm meta file (AMF) which describes the required inputs and the produced output of the algorithm. This piece of information is required by the Modelling Tool to do the mapping between the algorithm input/output and the Abstraction Layer input/output APIs.

The Behaviour Library will be described in D4.6 – Final Swarm modelling library and D7.2 – Final CPSwarm Abstraction Library.

[1] https://github.com/cpswarm

### 3.1.5 Simulation and Optimization Orchestrator (SOO)

This component orchestrates the simulation and optimization 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 D6.2 - Final Simulation Environment and 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 (OT) and the distributed Simulation Managers (SMs) using the eXtensible Messaging and Presence Protocol (XMPP)[2]. The interaction among the components is described in D6.4 - Final CPS system design optimization and Fitness function design guideline. The SOO can be used to perform either a simulation of an algorithm or optimization using an evolutionary algorithm. Both the SOO and the OT maintain a list of available simulators, so they can choose which of them to use for the simulations or optimization processes.

In the case of optimization, the SOO receives models of the CPSs and the environment from the Modelling Tool, both described using the SDF format[3]. Furthermore, it receives an XML file (based on SDF, with some extensions) that describes the inputs and outputs of the OT candidate that correspond to the sensors and actuators of each swarm member. Finally, it receives 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 SMs that manage these simulators. Furthermore, it sends the configuration file to the OT to configure the optimization process. Then, a set of XMPP chat messages are exchanged among the components (fully described in D6.4). When the optimization is finished, the OT sends the SOO an optimized controller.

In the case of simple simulation, simulators receive the algorithm's description and simulate the scenario typically displaying a GUI, allowing the user to see if the algorithm works properly. The last version of the Simulation and Optimization Environment architecture has introduced new scalability and deployment features (please refer to D6.2 for full details). In this last release, the SOO can also be used also to deploy the required set of SMs in a cluster of distributed machines using Kubernetes[4]. Using a web interface, the user is able to monitor and orchestrate a large set of distributed SMs. The SOO is currently implemented as a Maven-based Java application, which incorporates a XMPP client based on the Smack library[5].

### 3.1.6 Optimization Tool

The Optimization Tool (OT) optimizes a control algorithm for an agent using a modular approach, where the distinct steps of evolutionary design are split into different components (see Figure 5). It starts with a generic representation of a control algorithm and searches for a viable solution using a heuristic search algorithm. The control algorithm needs to be evolvable, i.e. modifiable by mutation and recombination. An example for such a representation is an Artificial Neural Network (ANN). To apply the iterative heuristic search to find an optimized configuration of the controller for a CPS, a measure of fitness needs to be defined for a given problem scenario. A fourth component takes care of evaluating a pool of possible algorithm candidates and provides a ranking for the evolutionary algorithm.

In the CPSwarm context, the result is a controller that implements local interaction rules that lead to the desired global behaviour of the system. The controller can be evaluated with the Optimization Simulator by testing it in a reference scenario or by performing a statistically significant number of simulations on a given scale of parameters under predefined conditions.

[2] https://xmpp.org/
[3] sdformat.org
[4] https://kubernetes.io/
[5] https://www.igniterealtime.org/projects/smack/

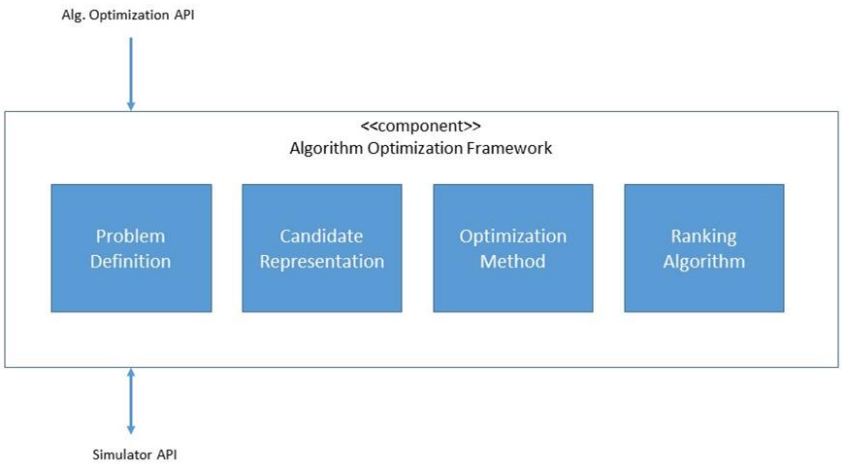


**Figure 5. Components of the Optimization Tool**

The OT integrates an XMPP client to communicate with the SOO and the SMs using the set of APIs defined in D6.4. It collects the presences of the available Simulation Managers and when it receives a start message from the SOO, it chooses the ones to use and communicate directly with them in parallel, sending the candidate controllers to be evaluated and receiving back fitness scores calculated using the designed fitness function. After it has finished the optimization process, the OT sends the optimized controller back to the SOO. The OT implements mechanisms that allow it to recover from failure during the optimization process.

### 3.1.7 Simulation Manager

The Simulation manager (SM) 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 SM to handle it. The SMs use XMPP to communicate both with the SOO and the OT. From the SOO, the SM receives the models to be used for the simulation, the description of the inputs and outputs of the candidate and the fitness function code. The OT provides the SM with the code of the candidate to be evaluated through simulation. The role of the SM is to integrate all the files into the simulation engine (using its own format) and to start the simulation. During optimization, the SM uses the fitness function to compute the fitness score once the simulation is finished, which is subsequently returned to the OT.

The SM is composed of a set of Maven-based Java applications, namely the generic SM project, which contains all the code common to all the managers, and the specific SM for each simulator. The partners are working on a refactored version of the SMs, based on the use of the ros-osgi project[6] that will be released by the end of the project and described in deliverable D6.7 – Final Integration of External Simulators. At the time of writing, specific SMs for Gazebo[7] and Stage[8] have been implemented. The integration of other simulators requires further evaluation and is envisioned by the end of the project. SMs can be automatically deployed as Docker containers using the SOO. While this option allows the user to rapidly deploy, monitor and orchestrate large set of SMs, it is also possible to run a SM by hand to integrate a local external simulator.

[6] https://github.com/ibcn-cloudlet/rososgi
[7] http://gazebosim.org/
[8] http://playerstage.sourceforge.net/

### 3.1.8 Code Generator

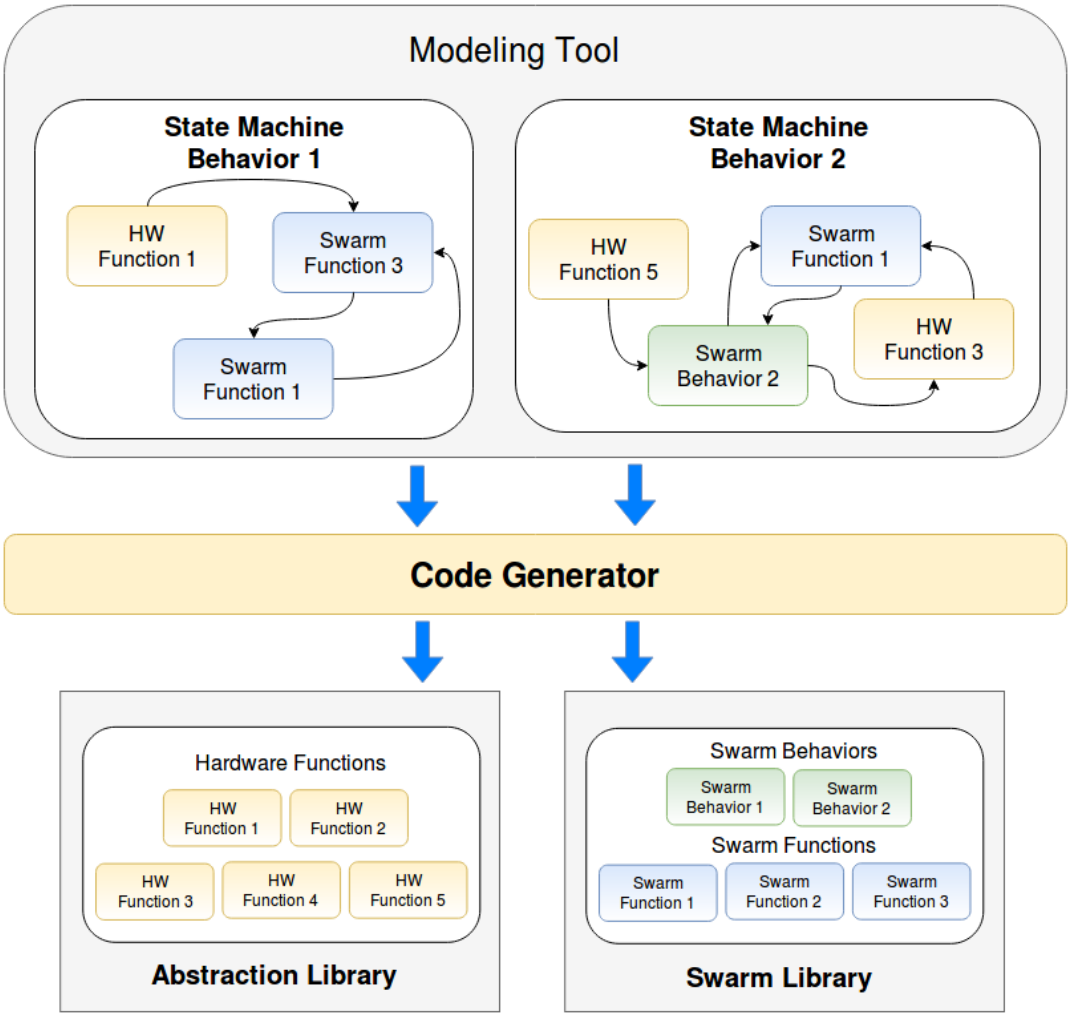


**Figure 6. Code Generator role in CPSwarm workbench**

One of the main purposes of the CPSwarm Workbench is to provide a framework able to ease and speed up the development of new CPS 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 for different platforms. 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.

In the continuation of tasks 4.3 and 5.3, the consortium has 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 high-level functionality that can be selected from three different sources:

1. The Abstraction Library
2. An optimized algorithm coming from the Optimization Tool
3. The Swarm Library which contains different types of swarm algorithms (simple functions or complex behaviours), generally handwritten taking inspiration from the biology.

For the first option, the Code Generator will serve as a "glue" between the platform-independent algorithm modelled as a state machine and the Abstraction Library which provides a set of APIs to access the basic functionalities of the CPS.

In the other two cases, the Code Generator generates a software wrapper to enable the defined algorithm to be called by a known interface supported by the CPS runtime environment. The information to generate this wrapper can be extracted from the Algorithm Meta File (AMF). For instance, in the ROS context, the Code Generator could generate a ROS action interface and a list of callbacks to receive/send the inputs/outputs listed in the algorithm meta file. Furthermore, this mechanism can also be exploited to generate interfaces for algorithm whose meta file (and possibly some pseudo-code description) has been realized in Modelling Tool and the implementation is left to a software developer.
In relation to these two identified roles, 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 that has been specifically extended in order to provide additional information to map each active state to the selected functionality. In addition to the state machine description, the Code Generator also receives the target runtime on which the generated code will be executed.

The final design of the Code Generator will be described as part of D5.4 – Final CPSwarm Modelling Tool.

### 3.1.9 Deployment Tool

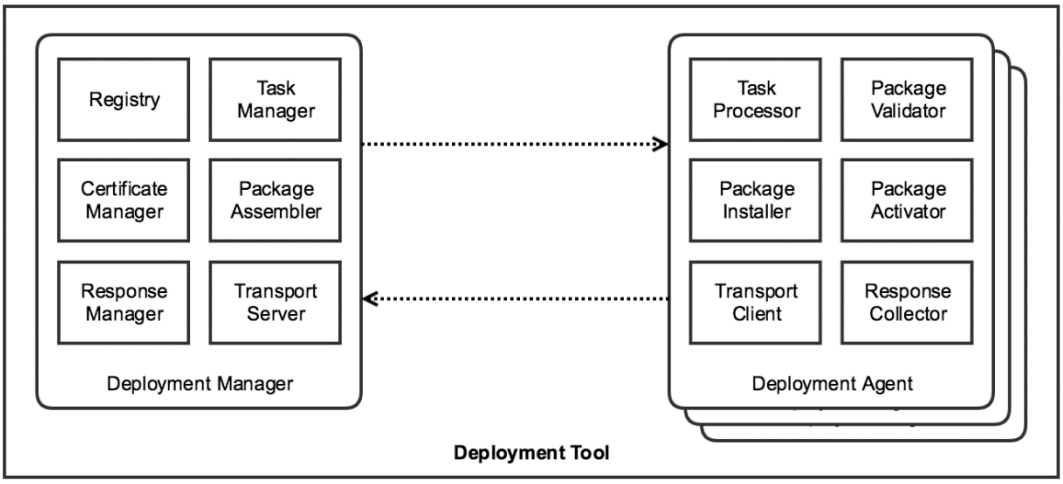


**Figure 7. Components of the CPSwarm Deployment Tool**

The CPSwarm Deployment Tool is responsible for the over-the-air deployment of generated code onto target devices. A deployment task consists of assembly (package preparation), transfer, installation, and activation (execution)[9]. Deliverables D3.1 and D3.2 elaborated on the 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 7 shows an abstract component structure of Deployment Tool. The Deployment Tool consists of two 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 which is responsible for installation, testing, and activation. The Deployment Agent is also responsible for advertising the target to the Deployment Manager.

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

Deployment Manager and Deployment Agents communicate with each other over the network using resource-friendly messaging protocols. Section 5.2.2 elaborates on the applied techniques that ensure that the Deployment Manager can coup with large number of Deployment Agents. Furthermore, sections 4.2.6 and 4.2.7 describe the applied methods to ensure secure deployment and application runtime.

The detailed description of the Deployment Tool, its requirements, and the first working design were presented in document D7.3 – Initial Bulk Deployment Tool. The final version uses state-of-the-art security techniques to ensure package integrity, authentication, and authorization. Furthermore, it addresses the usability aspects of software deployment using a graphical user interface. The final version will be described in D7.4 - Final Bulk deployment tool.

### 3.1.10 Abstraction Layer

The Abstraction Layer is an instantiation of a set of functions from the Abstraction Library on top of a swarm device. It fulfils standard functionalities such as communication with monitoring station and other swarm members. Moreover, it provides a uniform API for high level code to access the underlying heterogeneous hardware and basic functionalities. The Abstraction Layer is essential for the Code Generator, as the uniform API allows generated code to be reusable on different devices. In the previous design, the scope as well as the API provided by the Abstraction Layer were not clearly defined. Since the last deliverable, the consortium has refined the specification of the Abstraction Layer. Namely, the Abstraction Layer now provides two types of interfaces:

- Abstract access to heterogeneous sensors/actuators
- Abstract access to the basic functionalities of the swarm devices

A typical robot, such as a rover or a drone, would have multiple sensors to detect the environment as well as actuators to interact with the outside world. However, the sensors and actuators used on different devices are often heterogeneous and requires special drivers to access them. Therefore, the Abstraction Layer must hide the hardware specific details and provide a uniform, high-level abstract access to these resources. As an example, ROS uses the topic as an abstract access method to get information from sensors and send instructions to actuators. Only in this way, the high-level behaviour code generated by the Code Generator can be free from the hardware specific details and be ported to different devices without major difficulties.

In addition to sensors and actuators, a robot may come with some basic functions which it can perform out of the box. Drones, for example, may come with basic functions such as *take-off* and *landing*. The Abstraction Layer is also responsible for exposing such basic functionalities through a high-level abstract access. As an example, ROS uses the action interface to expose a function of the robot. Through the abstract access to the basic functions these may be assigned to a specific state in the behaviour state machine.

Besides that, the author of the Abstraction Layer for a specific device is also expected to provide a so-called abstraction description file (ADF). The ADF is a new addition to the Abstraction Layer. It describes the available APIs on the Abstraction Layer, e.g. what sensors/actuators are available, what basic functionalities the device can perform. Such information will be described in a standard way, so that different components within the CPSwarm system can interpret it uniformly. The information is essential during algorithm mapping in the Modelling Tool as well as during simulation and optimization.

Within CPSwarm, prototypical Abstraction Layers for rovers and drones used in demo are developed as proof of concept. In reality, the development of Abstraction Layer for different robots is expected to be carried out by manufacturers or the open source community. Indeed, for hardware to be supported by CPSwarm Workbench, it is assumed that Abstraction Layers for swarm members are already available and follow the convention defined by CPSwarm.

### 3.1.11 Monitoring & Command Tool

The Monitoring & Command Tool addresses the challenges related to the after-deployment phase, i.e., during 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 whole swarm with respect to the mission goal. In addition to monitoring, the Monitoring & Command Tool also tackles (re-)configuration of swarm members' parameters depending on external factors.

Swarm members can receive commands, for example to switch between pre-programmed behaviours, and/or configuration parameters through the channel established by the Monitoring & Command 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 & Command Tool runs exclusively in the Runtime Environment. After the deployment phase, the Monitoring & Command Tool is necessary to monitor the actual status of the swarm, as well as to send reconfiguration commands to modify the swarm behaviour, for e.g. to abort the mission or to re-purpose part of the swarm members. On one hand, it gathers live telemetry 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 using the GUI of the Monitoring & Command Tool.

Data exchanged between the swarm members and the Monitoring & Command 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 supports 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 to high-frequency / high-cardinality data streams.

The current design of the Monitoring & Command Tool will be described in D7.5 - Initial Monitoring and Configuration Framework.

## 3.2 Information View

To help the reader better understand the interaction between components within the CPSwarm system, a high-level information view of the system will be presented in this chapter. Figure 8 shows the information flow between components within the CPSwarm system. It is worth noting that most components do not communicate directly with each other, but instead exchange information with the help of the Launcher.

However, in Figure 8 the Launcher component is omitted to better highlight the information flow from one component to another. The information flows are numbered and explained in the following sections.

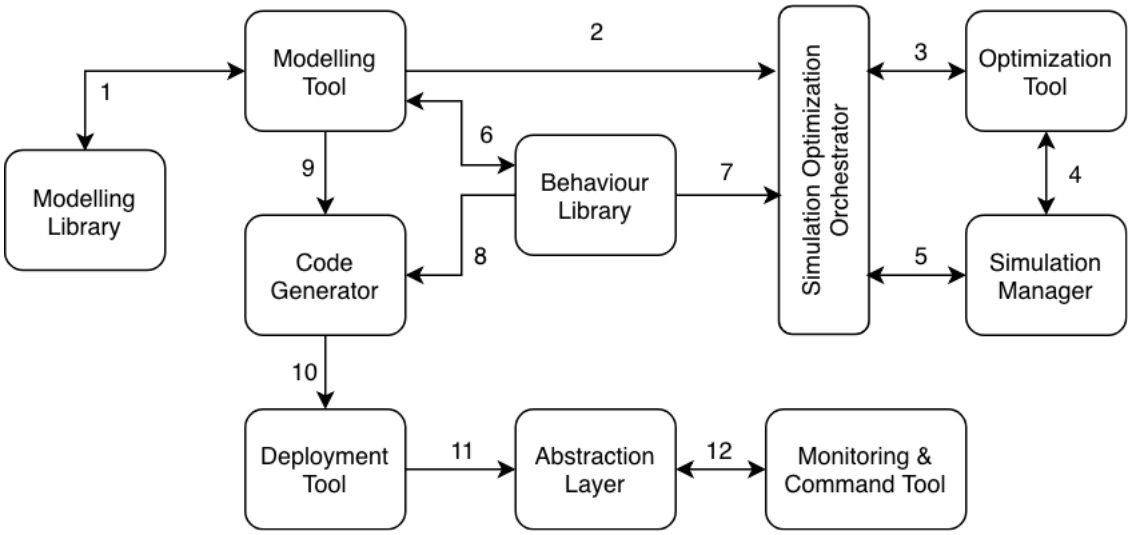


**Figure 8. Information flow between components within the CPSwarm system**

**Flow 1 – Between the Modelling Library and the Modelling Tool**

A set of prerequisite information is required by the Modelling Tool to carry out the swarm design activities. More information is passed for this purpose from the Behaviour Library (Flow 6). Flow 1 passes the following information:

- **CPS hardware specifications**: the hardware specifications, e.g. the 3D models, the equipped sensors/actuators and their properties.
- **Environment models**: the models of an environment in which simulation is carried out. This piece of information is needed by the simulator to build up the simulated environment to test the swarm algorithm.
- **Fitness function:** the function used to calculate the fitness score by the Simulation Manager.

**Flow 2 – The Modelling Tool to the Simulation Optimization Orchestrator**

The Modelling Tool outputs the following information to the Simulation and Optimization Environment:

- **Algorithm mapping**: the mapping between the pre-defined algorithm input/output and the Abstraction input/output. This piece of information is needed by the simulator run simulations using the given algorithm.
- **Swarm composition configuration**: the configuration indicating what devices the swarm consists of and how they interact with each other. This piece of information is needed by the simulator to populate the simulation environment with proper devices.
- **Fitness function implementation**: an implementation of the fitness function in code. It is required by the Simulation Managers to evaluate the performance of a candidate algorithm during the optimization phase.
- **Environment models**: the models of an environment in which simulation is carried out. This piece of information is needed by the simulator to build up the simulated environment to test the swarm algorithm.
- **CPS hardware specifications**: the hardware specifications, e.g. the 3D models, the equipped sensors/actuators and their properties.

**Flow 3 – Between the Simulation Optimization Orchestrator and the Optimization Tool**

This consists of bi-directional communication from SOO and Optimization Tool during the optimization phase:

- **Optimization Configuration**: this is the configuration which tells the Optimization Tool how to run the optimization process. It typically includes settings such as how many algorithm candidates to evaluate, how many iterations to run during optimization, what type of candidate algorithm should be generated (e.g. Artificial Neural Network), etc. This configuration is passed to the Optimization Tool at the beginning of the optimization phase.
- **Optimized candidate controller**: this is the optimized candidate controller implementation which is generated after the optimization phase has finished and it is passed from the Optimization Tool to the SOO. The optimized controller will be then further passed from the SOO to the Code Generator to be turned into platform dependent code.

**Flow 4 - Between the Optimization Tool and Simulation Managers**
Bi-directional communication between the Optimization Tool and Simulation Managers during the optimization phase. It contains the following information:

- **Candidate Controller**: this piece of information is passed from the Optimization Tool to the Simulation Manager. It is the implementation of an algorithm candidate, which is to be evaluated by the simulator. The candidate controller is passed to the Simulation Manager for each evaluation cycle.
- **Fitness Score**: this piece of information is passed from the Simulation Manager to the Optimization Tool. It is the result of the simulation with a candidate controller. It represents the performance of the candidate and is used by the Optimization Tool to pick the better performing candidates for further optimization.

**Flow 5 – The Simulation Optimization Orchestrator to Simulation Managers**
The SOO submits the following information to each Simulation Manager:

- **Simulation Configuration**: configuration necessary for each simulator to run. It typically includes the simulated environment models, the simulated swarm composition, the simulation steps, etc. This piece of information is passed to each Simulation Manager before the simulation phase begins to prepare the simulators for simulation activities.
- **Fitness Function Implementation**: this is the fitness function implementation from the Modelling Tool. In the case of optimization, this is passed to the Simulation Manager and used to calculate the performance score of a candidate controller during the simulation. This piece of information is passed to each Simulation Manager in the beginning of the optimization phase.

**Flow 6 – Between the Behaviour Library and the Modelling Tool**
Information passed between the Modelling Tool and the Behaviour Library during the modelling phase:

- **Abstraction Description File (ADF)**: The ADF is sent to the Modelling Tool for algorithm input/output mapping and mapping of basic functionalities.
- **Algorithm Meta File (AMF)**: The AMF is submitted to the Modelling Tool for the algorithm mapping.
- **Behaviour Algorithm**: the behaviour algorithm implementation and finite state machine (FSM) is passed to the Behaviour Library.

**Flow 7 – The Behaviour Library to the Simulation Optimization Orchestrator**
The prerequisite information needed by the Simulation and Optimization Environment:

- **CPS Abstraction Layer**: the Abstraction Layer implementation in the Behaviour Library. It is needed by each Simulation Manager to simulate the basic behaviour of a real device.
- **Behaviour Algorithm**: the behaviour algorithm implementation in the Behaviour Library. It is needed by each Simulation Manager to simulate the behaviour of the algorithm in the simulated environment.

**Flow 8 – Between the Behaviour Library and the Code Generator**
The input needed by the Code Generator from the Modelling Library. It includes the following information:

- **Behaviour Algorithm**: the behaviour algorithm implementation in the Behaviour Library. Typically, the behaviour algorithm is implemented with a single programming language. To support running the

algorithm on different target environments, the Code Generator needs to generate platform dependent code from the behaviour algorithm source code.

**Flow 9 – The Modelling Tool to the Code Generator**

The information passed to the Code Generator includes:

- Behaviour state machine model: this is the behaviour state machine built by the user using the Modelling Tool. The Code Generator needs the state machine model to generate the backbone behaviour code for specific devices.

**Flow 10 – The Code Generator to the Deployment Tool**

The output from the Code Generator to the Deployment Tool. It includes the following information:

- **Source Code:** the Code Generator should output platform dependent source code which is ready to be compiled locally or to be deployed directly to target devices. For example, the prototypical ROS-based Code Generator produces the deployable source code including a ROS-based behaviour state machine implementation, algorithm source code organized as ROS packages as well as the proper ROS launch files.

**Flow 11 – The Deployment Tool to the Abstraction Layer**

The deployment process from the Deployment Tool to the Abstraction Layer. The Deployment Tool supports two operating modes: 1) compiling the source code locally and deploying the compiled artefacts on target devices, and 2) deploying source code directly on target devices and running build scripts to compile it on target devices. Depending on the operating mode, different data will be sent in the data flow between the Deployment Tool and the Abstraction Layer. Typically, it could include the following information:

- **Compiled Artefacts:** in the first operating mode, the Deployment Tool builds the source code from the Code Generator into compiled artefacts. These artefacts are sent to devices to be installed and executed.
- **Source Code and Build Instructions:** in the second operating mode, the Deployment Tool sends source code as well as the build instructions to target devices. The source code will then be compiled according to instructions on the target device. The generated artefact can then run on the target device.

**Flow 12 – Between the Abstraction Layer and the Monitoring & Command Tool**

Bi-directional data exchange between swarm members and the Monitoring & Command Tool during runtime. It contains the following information:

- **Swarm member status**: this information flows from the swarm members to the Monitoring & Command Tool. During runtime, each of the swarm members keeps sending their real-time status, such as location, current mission, battery life, etc., to the Monitoring & Command Tool, so that the operator running the Monitoring & Command Tool will always have up-to-date knowledge of the status of the swarm.
- **Operator instructions**: this information flows from the Monitoring & Command Tool to the swarm members. It represents the instructions given by the operator to the swarm, such as changing the swarm behaviour, shutting down the swarm, etc. This allows the operator to have full control of the swarm during runtime.

### 3.3 Deployment View

Figure 9 shows the deployment view of the CPSwarm system. The three-dimensional rectangles represent physical hardware. The rectangles decorated with two small rectangles represent the software components deployed on each hardware instance. The lines between them highlight the interaction between components.

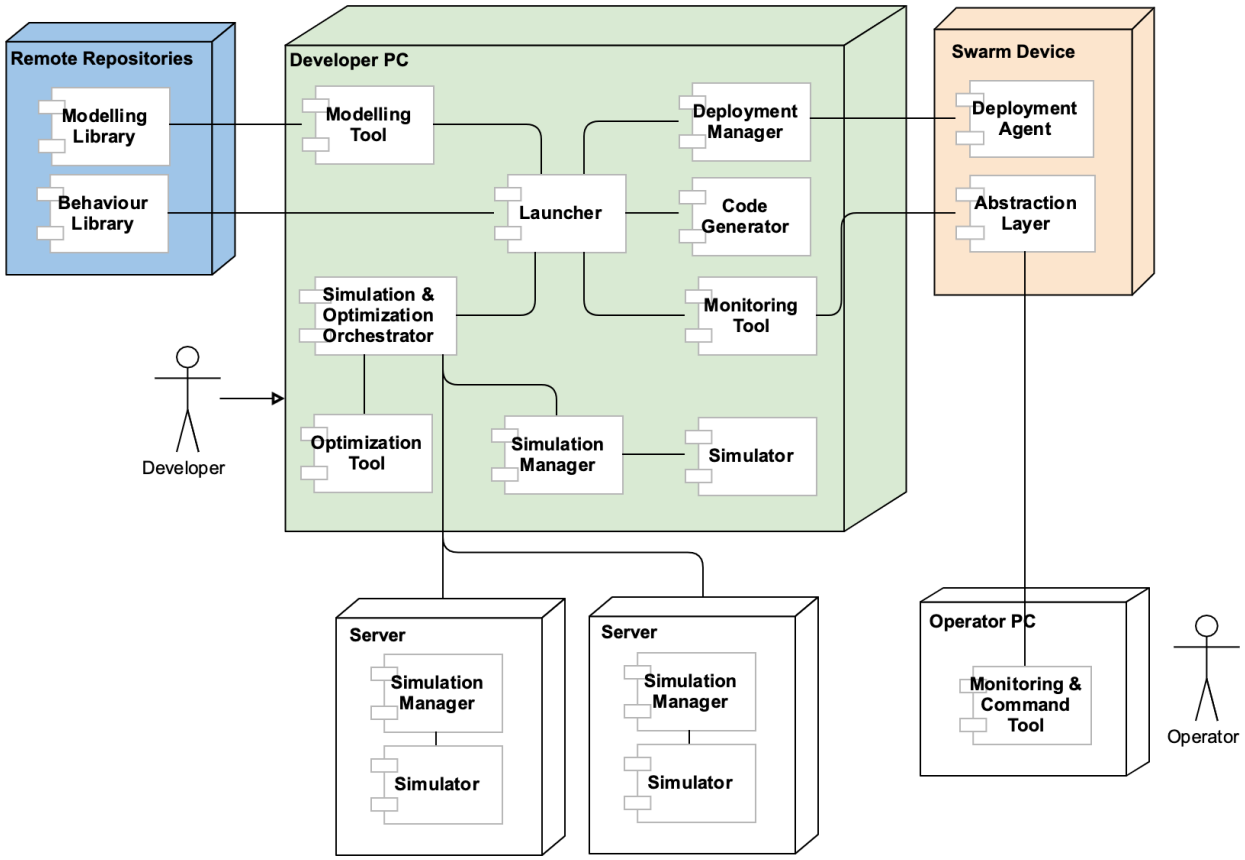


**Figure 9. CPSwarm deployment view. This figure borrows symbols from UML but it does not strictly follow the UML specification.**

Most development-time components such as the Modelling Tool, Code Generator, Deployment Tool are applications which run locally on the developer's PC. The remote repositories are hosted on a remote, public server, from which the developer can pull sources to use them locally during the swarm design phase.

Importantly, 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, the 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 is managed by the Simulation & Optimization Orchestrator. On the other hand, the Deployment Manager which consists of a backend and frontend component can be deployed on a server to cope with large scale deployment requirements. This also makes the Deployment Tool highly available, allowing the user to collect information about deployments even when the local PC is not available.

On each device within a swarm, a Deployment Agent as well as the Abstraction Layer is deployed. The Deployment Agent enables the interaction between the CPS swarm device and the Deployment Manager. The Abstraction Layer 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 platforms. 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 & Command Tool.

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

# 4 Security and Safety Analysis

## 4.1 Review of Previous Analysis

The initial architecture deliverable (D3.1) 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. The project 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. The intermediate architecture deliverable (D3.2), updated the security analysis and proposed planned countermeasures based on the second version of the CPSwarm Architecture. The Consortium organized two security workshops before M18. The first resulted in an agreement on the exact features that should be realized, upon SLAB's proposals of safety and security features. The second workshop focused on implementation details for a subset of these countermeasures to be included in the first demonstration of the CPSwarm Runtime Environment. As a result, the previous deliverable D3.2 provided an analysis of a unified framework for secure communications, the platform hardening, fault and tamper detection, contingency behaviours, emergency remote control and shutdown, code singing and signature validation and rights management.

## 4.2 Final Analysis

During the last phase of the CPSwarm project, the final analysis focuses on the actual vision scenarios and use cases. The project specified security threats, countermeasures and safety risks accordingly and as a result, fine-tuned the previous security components. Therefore, a shift of focus emerged towards secure communication, platform hardening, and emergency shutdown.

The results of the final security analysis will be presented in the upcoming D4.8 deliverable of Work Package 4 that will evaluate the use case scenarios from a security perspective. The document will outline possible attackers and their goals, visualize them using attack trees and provide potential countermeasures. Most countermeasures will be generic, and a subset of them will shape the way some components in the architecture behave. However, most of these solutions are foreseeable and the next chapters describe how these will be implemented.

### 4.2.1 Unified framework for secure communications

The 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 Command 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 that the partners 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), the Consortium decided to build a decentralized solution based on the Zyre[10] implementation of ZeroMQ[11], where swarm members talk to each other directly, without going through a central authority. While this will in turn requires 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. The Consortium agreed that the Confidentiality, Integrity and Availability of the assets should be partially protected by the Communication Library. The message types defined for the project will be protected according to Table 1.

**Table 1. Protection goals of message types**

| Message type | Confidential | Authenticated |
|---|---|---|
| Event<br>*An event has occurred on one of the swarm members that need to be propagated* | Yes | Yes |
| Command<br>*The Monitoring and Command Tool has raised a remote event on a specific swarm member* | Yes | Yes |
| Artefact<br>*The Deployment Tool has sent a software artefact that needs to be deployed on the swarm member* | Yes | Yes |
| Status<br>*The swarm member has made progress deploying the software artefact* | Yes | Yes |
| Set / Get<br>*The Monitoring and Command Tool has sent a request to get or set the value for a global parameter of the behaviour* | Yes | Yes |
| Subscribe / Unsubscribe<br>*The Monitoring and Command Tool wants to subscribe to or unsubscribe from updates on a property* | Yes | Yes |
| Telemetry<br>*The swarm member has sent an update for the value of a property to a subscriber* | Yes | Yes |

Please note that response messages, which only include a confirmation that the operation has completed successfully are not included, and that the descriptions in italic are only examples for how such a message might be used.

The Communication Library implements the requirements from the table and therefore covers a great deal of attack surface and provides sufficient countermeasures for numerous attack scenarios. The final implemented security features will be based on the libhydrogen[12] library using two cryptographic building blocks: the Curve25519 elliptic curve, and the Gimli permutation. Features include encryption, authentication and integrity protection along with a key exchange mechanism integrated into the discovery phase. The secure version of the Communication Library will be an extension of the current basic version of the library. The end users will be able to switch to the secure version by using a different endpoint class which will

[10] https://github.com/zeromq/zyre
[11] http://zeromq.org/
[12] https://github.com/jedisct1/libhydrogen

transparently enable the secure features without any other configuration necessary. The Communication Library will be described in more detail in the upcoming D7.4 and D7.6 documents.

### 4.2.2 Platform hardening

Used in two of the CPSwarm use cases, ROS (Robot Operating System) is the project's primary target software platform. Furthermore, the pilot partners select hardware platforms most relevant to the target use cases and vision scenarios. The platform hardening will start with a security analysis of the target platforms and end with testing and deployment. As a final result, a hardening guide and a optimized image will be delivered for each platform analysed at the end of the project.

### 4.2.3 Fault and tamper detection

For the sake of completeness, fault and tamper detection are mentioned as generally necessary features to be implemented for a final version of CPS in production. However, they are beyond the scope of this project. 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. Tamper detection can be done by the use case partners by adding hardware components to their system and defining software behaviours in case these are triggered. From a security perspective, these new hardware modules can provide a trusted environment within the devices. In case the devices are tampered with, their assets (and possibly the mission) can be compromised to cause safety or security breaches.
Figure 10 depicts an abstract model of a swarm member and describes how to handle faults and tampered data by changing behaviour – see Section 4.2.4.

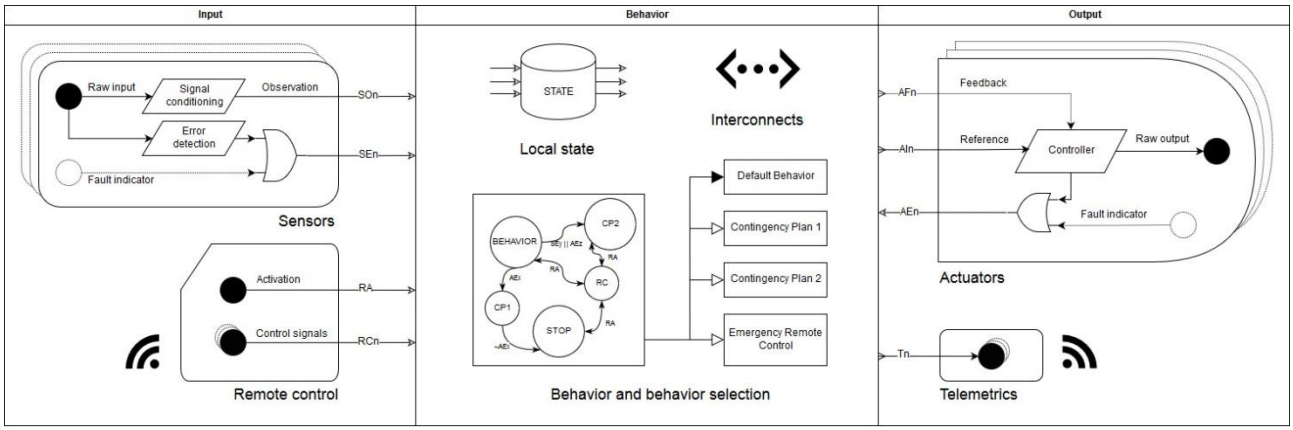


**Figure 10. High-level model of a swarm member**

### 4.2.4 Contingency behaviours

Contingency behaviours – as a safety feature – 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 Command Tool or by an external event sensed in the environment. This countermeasure primarily addresses safety concerns. 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.

### 4.2.5 Emergency remote control and shutdown

By switching to emergency remote control mode using the Monitoring and Command Tool, the (authenticated and authorized) 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.
Emergency shutdown can be viewed as a specific contingency behaviour that can be triggered by either the Operator through the Monitoring and Command 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. The partners aim to provide two different types of emergency shutdown:

- 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 Command 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. A proof of concept device has been successfully presented in operation during the M18 demo: a NanoPi NEO Air-based device capable of running ROS was used to send an emergency shutdown message to a drone via Wi-Fi using the project's communication protocol.

### 4.2.6 Secure initial deployment

During manufacturing, it is paramount to load the necessary certificates and generate public-private key pairs for the devices in a trusted environment. The process of key generation and key exchange will be done according to the Ironhouse[13] pattern using CurveZMQ[14]. The workflow for setting up a device (Agent) and register it with the Deployment Tool (Server) is as follows:

1. The deployment administrator authenticates (with credentials) and asks for a token from the Server over the RESTful API or GUI
2. The deployment administrator starts the Agent on device, generating a key pair locally.
3. The deployment administrator starts the Agent on device with the token as a process environment variable.
4. The Agent generates a key pair and contacts the Server's RESTful API, submitting the token (to authenticate) and its public key for CurveMQ.
5. The Server invalidates the token so it can no longer be used.
6. The Agent contacts the Server over the network. They establish a secure channel using CurveZMQ.

The steps from step 2 are automated. The token used has a sufficiently high entropy (48 bits) to resist any brute-force hacking attempts.

### 4.2.7 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. As code signing is a possible countermeasure for specific attacks regarding the upgrade procedure, this feature will be addressed in more detail in D4.8.

[13] https://github.com/pebbe/zmq4/blob/master/examples_security/ironhouse.go
[14] http://curvezmq.org/

### 4.2.8 Rights management

Being able to authorize entities that can affect the operation of the swarm adds an extra layer of security to the system. Revoking rights from compromised swarm members can isolate those members and prevent a breach in the whole swarm. Moreover, limiting rights to certain operations can minimize the possible damage done by a compromised swarm member. Authorization can enforce the separation of different maintenance and monitoring tasks of the operators, such as deployment, monitoring, configuration and remote control.

# 5 Scalability Analysis

## 5.1 Review of Previous Analysis

In D3.1, the importance of scalability within the CPSwarm project was discussed. In particular, two aspects which are likely to cause scalability issues and performance bottlenecks were 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 a 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 mechanism in modern smart phones was created in CPSwarm.

## 5.2 Final Analysis

### 5.2.1 Simulation scalability analysis

The final design of the Simulation and Optimization Environment has been presented in D6.2. The deliverable contains a detailed scalability analysis of the proposed solution and shows how the scalability issues identified in the previous version of the architecture (Micha Rappaport, 2018) have been solved. The following results have been obtained:

1) Leveraging XMPP for the communication among the components. Core XMPP features[15] include unique identifiers, presence mechanism and one-to-one chat messages; or some of the protocol extensions, like file transfer[16] and publish/subscribe[17]. This enables us to leverage the solid scalability features provided by the protocol[18, 19].

2) The introducion of a scalable discovery mechanism for the distributed Simulation Managers where the Simulation Managers announce themselves to the SOO when they are available, leveraging the XMPP presence mechanism.

3) In the approach described in D6.1, the evaluation of the controller was conducted within the Optimization Tool based on the continuous exchange of messages with the simulation environment (centralized approach). This increased the number of messages exchanged and decreased performance. To address this issue, the final version sends the controller to the Simulation Manager and evaluates it locally within the simulator. As only the controller and fitness score are transferred,

[15] https://xmpp.org/rfcs/rfc6120.html
[16] https://xmpp.org/extensions/xep-0096.html
[17] https://xmpp.org/extensions/xep-0060.html
[18] https://www.igniterealtime.org/about/OpenfireScalability.pdf
[19] https://www.isode.com/whitepapers/xmpp-performance-constrained.html

the number of messages exchanged is much more limited, thus increasing performance (distributed approach).

The analysis presented in D6.2 demonstrates that the performance of the distributed approach is better that of the centralized one in most scenarios. Furthermore, the partners have conducted an analysis of the performance of the approach as the number of Simulation Managers increases. Figure 11 shows the resulting optimization time with different number of Simulation Managers used. The chart shows measurements in line with the theoretically calculated performance. The performance scales well with the number of Simulation Managers. The performance improvement rate is minimized beyond $2^4$ Simulations Managers, possibly related to the limited number of cores (24) available on the test cluster.

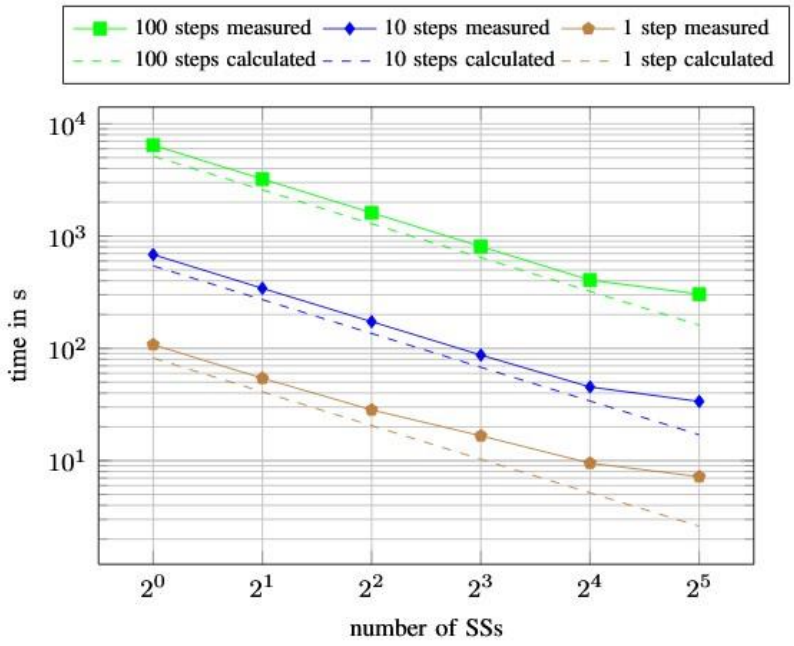


**Figure 11. Scalability with number of Simulation Managers of the optimization time of the distributed approach for varying simulation lengths. In this figure, the Simulation Managers are referred to as Simulation Servers (SSs).**

### 5.2.2 Deployment scalability analysis

The CPSwarm Deployment Tool is built on top of the initial OTA update concept. This provides tremendous benefit for deployment on multiple devices, relieving users from the burden of direct software deployment. However, an appropriate set of technologies are required to optimally deploy software on resource-constrained swarm devices.

The Deployment Tool's update system is based on a publish-subscribe messaging pattern tailored for the management of a large number of devices using a simple 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[20] protocol. This way, the

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

Deployment Manager sends a single copy of messages to a remote network which are then multicasted 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.

# 6 Conclusion

This deliverable presented the final system architecture design of CPSwarm. The project has evolved significantly since the initial phases, leading to discovery of new challenges and potential solutions. As a result, the architecture design went through an iterative process, reaching the final form that is documented in this deliverable. This document provided a high-level overview of the technical aspects of the CPSwarm architecture, leaving fine-grained technical details to the technical deliverables of the individual project tasks.

The final architecture paves the way for future development and integration activities in Task 3.3 - Continuous System Integration. Work in the following months will focus on component integration. The final status of CPSwarm Workbench and associated tools will be reported in deliverable D3.6.

## Acronyms

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

## List of Figures



## List of Tables

## Reference


Ahmed, H., & Glasgow, J. (2012). *Swarm intelligence: Concepts, models and applications.* School of Computing, Queen' University, Canada.

Binitha, S., & Sathya, S. (2012, 2). A survey of bio inspired optimization algorithms. *International Journal of Soft Computing and Engineering*, pp. 137-151.

Blum, C., & Li, X. (n.d.). Swarm intelligence in optimization.

Bonabeau, E., Dorigo, M., & Theraulaz, G. (2008). *Swarm intelligence: from natural to artificial systems.* Oxford University Press.

Brambilla, M., Ferrante, E., Birattari, M., & Dorigo, M. (2013, 7). Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, pp. 1-41.

Camazine, S., Franks, N., Sneyd, J., Bonabeau, E., Deneubourg, J., & Theraula, G. (2001). *Self-organization in Biological Systems.* Princeton University Press.

Floreano, D., & Mattiussi, C. (2008). *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies.* MIT Press.

Garnier, S., Gautrais, J., & Theraulaz, G. (2007, 1). The biologicl principles of swarm intelligence. *Swarm Intelligence*, pp. 3-31.

Green, D., Aleti, A., & Garcia, J. (2017). The nature of nature: Why nature-inspired algorithms work.

Hamann, H., & Schmickl, T. (2012). Modelling the swarm: Analysing biological and engineered swarm systems. *Mathematical and Computer Modelling of Dynamical Systems*, pp. 1-12.

Hassanien, A., & Alamry, E. (2015). *Swarm Intelligence: principles, Advances and Applications.* CRC Press.

IEEE. (2011). ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description.

Kolling, A., Walker, P., Chakraborty, N., Sycara, K., & Lewis, M. (2016). Human interaction with robot swarms: A survey. *IEEE Transactions on Human-Machine Systems*, pp. 9-26.

Krause, J., Cordeiro, J., Parpinelli, R., & Lopes, H. (2013). A survey of swarm algorithms applied to discrete optimization. *Swarm Intelligence and Bio-Inspired Computation*.

Lim, C. J., & Dehuri, S. (2009). *Innovations in Swarm Intelligence.* Springer.

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

Parpinelli, R., & Lopes, H. (2011). New inspirations in swarm intelligence: A survey. *International Journal of Bio-Inspired Computation*.

Shranz, M., Umlauft, M., Rappaport, M., & Elmenreich, W. (2018). A classification of basic swarm behaviors and their application in cyberphysical systems. *Swarm Intelligence*.

Yang, X., Cui, Z., Xiao, R., Gandomi, A., & Karamanoglu, M. (n.d.). *Swarm intelligence and bio-inspired computation: theoriy and application.* Elsevier.

Yang, X., Deb, S., Zhao, Y., Fong, S., & He, X. (2017). Swarm intelligence: past, present and future. *Soft Computing*.