D6.2 – FINAL SIMULATION ENVIRONMENT

Deliverable ID	**D6.2**
Deliverable Title	**Final Simulation Environment**
Work Package	**WP6 – Simulation and Performance Prediction**
Dissemination Level	**PUBLIC**
Version	**1.0**
Date	**03/05/2019**
Status	**Final**
Lead Editor	**Davide Conzon (LINKS)**
Main Contributors	**Micha Sende (LAKE), Arthur Pitman (UNI-KLU), Midhat Jdeed (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
0.0	2019-03-02	Davide Conzon (LINKS)	First Draft with TOC
0.1	2019-03-16	Davide Conzon (LINKS)	First Sections
0.2	2019-03-23	Davide Conzon (LINKS)	Integrated the description of the Initial Simulation Environment
0.3	2019-03-29	Davide Conzon (LINKS)	Integrated the implementation of the Final Simulation Environment
0.4	2019-04-02	Davide Conzon (LINKS)	Integrated UNI-KLU contributions for Section 5 and completed implementation in Section 7
0.5	2019-04-02	Davide Conzon (LINKS)	Fixed document format
0.6	2019-04-05	Davide Conzon (LINKS)	Integrated UNI-KLU contribution for Section 5
0.7	2019-04-19	Davide Conzon (LINKS)	Version ready for internal review
0.8	2019-04-27	Davide Conzon (LINKS)	Refactored some sections and integrated UNI-KLU review
0.9	2019-04-30	Davide Conzon (LINKS)	Integrated DIGISKY review
1.0	2019-05-03	Davide Conzon (LINKS)	Final version to be submitted

Internal Review History

Review Date	Reviewer	Summary of Comments
2019-04-30	Omar Morando (DIGISKY)	Approved with minor comments.
2019-04-23	Arthur Pitman, Midhat Jdeed (UNI-KLU)	Approved with minor comments.

Table of Contents

1 Executive Summary

This deliverable, *D6.2 - Final Simulation Environment*, gives a detailed description of the simulation environment and its integration into the algorithm optimization environment. Firstly, it describes the work done to build the initial version of the simulation environment released at M9 (see *D6.1 – Initial Simulation Environment*). Then, the deliverable describes the final simulation environment, presenting an evaluation of the performances, compared with the previous ones. Finally, the last set of features introduced in the architecture to increment its scalability will be described.

This deliverable reports the results of Task 6.1.

2 Introduction

The document *D3.2 - Updated System Architecture & Design Specification*, delivered at M18, describes the updated architecture of the CPSwarm system, see Figure 1.

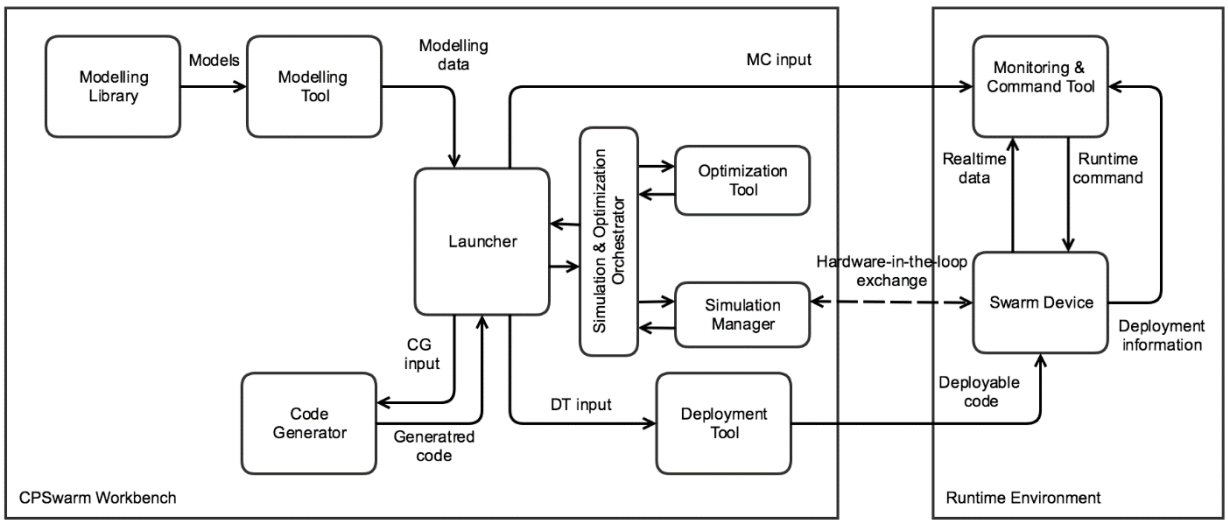

Figure 1 - Overview of components in CPSwarm system

Based on D3.2 and D6.1 which have introduced the initial version of the Simulation and Optimization Environment, this deliverable presents the work done by the Consortium to evolve the architecture from the first version designed at M9, to the final version proposed in this document.

The Simulation and Optimization Environment is the part of the CPSwarm Workbench responsible for the realization of an optimized controller that implements local interaction rules, which lead to the desired global behavior of the system. The intended approach is to develop customizable environments, which allow the most suitable tools to be used according to the specific use case. This means that different types of modelling, optimization and simulation tools can be used. This customization is realised by to the definition of generic APIs which decouple the tools from each other. In particular, the optimization phase requires frequent communication between the Optimization Tool (OT) and one or more Simulation Tools (ST)s.

Firstly, this deliverable recaps the Initial Simulation Environment, described in D6.1, presenting the two approaches proposed in that deliverable:

- The distributed approach: the controller to be optimized is executed directly in the ST, implemented with a filesystem-based implementation, which has allowed to analyze issues related to the interoperation between the OT and the ST.
- The centralized approach: the controller is executed in the OT, based on a network communication protocol and a broker architecture that decouples the OT from the STs, allowing different STs to be used as well as multiple instances of the same ST to run in parallel.

After conducting an analysis of the performance of these approaches, the CPSwarm Consortium developed the final version of the architecture of the Simulation and Optimization Environment, utilizing a distributed approach based on the eXtensible Messaging and Presence Protocol (XMPP). The deliverable describes this architecture and then analyzes and compares it to the first one, showing that performances has improved. Furthermore, the solution developed supports:

- Optimization; an optimization process may be run using an OT and a set of distributed STs to optimize a candidate controller and to replay the optimized candidate.

- Visual simulation: an optimized candidate may be demonstrated on a ST that can present it in a more realistic scenario and allow the user to visually evaluate its performance.
- Deployment: the optimized candidate may be deployed to the actual devices.

In the final part, the deliverable presents the features introduced to the architecture to improve the scalability of the Simulation and Optimization Environment.

1.1 Scope

This deliverable is limited to simulation environments that simulate robotics behavior with a focus on rovers and drones. Other types of simulators such as network communication simulators are not considered in this deliverable. Furthermore, only the Simulator API is covered in its entirety as it is part of the Simulation and Optimization Environment. The interfaces to the Modeling Environment are only briefly explained, because they are the focus of D5.2 Initial CPS Modelling Tool.

1.2 Document Organization

The rest of this deliverable is structured as follows. Firstly, Section 3 presents the first version of the architecture released at M9 and together with its performance. Then, Section 4 introduces the final architecture designed to address the issues identified in the initial one. In Section 5, the authors present a prototype of this architecture based on XMPP protocol. Section 6 analyzes the performances of this solution and compare it to the previous one, showing the results obtained and what can be still improved. Based on this analysis, Section 7 presents the features introduced in the final version of the architecture to furtherly improve the scalability of the solution. Finally, Section 8 concludes this deliverable.

2.1 Related documents

ID	Title	Reference	Version	Date
D5.2	Initial CPSwarm Modelling Tool	D5.2	1.0	30-09-2017
D6.1	Initial Simulation Environment	D6.1	1.0	05-10-2017
D6.5	Initial Integration of External Simulators	D6.5	1.0	30-06-2018
D3.2	Updated Updated System Architecture & Design Specification	D3.2	1.0	30-06-2018
D6.6	Updated Integration of External Simulators	D6.6	1.0	31-04-2019
D6.4	Final CPS System design optimization and Fitness function design guidelines	D6.4	1.0	30-06-2019
D6.7	Final Integration of External Simulators	D6.7	1.0	31-12-2019

3 Initial Simulation and Optimization Environment architecture

Deliverable D6.1, released in M9, introduced two different design of the Simulation and Optimization Environment. Both approaches share a common architecture (see Figure 2) consisting of two core components, the OT and one or more STs. The OT is responsible for performing the evolutionary optimization process [1] and the ST performs the required simulations in each step of the optimization process and evaluates the fitness of a controller candidate. The ST simulates a homogeneous swarm of CPSs, where each CPS is controlled by a controller generated by the OT. This controller translates the sensor inputs to actuator outputs. The two components are interconnected to each other through an interface that defines how they communicate during the optimization process. The generic nature of the interface allows many well-established STs that support simulation of swarms of CPSs, at different levels of detail, to be integrated into the architecture.

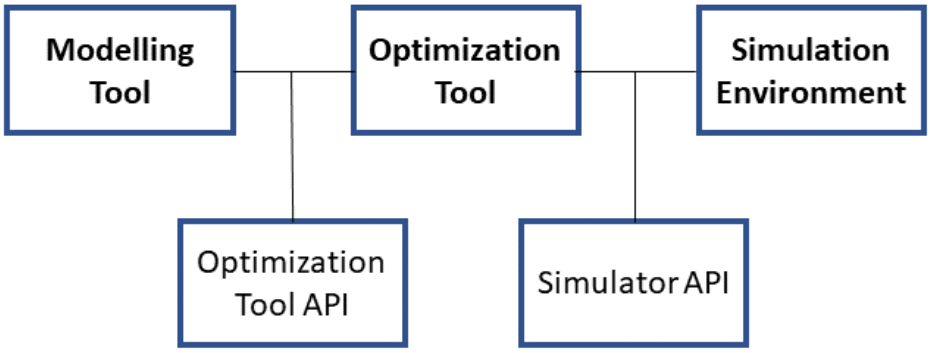

Figure 2 - Architecture of the optimization and simulation environment.

The key difference between the two approaches is where the candidate controller resides during simulation. The first approach, hereafter referred to as *distributed,* transfers the CPS controller from the OT to the ST, which can then independently run the simulation and returns the resulting fitness value of the controller. The other approach, hereafter referred to as *centralized*, executes the candidate controller centrally within the OT. In this case, the STs merely carry out the actions received from the controller and report back the sensor readings of the CPSs. With this latter approach, the STs are centrally controlled by the OT.

In D6.1 the two approaches were implemented in different ways. The *centralized* approach was implemented using network socket inter-process communication, allowing it to distribute the simulations among multiple STs, running on remote servers, connected through a broker (see Section 3.2.2). The drawback of this implementation was that there was a high messaging overhead and a repeated polling for available STs. This inhibited the system from scaling well with more than three STs. Instead, the *distributed* approach has been implemented based on file system inter-process communication (see Section 3.2.1), passing the complete controller to the STs, but was limited to execution on a single machine. Ignoring the implementation specifics, these approaches are subsequently referred to as *centralized* and *distributed*.

The following subsections introduce the API defined for the architecture, then introduce the prototype implemented to test the two approaches and finally presents the results of a performance analysis done.

3.1 Simulator API

While, both cases require an interface between the OT and the STs, there are some key differences. For the *centralized* approach, the OT and STs must exchange the sensor readings and actuator controls. On the OT side, the interface must therefore receive the sensor inputs from the ST and feed them into the controller. The resulting actuator commands are determined by the controller and transmitted back to the corresponding ST. On the {ST side, the interface must allow control of the CPS behaviour, using the actuator commands received from the OT. After the commands are executed by the ST, the sensor readings are transmitted back to the OT. Instead, the interface for the distributed approach requires exchanging the complete controller representation.

On the OT side the interface must therefore export the controller representation and transmit it to the STs. Once the ST completes the simulation, the OT retrieves the simulation results to assess the performance of the controller. Depending on the implementation, this can be either raw log data or pre-processed information in the form of a fitness value. The ST side of the interface receives the controller and integrates its representation into the CPS's code, allowing the controller to translate the sensor readings to the actuator commands. Once the simulation is finished, it sends back the result of the simulation. Computing the fitness value of a simulation can thus be done on either side, in the OT or the ST.

Regardless of the approach, there are several messages that need to be exchanged. At first, there is a setup phase, which allows the OT to be aware of the available STs. This requires a discovery process where the OT polls for STs stating the requirements on the ST to be used (e.g., number of dimensions supported, and the maximum number of agents supported). The ST that satisfies these requirements then need to report back to the OT stating their availability. Then, the optimization can be performed, where the OT communicates with the selected STs. Depending on the chosen approach, a different number and different types of messages are exchanged between the OT and the STs. This communication takes place over several iterations, until the OT finds a solution that cannot be further optimized. This optimal solution is represented by a candidate controller that might be then exported by the OT to be deployed on the CPSs. The optimized controller can be deployed to a ST or on actual CPS hardware, with the former allowing further improvements to performance, scalability, or robustness analysis as well as visual inspection of the CPS behaviour, while latter permits testing under real conditions. Regardless of whether the controller is used in simulation or on actual CPS hardware, the connection of CPS' sensor inputs and actuator outputs must follow the same specification as the interface implemented in the ST for the *distributed* optimization process.

3.2 Initial Simulation Environment Prototypes

As stated before, the *distributed* and *centralized* approaches, presented in D6.1, have been implemented in different ways. The next subsections present the *distributed* solution implemented based on file system inter-process communication and then the *centralized* solution implemented using a network socket inter-process communication.

3.2.1 Distributed approach

The first prototype has been used to test the *distributed* approach. In this case, the communication between the OT and the simulation environment is based on a filesystem inter-process communication technique and ROS, which is a middleware that can control robots in simulation and on physical hardware. The ROS simulations are launched from OT by executing a script that first compiles the ROS package implementing the simulation and then executes this package. The OT employs the ST to evaluate evolved candidate controllers through simulation. This communication is enabled by the simulator API, which passes the candidate representation and problem specific parameters from the OT to the ST and returns the performance of the candidate. The candidate controller is, typically, an artificial neural network (ANN), exported by the OT Framework for EVOlutionary design (FREVO) as a C source code file. The ROS implementations include this source code file to enable the agents in the simulation making decisions by translating the sensor readings to actuator commands. The parameters that need to be transferred from the OT are written into parameter files in the YAML format that is used by ROS. The performance of a candidate was measured by performance metrics defined in the modelling tool. The ST measured the metrics and wrote them to log files in text format. The OT read these log files and applied the fitness function to calculate the fitness score of a candidate controller. This first implementation was based on the use of Modelio as modeling tool, FREVO as OT, and ROS-based STs, such as Stage and Gazebo. This prototype has been used to show how to visually replay a candidate controller through simulation and how to use an optimized controller in simulation environments and on robotic

hardware, showing that it is possible to use the same candidate, thanks to the use of ROS, in a 2D ST like Stage, in a more realistic 3D ST like Gazebo and on actual hardware (such as turtlebots[1]).

3.2.2 Centralized approach

The previous prototype was unable to perform simulation using multiple parallel simulation environments running remotely. This leads to the design of a broker-based architecture. based on the *centralized* approach (see Figure 3).

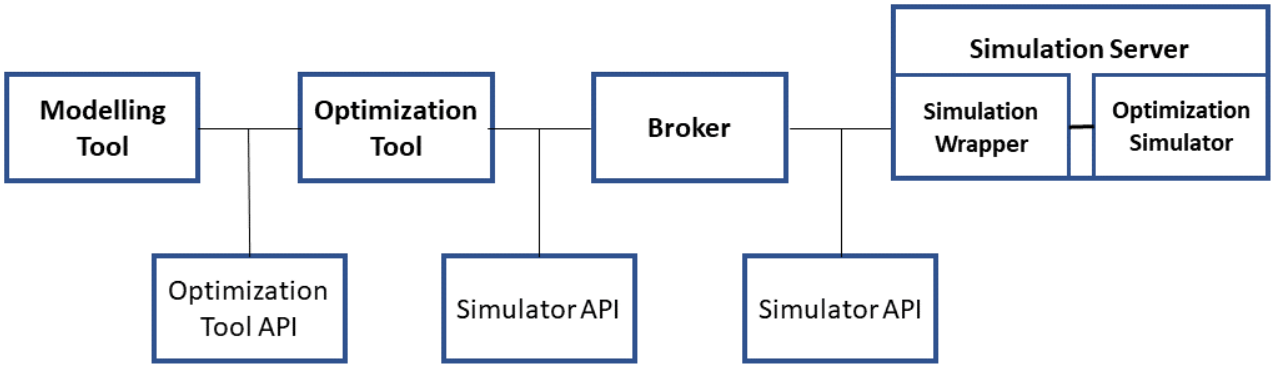

Figure 3 - Architecture of the broker-based optimization and simulation environment.

The implementation of this architecture has been based on the use of Modelio and FREVO as modeling and optimization tools and on the use of the Message Queue Telemetry Transport (MQTT) protocol to implement the Simulator API in a network-based way. This architecture included the Simulation Wrapper (SW), which was the software layer installed on every Simulation Server (SS), implementing the simulator API. The SW integrated a MQTT client that automatically subscribed itself to the topic where the OT published its messages for the SS. Thanks to this software layer, the OT was able communicate with the SS without knowing what type of ST was used. Several STs can be used in the same way also working in parallel, to reduce the simulation times. Indeed, after the OT has created one generation of controller candidates, it can send different candidates to different SSs, which perform the simulation and calculate the fitness score. When all the fitness scores are returned, the OT can perform the evolutionary steps to create the candidates for the next generation. Considering the complexity of the SW implementation for complex STs, such as Stage or Gazebo, a simpler ST, *Minisim* (see Section 3.2.2.1). was used during initial experiments. Using this approach, it was possible to investigate the parallelizaztion of the simulations of the different candidate controllers from the same generation.

3.2.2.1 Minisim Simulation

Minisim is a simple Java based simulation that runs without a Graphic User Interface (GUI). It implements a multi-agent simulation where one or more agents must reach a goal, before being caught by one or more defenders, placed between agents and the goal. The agents move according to the commands given by the optimization tool. The defenders always move slowly towards the closest agent.

3.3 Performance Analysis

After the development of these first protypes of the architecture, the Consortium in [1] has analysed the performance of the two approaches (reported in this section), leading to the definition of an enhanced architecture, which will be presented in Section 4.

When comparing the two approaches, both have their advantages and disadvantages. Looking at the *centralized* approach, the implementation of the ST is agnostic to the type of representation used for the

[1] http://www.turtlebot.com/

controller. This has the advantage that new controller representations can be added to the OT easily, without the need to update the ST interface. The disadvantage is that a lot of message exchange occurs between OT and the ST, throughout the simulation. When the number of ST is increased the OT becomes a bottleneck as it must communicate constantly with each ST. Therefore, the *distributed* approach is less portable but more performant.

The performance of either architectural approach can be measured as the total time taken for the optimization process. This time can be expressed as

$$t_{opt} = n_{gen} * (t_{evo} + \frac{n_{pop} * n_{eval}}{n_{sim}} * (t_{sim} + t_{ohd}))$$

Equation 1

consisting of three times:

- The time t_{evo}, which expresses the time required to perform the evolutionary calculations, such as selecting the best performing controllers and creating a new generation of controllers. Such tasks are executed for each generation of the optimization and hence are multiplied by the number of generations n_{gen}.
- The time t_{sim}, which expresses the time taken by one simulation run. For simplification purposes, it is assumed that this time is measured in discrete steps and constant, regardless of the number of CPS in the simulation. The simulation time can therefore be expressed as

  $$t_{sim} = n_{step} * t_{step}$$

  Equation 2

  based on the number of discrete time steps n_{step} and the time t_{step} required to simulate one step. A simulation is performed for each controller in the population of n_{pop} controller candidates. For robustness and statistical significance, each controller can be evaluated n_{eval} times as a different variant of the same problem. This results in several $n_{pop} * n_{eval}$ simulations that must be performed during each of the n_{gen} generations. Depending on the number of available STs n_{sim}, the optimization process can be accelerated by distributing the simulations among these STs. The upper limit for the number of required STs is therefore $n_{pop} * n_{eval}$.
- The time component t_{ohd} states the amount of overhead time required during simulation. Where the other two times are identical for both approaches, the overhead time varies between the *centralized* and the *distributed* approach. It can be generalized as

  $$t_{ohd} = t_{setup} + t_{run} + t_{finalize}$$

  Equation 3

  where the setup time t_{setup} is the time required to setup the STs, the run time t_{run} is the overhead time added while running the simulations, and the finalization time $t_{finalize}$ the time to finalize the simulation and gather the results. For the centralized approach, the overhead time is

  $$t_{ohd,c} = n_{msg,setup} * t_{msg} + n_{msg,run} * n_{step} * n_{cps} * t_{msg} + n_{msg,finalize} * t_{msg} * t_{fitness}$$

  Equation 4

  that contains two times. First, the message transmission time t_{msg} between the OT and the STs. During setup, there are $n_{msg,setup}$ messages to be exchanged. During run time, each of the n_{cps} CPSs in the STs communicates $n_{msg,run}$ messages for every simulation time step, where the simulation lasts for n_{step} steps. When finalizing a simulation, there are $n_{msg,fin}$ messages to be exchanged. Second, the time $t_{fitness}$ to compute the fitness value of a controller adds to the finalization time. For the distributed approach, the total overhead time sums up to

$$t_{ohd,d} = t_{export} + n_{msg,setup} * t_{msg} + t_{import} + n_{msg,finalize} * t_{msg} + t_{fitness}$$

Equation 5

that contains two additional times as compared to the *centralized* approach. First, the time t_{export} to export the controller representation from the OT into a format readable by the STs. Second, the time t_{import} to import the controller into a STs.

To compare the performance of both approaches, it is possible to calculate the ratio

$$r = \frac{t_{opt,c}}{t_{opt,d}}$$

Equation 6

which relates the total optimization run time of the *centralized* approach with the optimization time of the *distributed* approach. This ratio expresses which approach is more suitable for a specific setup of parameters. The partners determined the most relevant parameters for analysis to be the simulation length as number of simulated steps n_{step} and the number of CPSs n_{cps} that are being simulated. The number of parallel STs has a negligible influence as both approaches can use parallelization. To compare the performance scalability of both approaches, the partners set the other parameters to a fixed value that has been derived using measurements on the testbed described in [1]. They are summarized in Table 1 where the evolutionary parameters were chosen to yield good results.

Table 1 - Optimization parameters measured through simulations.

Parameter	Value
n_{gen}	200
n_{pop}	50
n_{eval}	1
$n_{msg,setup}$	4
$n_{msg,run}$	2
$n_{msg,finalize}$	1
t_{evo}	12ms
t_{msg}	30ms
t_{export}	35ms
t_{import}	8833ms
$t_{fitness}$	0.69ms
t_{steo}	100ms

The resulting ratio r (see Equation 6) using these values can be seen in Figure 4. A value of r>1 means that the *distributed* approach performs better whereas a value of r<1 means that the *centralized* approach is favourable. As both approaches can use parallelization, the resultant ratio is independent of the number of parallel STs used. For most cases the *distributed* approach performs better, even though the time for importing the controller is dominating in Table 1. This is because there is a lot of messaging overhead if all CPS controllers are executed in the OT. This creates a bottleneck, where most work still is performed by a single tool. As seen in Figure 4, this is not so crucial for small swarm sizes, but already for a swarm size of eight CPSs, the optimization with central control takes longer when simulations last more than eighteen steps.

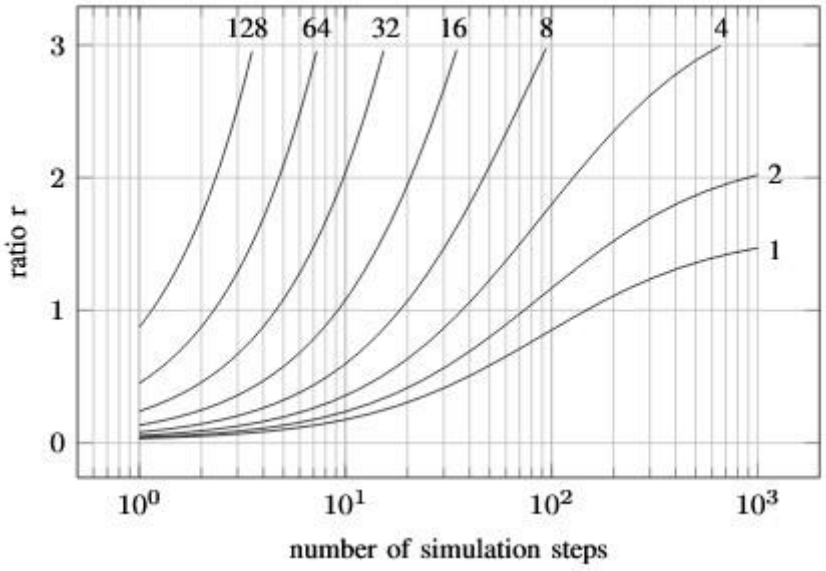

Figure 4 - Ratio of optimization times between central and distributed control.

To conclude the comparison between the two approaches, it can be stated that the *distributed* approach is favourable most of the time as it outperforms the centralized approach in terms of total time taken to run the optimization. If the communication between the OT and the STs is implemented using a network socket-based interface, the simulation workload can be well distributed onto different machines. In this case, the OT needs to be aware of the available ST. In the network-based implementation, previously presented in D6.1, the OT was polling for new STs, before each simulation run. This created a considerable amount of overhead, rendering it impractical to use with more than three STs. Therefore, the Consortium has implemented a solution with two separate phases. First, the *setup phase*, where all available ST are discovered and second, the actual optimization phase, which uses the available STs. The new architecture implementing this proposal design is presented in the next section.

4 Final Simulation and Optimization Environment architecture

This section presents the details of the distributed architecture designed and implemented starting from the lesson learned testing the architecture delivered in the first phase of the project and briefly summarized in the previous section.

4.1 Specification

The architecture uses a network socket-based approach to allow distribution of the STs among different machines. The communication is managed by a central broker, which keeps track of the available STs. The first problem of the Initial Simulation Environment architecture addressed is the discovery protocol responsible for determining the available STs. The previous approach required a discovery phase before each simulation, which affected the performance, because it did not scale well with the number of STs. For this reason, the new architecture introduces a setup phase, where the STs, through their SMs, announce themselves and their capabilities. This allows the other tools to maintain an updated list of available STs, also during the optimization phase. These real time updates eliminate the need for repeated discovery by the OT. The second problem previously encountered is the high number of messages that are exchanged during the optimization phase. To address this, the new architecture is based on the distributed approach described in section 3. The OT sends the controller to the STs, avoiding the exchange of messages between them. The STs can thus perform the simulation stand-alone, without the need to interact with the OT, during the simulation. This reduces considerably the number of exchanged messages, considerably.

This architecture is composed of four main components, as shown in Figure 5:

- The previously mentioned OT, which is responsible for performing the evolutionary optimization.
- Several STs, distributed on different machines, called SSs, each one wrapped by a SM. This latter component is a software layer installed on the SS that implements the network interface and acts as a client that connects the ST to the broker. This allows the {OT to communicate with the STs, without knowing the exact type of ST used.
- Th SOO, a new component in this architecture, which oversees all the SMs and coordinating the simulation and optimization tasks. It maintains a list of available SMs, together with the capabilities of the STs that they wrap. When it is launched, the user can indicate the requirements for the ST to be used, like dimensionality or minimum CPS cardinality. In this way, the SOO can select SMs that fulfil the requirements.
- Finally, the broker that handles all communication between the other components.

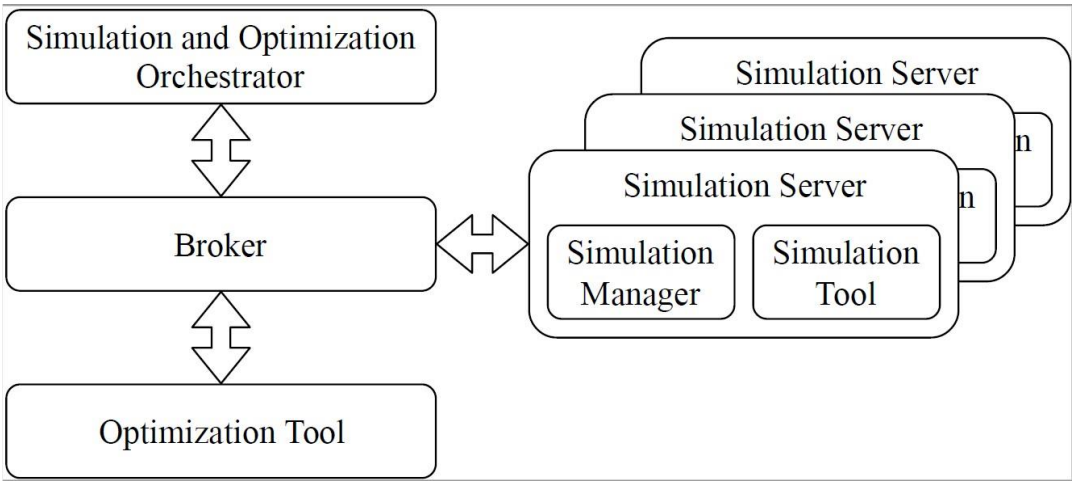

Figure 5 - The network-based architecture consisting of the components SOO, broker, OT, and SSs.

An overview of the final version of the API defined for this architecture (for the details of the format used for these messages, please refer to the deliverable D6.6) is presented in the next subsection. The proposed

architecture can perform simulation workflows, with or without optimization, as described in subsections 4.3 and 4.4. Finally, the last subsection explains the theoretical time complexity of optimizations carried out using this approach.

4.2 Simulator API

This subsection presents the Simulator API defined for the final architecture. The following details are provided for each API:

- Description of the API.
- Components using this API.
- Type of communication used (i.e., file transfer, text message).
- Data included.

4.2.1 Simulation Configuration

- Used to configure the selected STs.
- Sent by the SOO to all the selected SMs.
- File transfer.
- ZIP file including:
    - CPS models.
    - Environment models.
    - Other configurations.

4.2.2 StartOptimization

- Used to start an optimization task.
- Sent by the SOO to the Optimization Tool.
- Message.
- Fields:
    - ID: an ID to allow the optimization process to be tracked together the corresponding simulation environment (it is the name of the package to be optimized, for example *cpswarm_sar*).
    - Type: The type of the message (fixed value: *StartOptimization*).
    - Description: a human readable text to give hints about the message.
    - SimulationConfiguration: a string containing the parameters to be used in the simulation, if some additional parameter is needed.
    - OptimizationConfiguration: a JSON string containing the configuration of the Optimization Tool.
    - SimulationManagers: a list of JIDs to be used to communicate with Simulation Managers, through a chat message.

4.2.3 GetProgress

- Used to get the progress of a running optimization, specified by ID.
- Sent by the SOO to the OT.
- Message.
- Fields:
    - ID: an ID to allow the optimization process to be tracked together the corresponding simulation environment (it is the name of the package to be optimized, for example cpswarm_sar).
    - Type: The type of the message (fixed value: *GetProgress*).
    - Description: a human readable text to give hints about the message.

4.2.4 CancelOptimization

- Used to cancel a running optimization, specified by ID.
- Sent by the SOO to the OT.
- Message.
- Fields:
    - ID: an ID to allow the optimization process to be tracked together the corresponding simulation environment (it is the name of the package to be optimized, for example *cpswarm_sar*).
    - Type: The type of the message (fixed value: *CancelOptimization*).
    - Description: a human readable text to give hints about the message.

4.2.5 SimulationResult

- Used to return the result of a simulation done for optimization.
- Sent by a SM to the OT (in case of optimization).
- Message.
- Fields:
    - ID: an ID to allow the optimization process to be tracked together the corresponding simulation environment (it is the name of the package to be optimized, for example *cpswarm_sar*).
    - Type: The type of the message (fixed value: *SimulationResult*).
    - Description: a human readable text to give hints about the message.
    - SID: ID of the single simulation.
    - fitnessValue: value calculated through the fitness function.
    - Status: OK or Error.

4.2.6 SimualtorConfigured

- Used to reply to the attempt to configure the ST wrapped by the SM.
- Sent by a SM to the SOO.
- Message.
- Fields:
    - ID: an ID to allow the optimization process to be tracked together the corresponding simulation environment (it is the name of the package to be optimized, for example *cpswarm_sar*).
    - Type: The type of the message (fixed value: *SimulationConfigured*).
    - Description: a human readable text to give hints about the message.
    - Status: OK or Error.

4.2.7 OptimizationStarted

- Used to reply to the *StartOptimization* message, with ID for the optimization.
- Sent by OT to SOO.
- Message.
- Fields:
    - ID: an ID to allow the optimization process to be tracked together the corresponding simulation environment (it is the name of the package to be optimized, for example *cpswarm_sar*).
    - Type: The type of the message (fixed value: *OptimizationStarted*).
    - Description: a human readable text to give hints about the message.
    - Status: OK or Error.

4.2.8 OptimizationCancelled

- Used to reply to the *CancelOptimization* message.
- Sent by OT to SOO.

- Message.
- Fields:
    - ID: an ID to allow the optimization process to be tracked together the corresponding simulation environment (it is the name of the package to be optimized, for example cpswarm_sar).
    - Type: The type of the message (fixed value: *OptimizationCancelled*).
    - Description: a human readable text to give hints about the message.
    - Status: OK or Error.

4.2.9 OptimizationProgreess

- Used to reply to the GetProgress message (sent also automatically at the end of the optimization process with the optimized candidate controller).
- Sent by OT to SOO.
- Message
- Fields:
    - ID: an ID to allow the optimization process to be tracked together the corresponding simulation environment (it is the name of the package to be optimized, for example *cpswarm_sar*).
    - Type: The type of the message (fixed value: OptimizationCancelled).
    - Description: a human readable text to give hints about the message.
    - Status: OK or Error.
    - Progress: express in percent the progress reached by the optimization process.
    - FitnessValue: the best value calculated through the fitness function.
    - Candidate: the current best candidate controller.

4.2.10 RunSimulation

- Used to specify the candidate controller to be tested in the simulation, it can be sent by the OT in case of optimization process, or by the SOO in case of simple simulation.
- Message.
- Fields:
    - ID: an ID to allow the optimization process to be tracked together the corresponding simulation environment (it is the name of the package to be optimized, for example *cpswarm_sar*).
    - Type: The type of the message (fixed value: OptimizationCancelled).
    - Description: a human readable text to give hints about the message.
    - SID: ID of the simulation.
    - Configuration: the parameters to be used to run the simulation.

4.3 Optimization Process Workflow

The SOO can perform an optimization using the OT, where each controller candidate is simulated in a SS, as shown in Figure 6.

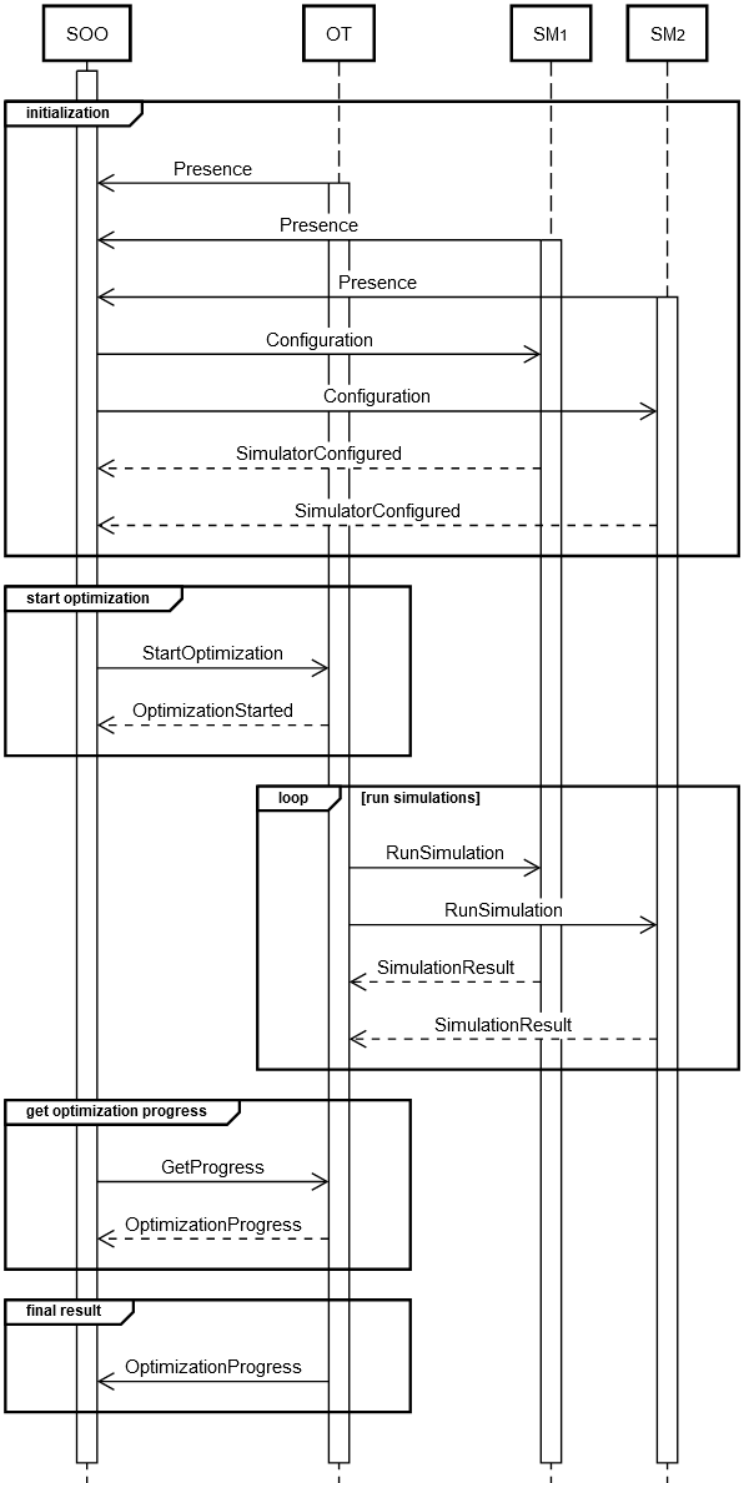

Figure 6 - The messaging sequence during the optimization process.

In this case, the SOO provides the OT with a list of SMs to be used for the optimization process. Then, the OT starts the optimization sending the controller candidates in parallel to all the SMs and collecting the fitness scores calculated during the simulations. Once the optimization is finished, the OT returns the optimized controller to the SOO.

Figure 6 illustrates the flow of messages between the SOO, the OT, and two exemplary SMs, during an optimization. In the initialization phase, all components announce their availability by broadcasting presence information. The SOO collects this information to create a list of available SMs and their capabilities to be used

in the selection phase. Similarly, the OT's presence informs the SOO that it is ready to perform optimization tasks. When the user starts the optimization, the SOO evaluates the available SMs, selecting the ones that fulfil the requirements. Then, it transmits configuration files, to them, to setup the simulation (including, the CPS and environment models received by the modelling tool, the maximum number of simulation steps to be performed, etc.). Once all the SMs have confirmed to have been configured with a *SimulationConfigued* message, the SOO sends a *StartOptimization* message to the OT, which replies with an *OptimizationStarted* message, which includes a unique ID, valid for the whole optimization process. It, then, begins the optimization, sending a sequence of *RunSimulation* messages to SMs, including the candidate controller to be evaluated. The SMs use the corresponding STs to evaluate the controllers and after having calculated the fitness score of the candidate, they send it to the OT, through a *SimulationResult* message Throughout the optimization process, the SOO may request the progress of the optimization process intermittently or even cancel it by sending the OT a *GetProgress* or *CancelOptimization* message respectively, receiving in response an *OptimizationProgress* message in the former case and an *OptimizationCancelled* message in the latter. Once the optimization process completed, the OT sends a final *OptimizationProgress* message to the SOO, which includes the optimized candidate.

4.4 Simulation Process Workflow

The SOO can also be used to send a specific controller candidate to a SM, for more in depth analysis. This allows a controller optimized by the OT to be evaluated more thoroughly, e.g., through visual replay using the ST GUI. In case of visual replay, the selected ST must run on a machine directly accessible for the user, so that they can see the GUI of the ST. In this case, the SOO must configure the selected SM by sending it, the required files and then send it the appropriate controller once the *SimulationConfigured* message has been received.

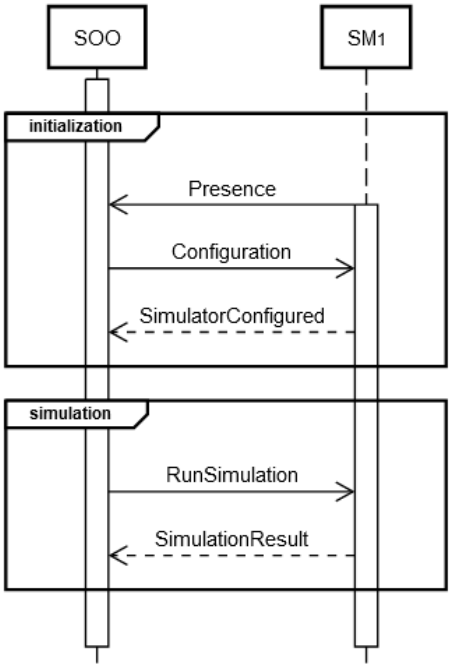

Figure 7 - The messaging sequence when simulating a specific CPS controller.

In this much simpler scenario, the OT is not involved, and SOO and SM communicate directly (see Figure 7).

4.5 Required optimization time

As this architecture introduces two separate phases for setup and optimization, the theoretical time required for optimization therefore changes from Equation 1 to

$$t_{opt} = t_{setup} + n_{gen} * (t_{evo} + \frac{n_{pop} * n_{eval}}{n_{sim}} * (t_{sim} + t_{ohd})$$

Equation 7

having the additional setup time

$$t_{setup} = n_{msg,presence} + n_{msg,config} + 2 * t_{msg}$$

Equation 8

being a multiple of the message transmission time t_{msg}}. The total setup time is made up of the number of *Presence* messages transmitted from the SMs and the OT to the SOO, the number of *Configuration* messages from the SOO to the SMs, and two messages to start the optimization and get the result (*StartOptimization* and *OptimizationProgress*). As the OT requires only one *Presence* message and each SM requires exactly one *Presence* message and one *Configuration* message, $n_{msg,presence} = n_{sim} + 1$ and $n_{msg,config} = n_{sim}$. Hence, the setup time can be rewritten as

$$t_{setup} = (2 * n_{sim} + 3) * t_{msg}$$

Equation 9

Based on this architecture, the next section presents an implementation using the XMPP protocol.

5 Final Simulation and Optimization Environment Prototype

The architecture described in the previous section is implemented using existing tools wherever possible with extensions only being developed where required by the proposed architecture. This section begins by describing the communication infrastructure that connects the different tools, then describes their implementation.

5.1 XMPP protocol

XMPP is a free and open-source real-time messaging protocol, based on XML, originally designed for chat application, but flexible enough to be used in many other domains [2]. Performance of XMPP in terms of latency, scalability and robustness has been tested and demonstrated [3]. The XMPP core has been standardized by IETF in 2004, while a large community of developers are continuously adding new extensions that are integrated in the protocol under the guide of the XMPP Standard Foundation (XSF). Its architecture is fully decentralized and extensible. Beside interoperability features, such as service discovery and server federation, XMPP embeds also security features, such as Simple Authentication and Security Layer (SASL)[2] for authentication and Transport Layer Security (TLS)[3] for encryption, both for client-to-server and server-to-server communications. Because XMPP owns features that allow fulfilling the architecture requirements described in Section 4.1, it has been selected as communication protocol. It is favoured over MQTT, used in the previous implementation (see Section 3.2.2**Error! Reference source not found.**), because MQTT does not provide all the required features, like one-to-one communication or an embedded presence protocol. Through XMPP, the proposed architecture can be implemented without having to leverage on many different protocols. This implementation uses many of the features provided by XMPP:

- The different architecture tools are identified through a Jabber Identifier (JID), allowing them to be addressed unambiguously.
- The presence mechanism is used by the SMs to announce their availability and to signal their status in real-time.
- Chat messages are used for the communication among the tools. Furthermore, some of the available XMPP Extension (XEP)s are used, specifically XEP-0095[4], XEP-0096[5], XEP-0065[6], and XEP-0047[7] for file transfer between the tools.

5.2 FREVO-XMPP

5.2.1 Overview

FREVO-XMPP was developed as a sample optimization tool that builds upon FREVO, a modular optimization system based on the principles of genetic algorithms [4]. Both the controller representations as well as the evolution method used during the optimization process, may be costumized. In this instance, we provide a sample implementation of the Neural Network Genetic Algorithm (NNGA) [5]. It begins by creating n_{pop} controller candidates. In each of the n_{gen} generations, the controllers are evaluated and ranked according to their performance. Successful controllers, i.e. those with high fitness values, are carried to the next generation as elite, or are crossed or mutated to produce new controllers. In addition, a small proportion of entirely new random controllers is introduced with the intention of maintaining diversity in the population. FREVO currently supports two execution strategies for the evaluation of controller candidates.

[2] https://tools.ietf.org/html/rfc4422
[3] https://tools.ietf.org/html/rfc5246
[4] https://xmpp.org/extensions/xep-0095.html
[5] https://xmpp.org/extensions/xep-0096.html
[6] https://xmpp.org/extensions/xep-0065.html
[7] https://xmpp.org/extensions/xep-0047.html

- Candidate pool: a work item is created for each controller candidate. Each work item creates n_{eval} problem variant instances to evaluate performance.
- Problem pool: a work item is created for each problem variant. Each work item evaluates each of the controller candidates sequentially.

The choice of strategy should be made depending on the type of problem. In general, a problem pool has the advantage that a single problem component is used to evaluate multiple controller candidates and may improve performance in situations where the overhead of starting a simulation instance is high or controller candidates may be substituted with minimal overhead. In instances where only a single problem variant is required, a candidate pool should be used to ensure parallelism.

5.2.2 FREVO-XMPP Implementation

FREVO-XMPP provides a wrapper around FREVO that supports XMPP. The optimization process proceeds as follows:

- Upon receiving a *StartOptimization* message, FREVO-XMPP creates an *OptimizationTask* to oversee the optimization process and runs it on a new thread. As the evaluation of controllers is conducted by the SMs, FREVO-XMPP is largely input/output bound and can thus execute multiple *OptimizationTasks* in parallel without any significant CPU load.
- The *OptimizationTask* deserializes FREVO-XMPP configuration, which specifies the type of evolution method, the controller representation, as well the operations to be performed on them throughout evolution. Furthermore, it receives a list of SMs which may be used to evaluate controller candidates. Communication with these is managed by an associated SM proxy.
- Rather than evaluating a controller locally, FREVO-XMPP's *GenericProblem* component forwards requests for evaluation to an available SM proxy, which in turn sends a *RunSimulation* message to its associated SM and blocks waiting for a *SimulationResult* message or a simulation timeout to occur.
- The optimization continues until completion resulting in an *OptimizationComplete* message being sent back to the SOO.

5.3 SOO

The SOO is implemented as Java application embedding an XMPP client (based on the open-source XMPP library Smack[8]) as briefly explained in the deliverable D6.5. The next subsections briefly describe the main classes of the SOO.

5.3.1 SimulationOrchestrator

The user runs the SOO using the CPSwarm Launcher, passing to it several parameters, which specify the requirements of the optimization task. These parameters include requirements for the evolutionary process as well as requirements for the simulation.

- *Generation count:* Number of generations used in the evolutionary optimization process.
- *Candidate count*: Number of controller candidates evaluated in each iteration of the evolution.
- *Seed*: Seed for random number generation in the OT.
- *Opt*: Whether the task to be executed requires optimization.
- *GUI*: Whether the simulation is executed showing the ST GUI.
- *Dimensions*: Number of dimensions required for the simulation.
- *Max*: Maximum number of CPSs required for the simulation.
- *Params*: Command line parameters to be passed to the ST.
- *Simulation timeout*: Time after which the simulation is terminated.

[8] https://www.igniterealtime.org/projects/smack/

The list of parameters will be extended in future releases.

When the SOO starts, it configures itself using the parameters passed through the Launcher and the values inserted in the embedded in its configuration file (see ANNEX N). Then, it starts the client connecting itself to the configured XMPP server. Furthermore, if its configuration indicates to start also the OT, it runs the configured executable. Once connected. the SOO collects the presences of the available SMs (both already known or new ones) and evaluates the capabilities provided by the STs wrapped by the available SMs and to compare them with the requirements requested for the simulation task (with or without optimization involved).

In case of simulation only: currently the SOO selects the first suitable SM and sends it the configuration files inserted in its configuration folder. When the *SimulatorConfigured* message is received, it sends a *RunSimulation* message to the SM, including the candidate to be replayed in the ST.

In case of optimization: the SOO selects all the suitable available SMs and sends the configuration files to all of them. When all corresponding *SimulationConfigured* messages have been received, it sends a *StartOptimization* message to the OT, including the list of SMs to be used. Then, the SOO waits to receive the *OptimizationProgreess* message from the OT, indicating that the optimization task is finished and saves the optimized candidate, in the *output* folder configured in the Launcher.

5.3.2 GetProgressSender

A runnable class that periodically sends a *GetProgress* message to the OT, to monitor the progresses of an optimization task.

5.3.3 OptimizationToolLauncher

A runnable class used to run the executable of the OT, if the SOO is configured to run also the OT.

5.3.4 Listeners

5.3.4.1 ConnectionListenerImpl

This listener is used to receive connection status notifications, so that it is able react when the connection goes down or, at least, notify the user).

5.3.4.2 MessageEventCoordinatorImpl

This listener is used to receive the chat messages sent by the Optimization Tool (described in Section 3.1.1). It handles the messages, taking the decisions based on this. If it is a positive *OptimizaitonStarted* message, it start to wait the optimized candidate; if it is a positive *OptimizationCanelled*, it reset the status of the current optimization; if it is a positive *optimizationProgress*, it forwards the message to who is required it and if it the progress is of 100% (the *optimizationProgress message* has been automatically sent by the OT to signal a finished optimization), the final best candidate controller is stored in the output folder. Finally, in case of errors, they are logged and handled accordingly.

5.3.4.3 PacketListenerImpl

This listener is used to receive presences from the OT and the SMs, to accept their subscription requests and to collect the SMs capabilities.

5.3.5 SOO configuration file

The SOO is configured using a custom XML file (see ANNEX N for the schema).

- *serverURI*: Uniform Resource Identifier (URI) of the XMPP server.
- *serverName*: Name of the XMPP server.

- *username:* Username to be used to connect to the XMPP Server.
- *serverPassword*: Password to be used from the SOO to connect to the XMPP server (temporary solution).
- *optimizationUser*: XMPP username of the OT.
- monitoring: Indication if the monitoring GUI must be used or not
- *configEnabled*: Indication if the configuration of the simulators must be done or not.
- *startingTimeout:* Time used to wait new Simulation Managers.
- *mqttBroker*: MQTT broker to be used if the monitoring is set to true.
- *localOptimization*: Indicates if the OT has to be launched by the SOO.
- *optimizationToolPath*: Path of the optimization tool executable.
- *optimizationToolPassword*; To be used if the Optimization Tool has to be launched from the SOO (temporary solution).

5.4 Simulation Manager API

The SM implementation is also done in Java and split into two parts, as described in Deliverable D6.5. First, a common part is implemented as abstract class to provide a base module for all SMs implementations. Each SM for a specific ST is derived from this class and shared as a separate component. The specific part of the SM defines how to handle the files received for the configuration and the messages received with the controllers to be simulated.

5.4.1 SimulationManager

The main class of the Simulation Manager is *SimulationManager*. This is an abstract class implemented by all the SMs. It starts loading all the values passed by the specific implementations (see Section 5.5.1). The *SimulationManager* encapsulates an XMPP client (based on Smack library). It begins by connecting itself to the XMPP server and it creates the account with the ID of the manager that is composed by the string *"manager_"* followed by a random Universally Unique Identifier (UUID). It adds to the connection a set of listeners (described in the following subsection). When a SM starts, it adds the SOO to its roster[9], which is the list of its ``friend accounts". It signals its availability by sending a *Presence* message including a list of features provided by the wrapped ST in the status, which is automatically received by all the ``friends", i.e., the SOO. This implementation features the number of dimensions and the maximum number of CPSs supported. After choosing the SM to be used, the SOO sends to the SM the files that are required for configuring the ST, using the XMPP file transfer. These include the models of CPSs and environment. The SM confirms the reception of the configuration with an acknowledgment. After this, the SM waits for messages instructing it to run the required simulations. To do this, several listeners are used, as described in the following subsection.

5.4.2 Listeners

5.4.2.1 AbstractFileTransferListener

This listener is fired when a request to transfer files is received. Typically, it contains the file to be used to setup the wrapped ST. The class is abstract because the implementation of the methods used to handle the configuration files is delegated to the specific SMs, since every SM will need to do different operations or conversions.

[9] https://xmpp.org/rfcs/rfc6121.html#roster

5.4.2.2 AbstractMessageEventCoordinator

This listener is fired when a message is received, specifically a *RunSimulation* message indicates to the SM to start a simulation. The class is abstract because the implementation of the methods used to handle the message is delegated to the specific SMs, since every SM will need to do different operations.

5.4.2.3 ConnectionListenerImpl

The same listener described in Section 5.3.4.1.

5.4.2.4 PresencePacketListener

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

After the description of the parts common to all the SMs, the next section will introduce the typical structure of a specific SM implementation.

5.5 Simulation Manager Implementation

The SM implementations are developed as Java applications that implement the abstract classes defined in the SM API to interact with the specific ST. The next subsections describe the typical component of a SM implementation. For a detailed description of the two SMs developed for Stage[10] and Gazebo[11], refer to deliverable D6.6.

5.5.1 SpecificSimulationManager

The main class is an implementation of the abstract class described in Section 5.4.1 that loads the specific configuration of the SM (see deliverable D6.6 for the configuration examples) configuring it and its superclass. It, then, instantiates the implementations of any needed listeners, as described in the next subsections.

5.5.2 Listeners

5.5.2.1 FileTransferListenerImpl

An implementation of *AbstractFileTransferListener* for the specific SM that receives the files sent for the configuration contained in a compressed file, extracts them in a temporary folder and puts them in the folders used by the wrapped ST. If needed, the files are converted to the format used by the ST.

5.5.2.2 MessageEventCoordinatorImpl

An implementation of the AbastractMessageEventCoordinator described in Section 5.4.2.2 that receives the messages and uses them to interact with the wrapped ST. For example, the *RunSimulation* message will be handled, saving the candidate in the ST and then running it to execute the simulation.

5.5.3 SM configuration file

The SMs are configured using a custom XML file (see ANNEX O and ANNEX P for the schema of the files of the Stage and Gazebo SM). The configuration file contains the following tags, as well as some additional custom tags for each ST:

[10] http://playerstage.sourceforge.net/

[11] http://gazebosim.org/

- *Uuid*: If present, indicates the Universally Unique Identifier (UUID) to be used in the username (it is useful to have fixed username).
- *serverURI*: URI of the XMPP server.
- *serverName*: Name of the XMPP server.
- serverPassword: Password to be used to connect to the XMPP server.
- dataFolder: Data folder where to store the data.
- *Dimensions*: Dimensions supported by the wrapped simulator.
- *maxAgents*: Maximum number of agents supported by the wrapped simulator.
- *optimizationUser*: XMPP user of the OT.
- *orchestratorUser*: XMPP user of the SOO.
- *rosFolder*: Folder of the ROS workspace, it must be the <src> folder.
- *monitoring*: Indication if the monitoring GUI must be used or not.
- *mqttBroker*: MQTT broker to be used if the monitoring is set to true.

5.6 Simulator API

The simulator API allows interconnecting the SOO and the OT (i.e., FREVO) to one or more SMs, which in turn, provide access to heterogeneous STs. All the components use a XMPP client to connect to a centralized XMPP server using the APIs described in Section 4.2. In the following subsections, a sample communication sequence for optimization and simulation are presented.

5.6.1 Sample Optimization messages

The SOO collects the presences sent by the SMs when they go online, which contain their capabilities as shown in Schema 1.

```
{
   "server":"manager_561fad07-bdb7-4f32-9fa7-1a5f0ae325f9@pert/cpswarm",
    "capabilities":
    {
      "dimensions":2,
      "max_agents":8
     }
}
```

Schema 1 - Exemplary SM presence status

The SOO selects the suitable SMs based on the requirements indicated by the user and then starts a new optimization by sending the OT a *StartOptimization* message, as shown in Schema 2.

```
{
    "optimizationConfiguration": "",
    "simulationConfiguration": "",
    "SimulationManagers": [
        "sm@xmpp.example"
    ],
    "ID": "optimization1",
    "type": "StartOptimization",
    "description": ""
}
```

Schema 2 – Exemplary *StartOptimization* message

Upon receiving this, the OT creates an optimization task and acknowledges this with an *OptimizationStarted* message, as shown in Schema 3.

```
{
    "status": "ok",
    "ID": "optimization1",
    "type": "OptimizationStarted"
}
```

Schema 3 - Exemplary *OptimizationStarted* Message

The OT creates candidate controllers and sends them to a SM for execution with a *RunSimulation* message, as shown in Schema 4.

```
{
    "SID": "0",
    "configuration": "",
    "candidate": "/* code */",
    "ID": "optimization1",
    "type": "RunSimulation"
}
```

Schema 4 - Exemplary *RunSimulation* message

The SM creates and executes a simulation with the controller and once execution is complete, replies to the OT with the fitness value via a *SimulationResult* message, as shown in Schema 5.

```
{
    "SID": "0",
    "fitnessValue": 55.66969150395627,
    "status": "ok",
    "ID": "optimization1",
    "type": "SimulationResult",
    "description": ""
}
```

Schema 5 - Exemplary *SimulationResult* message

The exchange of candidate controllers between the OT and SMs continues as the optimization progresses. The SOO may query the progress of the optimization using a *GetProgress* message, as shown in Schema 6.

```
{
    "ID": "optimization1",
    "type": "GetProgress",
    "description": ""
}
```

Schema 6 - Exemplary *GetProgress* message

The OT replies with the current progress and the best fitness achieved with an *OptimizationProgress* message, shown in Schema 7.

```
{
    "progress": 50.0,
    "fitnessValue": 55.66969150395627,
    "candidate": "/* code */",
    "status": "ok",
    "ID": "optimization1",
    "type": "OptimizationProgress"
}
```

Schema 7 – Exemplary required *OptimizationProgress* message

Once the optimization is complete, the OT sends a final *OptimizationProgress* message to the SOO, as shown in Schema 8.

```
{
    "progress": 100.0,
    "fitnessValue": 86.465897247841939,
    "candidate": "/* code */",
    "status": "ok",
    "ID": "optimization1",
    "type": "OptimizationProgress"
}
```

Schema 8 - Exemplary final OptimizationProgress message

5.6.2 Sample Simulation messages

The SOO collects the presences sent by the SMs when they go online, containing their capabilities, as shown in Schema 9.

```
{
   "server":"manager_5432ad07-bd47-4242-9fa7-1a5f0ae325f9@pert/cpswarm",
    "capabilities":
    {
      "dimensions":3,
      "max_agents":8
     }
}
```

Schema 9 - Exemplary SM presence status

Then it selects the first SM available that satisfies the requirements and sends to it the *RunSimulation* message, with the candidate controller to be replayed, as shown in Schema 10.

```
{
    "SID": "0",
    "configuration": "visual:true",
    "candidate": "/* code */",
    "ID": "simulation1",
    "type": "RunSimulation"
}
```

Schema 10 - Exemplary *RunSimulation* message

This is used to see the behavior of a candidate in the ST GUI, so no resulting message is expected.

5.7 EmergencyExit Simulation Environment

The *EmergencyExit* problem, a simple scenario in which three CPSs are required to find one of two exit points, was developed to test the different components and workflows of the architecture. The simulation runs in discrete time and space and thus in every time step, a CPS can move to one of the adjacent fields. Each CPS tries to reach one of the exits of the environment, while avoiding the fields occupied by obstacles or other CPSs. The performance of the simulation is measured after a given number of simulation steps as the distance of every CPS from the nearest exit. Additionally, the time taken to reach the exit is considered to reward faster solutions. These distances are recorded in a log file that is used by the SM to report the overall fitness of a given controller to the OT.

The *EmergencyExit* problem is implemented as a ROS package, composed mainly of two source files:

- The controller representation as C++ code, typically implementing an ANN generated by the OT. Based on the contents of the surrounding cells and the position of the nearest exit, it calculates two outputs, which are used by the CPS as instructions for vertical and horizontal movement.
- A ROS wrapper, which allows ROS to collect input from the simulation environment and apply the actions on the simulated CPS, based on the outputs calculated by the ANN. The C++ class allows the CPSs to communicate with each other and with the simulation environment using the ROS publish/subscribe communication paradigm.

The ROS wrapper must be adapted for different STs, while the controller C++ code can be reused seamlessly. During the optimization process, the wrapper remains fixed with only the controller code changing for each simulation.

It is planned to release the code of this implementation in 2019 as open source on the CPSwarm Github repository[12]. It will include FREVO-XMPP, SOO, SM API, and SM implementations for Stage and Gazebo.

Several testbeds have been setup to test the solution presented in this section and its performance. The description of the testbeds and the corresponding test cases are described in the next section, together with a performance analysis of this architecture, compared with the previous one.

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

6 Performance and Scalability evaluation

After having defined an updated architecture, the partners evaluated it, using the prototype described in the previous section. to test if the problems seen in the Initial Simulation and Optimization Environment architecture had been solved and to compare the performances of the two approaches. The following subsections present the testing of the solution and the analysis of the performance and scalability of the two approaches.

6.1 Testbeds

The solution presented in Section 4 and 5 was evaluated using three test cases. This section describes the testbed setup of these test cases. The first setup acts as a Proof of Concept (PoC) to demonstrate the provided features. The other two are used to evaluate the performance of the presented approaches.

6.1.1 Testbed for features evaluation

For the first test case, three SSs. running the Stage ST and the corresponding SM are used. Another computer is used both as SS, running the Gazebo ST with SM, and to run the SOO and the FREVO-XMPP. The XMPP server is installed in the cloud. Both Openfire[13] and Tigase[14] have been used. This setup is visualized in Figure 8.

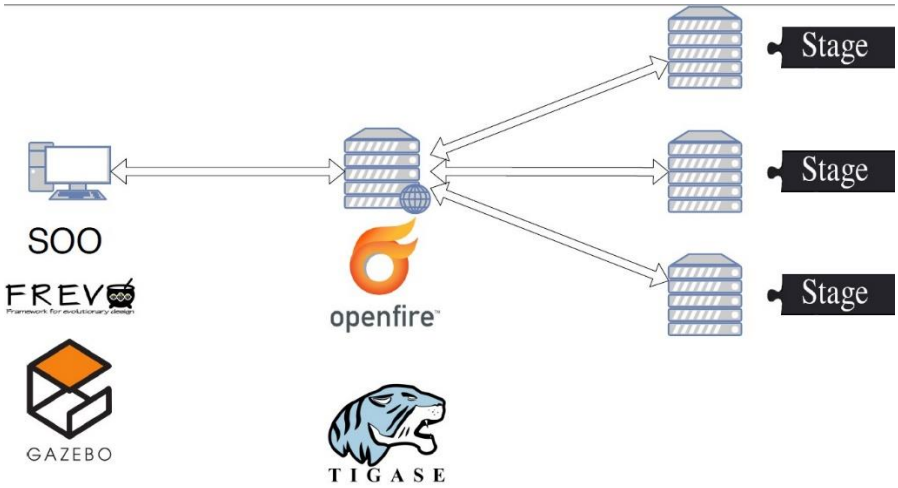

Figure 8 - Proof of concept testbed setup.

To test the different components and workflows of the architecture, first an optimization is performed and then the result is then replayed locally with a GUI. For this purpose, the authors implemented the *EmergencyExit* problem in simulation as ROS components, both for the Stage and Gazebo ST. The implementations are based on a simple scenario with three CPSs and two exits. The scenario setup for the Stage and Gazebo STs can be seen in Figure 9 and Figure 10. This simple setup can effectively test the architecture, without shifting the focus to the challenges related to performing complex multi-CPSs simulations. As shown in Figure 10, the two implementations feature a different level of abstraction: Stage implements the CPSs as simple squares, while Gazebo implements them as TurtleBot robots.

[13] https://www.igniterealtime.org/projects/openfire/
[14] https://tigase.net/content/tigase-xmpp-server

Figure 9 - Stage ROS implementation of the *EmergencyExit* simulation.

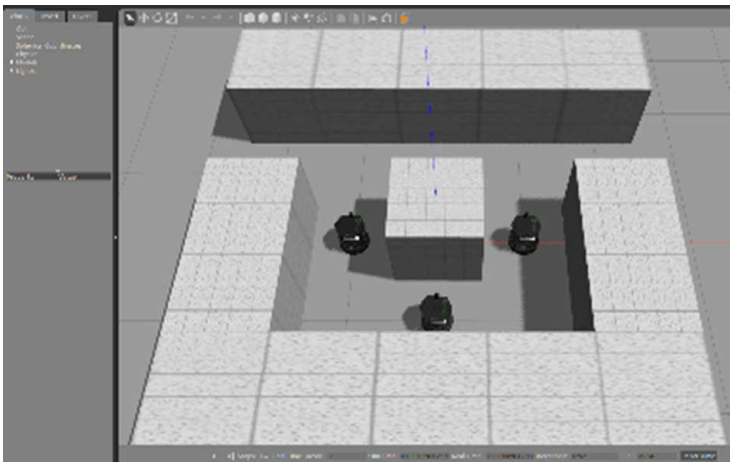

Figure 10 - Gazebo ROS implementation of the *EmergencyExit* simulation.

To perform the test, the authors launched the SOO, selecting the optimization workflow with the requirement to perform simulations in two dimensions, increasing the speed of each simulation. Consequently, the SOO successfully selected the three SSs running Stage and excluded the one running Gazebo and sent the *StartOptimization* message to FREVO-XMPP, together with the JIDs of the SMs to be used. FREVO-XMPP then carried out the optimization, distributing the simulation tasks to the SMs and then, returned the optimized controller to the SOO. To continue the test, the authors, then, launched the SOO again, this time selecting the simulation-only workflow with the requirement to perform the simulation in three dimensions with a GUI enabled. As a result, the SOO successfully launched the simulation locally in Gazebo replayed the final optimized candidate within a more realistic 3D environment. This test case showed the ability of the SOO to automatically choose the correct SS, based on the requirements specified by the user and the capabilities exported by the installed SM. It thus demonstrated how the simulators are seamlessly integrated through the proposed architecture. The use of ROS in the optimization and simulation process ensures the portability of the controller among different STs. In a final step, optimized controller was installed on a real TurtleBot robot and tested in an environment like the one used in simulation (see Figure 11). In this way, the complete chain was tested spanning optimization, simulation and deployment on a CPS hardware platform.

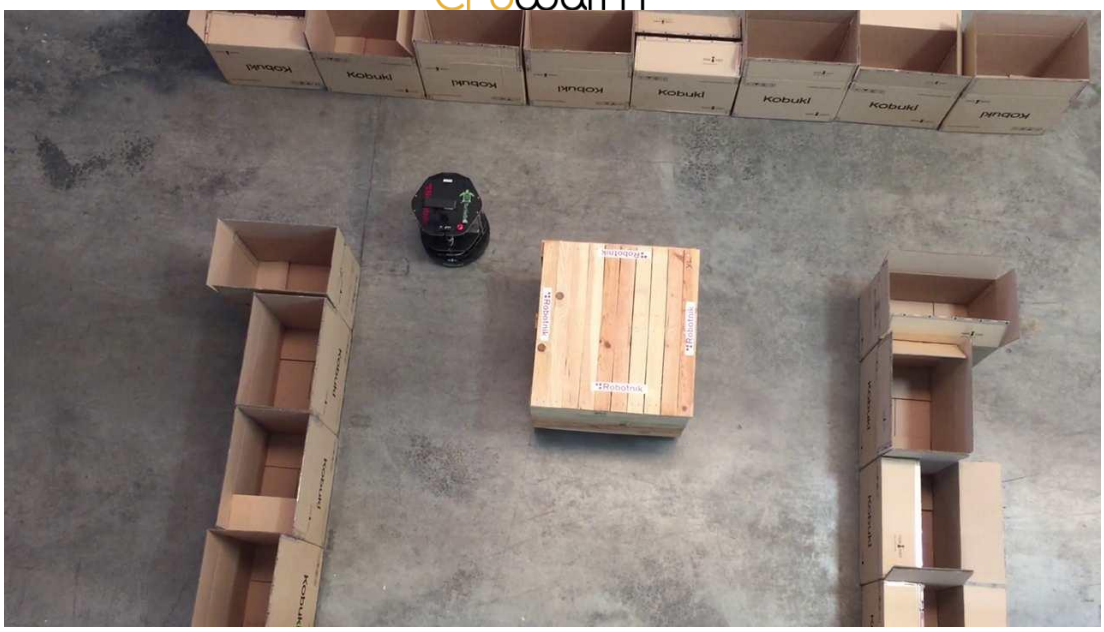

Figure 11 - Real world experiment of the of the *EmergencyExit* problem using TurtleBot robots.

6.1.2 Testbed for performance comparison

A second test case with four distributed SSs (see Figure 12) was constructed for a realistic comparison between the centralized and the distributed approaches. For technical reasons, the centralized approach is implemented using the *Minisim* Java simulation (see Section 3.2.2.1), while the distributed approach is based on the *EmergencyExit* ROS simulation (see Section 5.7). Nevertheless, both approaches are comparable as both perform simulations lasting for a given number of steps.

All four computers run the Stage ST with corresponding SM, the *EmergencyExit* ROS simulation and ROS Kinetic[15], with one additionally hosting The SOO and the FREVO-XMPP OT. The components are connected to a Tigase XMPP server running in the cloud. The test case can be used to test the complete optimization process, first using one SS and then parallelizing using two, three, and four SSs.

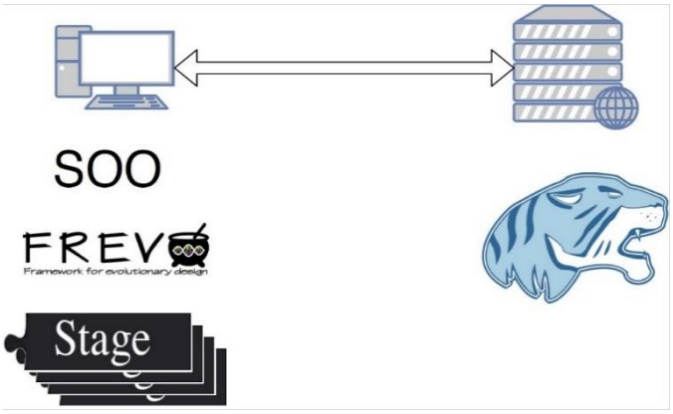

Figure 12 - Performance comparison testbed setup.

6.1.3 Testbed for scalability

Finally, a third test case was constructed to evaluate the scalability of the implementation for a given number of SSs with the objective of showing the maximum degree of parallelization possible, using the current

[15] http://wiki.ros.org/kinetic

distributed approach. To do this, all implemented tools (SOO, OT, and SMs) are executed on a single computer (see ANNEX A for its specifications). As shown in Figure 13. As before, the OT used is FREVO-XMPP and the Tigase XMPP server runs in the cloud.

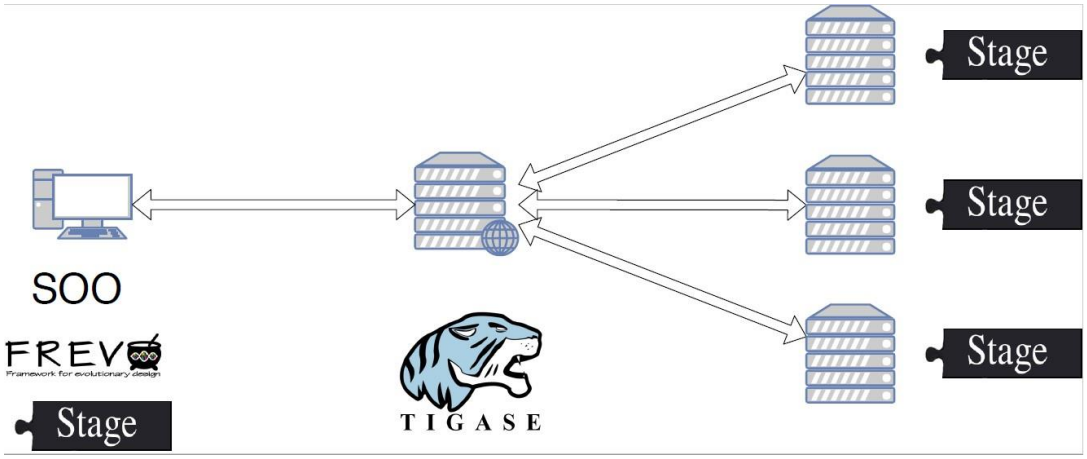

Figure 13 - Scalability evaluation testbed setup.

To rule out performance limitations of the test computer on FREVO-XMPP, the simulations used for the scalability analysis are just a sleep phase, which does not put any computational load on the computer. As it only serves to analyse the scalability of the network performance with the number of SSs, it emulates the overhead time as

$$t_{ohd} = \left(n_{msg,setup} + n_{msg,finalize}\right) * t_{msg}$$

Equation 10

excluding import, export, and fitness computation times. The performance measurements are discussed in the next section.

6.2 Performance and scalability analysis

This section presents the performance evaluation of the proposed architecture, leveraging the testbed described in the previous subsection. The parameter evaluated is total time taken for a complete optimization run. This optimization time is measured for a varying number of simulation steps n_{steps} and the number of available SSs n_{sim}. All other parameters are fixed. To get reliable results, each measurement is repeated at least five times until the relative error of the sample is at most 10%, with a confidence of 99.9%.

The analysis begins by comparing the *centralized* approach to the *distributed* approach introduced in Section 3.2.1. Then a more in-depth analysis of the distributed approach that analyzes its scalability with the number of SSs is presented.

6.2.1 Comparison of Centralized and Distributed Approach

To compare the centralized and the distributed approach, the partners first compute the total optimization time including setup time, based on Equation 1 and Equation 7 respectively. Then the partners perform measurements according to the testbed setup described in Section 6.1.2. To calculate the optimization time, the measurements presented in Table 1 are used. The number of CPSs being simulated is $n_{cps} = 8$ and the evolutionary parameters are set to $n_{gen} = 4$ and $n_{pop} = 4$. This yields the optimization times

$$t_{opt,c} = \frac{9.28\,s * n_{step} + 2.41s}{n_{sim}} + 0.048s$$

Equation 11

for the centralized approach and

$$t_{opt,d} = 0.06s * n_{sim} + \frac{1.6s * n_{step} + 142.39s}{n_{sim}} + 0.14s$$

Equation 12

for the distributed approach.

These optimizations times are plotted in Figure 14 as function of SSs and simulation steps. It shows the inverse proportionality between the simulation time and the number of SSs. Increasing the number of SSs is therefore well suited for reducing the total optimization time. The small term of direct proportionality of the distributed approach does not prevail for such low numbers of SSs. The major difference between the approaches lies within the dependency on the number of simulation steps. Here it becomes clear that the *distributed* approach becomes favourable as simulation lengths increase. In this example, using eight CPSs, the *centralized* approach is favourable only for short simulations in the order of ten steps. This is in line with the conclusion from the ratio shown in Figure 4.

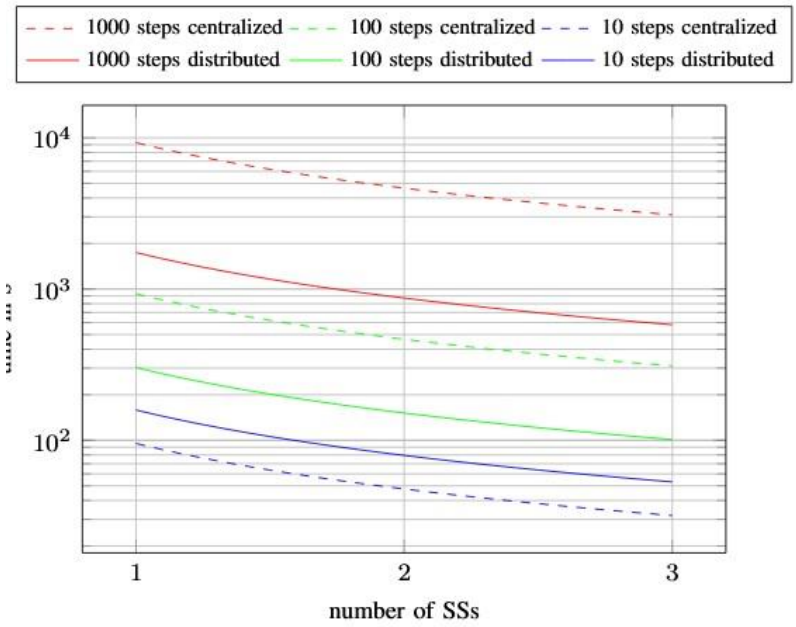

Figure 14 - Theoretical comparison of the scalability with number of SS of the optimization time between centralized and distributed approach for varying simulation lengths and eight CPSs.

Next, measurements using the testbed described in the previous section are performed. They are compared to the performance results of the centralized MQTT implementation presented in D6.1. The results can be seen in Figure 15. Showing that the *centralized* approach scales poorly beyond three SSs because it performs SS discovery before each simulation. The implementation of the distributed approach mitigates this problem by introducing a different presence mechanism. The performance is mostly in line with the calculations presented above. For short simulations the centralized approach is preferable, whereas for the other cases the distributed approach performs better.

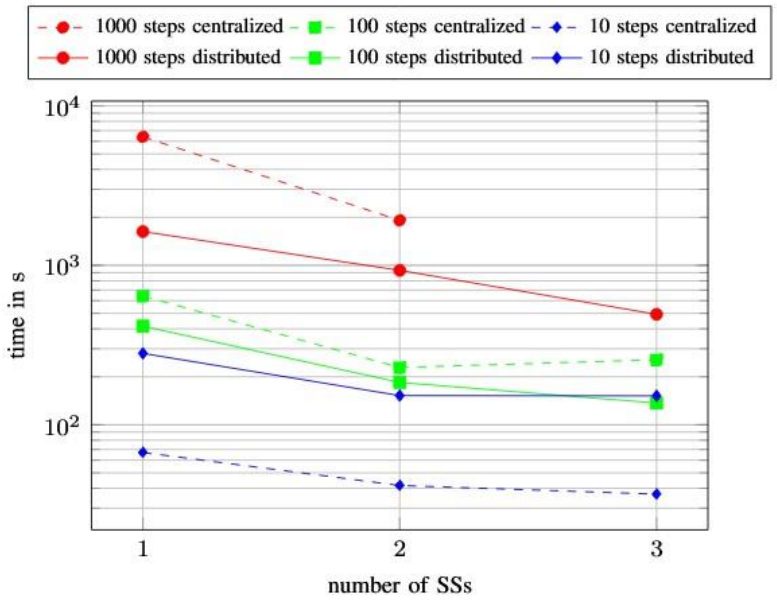

Figure 15 - Measured comparison of the scalability with number of SS of the optimization time between centralized and distributed approach for varying simulation lengths and eight CPSs.

6.2.1 Scalability Analysis

In the previous subsection, the authors explained the difference between the two approaches, and found that the *distributed* approach excelled in most scenarios. To further investigate how well the performance scales with a larger range of SS, measurements using the third test case described in Section 6.1.3 were performed. To be able to assess the parallelization, the parameters n_{gen}= 4 and n_{pop}= 32 were used. Because a typical optimization process includes only a single setup phase, only the optimization time is measured as the time between transmitting the *StartOptimization* message and receiving the final *OptimizationProgress* messages at the SOO. Figure 16 shows the resulting optimization time. The measurements are mostly in line with the theoretically calculated performance. The performance scales well with the number of SSs. The small difference the predicted and measured values is due to implementation details not captured in the model. When the number of SSs is increased beyond 16, performance does not scale well anymore, as the testbed computer used for the tests only has 24 cores.

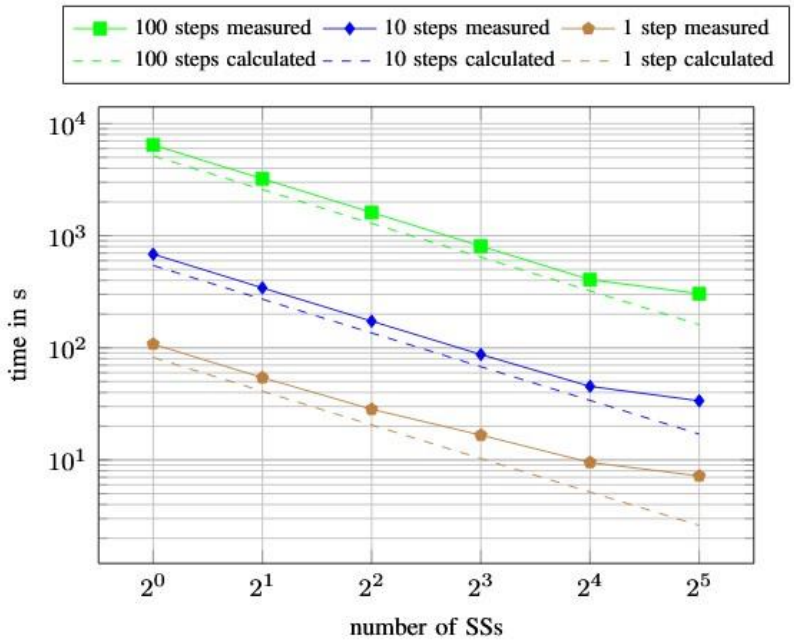

Figure 16 - Scalability with number of SS of the optimization time of the distributed approach for varying simulation lengths.

This comparison demonstrates that while the choices made by the Consortium are correct, some improvement is still possible. The next subsection will present the outcomes of such analysis.

6.3 Architecture evaluation outcomes

While testing the architecture presented in Section 4, it became clear that the scalability of the system needed to be improved. Indeed, currently, the system requires one dedicated machine for each ST. By using a container service (e.g., Docker[16]), multiple STs may be run by each SS. To do this, a set of containers both for ROS based STs and ad-hoc STs like the *Minisim* (see 3.2.2.1) as well as the associated SMs need to be created. Furthermore, this strategy combined with solutions like Docker Swarm[17] or Kubernetes[18] would allow the user to easily deploy and maintain large sets of STs. The system could also be deployed to the cloud to circumvent the inherently resource-bound nature of simulation. Corresponding improvements will also be made to the OT to support many SMs.

Considering these outcomes of the analysis, the partners have defined and implemented a set of deployment and scalability features to be added to the Simulation and Optimization Environment architecture. Such features will be described in the next Section 7.

A further weakness of the architecture presented in Section 4 is that it that the entire optimization process must be restarted from the beginning if it is disrupted. This is particularly time consuming in long running optimizations. In the next few months, solutions to address these issues will be investigated and will be documented in the upcoming WP6 deliverables *D6.4 - Final CPS System Design Optimization and Fitness Function and Design Guidelines* and *D6.7 - Final Integration of external Simulators*.

[16] https://www.docker.com/
[17] https://docs.docker.com/engine/swarm/
[18] https://kubernetes.io/

7 Deployment and scalability features

Based on the outcomes of the analysis presented in the previous section, the partners have introduced a set of deployment and scalability features to enhance the scalability of the solution implemented and to allow the rapid deployment of the distributed SMs.

7.1 Specification

This section describes the features that have been added to the final architecture of the Simulation and Optimization Environment (described in Section 4.1) to further improve its scalability.

The SOO, the OT and the STs are included in containers, so each one can be run in a container environment, such as Docker. In case of ROS-based STs, the containers need to contain:

- A ROS distribution.
- One or more ROS based STs (for example, one container could support both Stage and Gazebo).
- The ROS packages of the simulations.
- The SMs of the STs, with integrated the XMPP clients, to communicate with the other components.

The containers of ad-hoc STs (e.g., *Minisim* described in Section 3.2.2.1) instead need only to contain:

- The ad-hoc STs.
- The relative SMs.

In this way, it is possible to use one rapid deployment and orchestration tool (e.g., Docker Swarm or Kubernetes) to deploy the required set of STs using the CPSwarm workbench. To achieve this, the SOO will support three different operating mode:

- *Deployment mode*: if launched in this modality, the user will pass to the SOO, the desired set of STs to be deployed in the available SSs (potentially more than one for each SS), the SOO will use the chosen deployment solution to verify the current set of available STs and eventually to scale the environment to reach the desired state.
- *Running mode:* this is the same behaviour described in Section 4.1 to execute an optimization process or to replay a candidate in one ST.
- *Deployment & Running mode*: this is a combination of the behaviours described in the previous two points; it deploys the needed STs and then starts to use them.

Besides improving the scalability of the solution by allowing to install multiple instances of ST in the same SS, the tools can rapidly scale the set of available STs.

It is important to highlight that this is only an addition to the previous method of deploying STs and not a replacement. Indeed, thanks to the use of SMs, it is possible both to use the containerized STs and manually installed standalone STs (and abstracted with SMs).

7.2 Implementation

This section presents the prototype developed to test the new features presented in this section: firstly, the technologies analysed and chosen to implement them will be introduced; then, the next subsections will present the solutions implemented.

7.2.1 Technologies implemented

7.2.1.1 Container Runtimes

Docker

In 2018, 83% of production containers used Docker[19] (was 99% in 2017).

Docker is available in two versions:

- Docker Community Edition (CE) is the open-source version that is usually the right solution for users that need to experiment with Docker and, with container-based apps
- Docker Enterprise Edition (EE) is the commercial version suitable for users that need to build, ship, and run business-critical applications in production. Docker EE is certified to provide enterprises with a container platform suitable for commercial deployment, with security features like Role Based Access Control (RBAC). integrated image signing policies, and cluster management, supporting both Kubernetes and Docker Swarm orchestrators. Furthermore. It allows image promotion policies to be implemented as well as image mirroring, and vulnerability scanning images. Finally, it provides complete support for end users.

Docker works by creating isolated Linux processes using software fences. Currently, Docker is mainly controllable by a Command Line Interface (CLI), but there are a few GUIs that can be used, including the one provided by Docker EE. One of the basic components provided by Docker are images, which are snapshots of the contents of containers. When a developer changes the code, Docker automatically create a new version of the image, with a hash ID. Versioning between development, test and production is quick, seamless and predictable. Docker addresses many frequent software management issues:

- Management of applications: multiple versions of the same software may coexist on the same host machine, (e.g., different Java versions).
- Version control: the images are created using a text file, a Do*ckerfile,* ensuring the container deployment is retrievable and re-buildable.
- Low overhead: compared to virtual machine hypervisors, Docker is lightweight and faster, since containers are small and boot instantly.
- Distributed management: Docker EE provides a GitHub like repository to manage organization of images and deployment of application containers, through the Docker Universal Control Plane and Docker Trusted Registry.
- Containers sharing: Docker provides a cloud service, Docker Hub[20]. for finding and sharing container images both publicly and privately.

CoreOS rkt[21]

In 2018, 12% of production containers run under *rkt*. It supports two types of images: *Docker* and *App Containers* (*appc)*[22]. Rkt is a pod-based process that works out of the box with *Kubernetes*. Its unique features include: support for Trusted Platform Modules (TPM) and optimization for application containers. However, compared with Docker, developers may find fewer third-party integrations. The main advantage of this tool is compatibility, making it a good solution for public cloud portability and rapid deployment. Unfortunately, despite being on the project's roadmap, it, currently, laks of Open Containers Initiative (OCI) compliance. Recently, Red Hat acquired the company behind rkt, CoreOS.

Apache Mesos Containerizer[23]

In 2018, 4% of production containers used Mesos. Developed by Apache, *Mesos* offers quality performance, supporting both *Docker* and *appc* images. OCI support is not yet fully operative, but it is already planned. The

[19] https://sysdig.com/blog/2018-docker-usage-report/
[20] https://hub.docker.com/
[21] https://coreos.com/rkt/
[22] https://coreos.com/rkt/docs/latest/app-container.html
[23] http://mesos.apache.org/documentation/latest/containerizers/

best use of Mesos is combined with frameworks for big data applications, but also other use cases are possible. However, it is not possible to run the containers standalone, the Mesos framework is required to make them run.

LXC Linux Containers[24]
In 2018, 1% of containers run leveraging *LXC Linux Containers*. *LXC* has an active community, since it has been designed also before *Docker*. It is composed of three components:

- *LXC*, the runtime.
- *LXD*, a daemon written in Go that manages containers and images
- *LXFUSE*, which manages the file system.

LXD expands on the low-level LXC tools, offering a new User Interface (UI) and Command Line Interface (CLI) for container management. *LXD* emulates the experience of operating Virtual Machines (VM)s using containers, offering reduced overhead and support for both Windows or MacOS clients. However, it lacks Kubernetes integration and is currently not compliant with OCI.

OpenVZ[25]
It is an open source container-based virtualization extension of the Linux kernel, originally released in 2005. It can run multiple virtual environments and virtual private servers on a single Linux Operating System (OS). The advantage of this tool is that the hosts, sharing a single kernel, requires a lower memory footprint compared to other container runtimes. However, since it focuses on containers for whole operating systems, it is not ideal for single applications. Furthermore, id does not currently provide any Kubernetes integration.

CPSwarm Solution
While other solutions are becoming increasingly popular, Docker is still the de-facto standard for Container Runtime. For this reason, the Consortium has chosen to build the required the Simulation and Optimization Environment containers, using Docker images.

7.2.1.2 Containers deployment and orchestration tools

Docker Swarm
Docker Swarm is a popular open source standard for packaging and distributing containerized applications, providing native clustering for Docker while being fully integrated with the Docker Engine and using a standard API and networking processes. Furthermore, it is built into the Docker CLI and supports a host of tasks through multiple commands that are easy to pick up.

Swarms are a cluster of nodes that consist of the following components:

- Swarm managers, which oversee Control Orchestration, Cluster Management, and Task Distribution.
- Swarm Nodes, which must run containers and services that have been assigned by the Manager Node.

[24] https://linuxcontainers.org/
[25] https://openvz.org/

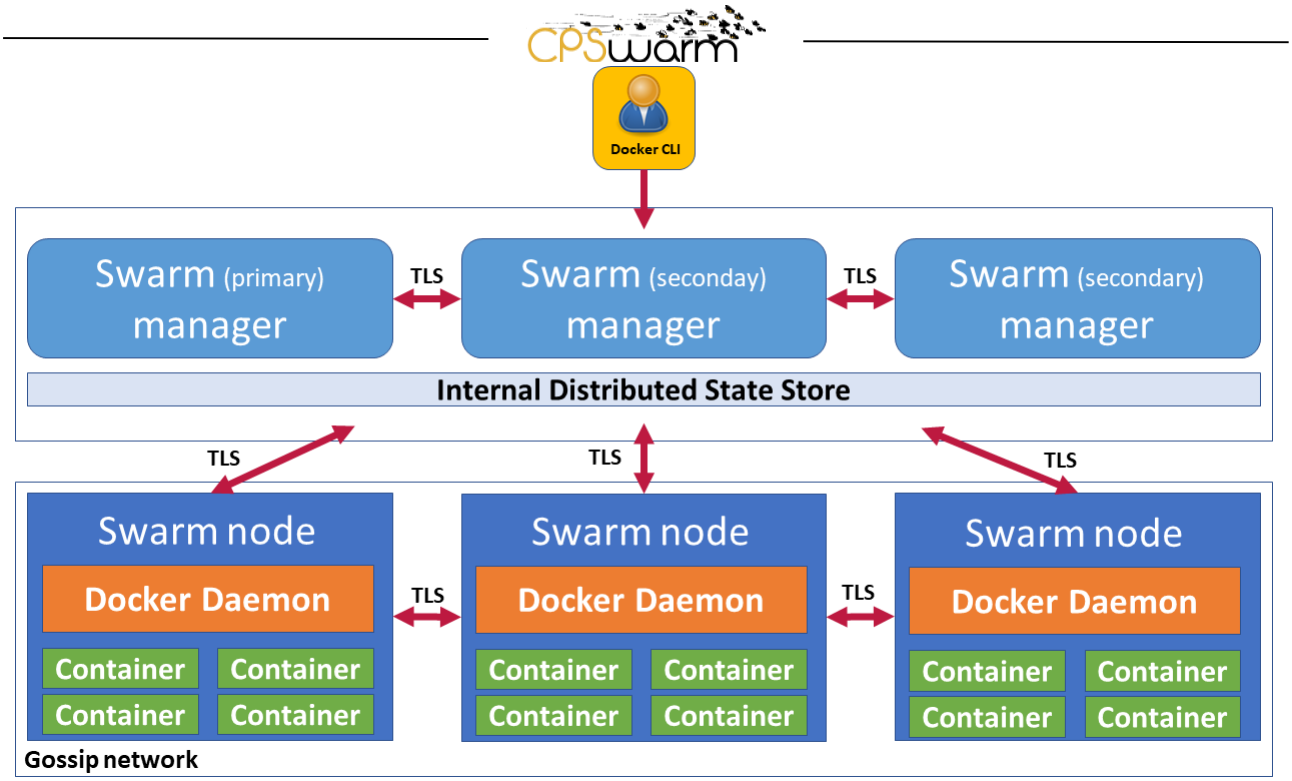

Figure 17 - Docker swarm architecture

Docker Swarm has the following features:

- Storage: For persistent storage, Docker uses the concept of storage volumes. Docker Swarm adds optional plugins that provide access to third-party storage systems.
- Networking: Each Docker Swarm node comes with an autoconfigured overlay network for container traffic. Inter node traffic encrypted using auto-configured TLS encryption with mutual authentication. The user can extend Docker networking capabilities using networking plugins.
- Scheduling: In a Swarm, each task maps to one container. Container placement decisions can be controlled using labels. The scheduler evaluates CPU and memory constraints when scheduling containers. Furthermore, Swarm supports Services, which are continuously monitored by the scheduler. In the event of node failure, the scheduler restarts the affected containers on another host.
- Service Discovery: in Docker Swarm, service discovery is handed by internal Domain Name System (DNS) components that assign a virtual IP and DNS entry to each service in the overlay network. Containers share DNS mappings using an internal gossip network and any container can access any other service by referencing its service DNS name. Docker swarm continuously monitors the health of the containers.
- Load Balancing: in Docker Swarm, when a service is deployed, it is assigned a virtual IP and DNS entry and load balancing is automatically handled by the Docker engine using IP Virtual Server (IPVS). When a service is requested through the DNS, Docker services the request through the IPVS and routes the traffic to all the healthy containers. The user is not required to implement any load balancing but can use an external load balancer to distribute traffic across the nodes.

Kubernetes

In 2014 Google introduced Kubernetes, an Information Technology (IT) management tool specifically designed to simplify the scalability of workloads using containers. The main features it provides are the ability to automate the deployment, scaling, and operation of application containers. Leveraging Kubernetes, it is possible to configure the running modes of the applications and their interactions with other applications. It allows great flexibility and reliability and provides the user with a GUI and composable platform primitives. The basic architecture of a Kubernetes cluster is shown in Figure 18.

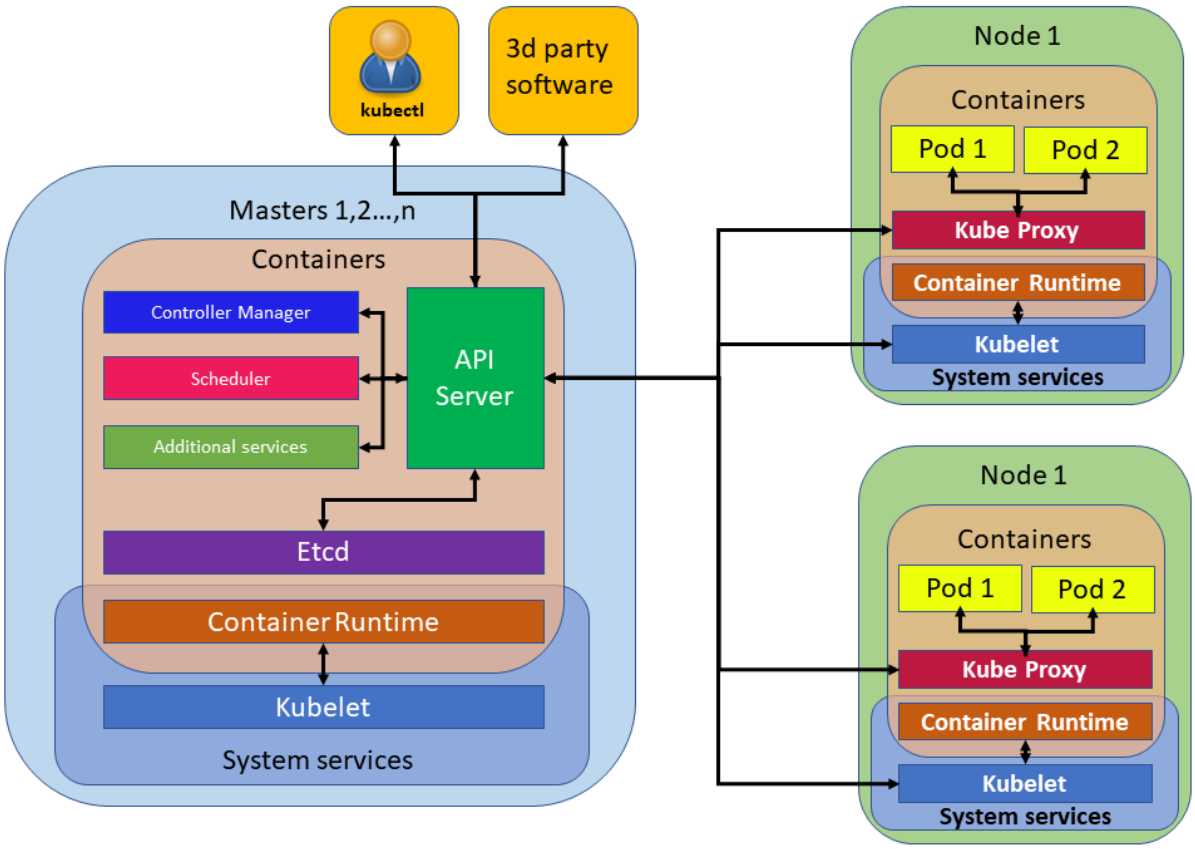

Figure 18 - Architecture of a Kubernetes cluster

A Kubernetes cluster - a collection of hosts that aggregate their available resources including CPU, RAM, disk, and their devices into a usable pool - is composed of two types of nodes (physical or virtual machines):

- *Master nodes:* The primary control plane for Kubernetes, responsible for running the API Server, scheduler, and cluster controller.
- *Worker nodes*: the 'workers' of a Kubernetes cluster that run the containers. They contain all the required services to manage networking among containers, the communication with the master nodes, and the assignment of the resources to the containers.

Specifically, within the *Master nodes:*

- The *API server* provides a Representational State Transfer (REST) interface into the Kubernetes control plane and datastore. All clients, including nodes, users and other applications interact with Kubernetes strictly through the API Server.
- *Etcd* acts as the cluster datastore; providing a strong, consistent and highly available key-value store used for persisting cluster state.
- The *Controller Manager* is the primary daemon that manages all core component control loops. It monitors the cluster state leveraging the *API Server* and orchestrate the cluster to reach the desired state.
- The *Scheduler* is a policy-rich engine, which evaluates workload requirements (i.e. general hardware requirements, affinity, anti-affinity, and other custom resource requirements) and attempts to place it on a matching resource.

Instead, on the other *Worker Nodes*:

- *Kubelet*, acts as the node agent responsible for managing pod lifecycle on its host.
- *Kube-proxy* manages the network rules on each node and performs connection forwarding or load balancing for Kubernetes cluster services.
- A Container Runtime Interface (CRI) compatible application executes and manages containers (e.g., a solution, as described in Section 7.2.1.1).
- One or more Pods, where a Pod is the smallest unit of work or management resource within Kubernetes. It is comprised of one or more containers that share their storage, network, and context.

Kubernetes has the following features:

- Storage: Like Docker Swarm, Kubernetes uses a concept like the Docker volume plugins, to provide persistent storage, with the advantage that Kubernetes allows using multiple volumes per container. Kubernetes supports different volume types just like Docker. One other major difference between Kubernetes and Docker is that the former's volumes are built outside the definition of the container itself, so the volume's lifecycle is independent of that of the container and may outlive it.
- Networking: Both Kubernetes and Docker Swarm uses overlay networks to allow containers on different hosts to communicate with each other. In Docker Swarm such networks are built using a sophisticated "mesh" network. Instead, Kubernetes implements a flat networking model, where there is no native implementation of overlay networks. Indeed, Kubernetes supports Container Network Interface Standard (CNI), a plugin architecture that allows using third party solutions[26], including:
    - Flannel: A simple overlay network that meets basic Kubernetes requirements.
    - Knitter: A network solution supporting multiple networking in Kubernetes.
    - Calico: A secure Layer 3 networking and network policy provider.
    - Contiv: An open-source project.

  Another major difference is that all containers in one single Kubernetes pod share a common IP address, which requires them to coordinate port usage. These addresses are only exposed internally within a Kubernetes cluster by default. To expose them externally, the creation of a specific Kubernetes ingress resource is required.
- Scheduling: Compared with Docker Swarm, Kubernetes has rich scheduling functionality:
    - Replica Sets: This scheduling option ensures that a specified number of pod replicas are running at any given time.
    - Deployments: This scheduling option is used to control replica sets and pods. It works in a declarative way: the user specifies what they want to accomplish and Kubernetes makes it happen. These can be used to create new replica sets, rollback, scale up, pause a deployment and clean up older replica sets.
    - Stateful Sets: This scheduling option is designed to be used with containers that have the requirement to maintain state also after the end of the lifecycle.
    - Daemon Sets: this scheduling option ensures that containers run on all the Kubernetes nodes of the cluster.
    - Cron Jobs: This scheduling option is used to schedule the run of the related containers for a certain time of day.
- Load Balancing: using Docker Swarm, load balancing is implemented within the internal "mesh" network. Rather than providing load balancing for front-end the Docker worker hosts. Kubernetes provides two different options:
    - A "load balancer" resource, which creates a load balancer in Google Cloud Engine (GCE), Amazon Web Service (AWS), Microsoft Azure or any supported cloud provider.
    - A combination of the "Ingress Resource" to configure the Ingress Controller Resource with the rules to be used to provide load balancing to the service that front ends the application pod (Rules –> Ingress –> Service –> Application Pod).

[26] https://kubernetes.io/docs/concepts/cluster-administration/addons/#networking-and-network-policy

- Service Discovery: In Kubernetes, *etcd,* a key value store, is used to store IP to service name mappings. Kubernetes maintains an internal DNS service for internal services. Health checks are handled by a process on each Kubernetes node. A Kubernetes user can also specify a custom health check using exec probe. Kubernetes can integrate with an external DNS service and, in this way, it can create DNS records during service creation.
- Autoscaling: Kubernetes allows auto-scaling based on application demand. This feature is not provided by Docker Swarm, which supports only manual scaling.
- Helm Charts[27]: A package manager used to simplify the complex task of describing containerized applications by providing a set of sharable containerized applications descriptions, written using best practices, ready to be installed in Kubernetes clusters.

CPSwarm solution

For the CPSwarm Simulation and Optimization Environment, the Consortium has chosen to use Kubernetes because, even if it is not as easy to install and get into production as Docker Swarm, it provides a more expansive feature set. In addition, there are more scale-out deployments for Kubernetes and as an orchestration engine, Kubernetes has a wider open source community that can provide support.

To allow the components of the Simulation and Optimization Environment to be completely integrated with the chosen technologies, all of them have been *dockerized*[28] and uploaded to *DockerHub* and a Kubernetes client has been integrated in the SOO for the deployment of SMs.

7.2.2 SOO dockerization and kubernetes integration

To implement the deployment and scalability features, the SOO provides these different operation modes: *deployment, running and deployment & running*, which can be configured with a parameter settable by the user through the Launcher. Then, a deployment file has been defined that specifies the STs to be deployed (see ANNEX B and ANNEX C for document schema and example). The structure of this file reflects the API defined by Kubernetes and it is used by the SOO to do the actual deployment. The most important parts of the file are: the *replicas* field that indicates the number of components that need to be deployed, the *containers* field that indicates the actual container to be deployed and the *nodeSelector* field that provides a key-value pair to identify the nodes of the cluster suitable for that component. For example, *component:system* is used for SOO and FREVO, *component:stage* or *component:gazebo* for the STs.

Although the SOO is launched through the Launcher and not deployed using Kubernetes, it has been dockerized for uniformity (see ANNEX D).

7.2.3 FREVO dockerization

Being a Java application the dockerization of FREVO is quite straightforward (see ANNEX E). The FREVO instance can be launched manually, launched as external process in the local machine through he SOO or deployed in the Kubernetes cluster, using SOO - Kubernetes integration (see ANNEX D).

7.2.4 Stage and Gazebo SMs dockerization

The dockerization of the SMs is a more involved process compared to the others, because its environment is more complex. In general, several images may be used as building blocks of the final container. The containers start from a common image (r*os-kinetic-maven,* see ANNEX F) that contains the ROS kinetic image, together with Java and Maven (used by the SMs). Two images, *gazebo-simulator* (see ANNEX G) and *stage-simulator* (see ANNEX H) may be used to add to this image the two STs already been integrated, to this image, while, *gazebo-simulation-manager* (see ANNEX I) and *stage-simulation-manager* (see ANNEX J) can be used to add

[27] https://github.com/helm/charts

[28] Dockerizing an application is the process of converting an application to run within a Docker container.

the relative SMs. These may be composed, together with custom images, to build an image for a specific scenario. As outlined in Figure 19, for the *emergency_exit* problem, additional images for both the Gazebo and Stage STs have been created for deployment, through the SOO. For Gazebo, *gazebo-em-ex-deps* (see ANNEX K) provides the needed dependencies, while *gazebo-em-ex* (see ANNEX L) provides the actual simulation package. For Stage, *stage-em-ex* (see ANNEX M) has been created.

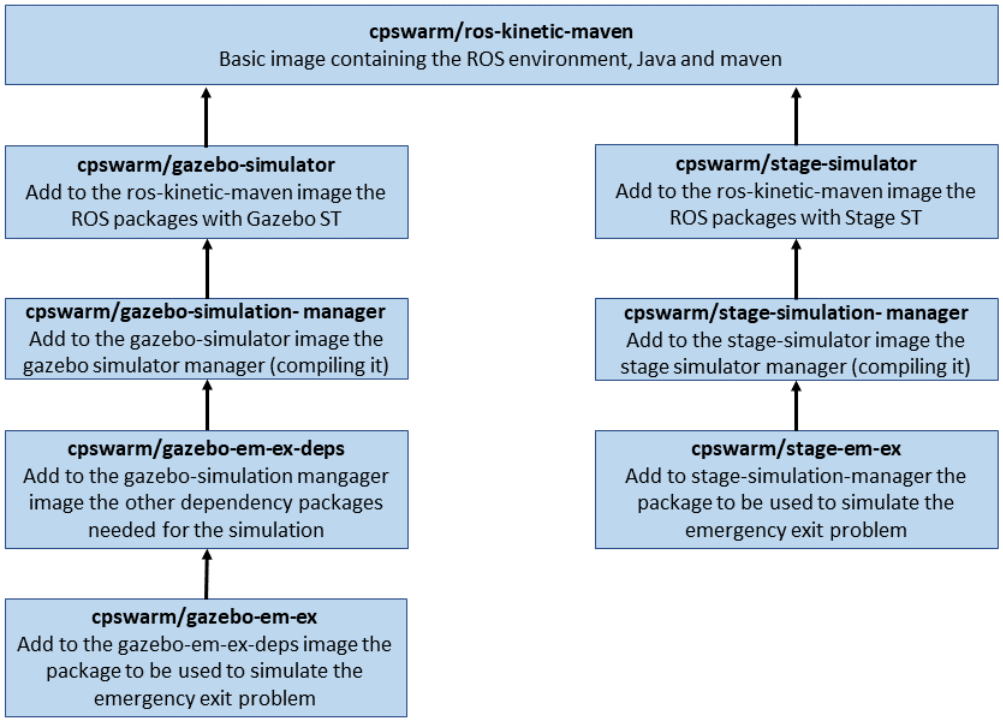

Figure 19 – Hierarchy of Docker images for the Emergency Exit problem.

8 Conclusion

This deliverable describes the process that has led the Consortium to the definition of the final architecture of the Simulation and Optimization Environment. This document presents both the initial version presented in D6.1 and the final version, describing the architectures and interfaces designed, the prototypes built using various tools as well as the performance and scalability of the different versions.

As M28 is the final month for T6.1, this deliverable describes the final version of the architecture. In the last months of the project, the partners will continue testing this architecture and will investigate ways to further improve reliability and scalability of the system. The results of this work will be presented in the upcoming deliverables D6.4 and D6.7.

Acronyms

Acronym	Explanation
FREVO	FRamework for EVOlutionary design
GUI	Graphical User Interface
ANN	Artificial Neural Network
ROS	Robot Operating System
URDF	Unified Robot Description Format
API	Application Programming Interface
MQTT	Message Queue Telemetry Transport
TCP	Transmission Control Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
POJO	Plain Old Java Object
AOE	Algorithm Optimization Environment
SOO	Simulation and Optimization Orchestrator
OT	Optimization Tool
ST	Simulation Tool
SS	Simulation Server
SW	Simulation Wrapper
SM	Simulation Manager
YAML	YAML Ain't Markup Language
CPS	Cyber-Physical System
NEAT	Neuroevolution of Augmenting Topologies
XMPP	eXstensible Messaging and Presence Protocol
CoAP	Constrained Application Protocol
HTTP	HyperText Transfer Protocol
IETF	Internet Engineering Task Force
XSF	XMPP Standard Foundation
SASL	Simple Authentication and Security Layer
TLS	Transport Layer Security
JID	Jabber Identifier
XEP	XMPP Extension
V-REP	Virtual Robot Experimentation Platform
STDR	Simple Two Dimensional Robot simulator

PoC	Proof of Concept
XML	Extensible Markup Language
NNGA	Neural Network Genetic Algorithm
CPU	Central Processing Unit
UUID	Universally Unique Identifier
Appc	Application containers
TPM	Trusted Platform Modules
OCI	Open Containers Initiative
UI	User Interface
CLI	Command Line Interface
VM	Virtual Machine
OS	Operating System
IT	Information Technology
IPVS	IP Virtual Server
CNI	Container Network Interface Standard
GCE	Google Cloud Engine
AWS	Amazon Web Service
REST	Representational State Transfer
DNS	Domain Name System

List of figures

References

[1] 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.

[2] D. C. P. B. R. R. C. P. M. J. P. K. C. K. D. S. E. Ferrera, «XMPP-based infrastructure for IoT network management and rapid services and applications development,» *Annals of Telecommunication Vol. 72,* pp. 443-457, Jul 2017.

[3] C. L. I. E. K. K. J. Kurniawan, «XMPP Performance Analysis using large volume traffic from honeypot sensor,» in *Proceeding of International Conference on Innovation, Enterpreneurship, and Technology ICONIET*, Tangerang City, Banten, Indonesia, November 2015 .

[4] I. F. a. W. E. A. Sobe, «FREVO: A tool for evolving and evaluating self-organizing systems,» in *Proceedings International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, Sep. 2012.

[5] L. C. J. a. R. P. J. A. J. F. Van Rooij, «Neural Network Training Using Genetic Algorithms,» in *World Scientific Publishing Co.,*, Mar. 1997.

ANNEX A – Testbed PC specification

- 12 Intel Xeon X5675 processors running at 3.07 GigaHertz,
- 16 GigaByte of memory.
- Using hyper-threading, it supports 24 threads that can run in parallel.
- Ubuntu 16.04 64 \bit
- OpenJDK 9 Java.

ANNEX B – SOO deployment file schema


```
{
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://example.com/root.json",
  "type": "object",
  "title": "The Root Schema",
  "required": [
    "deployments"
  ],
  "properties": {
    "deployments": {
      "$id": "#/properties/deployments",
      "type": "array",
      "title": "The Deployments Schema",
      "items": {
        "$id": "#/properties/deployments/items",
        "type": "object",
        "title": "The Items Schema",
        "required": [
          "metadata",
          "spec",
          "template"
        ],
        "properties": {
          "metadata": {
            "$id": "#/properties/deployments/items/properties/metadata",
            "type": "object",
            "title": "The Metadata Schema",
            "required": [
              "name",
              "namespace",
              "generation",
              "labels"
            ],
            "properties": {
              "name": {
                "$id": "#/properties/deployments/items/properties/metadata/properties/name",
                "type": "string",
                "title": "The Name Schema",
                "default": "",
                "examples": [
                  "stage"
                ],
                "pattern": "^(.*)$"
              },
              "namespace": {
                "$id": "#/properties/deployments/items/properties/metadata/properties/namespace",
                "type": "string",
                "title": "The Namespace Schema",
                "default": "",
```

```
        "examples": [
          "default"
        ],
        "pattern": "^(.*)$"
      },
      "generation": {
        "$id": "#/properties/deployments/items/properties/metadata/properties/generation",
        "type": "integer",
        "title": "The Generation Schema",
        "default": 0,
        "examples": [
          1
        ]
      },
      "labels": {
        "$id": "#/properties/deployments/items/properties/metadata/properties/labels",
        "type": "object",
        "title": "The Labels Schema",
        "required": [
          "k8s-app"
        ],
        "properties": {
          "k8s-app": {
    "$id": "#/properties/deployments/items/properties/metadata/properties/labels/properties/k8s-app",
            "type": "string",
            "title": "The K8s-app Schema",
            "default": "",
            "examples": [
              "stage"
            ],
            "pattern": "^(.*)$"
          }
        }
      }
    }
  },
  "spec": {
    "$id": "#/properties/deployments/items/properties/spec",
    "type": "object",
    "title": "The Spec Schema",
    "required": [
      "replicas",
      "selector"
    ],
    "properties": {
      "replicas": {
        "$id": "#/properties/deployments/items/properties/spec/properties/replicas",
        "type": "integer",
        "title": "The Replicas Schema",
        "default": 0,
        "examples": [
          1
```

```
            ]
           },
           "selector": {
            "$id": "#/properties/deployments/items/properties/spec/properties/selector",
            "type": "object",
            "title": "The Selector Schema",
            "required": [
             "matchLabels"
            ],
            "properties": {
             "matchLabels": {
              "$id":
"#/properties/deployments/items/properties/spec/properties/selector/properties/matchLabels",
              "type": "object",
              "title": "The Matchlabels Schema",
              "required": [
               "k8s-app"
              ],
              "properties": {
               "k8s-app": {
                "$id":
"#/properties/deployments/items/properties/spec/properties/selector/properties/matchLabels/properties/k8
s-app",
                "type": "string",
                "title": "The K8s-app Schema",
                "default": "",
                "examples": [
                 "stage"
                ],
                "pattern": "^(.*)$"
               }
              }
             }
            }
           }
          }
         },
         "template": {
          "$id": "#/properties/deployments/items/properties/template",
          "type": "object",
          "title": "The Template Schema",
          "required": [
           "metadata",
           "spec"
          ],
          "properties": {
           "metadata": {
            "$id": "#/properties/deployments/items/properties/template/properties/metadata",
            "type": "object",
            "title": "The Metadata Schema",
            "required": [
             "name",
```

```
            "labels"
           ],
           "properties": {
            "name": {
             "$id":
"#/properties/deployments/items/properties/template/properties/metadata/properties/name",
             "type": "string",
             "title": "The Name Schema",
             "default": "",
             "examples": [
              "stage"
             ],
             "pattern": "^(.*)$"
            },
            "labels": {
             "$id":
"#/properties/deployments/items/properties/template/properties/metadata/properties/labels",
             "type": "object",
             "title": "The Labels Schema",
             "required": [
              "k8s-app"
             ],
             "properties": {
              "k8s-app": {
               "$id":
"#/properties/deployments/items/properties/template/properties/metadata/properties/labels/properties/k8s
-app",
               "type": "string",
               "title": "The K8s-app Schema",
               "default": "",
               "examples": [
                "stage"
               ],
               "pattern": "^(.*)$"
              }
             }
            }
           }
          },
          "spec": {
           "$id": "#/properties/deployments/items/properties/template/properties/spec",
           "type": "object",
           "title": "The Spec Schema",
           "required": [
            "containers",
            "nodeSelector"
           ],
           "properties": {
            "containers": {
             "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers",
             "type": "array",
```

```
                "title": "The Containers Schema",
                "items": {
                 "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items",
                 "type": "object",
                 "title": "The Items Schema",
                 "required": [
                  "name",
                  "image",
                  "args"
                 ],
                 "properties": {
                  "name": {
                   "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items/properti
es/name",
                   "type": "string",
                   "title": "The Name Schema",
                   "default": "",
                   "examples": [
                    "stage"
                   ],
                   "pattern": "^(.*)$"
                  },
                  "image": {
                   "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items/properti
es/image",
                   "type": "string",
                   "title": "The Image Schema",
                   "default": "",
                   "examples": [
                    "cpswarm/stage-em-ex:1.0.9"
                   ],
                   "pattern": "^(.*)$"
                  },
                  "args": {
                   "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items/properti
es/args",
                   "type": "array",
                   "title": "The Args Schema",
                   "items": {
                    "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items/properti
es/args/items",
                    "type": "string",
                    "title": "The Items Schema",
                    "default": "",
                    "examples": [
                     "-n",
                     "pippo",
```

```
                    "-s",
                    "pluto"
                   ],
                   "pattern": "^(.*)$"
                  }
                 }
                }
               }
              },
              "nodeSelector": {
               "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/nodeSelector",
               "type": "object",
               "title": "The Nodeselector Schema",
               "required": [
                "component"
               ],
               "properties": {
                "component": {
                 "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/nodeSelector/properties/component",
                  "type": "string",
                  "title": "The Component Schema",
                  "default": "",
                  "examples": [
                   "stage"
                  ],
                  "pattern": "^(.*)$"
                 }
                }
               }
              }
             }
            }
           }
          }
         }
        }
       }
      }
     }
    }
   }
  }
}
```

ANNEX C – SOO Deployment file example


```
{
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://example.com/root.json",
  "type": "object",
  "title": "The Root Schema",
  "required": [
    "deployments"
  ],
  "properties": {
    "deployments": {
      "$id": "#/properties/deployments",
      "type": "array",
      "title": "The Deployments Schema",
      "items": {
        "$id": "#/properties/deployments/items",
        "type": "object",
        "title": "The Items Schema",
        "required": [
          "metadata",
          "spec",
          "template"
        ],
        "properties": {
          "metadata": {
            "$id": "#/properties/deployments/items/properties/metadata",
            "type": "object",
            "title": "The Metadata Schema",
            "required": [
              "name",
              "namespace",
              "generation",
              "labels"
            ],
            "properties": {
              "name": {
                "$id": "#/properties/deployments/items/properties/metadata/properties/name",
                "type": "string",
                "title": "The Name Schema",
                "default": "",
                "examples": [
                  "frevo"
                ],
                "pattern": "^(.*)$"
              },
              "namespace": {
                "$id": "#/properties/deployments/items/properties/metadata/properties/namespace",
                "type": "string",
                "title": "The Namespace Schema",
                "default": "",
```

```
            "examples": [
              "default"
            ],
            "pattern": "^(.*)$"
          },
          "generation": {
            "$id": "#/properties/deployments/items/properties/metadata/properties/generation",
            "type": "integer",
            "title": "The Generation Schema",
            "default": 0,
            "examples": [
              1
            ]
          },
          "labels": {
            "$id": "#/properties/deployments/items/properties/metadata/properties/labels",
            "type": "object",
            "title": "The Labels Schema",
            "required": [
              "k8s-app"
            ],
            "properties": {
              "k8s-app": {
                "$id":    "#/properties/deployments/items/properties/metadata/properties/labels/properties/k8s-
app",
                "type": "string",
                "title": "The K8s-app Schema",
                "default": "",
                "examples": [
                  "frevo"
                ],
                "pattern": "^(.*)$"
              }
            }
          }
        }
      },
      "spec": {
        "$id": "#/properties/deployments/items/properties/spec",
        "type": "object",
        "title": "The Spec Schema",
        "required": [
          "replicas",
          "selector"
        ],
        "properties": {
          "replicas": {
            "$id": "#/properties/deployments/items/properties/spec/properties/replicas",
            "type": "integer",
            "title": "The Replicas Schema",
            "default": 0,
            "examples": [
```

```
            1
           ]
          },
          "selector": {
           "$id": "#/properties/deployments/items/properties/spec/properties/selector",
           "type": "object",
           "title": "The Selector Schema",
           "required": [
            "matchLabels"
           ],
           "properties": {
            "matchLabels": {
             "$id":
"#/properties/deployments/items/properties/spec/properties/selector/properties/matchLabels",
             "type": "object",
             "title": "The Matchlabels Schema",
             "required": [
              "k8s-app"
             ],
             "properties": {
              "k8s-app": {
               "$id":
"#/properties/deployments/items/properties/spec/properties/selector/properties/matchLabels/properties/k8
s-app",
               "type": "string",
               "title": "The K8s-app Schema",
               "default": "",
               "examples": [
                "frevo"
               ],
               "pattern": "^(.*)$"
              }
             }
            }
           }
          }
         },
         "template": {
          "$id": "#/properties/deployments/items/properties/template",
          "type": "object",
          "title": "The Template Schema",
          "required": [
           "metadata",
           "spec"
          ],
          "properties": {
           "metadata": {
            "$id": "#/properties/deployments/items/properties/template/properties/metadata",
            "type": "object",
            "title": "The Metadata Schema",
            "required": [
```

```
            "name",
            "labels"
           ],
           "properties": {
            "name": {
             "$id":
"#/properties/deployments/items/properties/template/properties/metadata/properties/name",
             "type": "string",
             "title": "The Name Schema",
             "default": "",
             "examples": [
              "frevo"
             ],
             "pattern": "^(.*)$"
            },
            "labels": {
             "$id":
"#/properties/deployments/items/properties/template/properties/metadata/properties/labels",
             "type": "object",
             "title": "The Labels Schema",
             "required": [
              "k8s-app"
             ],
             "properties": {
              "k8s-app": {
               "$id":
"#/properties/deployments/items/properties/template/properties/metadata/properties/labels/properties/k8s
-app",
               "type": "string",
               "title": "The K8s-app Schema",
               "default": "",
               "examples": [
                "frevo"
               ],
               "pattern": "^(.*)$"
              }
             }
            }
           }
          },
          "spec": {
           "$id": "#/properties/deployments/items/properties/template/properties/spec",
           "type": "object",
           "title": "The Spec Schema",
           "required": [
            "containers",
            "nodeSelector"
           ],
           "properties": {
            "containers": {
             "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers",
```

```
                "type": "array",
                "title": "The Containers Schema",
                "items": {
                 "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items",
                 "type": "object",
                 "title": "The Items Schema",
                 "required": [
                  "name",
                  "image",
                  "args",
                  "stdin"
                 ],
                 "properties": {
                  "name": {
                   "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items/properti
es/name",
                   "type": "string",
                   "title": "The Name Schema",
                   "default": "",
                   "examples": [
                    "frevo"
                   ],
                   "pattern": "^(.*)$"
                  },
                  "image": {
                   "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items/properti
es/image",
                   "type": "string",
                   "title": "The Image Schema",
                   "default": "",
                   "examples": [
                    "cpswarm/frevo-docker:1.0.3"
                   ],
                   "pattern": "^(.*)$"
                  },
                  "args": {
                   "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items/properti
es/args",
                   "type": "array",
                   "title": "The Args Schema",
                   "items": {
                    "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items/properti
es/args/items",
                    "type": "string",
                    "title": "The Items Schema",
                    "default": "",
                    "examples": [
```

```
                    "-n",
                    "pert-demoenergy-virtus.ismb.polito.it",
                    "-ip",
                    "130.192.86.237",
                    "-p",
                    "5222",
                    "-r",
                    "cpswarm",
                    "-cid",
                    "frevo",
                    "-cp",
                    "blah",
                    "-c",
                    "/home/"
                  ],
                  "pattern": "^(.*)$"
                }
              },
              "stdin": {
                "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/containers/items/properti
es/stdin",
                "type": "string",
                "title": "The Stdin Schema",
                "default": "",
                "examples": [
                  "false"
                ],
                "pattern": "^(.*)$"
              }
            }
          },
          "nodeSelector": {
            "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/nodeSelector",
            "type": "object",
            "title": "The Nodeselector Schema",
            "required": [
              "component"
            ],
            "properties": {
              "component": {
                "$id":
"#/properties/deployments/items/properties/template/properties/spec/properties/nodeSelector/properties/c
omponent",
                "type": "string",
                "title": "The Component Schema",
                "default": "",
                "examples": [
                  "system"
                ],
```

```
                "pattern": "^(.*)$"
              }
            }
          }
        }
      }
    }
  }
 }
}
```

ANNEX D – SOO Dockerfile

```
FROM maven:3-jdk-8

COPY . /home

WORKDIR /home

RUN mkdir Desktop

RUN mkdir Desktop/cpswarm

RUN mkdir Desktop/optimized

RUN mkdir Desktop/conf

RUN mvn -B validate

RUN mvn install -DskipTests

ENV JAVA_HOME /usr/lib/jvm/java-1.8.0-openjdk-amd64/
```

ANNEX E – FREVO Dockerfile

```
FROM openjdk:10.0.2-jdk-slim

COPY . /home/

WORKDIR /home/

RUN keytool -noprompt -importcert -trustcacerts \

    -file pert-demoenergy-virtus.ismb.polito.it.pem -alias pert-demoenergy-virtus.ismb.polito.it \

    -storepass changeit -keystore -J-Duser.language=en $JAVA_HOME/lib/security/cacerts

ENTRYPOINT ["java", "-jar", "frevo.xmpp-0.0.1-SNAPSHOT-jar-with-dependencies.jar"]

CMD ["-n", "pippo.pluto.it", "-ip", "123.123.123.123", "-p", "5222", "-r", "cpswarm", "-cid", "test", "-cp", "1234",
"-c", "/home/"]
```

ANNEX F – ros-kinetic-maven Dockerfile

```
FROM ros:kinetic-ros-base

RUN sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

RUN apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116

RUN apt-get update

RUN apt-get install -y ros-kinetic-navigation

RUN apt-get install -y python-catkin-tools

RUN apt-get install -y openjdk-8-jdk maven
```

ANNEX G – gazebo-simulator Dockerfile

```
FROM cpswarm/ros-kinetic-maven:1.0.1

RUN apt-get install -y ros-kinetic-gazebo-ros-pkgs
```

ANNEX H – stage-simulator Dockerfile

```
FROM cpswarm/ros-kinetic-maven:1.0.1

RUN apt-get install -y ros-kinetic-stage-ros
```

ANNEX I – gazebo-simulation-manager Dockerfile

```
FROM cpswarm/gazebo-simulator:1.0.1

COPY . /home/

WORKDIR /home/
```

ANNEX J – stage-simulation-manager Dockerfile

```
FROM cpswarm/stage-simulator:1.0.1

COPY . /home/

WORKDIR /home/

RUN apt-get install psmisc
```

ANNEX K – gazebo-em-ex-deps Dockerfile

```
FROM cpswarm/gazebo-simulation-manager:1.0.6

RUN apt-get update

RUN apt-get install -y apt-utils

RUN apt-get install -y --fix-missing libgeographic-dev geographiclib-tools libopencv-dev libeigen3-dev udev

RUN apt-get install -y ros-kinetic-ecl-build ros-kinetic-kobuki-msgs ros-kinetic-kdl-conversions ros-kinetic-ecl-exceptions ros-kinetic-rqt-robot-dashboard ros-kinetic-ecl-threads ros-kinetic-yocs-controllers ros-kinetic-ecl-geometry ros-kinetic-xacro ros-kinetic-kobuki-dock-drive ros-kinetic-geographic-msgs ros-kinetic-kobuki-driver ros-kinetic-ecl-streams ros-kinetic-eigen-conversions ros-kinetic-image-geometry ros-kinetic-rqt-common-plugins ros-kinetic-openni2-launch ros-kinetic-openni2-launch ros-kinetic-openni2-launch ros-kinetic-create-description ros-kinetic-diagnostic-aggregator ros-kinetic-robot-state-publisher ros-kinetic-control-toolbox ros-kinetic-warehouse-ros ros-kinetic-yocs-velocity-smoother ros-kinetic-rocon-apps ros-kinetic-depth-image-proc ros-kinetic-rviz ros-kinetic-joy ros-kinetic-python-orocos-kdl ros-kinetic-depthimage-to-laserscan ros-kinetic-std-capabilities ros-kinetic-world-canvas-server ros-kinetic-rocon-bubble-icons python-lxml ros-kinetic-robot-pose-publisher ros-kinetic-stage-ros ros-kinetic-compressed-image-transport ros-kinetic-stdr-resources ros-kinetic-stdr-gui ros-kinetic-rplidar-ros ros-kinetic-rocon-app-manager ros-kinetic-stdr-robot pyqt5-dev-tools ros-kinetic-create-node ros-kinetic-kobuki-ftdi ros-kinetic-zeroconf-avahi ros-kinetic-stdr-server ros-kinetic-freenect-launch ros-kinetic-gmapping ros-kinetic-yocs-md-vel-mux ros-kinetic-yocs-virtual-sensor ros-kinetic-laptop-battery-monitor python-future ros-kinetic-astra-launch ros-kinetic-ros-control ros-kinetic-controller-manager ros-kinetic-position-controllers ros-kinetic-joint-state-controller ros-kinetic-actionlib-tutorials ros-kinetic-camera-calibration ros-kinetic-costmap-converter ros-kinetic-create-dashboard ros-kinetic-diagnostic-analysis ros-kinetic-diagnostic-common-diagnostics ros-kinetic-diff-drive-controller ros-kinetic-effort-controllers ros-kinetic-eigen-stl-containers ros-kinetic-filters ros-kinetic-forward-command-controller ros-kinetic-frontier-exploration ros-kinetic-geometric-shapes ros-kinetic-gl-dependency ros-kinetic-hector-gazebo-plugins ros-kinetic-husky-base ros-kinetic-husky-bringup ros-kinetic-husky-control ros-kinetic-husky-description ros-kinetic-husky-gazebo ros-kinetic-husky-msgs ros-kinetic-husky-navigation ros-kinetic-image-publisher ros-kinetic-image-rotate ros-kinetic-image-view ros-kinetic-imu-filter-madgwick ros-kinetic-imu-transformer ros-kinetic-interactive-marker-tutorials ros-kinetic-interactive-marker-twist-server ros-kinetic-joint-state-controller ros-kinetic-joint-state-publisher ros-kinetic-joint-trajectory-controller ros-kinetic-laser-assembler ros-kinetic-laser-filters ros-kinetic-libg2o ros-kinetic-libmavconn ros-kinetic-librviz-tutorial ros-kinetic-lms1xx ros-kinetic-master-discovery-fkie ros-kinetic-master-sync-fkie ros-kinetic-multimaster-launch ros-kinetic-multimaster-msgs ros-kinetic-multimaster-msgs-fkie ros-kinetic-nmea-comms ros-kinetic-nmea-msgs ros-kinetic-nmea-navsat-driver ros-kinetic-nodelet-tutorial-math ros-kinetic-octomap ros-kinetic-pluginlib-tutorials ros-kinetic-pointcloud-to-laserscan ros-kinetic-random-numbers ros-kinetic-robot-localization ros-kinetic-robot-upstart ros-kinetic-roscpp-tutorials ros-kinetic-roslint ros-kinetic-rqt-moveit ros-kinetic-rqt-pose-view ros-kinetic-rqt-robot-steering ros-kinetic-rqt-runtime-monitor ros-kinetic-rqt-rviz ros-kinetic-rqt-tf-tree ros-kinetic-rviz-imu-plugin ros-kinetic-rviz-plugin-tutorials ros-kinetic-rviz-python-tutorial ros-kinetic-self-test ros-kinetic-serial ros-kinetic-smach ros-kinetic-smach-msgs ros-kinetic-smach-ros ros-kinetic-stereo-image-proc ros-kinetic-teb-local-planner ros-kinetic-teleop-twist-joy ros-kinetic-tf2-geometry-msgs ros-kinetic-tf2-relay ros-kinetic-tf2-sensor-msgs ros-kinetic-tf-conversions ros-kinetic-theora-image-transport ros-kinetic-turtle-actionlib ros-kinetic-turtle-tf ros-kinetic-turtle-tf2 ros-kinetic-turtlesim ros-kinetic-twist-mux ros-kinetic-twist-mux-msgs ros-kinetic-um6 ros-kinetic-um7 ros-kinetic-urdf-parser-plugin ros-kinetic-urdf-tutorial ros-kinetic-visualization-marker-tutorials ros-kinetic-turtlebot-msgs ros-kinetic-librealsense
```

ANNEX L – gazebo-em-ex Dockerfile

```
FROM cpswarm/gazebo-em-ex-deps:1.0.4

COPY . /home/

RUN mkdir cpswarm

WORKDIR /home/ws/

RUN catkin init --workspace .

RUN /bin/bash ros.sh

WORKDIR /home/

RUN mvn -B validate

RUN mvn install -DskipTests

ENV JAVA_HOME /usr/lib/jvm/java-1.8.0-openjdk-amd64/

RUN keytool -noprompt -importcert -trustcacerts \

   -file pert-demoenergy-virtus.ismb.polito.it.pem -alias pert-demoenergy-virtus.ismb.polito.it \

   -storepass changeit -keystore -J-Duser.language=en $JAVA_HOME/jre/lib/security/cacerts

CMD java -jar /home/target/it.ismb.pert.cpswarm.simulation.gazebo-1.1.0-jar-with-dependencies.jar
```

ANNEX M – stage-em-ex Dockerfile

```
FROM cpswarm/stage-simulation-manager:1.0.45

RUN mkdir cpswarm

RUN mkdir ws

RUN mkdir ws/src

RUN mkdir ws/src/emergency_exit

COPY emergency_exit /home/ws/src/emergency_exit

COPY ros.sh /home/ws/

COPY pert-demoenergy-virtus.ismb.polito.it.pem /home/

WORKDIR /home/ws/

RUN catkin init --workspace .

RUN /bin/bash -c "source /opt/ros/kinetic/setup.bash"

RUN /bin/bash ros.sh

WORKDIR /home/

RUN mvn -B validate

RUN mvn install -DskipTests

ENV JAVA_HOME /usr/lib/jvm/java-1.8.0-openjdk-amd64/

RUN keytool -noprompt -importcert -trustcacerts \

   -file pert-demoenergy-virtus.ismb.polito.it.pem -alias pert-demoenergy-virtus.ismb.polito.it \

   -storepass changeit -keystore -J-Duser.language=en $JAVA_HOME/jre/lib/security/cacerts

CMD java -jar /home/target/it.ismb.pert.cpswarm.simulation.stage-1.1.0-jar-with-dependencies.jar
```

ANNEX N – SOO configuration file

```
<?xml encoding="UTF-8"?>
<!ELEMENT settings (serverURI,serverName,username,serverPassword,
                    optimizationUser,monitoring,configEnabled,
                    startingTimeout,mqttBroker,localOptimization,
                    optimizationToolPath,optimizationToolPassword)>
<!ATTLIST settings
 xmlns CDATA #FIXED ''>
<!ELEMENT serverURI (#PCDATA)>
<!ATTLIST serverURI
 xmlns CDATA #FIXED ''>
<!ELEMENT serverName (#PCDATA)>
<!ATTLIST serverName
 xmlns CDATA #FIXED ''>
<!ELEMENT username (#PCDATA)>
<!ATTLIST username
 xmlns CDATA #FIXED ''>
<!ELEMENT serverPassword (#PCDATA)>
<!ATTLIST serverPassword
 xmlns CDATA #FIXED ''>
<!ELEMENT optimizationUser (#PCDATA)>
<!ATTLIST optimizationUser
 xmlns CDATA #FIXED ''>
<!ELEMENT monitoring (#PCDATA)>
<!ATTLIST monitoring
 xmlns CDATA #FIXED ''>
<!ELEMENT configEnabled (#PCDATA)>
<!ATTLIST configEnabled
 xmlns CDATA #FIXED ''>
<!ELEMENT startingTimeout (#PCDATA)>
<!ATTLIST startingTimeout
 xmlns CDATA #FIXED ''>
<!ELEMENT mqttBroker (#PCDATA)>
<!ATTLIST mqttBroker
 xmlns CDATA #FIXED ''>
<!ELEMENT localOptimization (#PCDATA)>
<!ATTLIST localOptimization
 xmlns CDATA #FIXED ''>
<!ELEMENT optimizationToolPath (#PCDATA)>
<!ATTLIST optimizationToolPath
 xmlns CDATA #FIXED ''>
<!ELEMENT optimizationToolPassword (#PCDATA)>
<!ATTLIST optimizationToolPassword
 xmlns CDATA #FIXED ''>
```

ANNEX O – Stage SM configuration file

```
<?xml encoding="UTF-8"?>
<!ELEMENT settings (serverURI, serverName, serverPassword, dataFolder, dimensions, maxAgents,
        optimizationUser, orchestratorUser, rosFolder, monitoring, mqttBroker, timeout, fake)>
<!ATTLIST settings
 xmlns CDATA #FIXED ''>
<!ELEMENT serverURI (#PCDATA)>
<!ATTLIST serverURI
 xmlns CDATA #FIXED ''>
<!ELEMENT serverName (#PCDATA)>
<!ATTLIST serverName
 xmlns CDATA #FIXED ''>
<!ELEMENT serverPassword (#PCDATA)>
<!ATTLIST serverPassword
 xmlns CDATA #FIXED ''>
<!ELEMENT dataFolder (#PCDATA)>
<!ATTLIST dataFolder
 xmlns CDATA #FIXED ''>
<!ELEMENT dimensions (#PCDATA)>
<!ATTLIST dimensions
 xmlns CDATA #FIXED ''>
<!ELEMENT maxAgents (#PCDATA)>
<!ATTLIST maxAgents
 xmlns CDATA #FIXED ''>
<!ELEMENT optimizationUser (#PCDATA)>
<!ATTLIST optimizationUser
 xmlns CDATA #FIXED ''>
<!ELEMENT orchestratorUser (#PCDATA)>
<!ATTLIST orchestratorUser
 xmlns CDATA #FIXED ''>
<!ELEMENT rosFolder (#PCDATA)>
<!ATTLIST rosFolder
 xmlns CDATA #FIXED ''>
<!ELEMENT monitoring (#PCDATA)>
<!ATTLIST monitoring
 xmlns CDATA #FIXED ''>
<!ELEMENT mqttBroker (#PCDATA)>
<!ATTLIST mqttBroker
 xmlns CDATA #FIXED ''>
<!ELEMENT timeout (#PCDATA)>
<!ATTLIST timeout
 xmlns CDATA #FIXED ''>
<!ELEMENT fake (#PCDATA)>
<!ATTLIST fake
 xmlns CDATA #FIXED ''>
```

ANNEX P – Gazebo SM configuration file

```
<?xml encoding="UTF-8"?>
<!ELEMENT settings (serverURI,serverName,serverPassword,dataFolder,
                    dimensions,maxAgents,optimizationUser,
                    orchestratorUser,rosFolder,monitoring,mqttBroker)>
<!ATTLIST settings
 xmlns CDATA #FIXED ''>
<!ELEMENT serverURI (#PCDATA)>
<!ATTLIST serverURI
 xmlns CDATA #FIXED ''>
<!ELEMENT serverName (#PCDATA)>
<!ATTLIST serverName
 xmlns CDATA #FIXED ''>
<!ELEMENT serverPassword (#PCDATA)>
<!ATTLIST serverPassword
 xmlns CDATA #FIXED ''>
<!ELEMENT dataFolder (#PCDATA)>
<!ATTLIST dataFolder
 xmlns CDATA #FIXED ''>
<!ELEMENT dimensions (#PCDATA)>
<!ATTLIST dimensions
 xmlns CDATA #FIXED ''>
<!ELEMENT maxAgents (#PCDATA)>
<!ATTLIST maxAgents
 xmlns CDATA #FIXED ''>
<!ELEMENT optimizationUser (#PCDATA)>
<!ATTLIST optimizationUser
 xmlns CDATA #FIXED ''>
<!ELEMENT orchestratorUser (#PCDATA)>
<!ATTLIST orchestratorUser
 xmlns CDATA #FIXED ''>
<!ELEMENT rosFolder (#PCDATA)>
<!ATTLIST rosFolder
 xmlns CDATA #FIXED ''>
<!ELEMENT monitoring (#PCDATA)>
<!ATTLIST monitoring
 xmlns CDATA #FIXED ''>
<!ELEMENT mqttBroker (#PCDATA)>
<!ATTLIST mqttBroker
 xmlns CDATA #FIXED ''>
```