Distributed Simulation for Evolutionary Design of Swarms of Cyber-Physical Systems

Micha Rappaport, Melanie Schranz
Lakeside Labs GmbH
Klagenfurt, Austria
Email: lastname@lakeside-labs.com

Davide Conzon, Enrico Ferrera
Pervasive Technologies
Istituto Superiore Mario Boella
Torino, Italy
Email: lastname@ismb.it

Midhat Jdeed, Wilfried Elmenreich
Institue of Networked and Embedded Systems
Alpen-Adria-Universität Klagenfurt
Klagenfurt, Austria
Email: firstname.lastname@aau.at

***Abstract*—Swarms of Cyber-Physical Systems (CPSs) 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 complex system that is extremely hard to design manually. Therefore, an automated process is required for designing the local interactions between the agents that lead to the desired swarm behavior. In this work, the authors employ evolutionary design methodologies to generate the local controllers of the agents. This requires many simulation runs and, as a consequence, distributed simulation. The paper first proposes a network-based Application Programming Interface (API) that employs a publish / subscribe broker architecture to distribute simulations among multiple Simulation Servers (SSs). Following this, a file-based API is proposed, which exports the agent controller to the simulator enabling deployment of the evolved solution on CPSs. Both approaches are compared in terms of time needed for the evolutionary optimization process with the support of simulations. A proof of concept demonstrates the portability to CPSs using TurtleBot robots. The results suggest that for most scenarios it is beneficial to export the agent controller to the simulator to avoid the vast communication overhead. The presented network-based approach currently lacks this feature but is well suited to offload computation-heavy simulations to a cluster of SSs.**

Keywords–Swarms; Evolution; Optimization; Cyber-Physical Systems (CPSs); Simulation; Architecture; Robot Operating System (ROS).

I. INTRODUCTION

Over the last decade, the phenomenon of self-organizing systems has gained significant traction in the research community, being observed in disciplines as diverse as physics and biology. Inspired by nature, swarm robotics is also seeing increased interest. On the one hand, coordinating multi-robot systems using swarm approaches offers many opportunities, such as self-organization, self-learning and self-reassembly [1]. On the other hand, it necessitates the difficult process of designing the individual agents to achieve the desired swarm behavior.

Designing swarms of Cyber-Physical Systems (CPSs) poses two main challenges. First, selecting the hardware that best suits the requirements of the swarm (see [2]–[5] for a further examination of this problem), and second, designing the control algorithm defining the behavior of the individual swarm agents. This paper focuses on the latter problem because many platforms for swarm research already exist, e.g., Spiderino [6] and Colias [7].

Approaches for designing local controllers of swarm agents, or more generally self-organizing systems [8], can be categorized into two approaches. First, hierarchical top-down design starting from the desired global behavior of the swarm and second, bottom-up design by defining the swarm agents and observing the resulting global behavior [9]. The design using either approach is still a difficult process as neither can predict the resulting swarm behavior based on the complex interactions between the agents [10]. This is especially true in dynamic environments. Evolutionary methods can be used to tackle such design challenges.

In this paper, we 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 [11]. In the context of swarm robotics, a solution refers to a control algorithm of individual agents that is gradually improved during the optimization process. As experiments with real robots require an extensive amout of time, such methods typically employ accurate and fast simulation to evaluate the performance of candidate solutions in the evolutionary process [12]. The evaluation of algorithms in evolutionary optimzation can be easily executed in parallel, which is for example supported in the FRamework for EVOlutionary design (FREVO) [13] by using multiple cores on the same machine. A further step would be the distribution of evolutionary optimization with a client-server-protocol, as exemplified by Kriesel [14]. This work introduces an architecture for parallel distributed simulations on remote Simulation Servers (SSs) and shows how the resulting agent controllers can be deployed on actual Robot Operating System (ROS)-based hardware platforms using TurtleBots [15]. Finally, the paper describes a performance analysis of the presented implementation.

The paper is organized as follows: In Section II, the evolutionary approach for designing swarms is reviewed. Section III introduces the proposed architecture and two implementations are described in Section IV. The performance of the different approaches is analyzed in Section V. Section VI provides a discussion 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 [16], as well as adaptive self-organization [17]. Both issues are not easy to handle, especially in changing environments and with dynamic interactions among individual agents of a system or a swarm.

Designing a swarm by using evolution is an automatic design method that creates an intended swarm behavior in a bottom up process starting form very small interacting components. This process modifies potential solutions until

 ISBN: 978-1-61208-610-1

a satisfying result is achieved. Such an evolutionary design approach is based on evolutionary computation techniques [18] [19] and mimics the Darwinian principle [20]. It describes the process of natural selection by recombining the most proper solutions to a defined problem. Evolution can be done either on individual or on swarm level. Typically, the process of evolving a behavior starts with the generation of a random population of individual behaviors. Each of these individual swarm-level behaviors is evaluated, typically through simulations. This evaluation is performed by a fitness function that allows to rank the behavior's performance. The higher a behavior is ranked, the higher is the chance for the behavior to be modified with genetic operators, like cross-over or mutation, to form a next generation of agent behaviors. These serve as input for the next iteration. Finally, through multiple iterations an agent behavior is evolved that exhibits the desired global swarm behavior.

Nevertheless, designing by evolution poses several challenges, including no guaranteed, predictable convergence, complex data structures, and the high costs of evolutionary computation itself.

Design by evolution asks for several tasks a designer must face during designing a system model. Adapted from Fehervari and Elmenreich [21], we distinguish six tasks: (i) The *problem description* gives a high abstracted vision of the problem. This includes constraints and the desired objectives for such a problem. (ii) The *simulation setup* transfers the problem description into an abstracted problem model. This model specifies the system components, i.e., details about the agents and the environment. (iii) The *interaction interface* defines the interactions among agents and their interactions with the environment. For instance, the agents sensors and actuators as well as the communication protocols should be specified here. (iv) The *evolvable decision unit* represents the agent controller and is responsible for achieving the desired objectives, i.e., the global behavior of a swarm to achieve a common goal. Such a decision unit must be evolvable to allow genetic operations as cross over or 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, HebbNets, or Neuroevolution of Augmenting Topologies (NEAT) ANNs [22]. (v) The *search algorithm* performs the optimization using evolutionary algorithms by applying the results from the above steps. During this task, an iterative mathematical model will be used to find the optimal solution. The optimization result is dependent on the fitness function of the problem. (vi) 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 highly depends on the problem description. The main purpose of this function is to guide the search algorithm to find the best solution.

This paper describes how the evolutionary design process is performed using the architecture proposed in the EU H2020 CPSwarm project [23]. In this architecture, the Algorithm Optimization Environment (AOE) is responsible for generating the individual agent controllers that lead to the desired global swarm behavior. This architecture is described in detail in the next section.

III. ARCHITECTURE

The following requirements exist for the design of the AOE:

- Multiple SSs, even remotely located, offer simulation capabilities to the Optimization Tool (OT) through a broker.

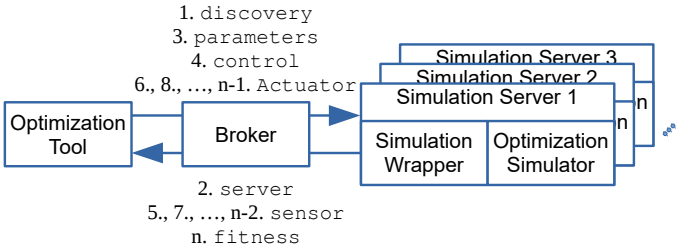

Figure 1. Network-based API.

- Each SS offers one or more Optimization Simulators (OSs).
- An OS exhibits certain characteristics but is also configurable to some extent by the OT.
- Candidate controllers of one generation can be evaluated in parallel.
- The OT can respond to requests from the OS at any point in time.

To fulfill these requirements, the AOE consists of two components: The OT, which is responsible for evolving candidate controllers using the mechanisms explained in Section II and the OS, which evaluates the behavior of each candidate through simulation. These two components are interconnected to each other through a set of interfaces called the Simulator Application Programming Interface (API), which allows them to communicate during the optimization process. Employing an OS as opposed to OT internal simulations gives the opportunity to build on well established simulators that support accurate simulation of swarms of CPSs with different levels of detail.

This section introduces two different approaches for the Simulator API. The first one leverages network socket-based inter-process communication to allow multiple simulations to be remotely run in parallel OSs. Since executing such a large number of simulation runs requires a significant amount of time, parallel distributed execution enhances the scalability and performance of the optimization process. The second approach is a file system-based inter-process communication technique which hands over full control to the OS. This approach requires no further communication between the tools and is therefore well-suited for deploying the generated candidate controllers on CPSs.

A. Network-based Approach

The network-based approach aims to improve scalability and performance through parallelization of simulations during the optimization process. This is achieved from two sides: on OT side, the candidates belonging to the same generation are evaluated in parallel using multi-threading. Each thread uses a different OS to perform the required simulation. The OSs are run in parallel, possibly on different, remote SSs.

The network communication is managed by a broker that offers a publish / subscribe infrastructure to the subscribing clients: OT and SS. Figure 1 gives an overview of the functional architecture of this approach, indicating the messages exchanged, numbered by the order in which they are sent (see Section IV for more details). The SS is decoupled from the OT via the broker. In this way, every SS can be on a dedicated machine with the hardware requirements needed to execute the simulations.

Every SS offers one or more OSs that communicate with the broker through a Simulation Wrapper (SW). The SW is a

 ISBN: 978-1-61208-610-1

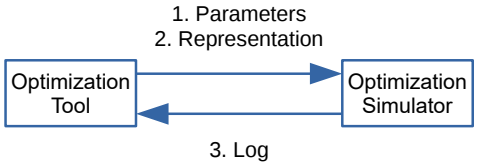

Figure 2. File-based approachAPI.

software layer installed on the SS that implements the Simulator API and acts as a client connecting to the broker. This allows the OT to communicate with the SS without knowing what type of OS is actually used. 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. When the fitness is returned from every SS, the OT can perform the evolutionary steps to create the candidates for the next generation.

Several SSs can work in parallel to reduce the time required to complete the simulations in one generation of the evolutionary optimization. Through the SW, different OSs can be employed, also if the OSs are heterogeneous among each other. Importantly, a SS can be used by several OT instances, but only by one at a time.

B. File-based Approach

Figure 2 shows the file-based approach and the files exchanged, numbered by the order in which they are sent. This solution aims to reduce the communication between OT and OS by passing a generated candidate controller as a file from the OT to the OS. The following three files are passed between the tools:

- Parameters: The parameters that need to be transferred from the OT to the OS to setup the OS.
- Representation: The representation of the candidate controller exported by the OT. The OS implementation includes this source code file to enable the agents in the simulation making decisions by translating the sensor readings to actuator commands.
- Log: Every agent in the simulation generates a log file containing the metrics for measuring the performance of a candidate. The log files are used by the fitness function to calculate the fitness of a candidate controller.

IV. Implementation

A set of existing tools was extended to implement the architecture presented in the previous section. These tools are interconnected using the two approaches of the Simulator API, the network-based approach and the file-based approach. This section first introduces the concepts and tools used for the development of the proposed solutions and then details the specifics of the Simulator API.

FREVO is selected as the OT since it is a very modular optimization tool that satisfies the principles addressed in Section II [13].

The current implementation of FREVO relies on simulations implemented in the problem component. This implies that each problem needs an implementation of the simulation including models of the agents. This can be avoided by using state-of-the-art robotics simulators that build on standard models. The integration of these tool into the CPSwarm architecture requires the current implementation to be improved, enabling it to use multiple SSs, as addressed in the list of requirements in Section III.

A. Network-based Implementation

As mentioned in Section III-A, the network-based approach is implemented using a broker architecture. The broker employs the Message Queue Telemetry Transport (MQTT) protocol [24] which is based on the publish / subscribe paradigm. In recent times, this solution has been recognized as the de-facto standard for event-driven architectures in the Internet of Things (IoT) domain. MQTT has been chosen 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. Therefore, both FREVO and the SW implement a client for the MQTT protocol.

FREVO implements the client as a helper class called `simMQTT` that contains the MQTT callbacks for receiving messages from the broker. The class is instantiated by the problem component in FREVO that evaluates a candidate controller through simulation. The `simMQTT` class handles all the communication with the broker.

As described above, the SS consists of an OS and a SW. The OS evaluates a specific candidate controller in the optimization process. The SW serves as client that handles the connection to the MQTT broker. The SW is implemented as a Java library. It embeds the MQTT Paho client [25] for MQTT communication. The library exports an abstract class called `SimulationWrapper`, which implements the behavior that is common to all the simulators. It provides a set of API functions to be used by the SS to handle the messages received from and to sent to FREVO. To test the implementation, the `SimulationWrapper` has been integrated with a very basic Java simulator called *Minisim*, which is a command-line, multi-agent simulator simulating a capture-the-flag game with multiple robots on a two-dimensional grid. *Minisim* has been specifically developed for testing the network communication between FREVO and the SS [26].

The messages exchanged between FREVO and the SS through the Simulator API are explained in the following. When FREVO needs to evaluate a candidate, it queries for available SSs by sending a `discovery` message containing the requirements for the OS. Every SS that has an OS fulfilling these requirements and has enough resources available answers the request with a `server` message. The server message contains the OS capabilities and ID. FREVO selects a suitable SS and initiates the simulation by sending a `parameters` message to setup the OS followed by a `control` message to start the simulation. The `parameters` message contains the parameters that describe the models as well as the necessary configuration parameters for the simulation environment. As a direct reaction to the `control` message, the addressed OS starts the simulation with the model parameters received earlier and publishes the `sensor` messages of the first simulation time step. The `sensor` message transmits the sensor readings of one agent to FREVO, which uses this information to compute the next actuator commands for this agent and replies with an `actuator` message to transmit the corresponding actuator commands. The process continues until the end of the simulation is reached. The `fitness` message is the final message of a simulation run, calculated by the SS once the maximum number of simulation steps has been reached. Every message contains a server ID and a simulation hash that can be used to uniquely identifying the OS and the simulation it is running.

 ISBN: 978-1-61208-610-1

B. File-based Implementation

With the file-based approach, FREVO communicates with the OS through the file system. The OS is based on ROS, a middleware that can control CPSs in simulation and on physical hardware. Hence, all simulators that are compatible with ROS can be used with this implementation. Two very popular simulators, namely Stage [27], a low-fidelity two-dimensional robot simulator and Gazebo [28], a high-fidelity three-dimensional robot simulator have been tested successfully with this implementation.

To perform the ROS simulations, FREVO first exports the candidate representation of the agent controller and problem specific parameters into the ROS workspace. Then FREVO executes a script that compiles and runs the simulation. When the simulation terminates, FREVO reads the log files created by ROS to compute the fitness of the simulated candidate. The API is implemented in the helper class `simROS` of FREVO, allowing every problem component to access ROS.

The three files described in the previous section are implemented as follows:

- Parameters: The parameters that need to be transferred from FREVO are written into parameter files in the YAML Ain't Markup Language (YAML) [29] format that is used by ROS. These are problem specific parameters such as description of the agents in terms of hardware or positioning.
- Representation: The candidate controller is represented as a fully meshed ANN. It is exported by FREVO into a C source code file. ROS includes this file into the source code of the package that implements the agent behavior. Once this file is exported to ROS, a recompilation of the ROS package becomes necessary.
- Log: The performance of a candidate is measured by performance metrics defined in FREVO. The simulator measures the metrics and writes them to log files in text format. FREVO reads these log files and applies the fitness function to calculate the fitness of a candidate controller.

This implementation uses a simple multi agent simulation called *EmergencyExit* [26]. On one hand, this implementation enables the communication between FREVO and ROS for evolving a controller using ROS-based simulations. On the other hand it allows the evolved result, i.e. the ANN, to be exported from FREVO and run in ROS standalone on actual CPS. As a proof of concept, an evolved ANN is used to guide TurtleBot robots in Gazebo simulations and in real world experiments.

V. Performance Analysis

The previous sections presented two different approaches for distributed simulation during the optimization process. The main difference between the presented approaches is that with the network-based approach the agent controllers reside within the OT FREVO and communicate with the simulator by exchanging messages, whereas with the file-based approach the agent controller is exported to the simulator and communication takes place only at the beginning and the end of the simulation. This section compares both approaches in terms of scalability. The relevant parameters for this analysis are the number of parallel threads n_{threads}, the number of agents n_{agents}, and the length of a simulation in terms of steps n_{steps}. First, a theoretical comparison is performed that is complemented by simulation results using FREVO with the above mentioned *Minisim* and *EmergencyExit* simulations. The performance is measured in time to perform a complete optimization.

A. Theoretical Comparison

For the first part of this analysis the approaches are abstracted to represent the different locations where the agent controller can reside. This is either internal within the OT (network-based implementation) or external within the simulator (file-based implementation).

In the evolutionary optimization process, there is a population of n_{pop} candidate controllers that can be evaluated in parallel. They are evaluated through simulation consisting of n_{step} steps. When all candidates have been evaluated, a new generation is created using evolutionary operators. The total number of generations is n_{gen}. This results in a total number of simulations for a complete optimization of

$$n_{\text{sim}} = n_{\text{gen}} \cdot n_{\text{pop}} \cdot n_{\text{eval}} \tag{1}$$

where each candidate is evaluated in n_{eval} simulations. When simulations are parallelized on n_{threads}, the number of simulations that need to be performed sequentially results to

$$n_{\text{sim}} = \frac{n_{\text{gen}} \cdot n_{\text{pop}} \cdot n_{\text{eval}}}{n_{\text{threads}}} \tag{2}$$

given that n_{thread} is within the range $[1, n_{\text{pop}} \cdot n_{\text{eval}}]$.

In general, the time t_{opt} needed for one optimization run consists of the time needed for performing the evolutionary calculations t_{evo}, the time needed for performing the simulations t_{sim}, and the overhead t_{overhead}

$$t_{\text{opt}} = n_{\text{gen}} \cdot t_{\text{evo}} + n_{\text{sim}} \cdot (t_{\text{sim}} + t_{\text{overhead}}) \tag{3}$$

where $t_{\text{sim}} = n_{\text{step}} \cdot t_{\text{step}}$. The difference between both approaches lies in the overhead t_{overhead}.

With the OT internal controller, the number of messages that need to be exchanged between the OT and the simulator depends on the number of agents n_{agent} in the simulation. The `discovery`, `server`, `parameters`, `control`, and `fitness` messages are only transmitted once for every simulation. The `sensor` and `actuator` messages are transmitted in every simulation step for every agent. Therefore, the number of messages exchanged in each simulation is

$$n_{\text{msg}} = 5 + 2 \cdot n_{\text{step}} \cdot n_{\text{agent}} \tag{4}$$

which gives us an overhead based on the communication time between the OT and the simulator:

$$t_{\text{overhead,int}} = (5 + 2 \cdot n_{\text{step}} \cdot n_{\text{agent}}) \cdot t_{\text{msg}}. \tag{5}$$

For the external controller, the overhead is

$$t_{\text{overhead,ext}} = t_{\text{export}} + t_{\text{compile}} + t_{\text{fitness}} \tag{6}$$

based on the time needed for exporting representation and simulation parameters t_{export}, the time needed for recompiling the simulator t_{compile}, and the time for reading the log files and computing the fitness t_{fitness}.

The total optimization time of both approaches is therefore

$$t_{\text{opt,int}} = n_{\text{gen}} \cdot t_{\text{evo}} + \frac{n_{\text{gen}} \cdot n_{\text{pop}} \cdot n_{\text{eval}}}{n_{\text{threads}}} \cdot (n_{\text{step}} \cdot t_{\text{step}} + (5 + 2 \cdot n_{\text{step}} \cdot n_{\text{agent}}) \cdot t_{\text{msg}}) \tag{7}$$

$$t_{\text{opt,ext}} = n_{\text{gen}} \cdot t_{\text{evo}} + \frac{n_{\text{gen}} \cdot n_{\text{pop}} \cdot n_{\text{eval}}}{n_{\text{threads}}} \cdot (n_{\text{step}} \cdot t_{\text{step}} + t_{\text{export}} + t_{\text{compile}} + t_{\text{fitness}}). \tag{8}$$

 ISBN: 978-1-61208-610-1

TABLE I. Optimization parameters measured through simulations.

parameter	value
n_{gen}	200
n_{pop}	50
n_{eval}	1
t_{evo}	12 ms
t_{msg}	30 ms
t_{export}	5.35 ms
t_{compile}	8833 ms
t_{fitness}	0.69 ms
t_{step}	100 ms

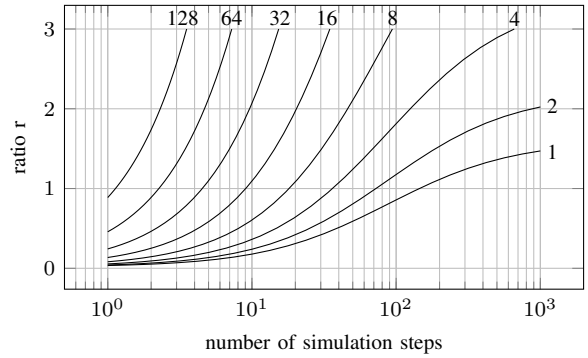

Figure 3. Ratio of optimization times between OT internal agent controller and OT external controller for different numbers of agents.

Table I shows specific parameters and execution times measured on a dual Intel Xeon X5675 3.07 GHz system with 16 GB memory and 12 cores in total, supporting up to 24 threads with hyper-threading. The operating system is Debian 9. The evolutionary parameters were chosen to yield good results. The times are measurements of the MQTT broker implementation and the ROS-based implementation.

Using these values with (7) and (8) the specific optimization times are

$$t_{\text{opt,int}} = 2.4\,\text{s}\left(1 + \frac{250}{n_{\text{thread}}}\left(2.5 + n_{\text{step}}\left(1.67 + n_{\text{agent}}\right)\right)\right) \tag{9}$$

$$t_{\text{opt,ext}} = 2.4\,\text{s}\left(1 + \frac{416.67}{n_{\text{thread}}}\left(88.39 + n_{\text{step}}\right)\right). \tag{10}$$

To decide whether to run optimization with internal or external agent controller, the ratio between both optimization times is a suitable metric.

Figure 3 shows the ratio $r = \frac{t_{\text{op,int}}}{t_{\text{opt,ext}}}$. A value of $r > 1$ means that the external approach performs better whereas a value of $r < 1$ means that the internal approach is favorable. As both approaches can use parallelization, the resultant ratio is independent of the number of parallel threads used. It can be seen that for most cases the controller should reside externally within the simulator. This is due to the fact that if all agent controllers are executed in a single tool it creates a bottleneck situation. This is not so crucial for small swarms but already for a swarm size of eight agents, the optimization with internal control takes longer when simulations last more than 18 steps.

B. Scalability of the network-based approach

This study analyzes the scalability of the network-based approach where the agent controller resides locally within FREVO. The optimization is performed with a population size

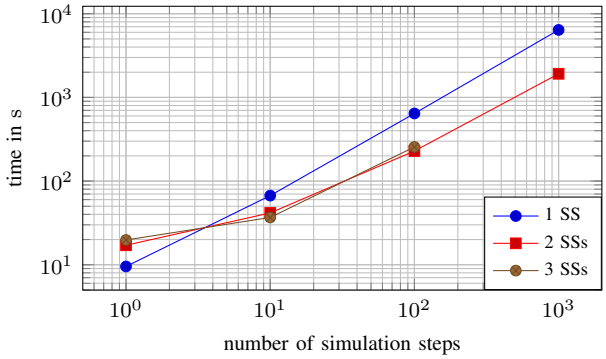

Figure 4. Optimization time using the network-based approach for varying number of SSs and 8 agents.

of $n_{\text{pop}} = 4$ and $n_{\text{gen}} = 4$ generations. Figure 4 shows the time needed for optimization with $n_{\text{agent}} = 8$ agents. As expected, the time needed for a complete optimization run scales linearly with the length of a single simulation. Only for extremely short simulations the overhead introduced from discovering available SSs renders the parallelism pointless. In fact, the discovery process limits the scalability of the proposed implementation. As the broker is the central point that enables the discovery, the optimization time does not scale for more than two SSs, because the `discovery` and `server` messages already account for most of the traffic. Therefore, the discovery procedure needs improvements as described in Section VI. Nevertheless, using two SSs rather than one speeds up the optimization significantly. The speed-up continues to increase as the number of simulation steps increases as longer simulations decrease the relative time spent in the discovery phase.

VI. Conclusion and Future Work

This paper presents a solution for the evolutionary design of swarms of CPSs based on remote simulation tools. The architecture designed for this solution is composed of three main components: The OT that is responsible for evolving candidate controllers, the OS evaluating the behavior of each candidate through simulation, and the Simulator API that connects the OT and the OS. For the latter component, two different approaches and related implementations are presented: A broker-based approach, which parallelizes the simulations of the optimization process to improve scalability and performance, and a file-based approach, which is used to prove the compatibility with CPSs.

A performance analysis shows that the broker-based approach does not scale well for more than two SSs because of network congestion at the OT when it discovers the SSs. Therefore, the current implementation cannot exploit the full potential of this approach. Furthermore, in this approach all agent controllers are executed in a single OT which creates a bottleneck. Hence, this implementation of the broker-based approach is suitable for simple simulations. This could be either short simulations with few time steps or simulations with only a small number of agents, e.g., swarms of up to eight agents and up to 18 time steps. The file-based approach is a viable alternative where the agent controller is exported to the OS and is especially suited for large swarms of robots as the messaging overhead is drastically reduced. In any case, the broker-based solution is the better choice if simulations are performed where OT and OS are not located on the same

 ISBN: 978-1-61208-610-1

machine.

Considering this, further improvements could be made to increase the performance of the broker-based approach. First, the optimization process could be split into two phases. In the first phase, a handshake between OT and SS would take place, where the OT would discover the SSs fulfilling the requirements and would select one of them for simulations. In the second phase, the OT would perform the optimization and would execute the simulations using the selected SSs. This approach is expected to reduce the number of `discovery` and `server` messages and to avoid the congestions that inhibit the scalability with more than two SSs. Second, the controller candidate represented as ANN could be exported to the SS as done in the file-based approach. In this way, it would be possible to avoid the use of `sensor` and `actuator` messages, reducing the overall number of messages exchanged and the time required to complete the optimization drastically. The drawback of the file-based implementation of this approach is the time required for recompiling the simulator code to include the ANN source code. This could be avoided by creating a generic ANN wrapper that would only read the ANN parameters from a configuration file avoiding the need for recompilation. Hence, a combination of both the presented approaches is needed for a distributed and scalable network architecture that can support the large amount of simulations during the optimization process.

ACKNOWLEDGMENT

We thank Robotnik Automation S.L.L. for porting the implementation to TurtleBot robots and the Gazebo simulator. We also thank Arthur Pitman for proofreading the text. 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. Patil, T. Abukhalil, S. Patel, and T. Sobh, "UB robot swarm – design, implementation, and power management," in Proc. 12th IEEE International Conference on Control and Automation (ICCA), Jun. 2016, pp. 577–582, ISBN: 978-1-5090-1738-6.

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

[3] R. Goldsmith, "Real world hardware evolution: A mobile platform for sensor evolution," in Proc. 5th International Conference on Evolvable Systems: From Biology to Hardware (ICES), Mar. 2003, pp. 355–364, ISBN: 978-3-540-36553-2.

[4] J. Bongard, "Exploiting multiple robots to accelerate self-modeling," in Proc. 9th Annual Conference on Genetic and Evolutionary Computation (GECCO), Jul. 2007, pp. 214–221, ISBN: 978-1-59593-697-4.

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

[6] M. Jdeed, S. Zhevzhyk, F. Steinkellner, and W. Elmenreich, "Spiderino – A low-cost robot for swarm research and educational purposes," in 13th Workshop on Intelligent Solutions in Embedded Systems (WISES), Jun. 2017, pp. 35–39, ISBN: 978-1-5386-1157-9.

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

[8] W. Elmenreich, R. D'Souza, C. Bettstetter, and H. de Meer, "A survey of models and design methods for self-organizing networked systems," in Proc. 4th International Workshop on Self-Organizing Systems (IWSOS), Dec. 2009, pp. 37–49, ISBN: 978-3-642-10865-5.

[9] 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, ISSN: 1573-7527.

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

[11] 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.

[12] L. Winkler and H. Wörn, "Symbricator3D – A distributed simulation environment for modular robots," in Proc. 2nd International Conference on Intelligent Robotics and Applications (ICIRA), Dec. 2009, pp. 1266–1277, ISBN: 978-3-642-10817-4.

[13] A. Sobe, I. Fehervari, and W. Elmenreich, "FREVO: A tool for evolving and evaluating self-organizing systems," in Proc. 1st International Workshop on Evaluation for Self-Adaptive and Self-Organizing Systems (Eval4SASO), Sep. 2012, pp. 105–110, ISBN: 978-0-7695-4895-1.

[14] D. Kriesel, "Verteilte, evolutionäre Optimierung von Schwärmen [Distributed, evolutionary optimization of swarms]," Diplomarbeit, Rheinische Friedrich-Wilhelm-Universität Bonn, Mar. 2009.

[15] Open Source Robotics Foundation, Inc., "Turtlebot," http://www.turtlebot.com/, accessed: 2017-12-07.

[16] M. Dorigo et al., "Evolving self-organizing behaviors for a swarm-bot," Autonomous Robots, vol. 17, no. 2, Sep. 2004, pp. 223–245, ISSN: 1573-7527.

[17] 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.

[18] J. H. Holland, Adaptation in Natural and Artificial Systems. Cambridge, MA, USA: MIT Press, Mar. 1992, ISBN: 9780262082136.

[19] I. Rechenberg, Evolutionsstrategie – Optimierung technischer Systeme nach Prinzipien der biologischen Evolution [Evolution strategy – Optimization of technical systems according to the principles of biological evolution]. Stuttgart, Germany: Fromman-Holzboog, 1973, ISBN: 978-3772803734.

[20] 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, ISSN: 1935-3820.

[21] I. Fehervari and W. Elmenreich, "Evolution as a tool to design self-organizing systems," in Revised Selected Papers from 7th IFIP TC 6 International Workshop on Self-Organizing Systems (IWSOS), Jan. 2014, pp. 139–144, ISBN: 978-3-642-54140-7.

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

[23] J. Liang et al., "D3.1 – initial system architecture & design specification," EU H2020 CPSwarm Consortium, Public deliverable, Aug. 2017.

[24] "Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1," Geneva, Switzerland, Standard ISO/IEC 20922:2016, Jun. 2006.

[25] "Eclipse paho," https://www.eclipse.org/paho/, accessed: 2017-12-07.

[26] M. Rappaport, D. Conzon, and E. Ferrera, "D6.1 – initial simulation environment," EU H2020 CPSwarm Consortium, Public deliverable, Oct. 2017.

[27] R. Vaughan, "Massively multiple robot simulations in stage," Swarm Intelligence, vol. 2, no. 2-4, Dec. 2008, pp. 189–208, ISSN: 1935-3820.

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

[29] "YAML Ain't Markup Language (YAML)," http://www.yaml.org/spec/, Specification Version 1.2, Sep. 2009.

 ISBN: 978-1-61208-610-1