# Scalable Distributed Simulation for Evolutionary Optimization of Swarms of Cyber-Physical Systems

Micha Sende,
Melanie Schranz

Lakeside Labs GmbH
Klagenfurt, Austria
lastname@lakeside-labs.com

Davide Conzon,
Enrico Ferrera,
Claudio Pastrone

Pervasive Technologies
LINKS Foundation
Turin, Italy
firstname.lastname
@linksfoundation.com

Arthur Pitman,
Midhat Jdeed,
Wilfried Elmenreich

Institute of Networked and Embedded Systems
University of Klagenfurt
Klagenfurt, Austria
firstname.lastname@aau.at

***Abstract*—Swarms of cyber-physical systems can be used to tackle many challenges that traditional multi-robot systems fail to address. In particular, the self-organizing nature of swarms ensures they are both scalable and adaptable. Such benefits come at the cost of having a highly complex system that is extremely hard to design manually. Therefore, an automated process is required for designing the local interactions between the cyber-physical systems that lead to the desired swarm behavior. In this work, the authors employ evolutionary design methodologies to generate the local controllers of the cyber-physical systems. This requires many simulation runs, which can be parallelized. Two approaches are proposed for distributing simulations among multiple servers. First, an approach where the distributed simulators are controlled centrally and second, a distributed approach where the controllers are exported to the simulators running standalone. The authors show that the distributed approach is suited for most scenarios and propose a network-based architecture. To evaluate the performance, the authors provide an implementation that builds upon the eXtensible Messaging and Presence Protocol (XMPP) and supersedes a previous implementation based on the Message Queue Telemetry Transport (MQTT) protocol. Measurements of the total optimization time show that it outperforms the previous implementation in certain cases by a factor greater than three. A scalability analysis shows that it is inversely proportional to the number of simulation servers and scales very well. Finally, a proof of concept demonstrates the ability to deploy the resulting controller onto cyber-physical systems. The results demonstrate the flexibility of the architecture and its performance. Therefore, it is well suited for distributing the simulation workload among multiple servers.**

***Keywords–Cyber-Physical System (CPS); Swarm; Evolutionary optimization; Distributed simulation; eXtensible Messaging and Presence Protocol (XMPP).***

## I. Introduction

Over the last decade, the paradigm of self-organization has gained significant traction in many research communities. Inspired by nature, swarm robotics is also seeing increased interest. This concept can be generalized from swarm robotics to swarms of Cyber-Physical Systems (CPSs) [1]. Applying self-organization to coordinate swarms of CPSs is a rather new approach, which aims at handling the highly complex systems, currently available. On the one hand, coordinating multiple CPSs using swarm approaches offers many opportunities, such as adaptability, scalability, and robustness [2]. On the other hand, it necessitates the difficult process of designing the individual CPSs to achieve the desired swarm behavior.

Designing swarms of CPSs poses two main challenges. First, selecting the hardware that best suits the requirements of the swarm (see [3], [4], [5], [6] for a further examination of this problem) and second, designing the control algorithm defining the behavior of the individual CPSs. This paper focuses on the latter problem because many platforms for swarm robotic research already exist, e.g., Kilobot [7], Spiderino [8], or Colias [9] and designing the hardware itself is out of scope of this work. Other platforms developed for traditional robotic applications such as the TurtleBot [10] are also suitable for executing swarm algorithms. Approaches for designing local controllers of individual CPSs in a swarm can be categorized into two types. First, hierarchical top-down design starting from the desired global behavior of the swarm and second, bottom-up design defining the individual CPS behaviors and observing the resulting global behavior [11]. Design using either one of the mentioned approaches is still a difficult process as neither can predict the resulting swarm behavior based on the complex interactions between the CPSs [12]. This is especially true in dynamic environments. One method to tackle such design challenges is evolutionary optimization [13].

In this paper, the authors employ the bottom-up design process based on evolutionary algorithms. Generally, evolutionary algorithms aim to mimic the process of natural selection by recombining the most successful solutions to a defined problem [14]. In the context of swarms of CPSs, a solution refers to a control algorithm of the individual CPSs that is gradually improved during the optimization process. As experiments with real CPSs require an extensive amount of time, such methods typically employ accurate and fast simulations to evaluate the performance of candidate solutions in the evolutionary process [15]. Using state-of-the-art simulators allows to build upon standard models and perform optimizations with varying level of detail. The evaluation of control algorithms in evolutionary optimization can be executed in parallel, which is for example supported in the FRamework for EVOlutionary design (FREVO) [16] by using multiple cores on the same ma-

chine. A further step would be the distribution of evolutionary optimization with a client-server-protocol, as exemplified by Kriesel [17].

To tackle this problem, the authors propose an architecture to distribute the simulations of an evolutionary optimization process onto multiple servers. Based on the work presented in [1], this paper describes an improved, extensible architecture that considers the lessons learned. This architecture builds on different generic tools for performing the optimization, evaluating the controller candidates through simulation, and managing the communication network. An implementation is provided, partially based on existing tools. The implementation is evaluated by demonstrating its usability among different test scenarios. First, the optimization process in heterogeneous network setups is demonstrated. Second, the deployment of the obtained controllers onto Robot Operating System (ROS) [18] based platforms is demonstrated, including simulations and hardware platforms. Finally, the scalability of the optimization performance is analyzed in terms of total time taken. There is a significant performance improvement compared to the previously presented architecture yielding an effective scalability with high numbers of simulators.

The proposed architecture is implemented as part of the *CPSwarm workbench* [19] which is a tool chain developed in the H2020 research project CPSwarm. This workbench aids in developing self-organizing swarm behavior for CPSs. It starts from modeling and design, goes over simulation and optimization, to deployment and monitoring. It is built around a central launcher application that allows to graphically access and configure the different tools. This work describes the simulation and optimization section of the architecture, known as the *simulation and optimization environment*.

The paper is organized as follows. In Section II, the evolutionary approach for designing swarms is briefly reviewed. Section III describes the two proposed approaches for distributing the simulations in an evolutionary optimization process by comparing them based on the results of the previous paper. Section IV introduces the newly proposed architecture that eliminates the problems previously experienced. Next, an implementation of this architecture is described in Section V. Section VI describes the testbeds that are used to evaluate the features provided and to measure the performance of the presented solution. The results of the performance analysis are detailed in Section VII, including a comparison to the previous approach. Finally, Section VIII provides a discussion, presents future work, and concludes the paper.

## II. Designing Swarms by Evolution

As described in the previous section, design by evolution can be used to tackle challenges such as scalability and generality [20], as well as adaptive self-organization [21]. These issues are not easy to handle, especially in changing environments and with dynamic interactions among the individual CPSs in a swarm.

Designing a swarm using evolution is an automatic design method that creates an intended swarm behavior in a bottom-up process starting from very small interacting components. This process modifies potential solutions until a satisfactory result is achieved. Such an approach is based on evolutionary computation techniques [22],[23] and mimics the Darwinian principle [2]. It describes the process of natural selection by recombining the most proper solutions to a defined problem. Evolution can be applied either on the level of individuals or the swarm as a whole. Typically, the process of evolving a behavior starts with the generation of a random population of individual behavior control algorithms. Each member of the selected population is evaluated using simulations and graded by a fitness function that allows ranking the behaviors' performances on the swarm level. The higher a behavior is ranked, the more likely it is to survive to the next generation. This selection process makes sure that only the best performing behaviors survive to serve as input for the next iteration of the evolutionary process. They are reproduced using genetic operators like crossover or mutation. This process is iterated for a specific number of generations or until a CPS controller emerges that exhibits the desired global swarm behavior. Nevertheless, design by evolution poses several challenges, including no guaranteed or predictable convergence to the global optimum, complex data structures, and the high costs of evolutionary computation itself.

Design by evolution dictates several tasks that a designer must face during the design of a system model. According to Fehervari and Elmenreich [24], there are six tasks to consider:

1) The *problem description* gives a highly abstracted vision of the problem. This includes constraints and the desired objectives for such a problem.
2) The *simulation setup* transfers the problem description into an abstracted problem model. This model specifies the system components, i.e., details about the CPSs and the environment.
3) The *interaction interface* defines the interactions among CPSs and their interactions with the environment. For instance, the CPSs' sensors and actuators as well as the communication protocols should be specified here.
4) The *evolvable decision unit* represents the CPS controller and is responsible for achieving the desired objectives, i.e., the global behavior of the swarm to achieve a common goal. Such a decision unit must be evolvable to allow genetic operations as crossover and mutation. It is most commonly represented by an Artificial Neural Network (ANN). There are different types of ANNs, e.g., fully-meshed ANNs, feed-forward ANNs, or HebbNets [25].
5) The *search algorithm* performs the optimization using evolutionary algorithms by applying the results from the above steps. During this task, an iterative mathematical model is used to find the optimal solution. The optimization result is dependent on the fitness function of the problem.
6) The *fitness function* represents the quality of the optimization result in a numerical way. There is no specific way or rule to design such a function as it is highly dependent on the problem description. The main purpose of this function is to guide the search algorithm to find the best solution. In general, the applicability and performance of a fitness function depends on the employed Optimization Tool (OT), thus there are no universally suitable fitness functions [26]. However, many studies in the field of evolutionary optimization have considered generic methods for fitness function design [27], [25].

Recently, several software frameworks have been implemented to support the procedure of evolutionary design. AutoMoDe [28] is a software for automatic design, which generates modular control software in the form of a probabilistic finite state machine. JBotEvolver is a Java-based versatile open-source platform for education and research-driven experiments in evolutionary robotics [29], which has been used in many studies [30], [31], [32]. Gehlsen and Page [33] addressed the topic of parallel execution of experiments for heuristic optimum seeking procedures. Their approach, DIstributed SiMulation Optimization (DISMO), supports distributed execution of Java simulations for optimization projects. As simulation core, the framework for DEveloping object-oriented Simulation MOdels in Java (DESMO-J) is used. The SImulation-based Multi-objective Evolutionary OptimizatioN (SIMEON) framework [34] implements several components in Java for providing an evolutionary optimization of problems modeled via simulation. Examples involve supply chain optimization and flexible manufacturing scheduling. However, the SIMEON framework does not provide a strategy for distributed simulation across multiple servers. In addition, the previously mentioned tool FREVO supports creating and evaluating swarm behavior by evolution and has been used in several studies as an evolution tool including robotics [12] and pattern generation [35].

Designing swarm behaviors using evolutionary optimization requires a large number of simulation runs. The next section summarizes and compares two approaches proposed in [1] for distributing these simulations onto different machines.

## III. Distributed Simulation

Depending on the level of realism in simulation of swarms of CPSs, a considerable amount of computational power is required [36]. This is especially true for the high number of simulations needed to complete an evolutionary optimization process. This can be accelerated by parallelizing the simulations that are carried out within one generation. In previous work [1], the authors proposed two architectural approaches on how to perform this parallelization and to distribute the workload. Both approaches have a common architecture composed by two core components, the OT and one or more Simulation Tools (STs). This concept is visualized in Figure 1. The OT is responsible for performing the evolutionary optimization process explained in Section II. The STs perform the required simulations in each step of the optimization process and evaluate the fitness value of a controller candidate. The interconnection between the OT and the STs allows to pass the simulations required during the evolutionary optimization from the OT to the STs where they are executed in parallel. The STs simulate 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 definition of a generic interface gives the opportunity to build on well established STs that support accurate simulation of swarms of CPSs with different levels of detail.

The key difference between the two approaches is the location where the CPS controller resides during the simulation. The centralized approach lets the CPS controller reside

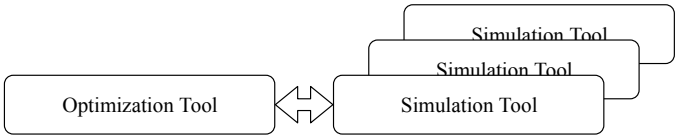


Figure 1. The concept of distributed simulation based on the components OT and ST.

centrally within the OT. The STs are merely executing the actions received from the controller in the OT and report back the sensor readings of the CPSs. With this approach the STs are hence centrally controlled by the OT. Instead, the distributed approach transfers the CPS controller from the OT to the STs, which can then independently run the simulation and only return back the resulting fitness value of the controller. This approach therefore distributes the control from the OT to the STs.

In [1] the two approaches have been implemented differently. The centralized approach has been implemented based on network socket inter-process communication. It allowed to distribute the simulations among multiple STs located on remote servers. The drawback of this implementation was that there was a high messaging overhead and a repeated polling for available STs. This inhibited the system to scale well with more than three STs. The distributed approach has been implemented based on file system inter-process communication. It allowed to fully pass over the control to the STs, but was limited to execution on a single machine. The authors now abstract away from the implementation specifics and thus refer to these approaches only as centralized and distributed.

Both cases require an interface between the OT and the STs, but there are some key differences. For the centralized approach, 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 again to the corresponding ST. On the ST side, the interface must allow to control the CPS behavior using the actuator commands received from the OT. Once the commands are executed by the ST, the interface transmits back the sensor readings to the OT as they are perceived by the CPSs. The interface for the distributed approach requires to exchange the controller representation. On the OT side the interface must therefore export the controller representation and transmit it to the STs. Once the STs completed the simulations, it receives the result of the simulations to assess the performance of the controller. Depending on the implementation, this can be either raw log data or already processed information in form of a fitness value. The ST side of the interface receives the controller and integrates its representation into CPS behavior code. This allows the ST 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 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 some kind of discovery process where the OT polls for STs, stating the requirements on the ST to be used (e.g., number

of dimensions or maximum number of agents supported). The STs satisfying 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 has found a solution that it cannot further optimize. This optimal solution is represented by a CPS controller that can then be exported by the OT to be deployed on the CPSs.

The deployment of the optimized controller can take place in STs or on actual CPS hardware. The former can be used to further inspect the resulting controller. This allows to run further performance, scalability, or robustness analysis as well as visual inspection of the CPS behavior. The latter allows deploying the resulting controller to the CPSs and test it under real conditions. Whether the controller is further used in simulations or on actual CPS hardware, there is the requirement for an interface that allows connecting the CPSs' sensor inputs and actuator outputs to this controller. This interface must follow the same specification as the interface implemented in the STs used during the optimization process with the distributed approach. Therefore, this interface needs to be defined only once and can then be used for optimization and deployment.

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 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 there is a lot of message exchange between OT and the STs, throughout the simulation. When the number of STs is increased, the OT can become the bottleneck as it has to communicate constantly with each ST. Therefore, the distributed approach is less portable than the centralized approach but can reach a higher performance [1].

The performance of either architectural approach can be measured as the total time taken for the optimization process. As the authors shown in [1], this time can be expressed as

$$t_{\mathrm{opt}} = n_{\mathrm{gen}} \cdot \left( t_{\mathrm{evo}} + \frac{n_{\mathrm{pop}} \cdot n_{\mathrm{eval}}}{n_{\mathrm{sim}}} \cdot (t_{\mathrm{sim}} + t_{\mathrm{ohd}}) \right) \tag{1}$$

consisting of three time components. First, the time $t_{\mathrm{evo}}$, 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 $n_{\mathrm{gen}}$. Second, the time $t_{\mathrm{sim}}$, 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 CPSs in the simulation. The simulation time can therefore be expressed as

$$t_{\mathrm{sim}} = n_{\mathrm{step}} \cdot t_{\mathrm{step}} \tag{2}$$

based on the number of discrete time steps $n_{\mathrm{step}}$ and the time $t_{\mathrm{step}}$ required to simulate one step. A simulation is performed for each controller in the population of $n_{\mathrm{pop}}$ controller candidates. For robustness and statistical significance, each controller can be evaluated $n_{\mathrm{eval}}$ times in a different variant of the same problem. This results in a number of $n_{\mathrm{pop}} \cdot n_{\mathrm{eval}}$ simulations that have to be performed during each of the $n_{\mathrm{gen}}$ generations. Depending on the number of available STs $n_{\mathrm{sim}}$, the optimization process can be accelerated by distributing the simulations among these STs. The upper limit for the number of required STs is therefore $n_{\mathrm{pop}} \cdot n_{\mathrm{eval}}$. Finally, the third time component $t_{\mathrm{ohd}}$ states the amount of overhead time required during simulation. Where the other two time components are identical for both approaches, the overhead time varies between the centralized and the distributed approach. In [1] the authors differentiated between two implementations when calculating the overhead time. It is now generalized by calculating it for the centralized and distributed approach. This abstracts the implementation details and yields the overhead time as

$$t_{\mathrm{ohd}} = t_{\mathrm{setup}} + t_{\mathrm{run}} + t_{\mathrm{finalize}} \tag{3}$$

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

$$\begin{aligned} t_{\mathrm{ohd,c}} = {} & n_{\mathrm{msg,setup}} \cdot t_{\mathrm{msg}} \\ & + n_{\mathrm{msg,run}} \cdot n_{\mathrm{step}} \cdot n_{\mathrm{cps}} \cdot t_{\mathrm{msg}} \\ & + n_{\mathrm{msg,finalize}} \cdot t_{\mathrm{msg}} + t_{\mathrm{fitness}} \end{aligned} \tag{4}$$

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

$$\begin{aligned} t_{\mathrm{ohd,d}} = {} & t_{\mathrm{export}} + n_{\mathrm{msg,setup}} \cdot t_{\mathrm{msg}} + t_{\mathrm{import}} \\ & + n_{\mathrm{msg,finalize}} \cdot t_{\mathrm{msg}} + t_{\mathrm{fitness}} \end{aligned} \tag{5}$$

that contains two additional time components as compared to the centralized approach. First, the time $t_{\mathrm{export}}$ to export the controller representation from the OT into a format readable by the STs. Second, the time $t_{\mathrm{import}}$ to import the controller into a ST. To compare the performance of both approaches, it is possible to calculate the ratio

$$r = \frac{t_{\mathrm{opt,c}}}{t_{\mathrm{opt,d}}}, \tag{6}$$

which relates the total optimization run time of the centralized approach $t_{\mathrm{opt,c}}$ with the optimization time of the distributed approach $t_{\mathrm{opt,d}}$. This ratio expresses, which approach is more suitable for a specific setup of parameters. The authors determined the most relevant parameters for analysis to be the simulation length as number of simulated steps $n_{\mathrm{step}}$ and the number of CPSs $n_{\mathrm{cps}}$ 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 authors 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 I where the evolutionary parameters were chosen to yield good results.

The resulting ratio $r$ using these values can be seen in Figure 2 for a varying number of CPSs. A value of $r > 1$ means that the distributed approach performs better whereas a value of $r < 1$ means that the centralized approach is favorable. As both approaches can use parallelization, the resultant ratio is independent of the number of parallel STs used. It can be seen that for most cases the distributed approach performs better, even though the time for importing the controller is dominating in Table I. 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 2, 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 18 steps.

To conclude the comparison between the two approaches, it can be stated that the distributed approach is favorable 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 STs. In the network-based implementation, previously presented in [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, this paper now proposes to have two separate phases. First, the setup phase, where all available STs are discovered and second, the actual optimization phase, which uses the available STs. The new architecture realizing this proposal based on the previous experience is presented in the next section.

## IV. Architecture

This section presents a distributed architecture that 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. It builds on the experience of the previously proposed architecture described in [1].

The first problem addressed is the discovery protocol responsible for determining the available STs. The previous architecture required repeated discovery before each simulation, hence the performances did not scale well with the number of STs. The new architecture therefore introduces a setup phase, where the STs announce themselves and their capabilities. Any updates to the available STs are further communicated,

TABLE I. Optimization parameters measured through simulations.

| parameter | value |
|---|---|
| $n_{\text{gen}}$ | 200 |
| $n_{\text{pop}}$ | 50 |
| $n_{\text{eval}}$ | 1 |
| $n_{\text{msg,setup}}$ | 4 |
| $n_{\text{msg,run}}$ | 2 |
| $n_{\text{msg,finalize}}$ | 1 |
| $t_{\text{evo}}$ | 12 ms |
| $t_{\text{msg}}$ | 30 ms |
| $t_{\text{export}}$ | 5.35 ms |
| $t_{\text{import}}$ | 8833 ms |
| $t_{\text{fitness}}$ | 0.69 ms |
| $t_{\text{step}}$ | 100 ms |

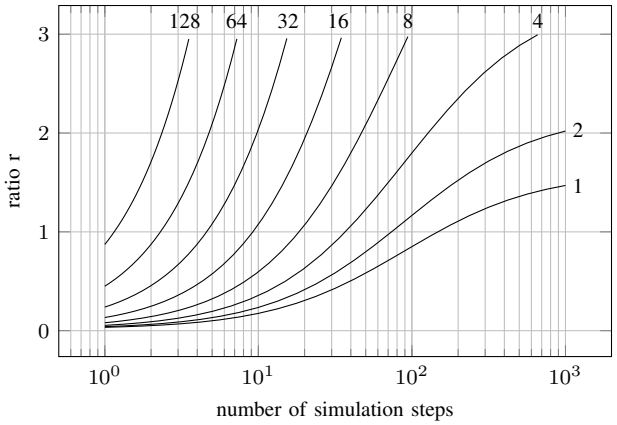


Figure 2. Ratio of optimization times between central and distributed control for different numbers of CPSs.

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 amount of messages that are exchanged during the optimization phase. The new architecture is therefore based on the distributed approach described in the previous section. The OT exports the controller to the STs to avoid exchanging the sensor readings and actuator commands. The STs can thus perform the simulation stand-alone without relying on the OT. This reduces the amount of exchanged messages considerably.

The architecture consists of four main components, as shown in Figure 3. First, it includes the previously mentioned OT, which is responsible for performing the evolutionary optimization. Second, there are several STs, distributed in different machines, called Simulation Servers (SSs), each one wrapped by a Simulation Manager (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 actually used. Third, there is a component called Simulation and Optimization Orchestrator (SOO), newly introduced in this architecture. The SOO is in charge of keeping track of all the SMs and coordinating the simulation 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 on the STs, such as dimensionality or minimum CPS cardinality. In this

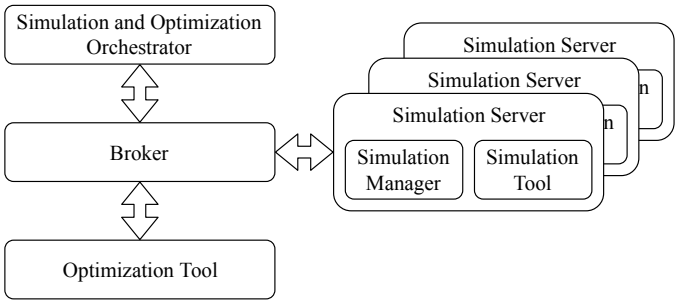


Figure 3. The network-based architecture consisting of the components SOO, broker, OT, and SS.

way, the SOO can select the SMs that fulfill the requirements. Finally, there is a broker that handles all communication between the other components.

The proposed architecture can perform two different kinds of workflows requiring simulations. First, the SOO can perform an optimization using the OT, where each controller candidate is simulated in a SS. Figure 4 illustrates the flow of messages between the SOO, the OT, and two exemplary SMs during the optimization process. 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 optimization 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, selects the ones that fulfill the indicated requirements, and transmits to them the configuration files needed to setup the ST. Once all the SMs have confirmed to be configured, the SOO sends a *StartOptimization* message to the OT, containing the list of SMs to be used for the optimization process. The OT replies with an *OptimizationStarted* message that includes a unique IDentifier (ID) of the optimization process. Then, the OT starts the optimization, using the STs that satisfy the requirements. It uses all the configured SMs in parallel by sending a sequence of *RunSimulation* messages to them. Each of these messages contains a candidate controller to be evaluated. The OT awaits the corresponding *SimulationResult* messages from the SMs. 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. Once the optimization process completed, the OT sends a final *OptimizationProgress* message to the SOO, containing the optimized controller.

Second, the SOO can send a specific controller candidate to a SM for more in depth analysis. This allows to evaluate a controller found by the OT more thoroughly, e.g., through visual replay using the ST Graphical User Interface (GUI). In case of visual replay, the selected ST must run on one machine directly accessible to the user, who needs to see the GUI of the ST. In this case, the SOO is responsible for sending the controller to the selected SM. This is visualized in Figure 5. In this much simpler scenario, the OT is not involved and SOO and SM communicate directly.

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

$$t_{\mathrm{opt}} = t_{\mathrm{setup}} + n_{\mathrm{gen}} \cdot \left( t_{\mathrm{evo}} + \frac{n_{\mathrm{pop}} \cdot n_{\mathrm{eval}}}{n_{\mathrm{sim}}} \cdot (t_{\mathrm{sim}} + t_{\mathrm{ohd}}) \right) \tag{7}$$

having the additional setup time

$$t_{\mathrm{setup}} = (n_{\mathrm{msg,presence}} + n_{\mathrm{msg,config}} + 2) \cdot t_{\mathrm{msg}} \tag{8}$$

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

$$t_{\mathrm{setup}} = (2 \cdot n_{\mathrm{sim}} + 3) \cdot t_{\mathrm{msg}}. \tag{9}$$

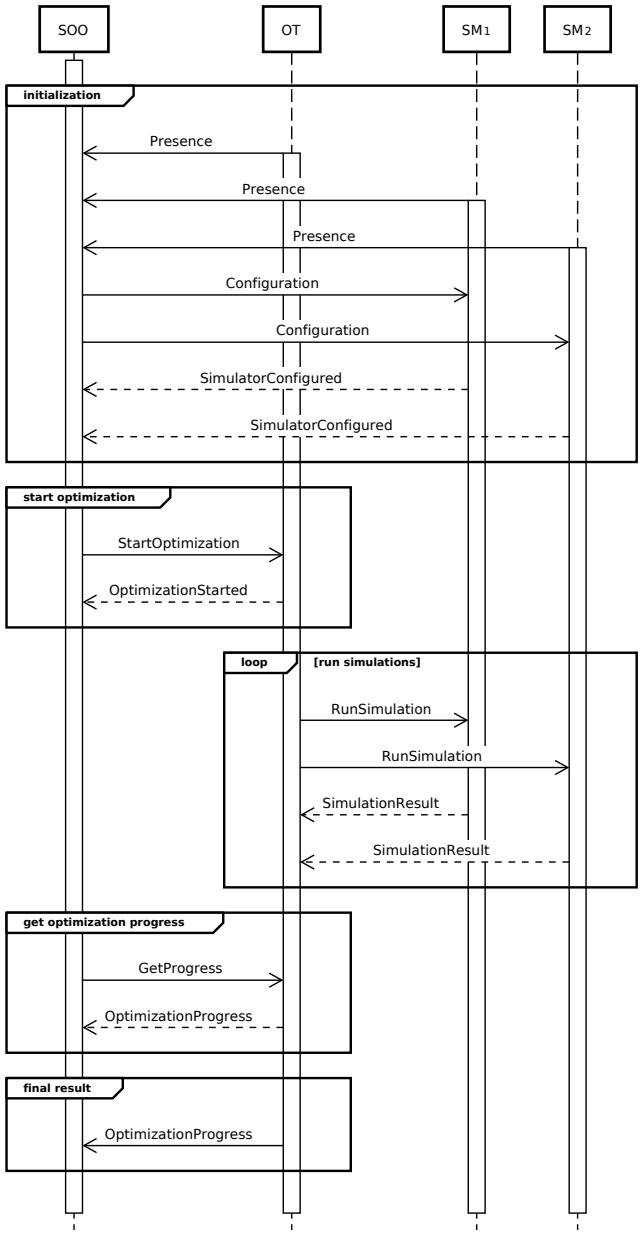


Figure 4. The messaging sequence during the optimization process.

Based on this architecture, the next section presents an implementation using the eXtensible Messaging and Presence Protocol (XMPP).

## V. Implementation

The architecture presented in the previous section is implemented using existing tools wherever possible and developing new tools where necessary. The tools that where developed completely from scratch are the SOO as well as the SMs. The

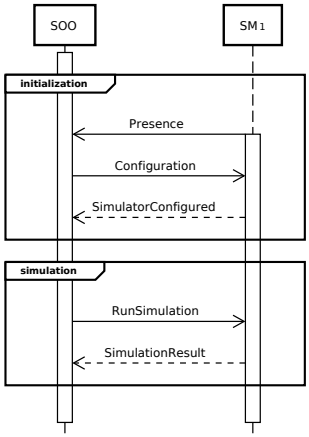


Figure 5. The messaging sequence for simulating a specific CPS controller.

existing tools are extended by a wrapper to enable integration with the proposed architecture. These are the OT and the STs. This section firstly describes the communication infrastructure that connects the different tools, followed by a description of their implementation.

The communication among the different tools happens according to the messaging sequence shown in Figures 4 and 5. To enable all tools to communicate with each other, the architecture relies on a central broker that manages the connections between them. The two messaging protocols Message Queue Telemetry Transport (MQTT) [37] and XMPP [38] have been tested extensively and are used for the implementations. MQTT is recognized as the de facto standard for event-driven architectures in the Internet of Things (IoT) domain. It has been chosen for the implementation previously presented in [1] because of its extreme simplicity. Its design principles attempt to minimize network bandwidth and device resource requirements whilst also ensuring reliability and some degree of assurance of delivery. Because XMPP owns features that allow to fulfill the requirements of the architecture described in Section IV it is selected as alternative communication protocol. It is favored over MQTT in the current implementation because MQTT does not offer all required features, such as one-to-one communication or a standardized presence protocol. Using XMPP, the proposed architecture can be implemented without having to rely on multiple different protocols.

In this work, the OT is implemented based on FREVO, a modular optimization system that applies the principles of genetic algorithms addressed in Section II [16]. FREVO divides the optimization task into several steps. First, it allows to model the problem as a simulation alongside an evaluating fitness function. Second, it allows to select different evolvable controller representations for the CPSs' controllers. Third, it allows to choose the evolution method used during the optimization process. Therefore, FREVO offers exceptional flexibility and allows many different setups to be explored. Currently, FREVO provides an implementation of the Neural Network Genetic Algorithm (NNGA) method [39]. It begins by creating $n_{\mathrm{pop}}$ controller candidates. In each of the $n_{\mathrm{gen}}$ 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. For integrating FREVO in this architecture, it is extended with a layer supporting XMPP communication and hence called FREVO-XMPP.

The SOO is implemented as Java application embedding an XMPP client. The SOO can be configured with several parameters. They specify the requirements of the optimization task executed by the user. These parameters include requirements for the evolutionary process as well as requirements on the simulation.

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

When a SM starts, it adds the SOO to its roster, which is the list of "friend accounts". It signals its availability by sending a *Presence* message including a list of features provided by the wrapped ST, which is automatically received by all the "friends", i.e., the SOO. The current implementation features the number of dimensions and the maximum number of CPSs supported, but the list will be updated in future releases. In this way, the SOO receives and collects the availability of the SMs and it is able to choose the ones that fulfill the requirements for the execution of new tasks. After choosing the SMs to be used, the SOO sends to the SMs the files that are required for configuring the STs, using the XMPP file transfer. These include the models of CPSs and environment. The SMs confirm the reception of the configuration with an *SimulationConfigured* message. In case of simulation only, the SOO sends the controller to be replayed directly to the SM. In case of optimization, it sends a *StartOptimization* message to the OT, indicating the Jabber Identifiers (JIDs) of the SMs to be used.

Upon receiving a *StartOptimization* message, FREVO-XMPP creates an *OptimizationTask* to oversee the optimization process. 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 Central Processing Unit (CPU) load. The *OptimizationTask* deserializes a 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. Rather than evaluating a controller locally as is typically done in FREVO implementations, it sends a *RunSimulation* message to one of the associated SMs and blocks waiting for a *SimulationResult* message or a simulation timeout to occur.

As common basis for the STs, ROS [18] is chosen because it is an open-source solution, widely supported by several robotic platforms and many existing STs. It provides modularity and interoperability. In this way, the same CPS controllers can be tested on different STs and then deployed on actual ROS-based hardware platforms. Two specific SMs are implemented integrating the ROS simulations based on the

STs Stage [36] and Gazebo [40]. Several other STs have been considered as well, e.g., V-REP [41], ARGoS [42], jMAVSim [43], and STDR [44]. Resulting from an analysis of their controllability, configurability and support for standard models, the authors selected Stage and Gazebo for integration in the proposed architecture. Nevertheless, the modular approach of our architecture allows to integrate other ROS STs as well, with only little additional effort.

The communication between SM and ST is built on the work done in [1]. The ST is ROS-based and executed by the SM as ROS node using the standard ROS facilities such as launch files. The simulation to be run is installed beforehand on the SS as a ROS package. The CPS controller it is transmitted from FREVO-XMPP to the SM as C++ code, which allows them to be efficiently integrated into the simulation image. When the SM receives the controller, it forwards it to the ST, which is recompiled with the new controller. The ROS package contains a launch file that launches the required ST. The SM runs this launch file to launch the simulations, passing the parameters that the user indicates through the SOO. In case of optimization, the SMs are also in charge of calculating the fitness values of the tested controllers. This is achieved by parsing the ROS log files and computing the fitness value accordingly.

As a test case, a simple multi-CPS simulation called *EmergencyExit* is implemented in ROS. It realizes a problem where the CPSs have to escape from the environment while avoiding collisions with other CPSs running in discrete time and space. It is implemented as ROS package consisting of the controller representation and a wrapper class. The wrapper is adapted for different STs, while the controller can be reused seamlessly. During the optimization process, the wrapper stays fixed. The part that changes for each simulation is the controller code. Every CPS creates log files that are used by the SM to report to FREVO-XMPP the overall fitness value of the controller used in that simulation.

A third SM is implemented based solely on Java without the need for ROS to demonstrate the flexibility of the architecture. It is used by the centralized approach previously introduced in [1]. The ST is a very basic stand-alone Java simulator called *Minisim* [1]. It is a command-line, multi-CPS ST simulating a capture-the-flag game with multiple CPSs on a two-dimensional grid. *Minisim* has been specifically developed for testing the network communication between FREVO-MQTT and the SMs.

It is planned to release the code of this implementation in 2019 as open source on the CPSwarm Github repository (https://github.com/cpswarm). It will include the OT FREVO-XMPP, the SOO, and SM implementations for the STs Stage and Gazebo.

Several testbeds are setup to test the solution presented in this section and evaluate its performance. The description of the testbeds and the corresponding test cases are described in the next section.

## VI. Testbed

The solution that has been presented in the previous chapters is evaluated through 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.

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 OT FREVO-XMPP. The XMPP server is installed in the cloud. Both Openfire and Tigase have been used. This setup is visualized in Figure 6.

To test the different components and workflows of the architecture, first an optimization is performed and the result is then replayed locally using a GUI. For this purpose, the authors implemented the *EmergencyExit* problem in simulation as ROS components, both for the Stage ST and the Gazebo ST. The implementations are based on a simple scenario with three CPSs and two exits. The scenario setup can be seen in Figures 7 and 8 for the Stage and Gazebo STs, respectively. This simple setup allows to effectively test the architecture without shifting the focus on the challenges related to performing complex and large-numbered multi-CPS simulations. The two implementations feature a different level of abstraction. First Stage, which implements the CPSs as simple squares and second Gazebo, which implements the CPSs as TurtleBot robots.

To perform the test, the authors launched the SOO selecting the optimization workflow with the requirement to perform simulations in two dimensions. This requirement was defined because a more abstract simulation yields better performance of the optimization, which includes a high number of simulations. As a result, the SOO successfully selected the three SSs running Stage and excluded the one running Gazebo. Then, the SOO launched FREVO-XMPP indicating to it the SSs to use and FREVO-XMPP distributed the simulation tasks onto them. Once the optimization finished, FREVO-XMPP 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. As a result, the SOO successfully launched the simulations locally in Gazebo displaying the GUI with the 3D environment. The more detailed ST Gazebo allowed to replay and test the final optimization result under more realistic conditions. 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 SMs. It thus demonstrated

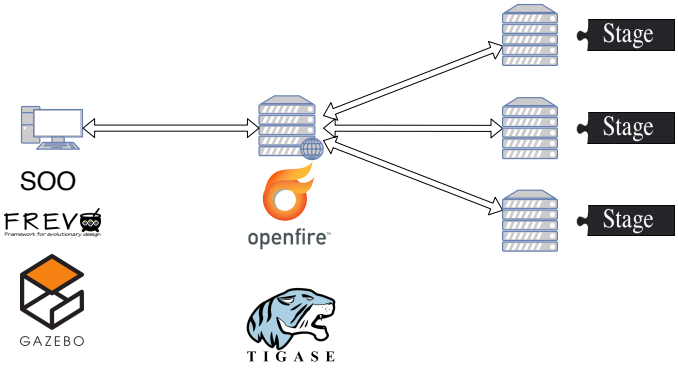


Figure 6. PoC testbed setup.

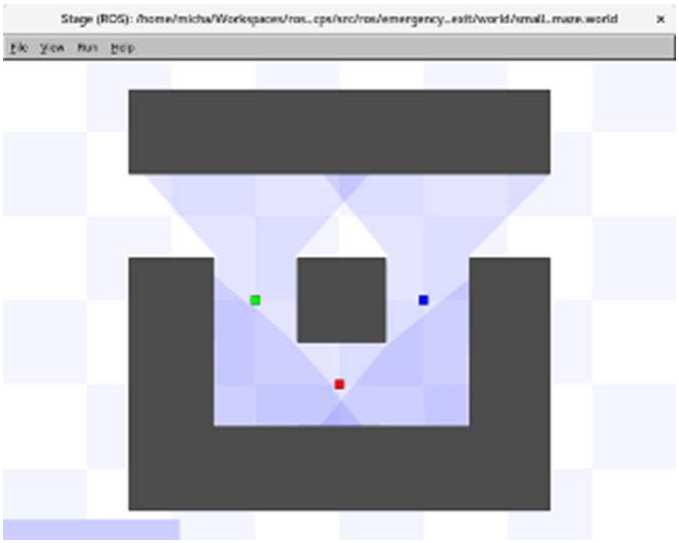


Figure 7. Stage ROS implementation of the *EmergencyExit* simulation.

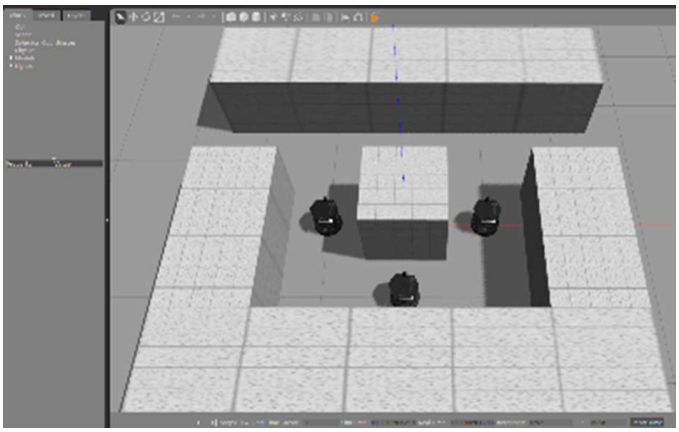

Figure 8. Gazebo ROS implementation of the *EmergencyExit* simulation.

how the STs are seamlessly integrated through the architecture described in this paper.

This test is complemented by using the optimized CPS controller for deployment on real hardware. By using ROS during the optimization and simulation process the authors have already demonstrated the portability of the controller among different STs. To take this even one step further, the same controller has been installed onto TurtleBot robots. The authors performed tests in an environment similar to the one used in simulation, see Figure 9. In this way, it has been demonstrated the complete chain from optimization, over simulation, up to deployment on a CPS hardware platform.

A second test case is constructed for a realistic distributed comparison between the centralized approach and the distributed approach. For this objective, the setup is the one shown in Figure 10 with four distributed SSs. For technical reasons, the centralized approach is implemented using the *Minisim* Java simulation, while the distributed approach is based on the *EmergencyExit* ROS simulation. Nevertheless, both approaches are comparable as both perform simulations lasting for the given number of steps. Specifically, for this test case, four

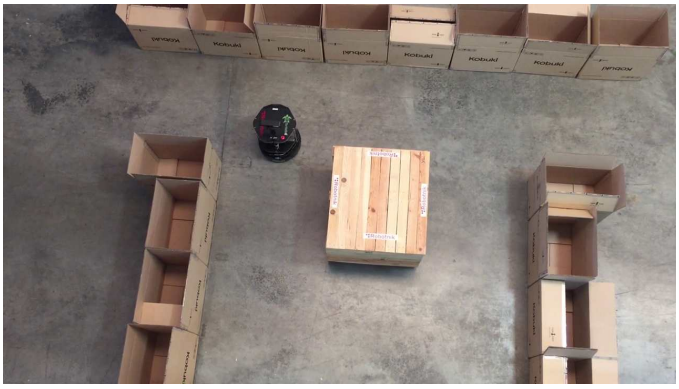

Figure 9. Real world experiment of the of the *EmergencyExit* problem using TurtleBot robots.

distributed computers are used: three SSs having installed the Stage ST with corresponding SM, the *EmergencyExit* ROS simulation and ROS Kinetic. The SOO and the OT FREVO-XMPP run on another computer that acts also as one SS, with the same setup as the others. All the components have been connected to a Tigase XMPP server running in the cloud. With this test case it is possible to test the complete optimization process, first using one SS and then parallelizing it on two, three, and four SSs.

Finally, a third test case is constructed to evaluate the scalability of the implementation with the number of SSs. Specifically, the objective is to show what degree of parallelization is possible with the current implementation, based on the distributed approach. To do this, all implemented tools (SOO, OT, and SMs) are executed on a single computer with 12 Intel Xeon X5675 processors running at $3.07\,\mathrm{GHz}$ and $16\,\mathrm{GB}$ of memory. Using hyper-threading, it supports 24 threads that can run in parallel. The operating system is Ubuntu 16.04 $64\,\mathrm{bit}$ running OpenJDK 9 Java. This setup is visualized in Figure 11. As before, the OT used is FREVO-XMPP and the Tigase XMPP server runs in the cloud. 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 analyze the scalability of the network performance with the number of SSs it emulates the overhead time, which changes from (5) to

$$t_{\mathrm{ohd}} = (n_{\mathrm{msg,setup}} + n_{\mathrm{msg,finalize}}) \cdot t_{\mathrm{msg}} \tag{10}$$

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

## VII. Performance Evaluation

This section presents the performance evaluation of the proposed architecture acquired using the testbed described in the previous section. The performance is measured in total time taken for a complete optimization run. This optimization time is measured for a varying number of simulation steps $n_{\mathrm{step}}$ and a varying number of SSs $n_{\mathrm{sim}}$. 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 $99.9\,\%$. As a first step, the centralized approach is compared to the distributed approach introduced

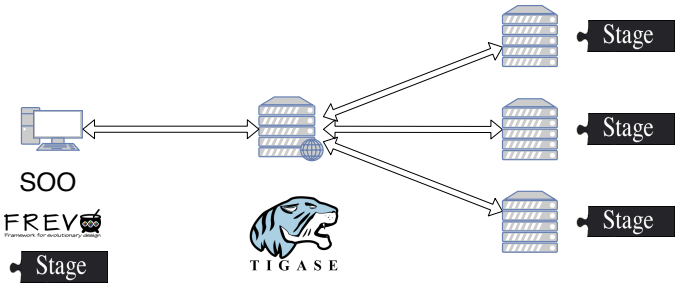


Figure 10. Performance comparison testbed setup.

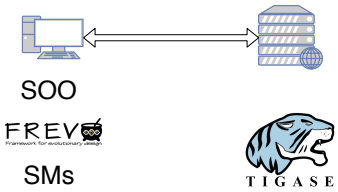


Figure 11. Scalability evaluation testbed setup.

in Section III. Then follows a more in depth analysis of the distributed approach that analyzes its scalability with the number of SSs.

### A. Comparison of Centralized and Distributed Approach

To compare the centralized and the distributed approach, the authors first compute the total optimization time including setup time, based on (1) and (7), respectively. Then the authors perform measurements according to the testbed setup described as the second test case in the previous section. For calculating the optimization time, the measurements presented in Table I are used. The number of CPSs being simulated is $n_{\text{cps}} = 8$ and the evolutionary parameters are set to $n_{\text{gen}} = 4$ generations and $n_{\text{pop}} = 4$ controller candidates per generation. This yields the optimization times

$$t_{\text{opt,c}} = \frac{9.28\,\text{s} \cdot n_{\text{step}} + 2.41\,\text{s}}{n_{\text{sim}}} + 0.048\,\text{s} \qquad (11)$$

for the centralized approach and

$$t_{\text{opt,d}} = 0.06\,\text{s} \cdot n_{\text{sim}} + \frac{1.6\,\text{s} \cdot n_{\text{step}} + 142.39\,\text{s}}{n_{\text{sim}}} + 0.14\,\text{s} \qquad (12)$$

for the distributed approach with $n_{\text{sim}}$ SSs. These optimizations times are plotted in Figure 12 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 longer the simulation, the more favorable becomes the distributed approach. In this example with eight CPSs, the centralized approach is favorable only for short simulations in the order of ten steps. This is in line with the conclusions from the ratio shown in Figure 2.

Next, measurements using the testbed described in the previous section are performed. They are compared to the

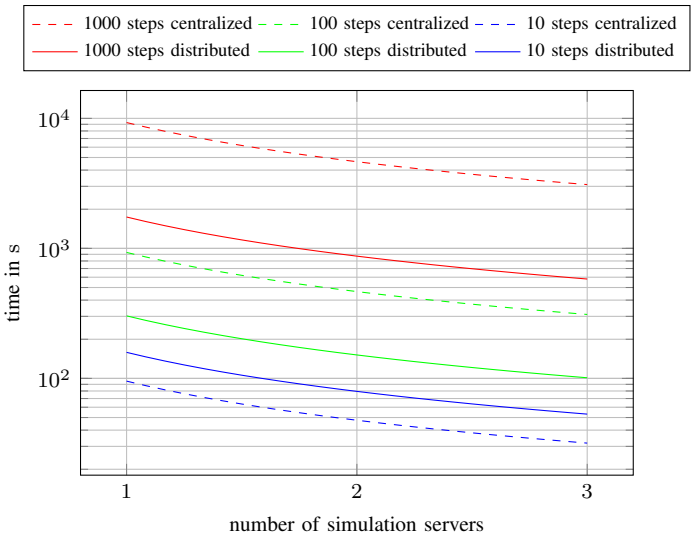


Figure 12. Theoretical comparison of the scalability with number of SSs of the optimization time between centralized and distributed approach for varying simulation lengths and eight CPSs.

performance results of the centralized MQTT implementation presented in [1]. The results can be seen in Figure 13. They show the limitations of the implementation of the centralized approach. Because it performs SS discovery before each simulation it scales only up to three SSs. The implementation of the distributed approach mitigates this problem by introducing a different presence mechanism. It can be seen that 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.

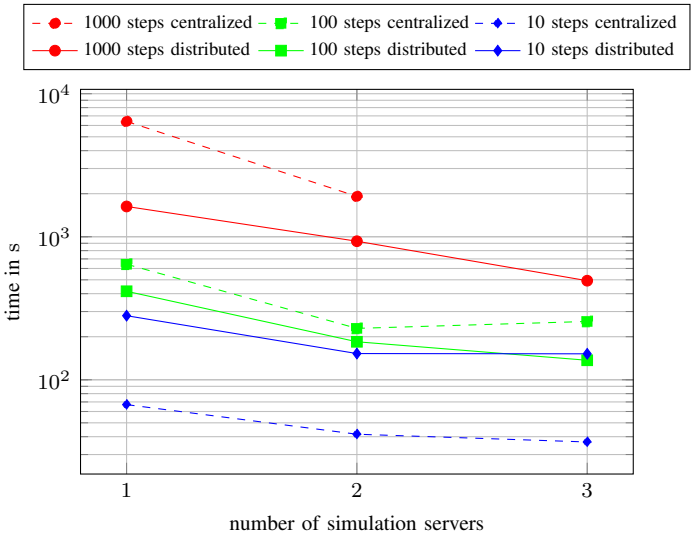


Figure 13. Measured comparison of the scalability with number of SSs of the optimization time between centralized and distributed approach for varying simulation lengths and eight CPSs.

### B. Scalability Analysis

So far the authors showed the difference between the two approaches where the distributed approach excelled in most scenarios. To further investigate how well the performance scales with a larger range of SSs, measurements using the third test case described in Section VI are performed. To be able to assess the parallelization, the evolutionary parameters are set to $n_{\text{gen}} = 4$ and $n_{\text{pop}} = 32$. Because a typical optimization process includes only a single setup phase, only the optimization time is measured, which is the time between transmitting the *StartOptimization* message and receiving the final *OptimizationProgress* messages at the SOO. Figure 14 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. There is only a small offset between measurements and theory, which is due to implementation details not captured in the model. When increasing the number of SSs to more than 16, it can be seen that the performance does not scale well anymore. This is due to the fact that the testbed reaches its limitations as the computer used for running the tests has only 24 cores.

## VIII. Conclusion and Future Work

This paper presents a solution for the evolutionary design of swarms of CPSs based on remote simulation tools. The principal idea is to parallelize the simulations at each iteration of the evolutionary optimization process. The architecture designed for this solution builds upon the lessons learned from previous work [1], which introduced two different approaches with corresponding implementations. Starting from the evaluation of these approaches, the authors describe an XMPP based implementation of the architecture that combines the strengths of the two approaches previously presented in [1] and, at the same time, mitigates their weaknesses.

The new XMPP based implementation that uses the distributed approach is then compared to the previous implementation, which was based on the centralized approach. The

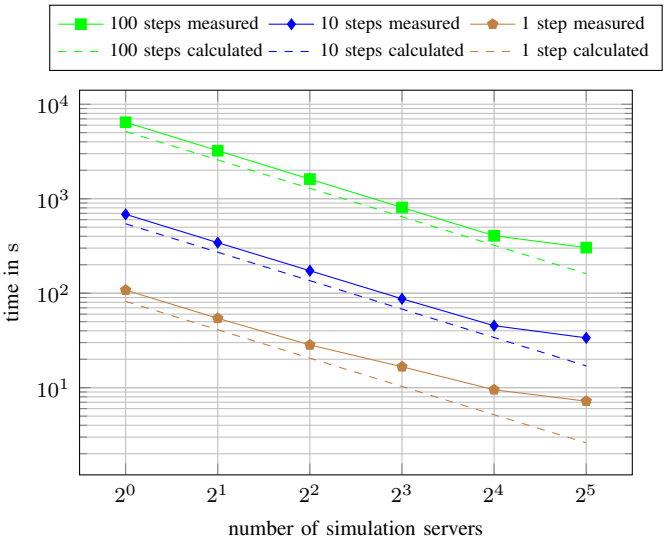


Figure 14. Scalability with number of SSs of the optimization time of the distributed approach for varying simulation lengths.

results in terms of total optimization time show that the new implementation is favorable in most cases. Only optimizations where the simulations have a short duration in the order of 1 s perform better using the previously implemented centralized approach. To take the performance analysis one step further, the new implementation is analyzed for its scalability with higher numbers of SSs. The results show that it scales well up to 32 SSs. The performance is expected to further scale well with even more SS, only the available hardware for creating such a large test setup was not available to the authors. Based on the new implementation, the authors demonstrate the ability of the architecture to successfully perform an optimization resulting in an optimized CPS controller. This controller is then replayed visually using the GUI of a locally installed ST connected to the OT. As a final step this controller is then exported and deployed on a TurtleBot robot to show that the resulting controller can bridge the gap from simulation to real world experiments.

Regardless of these achievements, the authors plan to extend the system in future to address several aspects. First, the compilation time within the STs could be reduced. Compiling the simulation code prior to testing a controller candidate is a major proportion of the overall time required by the optimization process. Incremental builds, i.e., recompiling only parts of the simulation that have changed, may reduce this significantly. In general, simulation code may be separated into four categories:

- Code specific to the simulation type. In the case of a ROS simulation, this includes the majority of ROS code. This code needs only be compiled once during the initial configuration of a SM.
- Code specific to the problem variant. This includes code such as randomly generated terrain and initial CPS positions. This code needs only be compiled during the initialization of a problem variant.
- Code specific to the controller representation type. This code implements a given type of representation, such as an ANN. This code needs only be compiled once for each representation type.
- Code specific to a given controller. This code is unique to a particular controller candidate, such as the weights and biases of an ANN, which can be updated without compilation.

A more optimized compilation strategy would separate code into these categories and would only recompile the modified pieces for a given simulation.

Second, the ST performance could be improved. In addition to compilation, the time required to start and stop ST instances adds significant overhead to simulation runs. Performance may also be improved by keeping the ST running and resetting its state for each individual run, e.g., returning CPSs to their start positions and resetting timer and counters. This would be particularly advantageous in situations where a controller may be changed purely by setting parameters, such as the weights and biases of an ANN. In any case, to implement this approach, improvements would need to be made to the simulation protocol to allow STs to distinguish the first simulation run from subsequent ones in a sequence, i.e., those from where compilation and simulation set up is required from those where it may be reused.

Third, the system robustness could be increased. One weakness of the current system is that it requires to restart from the beginning the (very long) optimization process, if it is disrupted. To address this, the authors propose saving the optimization state periodically by requesting the OT to send the last entire generation of controller candidates back to the SOO every $m$ generations. Furthermore, the OT must be extended to allow it start optimization with such a set of controller candidates, rather than just random ones. Communication with the SMs could also be improved by continuously monitoring XMPP presence notifications. If a SM goes offline, the OT can immediately identify this and ask to the SOO if there is another SS available to use. Also additional policies must be defined for retrying controller candidates in the event of a timeout.

Fourth, the system scalability could be extended. Currently, the system requires to instantiate one dedicated machine for each ST. By using a container service such as Docker [45], multiple STs may be run by each SS. To do this, a set of docker images containing the ROS based STs and related SMs need to be created. Furthermore, this strategy combined with solutions like Docker Swarm [46] or Kubernetes [47] would open the possibility to allow the user to rapidly deploy and easily maintain large-scale sets of STs. This would allow deploying the system to the cloud and thus addressing the inherently resource-bound nature of simulation. Corresponding improvements could also be made to the OT to allow it to support a large number of SMs.

Fifth, the optimization algorithms could be improved. Currently, the system offers the NNGA as the optimization method. While this algorithm produces a predicable number of simulation runs, other algorithms may offer improved performance. Similarly, the authors also plan to implement more advanced controller representations.

Finally, the tools and protocols could be generalized. In the future, the authors have already planned to support a greater range of OTs and STs (e.g., V-REP and ARGoS) and thus improve the value of the system to the community as a whole.

## ACKNOWLEDGMENT

The authors thank Robotnik Automation S.L.L. for porting the implementation to TurtleBot robots and the Gazebo ST. The research leading to these results has received funding from the European Union Horizon 2020 research and innovation program under grant agreement no. 731946.

## REFERENCES

[1] M. Rappaport, D. Conzon, M. Jdeed, M. Schranz, E. Ferrera, and W. Elmenreich, "Distributed simulation for evolutionary design of swarms of cyber-physical systems," in Proc. Int. Conf. on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE). IARIA, Feb. 2018, pp. 60–65, ISBN: 978-1-61208-610-1.

[2] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics: a review from the swarm engineering perspective," Swarm Intelligence, vol. 7, no. 1, Mar. 2013, pp. 1–41, DOI: 10.1007/s11721-012-0075-2.

[3] I. Fehérvári, V. Trianni, and W. Elmenreich, "On the effects of the robot configuration on evolving coordinated motion behaviors," in Proc. Congress on Evolutionary Computation (CEC). IEEE, Jun. 2013, pp. 1209–1216, ISBN: 978-1-4799-0452-5.

[4] R. Goldsmith, "Real world hardware evolution: A mobile platform for sensor evolution," in Proc. Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES). Springer, Mar. 2003, pp. 355–364, ISBN: 978-3-540-36553-2.

[5] J. Bongard, "Exploiting multiple robots to accelerate self-modeling," in Proc. Conf. on Genetic and Evolutionary Computation (GECCO). ACM, Jul. 2007, pp. 214–221, ISBN: 978-1-59593-697-4.

[6] D. Floreano and F. Mondada, "Hardware solutions for evolutionary robotics," in Proc. European Workshop on Evolutionary Robotics (EvoRobot). Springer, Apr. 1998, pp. 137–151, ISBN: 978-3-540-49902-2.

[7] M. Rubenstein, C. Ahler, and R. Nagpal, "Kilobot: A low cost scalable robot system for collective behaviors," in Int. Conf. on Robotics and Automation (ICRA). IEEE, May 2012, pp. 3293–3298, ISBN: 978-1-4673-1404-6.

[8] M. Jdeed, S. Zhevzhyk, F. Steinkellner, and W. Elmenreich, "Spiderino - a low-cost robot for swarm research and educational purposes," in Proc. Workshop on Intelligent Solutions in Embedded Systems (WISES). IEEE, Jun. 2017, pp. 35–39, ISBN: 978-1-5386-1157-9.

[9] F. Arvin, J. Murray, C. Zhang, and S. Yue, "Colias: An autonomous micro robot for swarm robotic applications," Int. J. Advanced Robotic Systems, vol. 11, no. 7, Jul. 2014, pp. 1–10, DOI: 10.5772/58730.

[10] Open Source Robotics Foundation, Inc. TurtleBot2. [Online]. Available: https://www.turtlebot.com/turtlebot2/ [retrieved: May, 2019]

[11] V. Crespi, A. Galstyan, and K. Lerman, "Top-down vs bottom-up methodologies in multi-agent system design," Autonomous Robots, vol. 24, no. 3, Apr. 2008, pp. 303–313, DOI: 10.1007/s10514-007-9080-5.

[12] I. Fehérvári and W. Elmenreich, "Evolving neural network controllers for a team of self-organizing robots," Journal of Robotics, vol. 2010, Mar. 2010, pp. 1–10, DOI: 10.1155/2010/841286.

[13] J. Xiao, Z. Michalewicz, L. Zhang, and K. Trojanowski, "Adaptive evolutionary planner/navigator for mobile robots," IEEE Transactions on Evolutionary Computation, vol. 1, no. 1, Apr. 1997, pp. 18–28, DOI: 10.1109/4235.585889.

[14] C. M. Fernandes and A. C. Rosa, "Evolutionary algorithms with dissortative mating on static and dynamic environments," in Advances in Evolutionary Algorithms. InTech, Nov. 2008, pp. 181–206, ISBN: 978-953-7619-11-4.

[15] L. Winkler and H. Wörn, "Symbricator3D – a distributed simulation environment for modular robots," in Proc. Int. Conf. on Intelligent Robotics and Applications (ICIRA). Springer, Dec. 2009, pp. 1266–1277, ISBN: 978-3-642-10817-4.

[16] A. Sobe, I. Fehérvári, and W. Elmenreich, "FREVO: A tool for evolving and evaluating self-organizing systems," in Proc. Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops (SASOW). IEEE, Sep. 2012, pp. 105–110, ISBN: 978-0-7695-4895-1.

[17] D. Kriesel, "Verteilte, evolutionäre Optimierung von Schwärmen [distributed evolution of swarms]," Master's thesis, Rheinische Friedrich-Wilhelm-Universität Bonn, Mar. 2009, URL: http://www.dkriesel.com/_media/science/diplomarbeit-de-1column-11pt.pdf.

[18] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in Proc. ICRA Workshop on Open Source Software in Robotics, May 2009, URL: http://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf.

[19] A. Bagnato, R. K. Bíró, D. Bonino, C. Pastrone, W. Elmenreich, R. Reiners, M. Schranz, and E. Arnautovic, "Designing swarms of cyber-physical systems: The H2020 CPSwarm project: Invited paper," in Proc. Computing Frontiers Conf. ACM, May 2017, pp. 305–312, ISBN: 978-1-4503-4487-6.

[20] M. Dorigo, V. Trianni, E. Şahin, R. Groß, T. H. Labella, G. Baldassarre, S. Nolfi, J.-L. Deneubourg, F. Mondada, D. Floreano, and L. M. Gambardella, "Evolving self-organizing behaviors for a swarm-bot," Autonomous Robots, vol. 17, no. 2, Sep. 2004, pp. 223–245, DOI: 10.1023/B:AURO.0000033973.24945.f3.

[21] Y. Yao, K. Marchal, and Y. Van de Peer, "Improving the adaptability of simulated evolutionary swarm robots in dynamically changing environments," PLoS ONE, vol. 9, no. 3, Mar. 2014, pp. 1–9, DOI: 10.1371/journal.pone.0090695.

[22] J. H. Holland, Adaptation in Natural and Artificial Systems. MIT Press, Apr. 1992, ISBN: 9780262082136.

[23] I. Rechenberg, Evolutionsstrategie – Optimierung technischer Systeme nach Prinzipien der biologischen Evolution [Evolution strategy – Optimization of technical systems according to the principles of biological evolution]. Fromman-Holzboog, 1973, ISBN: 978-3-772-80373-4.

[24] I. Fehérvári and W. Elmenreich, "Evolution as a tool to design self-organizing systems," in Proc. Int. Workshop on Self-Organizing Systems (IWSOS). Springer, Jan. 2014, pp. 139–144, ISBN: 978-3-642-54140-7.

[25] I. Fehérvári, "On evolving self-organizing technical systems," Ph.D. dissertation, Alpen-Adria-Universität Klagenfurt, Nov. 2013.

[26] A. J. Lockett, "Insights from adversarial fitness functions," in Pro. Conf. on Foundations of Genetic Algorithms (FOGA). ACM, Jan. 2015, pp. 25–39, ISBN: 978-1-4503-3434-1.

[27] D. Floreano and J. Urzelai, "Evolutionary robots with on-line self-organization and behavioral fitness," Neural Networks, vol. 13, no. 4-5, Jun. 2000, pp. 431–443, DOI: 10.1016/S0893-6080(00)00032-0.

[28] G. Francesca, M. Brambilla, A. Brutschy, V. Trianni, and M. Birattari, "AutoMoDe: A novel approach to the automatic design of control software for robot swarms," Swarm Intelligence, vol. 8, no. 2, Jun. 2014, pp. 89–112, DOI: 10.1007/s11721-014-0092-4.

[29] M. Duarte, F. Silva, T. Rodrigues, S. M. Oliveira, and A. L. Christensen, "JBotEvolver: A versatile simulation platform for evolutionary robotics," in Proc. Int. Conf. on the Synthesis and Simulation of Living Systems (ALIFE). MIT Press, Jul. 2014, pp. 210–211, ISBN: 978-0-262-32621-6.

[30] M. Duarte, A. L. Christensen, and S. Oliveira, "Towards artificial evolution of complex behaviors observed in insect colonies," in Proc. Portuguese Conference on Artificial Intelligence (EPIA). Springer, Oct. 2011, pp. 153–167, ISBN: 978-3-642-24769-9.

[31] J. Gomes, P. Urbano, and A. L. Christensen, "Evolution of swarm robotics systems with novelty search," Swarm Intelligence, vol. 7, no. 2-3, Sep. 2013, pp. 115–144, DOI: 10.1007/s11721-013-0081-z.

[32] T. Rodrigues, M. Duarte, S. Oliveira, and A. L. Christensen, "What you choose to see is what you get: an experiment with learnt sensory modulation in a robotic foraging task," in European Conf. on the Applications of Evolutionary Computation (EvoApplications). Springer, Apr. 2014, pp. 789–801, ISBN: 978-3-662-45523-4.

[33] B. Gehlsen and B. Page, "A framework for distributed simulation optimization," in Proc. Winter Simulation Conf. (WSC). ACM, Dec. 2001, pp. 508–514, ISBN: 0-7803-7309-X.

[34] R. A. Halim and M. D. Seck, "The simulation-based multi-objective evolutionary optimization (SIMEON) framework," in Proc. Winter Simulation Conf. (WSC). IEEE, Dec. 2011, pp. 2834–2846, DOI: 10.1109/WSC.2011.6147987.

[35] W. Elmenreich and I. Fehérvári, "Evolving self-organizing cellular automata based on neural network genotypes," in Proc. Int. Workshop on Self-Organizing Systems (IWSOS). Springer, Feb. 2011, pp. 16–25, ISBN: 978-3-642-19167-1.

[36] R. Vaughan, "Massively multiple robot simulations in Stage," Swarm Intelligence, vol. 2, no. 2-4, Dec. 2008, pp. 189–208, DOI: 10.1007/s11721-008-0014-4.

[37] A. Banks and R. Gupta, "MQTT version 3.1.1 plus errata 01," Organization for the Advancement of Structured Information Standards (OASIS), Standard, Dec. 2015, URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[38] P. Saint-Andre, "Extensible messaging and presence protocol (XMPP): Core," Internet Engineering Task Force (IETF), RFC 6120, Oct. 2004, DOI: 10.17487/RFC6120.

[39] A. J. F. Van Rooij, L. C. Jain, and R. P. Johnson, Neural Network Training Using Genetic Algorithms. World Scientific Publishing Co., Mar. 1997, ISBN: 978-9-810-22919-1.

[40] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in Int. Conf. on Intelligent Robots and Systems (IROS). IEEE/RSJ, Sep. 2004, pp. 2149–2154, ISBN: 0-7803-8463-6.

[41] M. Freese, S. Singh, F. Ozaki, and N. Matsuhira, "Virtual robot experimentation platform V-REP: A versatile 3d robot simulator," in Proc. Int. Conf. on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR). Springer, Nov. 2010, pp. 51–62, ISBN: 978-3-642-17319-6.

[42] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, T. Stirling, A. Gutirrez, L. Maria Gambardella, and M. Dorigo, "ARGoS: A modular, multi-engine simulator for heterogeneous swarm robotics," in Proc. Int. Conf. on Intelligent Robots and Systems (IROS). IEEE, Sep. 2011, pp. 5027–5034, ISBN: 978-1-61284-456-5.

[43] A. Driss, L. Krichen, F. Mohamed, and L. Fourati, "Simulation tools, environments and frameworks for UAV systems performance analysis," in Proc. Int. Wireless Communications and Mobile Computing Conf. (IWCMC). IEEE, Jun. 2018, pp. 1495–1500, ISBN: 978-1-5386-2070-0.

[44] F. M. Noori, D. Portugal, R. P Rocha, and M. Couceiro, "On 3D simulators for multi-robot systems in ROS: MORSE or Gazebo?" in Proc. Int. Symposium on Safety, Security and Rescue Robotics (SSRR). IEEE, Oct. 2017, pp. 19–24, ISBN: 978-1-5386-3923-8.

[45] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," Int. J. Computer Science and Network Security (IJCSNS), vol. 17, no. 3, Mar. 2017, pp. 228–235, ISSN: 1738-7906.

[46] F. Soppelsa and C. Kaewkasi, Native Docker Clustering with Swarm. Packt Publishing, Dec. 2016, ISBN: 978-1-786-46975-5.

[47] D. K. Rensin, Kubernetes - Scheduling the Future at Cloud Scale. O'Reilly, Sep. 2015, ISBN: 978-1-492-04871-8.