Autonomous Agents and Multi-Agent Systems (2021) 35:10
https://doi.org/10.1007/s10458-020-09489-0

Enabling scalable and fault-tolerant multi-agent systems by utilizing cloud-native computing

Stefan Dähling[1] · **Lukas Razik**[2] · **Antonello Monti**[1]

Accepted: 17 December 2020 / Published online: 25 January 2021

Abstract
Multi-agent systems (MAS) represent a distributed computing paradigm well suited to tackle today's challenges in the field of the Internet of Things (IoT). Both share many similarities such as the interconnection of distributed devices and their cooperation. The combination of MAS and IoT would allow the transfer of the experience gained in MAS research to the broader range of IoT applications. The key enabler for utilizing MAS in the IoT is the ability to build large-scale and fault-tolerant MASs since IoT concepts comprise possibly thousands or even millions of devices. However, well known multi-agent platforms (MAP), e. g., Java Agent DE-velopment Framework (JADE), are not able to deal with these challenges. To this aim, we present a cloud-native Multi-Agent Platform (cloneMAP) as a modern MAP based on cloud-computing techniques to enable scalability and fault-tolerance. A microservice architecture is used to implement it in a distributed way utilizing the open-source container orchestration system Kubernetes. Thereby, bottlenecks and single-points of failure are conceptually avoided. A comparison with JADE via relevant performance metrics indicates the massively improved scalability. Furthermore, the implementation of a large-scale use case verifies cloneMAP's suitability for IoT applications. This leads to the conclusion that cloneMAP extends the range of possible MAS applications and enables the integration with IoT concepts.

Keywords Cloud computing · Multi-agent systems · Internet of things

✉ Stefan Dähling
sdaehling@eonerc.rwth-aachen.de

Lukas Razik
lrazik@eonerc.rwth-aachen.de

Antonello Monti
amonti@eonerc.rwth-aachen.de

[1] Institute for Automation of Complex Power Systems, E.ON Energy Research Center, RWTH Aachen University, Mathieustraße 10, 52074 Aachen, Germany

[2] Energy Systems Engineering (IEK-10), Jülich Research Centre, Jülich, Germany

1 Introduction

Multi-Agent Systems (MAS) have been around for decades and have been applied to various use cases [18]. They allow for a distributed, robust and scalable design of applications in the field of, e. g., simulation, monitoring and control. For the implementation and execution of MASs many different Multi-Agent Platforms (MAP) have been proposed. One of the most popular MAPs is the Java Agent DEvelopment Framework (JADE) [20].

Besides the wide-spread use of MASs in the scientific community, only few real world use cases adopt MAS based approaches [40]. Moreover, these applications often comprise only a small number of agents. Additionally, literature shows that MAPs do not offer good performance when it comes to large-scale and distributed MAS applications. Besides, such platforms have single-points of failure, which may prevent them from being used in critical real world applications [21].

Cloud computing offers techniques to achieve high scalability and fault-tolerance, i. e., the recovery from hardware or software failures. It allows for an efficient, easy and on-demand provisioning of resources on infrastructure, platform and application layer. Orchestration tools such as *Kubernetes* in conjunction with container technology allow to compose distributed applications consisting of many distinct components, often referred to as *microservices* [15].

MAS and cloud computing, have the potential to complement each other. While cloud computing enables resource provision and management on platform level, MAS constitutes a paradigm for a distributed application design utilizing possibly distributed resources.

The challenges in exploiting this potential synergy are the following. First, an additional layer between the cloud platform level (often container orchestration) and the MAS application is required for managing the MAS and translating its requirements to the underlying orchestration tool. Second, it has to implement standardized tools for MAS developers, such as an agent management system (AMS) or the directory facilitator (DF) [14]. As a key feature of MASs, messaging among agents has to be enabled while hiding the underlying complexity of the network.

An emerging field of application for large-scale MASs is the Internet of Things (IoT). In fact these approaches show strong similarities. Geographically distributed devices, or *things*, are connected to the Internet and therefore able to communicate. However, many different frameworks for IoT-devices exist, making interoperability difficult. Furthermore, IoT devices often have restricted hardware capabilities limiting the execution of complex applications. Both, MAS and IoT, consist of distributed autonomous entities able to exchange information. Given the tremendous amount of research work done in the field of MASs, the IoT can benefit from that experience [36].

This work proposes a cloud-native Multi-Agent Platform (cloneMAP). cloneMAP is designed to tackle the aforementioned challenges. Its design goals are high scalability and fault-tolerance. As a result, cloneMAP is an enabler for the combination of MAS and the IoT. The requirements for this platform are derived from a review of literature, identifying the shortcomings of existing platforms. Additionally, a specific use case, called SwarmGrid-X, in the field of electrical power grids, motivated the development of cloneMAP. Similar to the IoT in general, SwarmGrid-X introduces the requirements of scalability and fault-tolerance. Nevertheless, cloneMAP is a general-purpose MAP and independent of a specific use case. Therefore, our contributions are:

- Conceptualization and implementation of a MAP based on a microservice architecture,

- Scalability and fault-tolerance to prevent bottlenecks and single-points of failure through the adoption of cloud computing techniques, and
- Integration of an interface between MAS and the IoT.

All three contributions open up a promising field of application. We do not aim at proposing new implementation concepts for cloud computing in general but rather to apply established concepts to the field of MASs. In order to allow researches and MAS developers to benefit from our contributions, cloneMAP is available as open-source software.[1]

The remainder of this paper is organized as follows. Section 2 gives an overview about the related work. Section 3 introduces the concept of cloneMAP in detail. In Sect. 4 we define metrics for an evaluation of the proposed features of cloneMAP. Section 5 analyzes the results of the defined benchmarks and compares them to the established MAP JADE. Additionally, we implement a large-scale use case on cloneMAP to validate its functionality and features. Finally, Sect. 6 provides a conclusion and an outlook of future work.

2 Literature review

This section reviews the literature regarding MAPs, an integration of MAS and the IoT as well as cloud computing and its potential combination with MAS.

2.1 Multi-agent systems and platforms

MASs are a distributed computing paradigm originated in the 1990s. Multiple software entities, so-called agents, operate individually but in cooperation to solve a global task. An agent is characterized by four main characteristics: autonomy, social ability (i. e. information exchange with other agents), reactivity and pro-activeness [48]. Many MAPs have been developed since then, enabling the implementation of MASs applications. An overview is given in [20]. To the best of our knowledge no MAP is especially designed using cloud computing techniques, especially container orchestration.

The design of MAPs is standardized by the Foundation for Intelligent Physical Agents (FIPA). Figure 1 shows the reference model of a MAP. It contains the AMS which manages the MAS and the agent it contains. All agents are registered with the AMS and can use it for white pages requests. The DF is an optional component that enables service discovery and hence, provides agents with yellow pages functionality. The Message Transport System (MTS) provides agents with the ability to communicate with each other. A FIPA compliant MAP has to follow this architecture and further specifications, e.g., for message construction.

While scientific literature provides a rich overview of different fields of applications of MAS, a broad adoption in industry and production systems seems to be missing [21]. A lack of scalability and fault-tolerance regarding the available MAPs has been identified as main reason for this [21, 40]. In the case of power engineering an IEEE working group has identified possible applications of MAS and provided recommendations on their design [24, 25]. Again, fault-tolerance is reported as one of the main requirements for MAPs. Besides, the working group also highlights the importance of interoperability

[1] https://www.fein-aachen.org/en/projects/clonemap/.

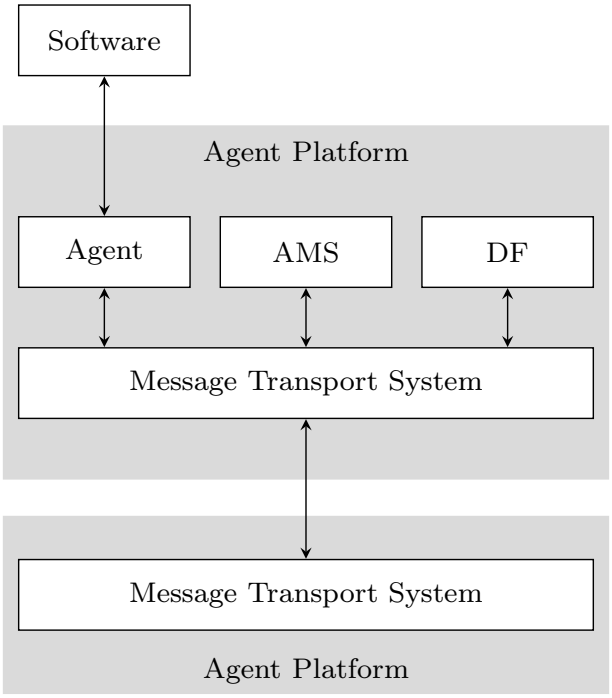

Fig. 1 FIPA reference model [14]

and extensibility. They recommend to achieve these by complying with standards defined by the FIPA [14].

The most-widely used MAP in literature is JADE. It allows the implementation of a MAS through a FIPA compliant middleware [5]. A JADE-based MAS can be deployed across multiple computer hosts.

In [9] the scalability of JADE is discussed. The average round-trip time (av-gRTT) of a ping-pong benchmark is measured for 1 to 1000 sender-receiver agent pairs. The authors conclude a linear scalability just because of a proportional increase of the avgRTT with the number of agent pairs on one single processor core system. Nevertheless, the measurements on two different hosts (i. e. 2 cores) show greater average avgRTT compared to the case of one host. Since no speedup measurements are performed using multiple hosts and processors, no conclusions on the scalability of JADE on a distributed/parallel computing system can be made (with respect to the scalability definition for a multi processor system [34]). Another JADE performance study can be found in [28], performed on a cluster of up to eight nodes. One main outcome is that the DF should be horizontally scalable to avoid bottlenecks.

In [38] a methodology for the evaluation of MAPs with criteria, such as performance measured as avgRTT, for MAP characterization is introduced and applied to eight different MAPs. JADE is identified as the one with the best messaging performance. A comparative analysis of JADE, AgentScape [31] and MadKit [17] is performed in [4], using different performance benchmarks. The results indicate that the messaging performance of JADE is superior to the one of the other MAPs. Further, the centralized

implementation of the DF in JADE and AgentScape is identified as a potential bottleneck. The distributed implementation in MadKit enables higher scalability.

The same authors present a more scalable MAP, called Magentix, that has been developed on Operating System (OS) level [3]. The MAP can be distributed on different machines. The AMS and DF are implemented in a distributed fashion. Each agent is executed as three OS threads, one for sending of messages, receiving of messages and for the agent behavior. A comparison with JADE shows that the messaging performance is improved. However, for a large-scale MASs the number of agents might exceed the number of available Central Processing Units (CPUs). The necessary scheduling of threads performed by the OS might impose performance issues. The performance of Magentix assessed by means of a comparison with JADE. An experiment, similar to the ping-pong benchmark, reveals that Magentix executes roughly six times faster compared to JADE for up to 1000 agents [3].

In [40] fault-tolerance is analyzed as an important performance indicator. Therefore, different levels of fault-tolerance (MAP vs. agent level) are introduced. While a fault on agent level is caused by the agent behavior itself and possibly results in a malfunctioning agent, a fault on platform level is caused by the platform or the hardware the platform is running on. As a result the latter should also be handled by the MAP. One MAP that offers fault-tolerance on platform level is SAGE [1]. Fault-tolerance is achieved by means of a decentralized architecture. The platform service of SAGE can be distributed over multiple hosts. In case one host fails the rest of the platform can still continue its operation. However, the recovery of single agents in case of a failure is not handled by SAGE. The authors of [1] also perform a comparison with JADE. The ping-pong benchmark shows that SAGE performs roughly twice as fast as JADE for up to 400 agents.

Based on the literature review the main requirements for cloneMAP are scalability and fault-tolerance. Scalability refers to the performance of the messaging among agents for large-scale MASs as well as the distributed implementation of platform components, i. e., the AMS and the DF. Fault-tolerance is addressed on platform level, enabling the recovery from faults of platform components. Additionally, cloneMAP supports the MAS developer in implementing fault-tolerance on agent level as well. Section 3 explains in detail how the requirements for cloneMAP are mapped to the architectural design.

In this paper we use JADE as term of comparison for cloneMAP due to its widespread use and because it is typically adopted for comparison by developers of MAPs who claim better scalability (e. g. [1, 3]).

2.2 MAS and the internet of things

In the following, the relation between MASs and the IoT is covered. The IoT consists of devices which can interact with their environment by means of sensors and actors. These devices are equipped with an interface to a common communication network, e. g., the Internet. Devices can then interact with each other or any remote service. Literature provides many examples for possible applications of the IoT [22, 41].

However, simply connecting devices to a common infrastructure does not automatically enable them to interact and cooperate. The MAS paradigm has been proposed to overcome this limitation and to allow for interoperability among devices [16, 29]. Every device is represented by one intelligent agent that is able to process the device's sensor data, interact with other agents, reason about its environment and make decisions on the control of the device's actuators. This architectural model is the *Internet of Agents* [33] or *agent-based*

IoT [36] in literature. A combination of IoT and MAS requires the implementation of large-scale MASs [37]. Other work has been done towards the integration of MAS and IoT [12, 13, 46], dealing with the modeling of the MAS architecture for IoT use cases. However, these examples use JADE for the implementation of MASs, comprising the discussed limitations. These limitations are identified as a major hindrance for the usage of MAS technology in industrial cyber-physical systems [21]. The combination of MAS and IoT requires not only research on the architectural models of MASs, also research on the implementation of agents, i. e., the MAP, is essential. We contribute to this by the presentation of cloneMAP's architecture and the demonstration of its capabilities.

Regarding the implementation of MASs for the IoT, the design choice is between two main concepts: embedding the agents in the devices or executing agents outside the devices, e. g., on a cloud-platform [21, 46]. Additionally, the concept of mobile agents exists [2]. Mobile agents are able to migrate from one hardware to another. Hence, they can move among different IoT devices and execute close to the environment they interact with. However, executing agents on IoT devices, either stationary or mobile, has some disadvantages [35]. IoT devices typically have limited computing resources limiting the complexity of agent algorithms that can be implemented on them. Mobile agents require IoT devices to be able to execute their code which requires a substantial level of interoperability among devices.

We believe that the execution outside the devices on a platform with shared resources has several benefits. Using shared cloud resources (i. e. computational performance, storage, etc.) allows for higher resource demands and a more efficient resource allocation. Further, agents executed outside the devices add another abstraction layer to IoT applications. In this sense, agents are logical entities that can interact with one, multiple or even no device. This way a system consisting of several components that logically belong together can be represented and controlled by a single agent according to its logical functionality. Executing agents directly on IoT devices would make this abstraction more difficult since it implies a fixed mapping between IoT devices and the agents they host. Lastly, the execution of agents in a cloud platform eliminates the need for interoperability directly among IoT devices. The devices only interact with the platform and possibly execute small tasks locally. Based on these considerations we decided to implement cloneMAP as a platform independent from the IoT devices but designed for a cloud environment. A similar perception can be found in [35] where agents are described as microservices running in a cloud environment.

2.3 Cloud computing

The term *cloud computing* appears in many areas besides information technology such as politics, economics, etc. We adopt the National Institute of Standards and Technology (NIST) definition of cloud computing [27]. Here, the cloud computing model has three service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

The IaaS layer deals with the provision of computing resources, typically through VirtualMachines (VMs) [43]. The PaaS layer provides certain services, that can be utilized for application development and deployment. It is the layer on which cloneMAP is located. *Container orchestration* is one of the key concepts of many cloud platforms (e. g. Docker Swarm and Kubernetes) located at PaaS layer. SaaS refers to the provision of cloud-based applications.

Orchestration in the cloud refers to the coordination of containerized services in order to maintain the desired state of the cloud system and to handle faults [8, 45]. This is why orchestration is actively used by cloneMAP. It enables the execution of applications which consist of many distinct and loosely coupled pieces, so-called microservices. The microservice architecture is the counter part of monolithic applications. Instead of one large application built from one code base, many small microservices are used. Microservices are often stateless. Stateless applications can be easily scaled horizontally by starting several instances. Moreover, a stateless application can be simply restarted in case of a fault since there is no state that would be lost. The described methods are associated with so called *cloud-native* applications. While the term is often used with different meanings, some key characteristics have been identified [7, 15]. They have in common, that cloud-native applications exploit the features of cloud infrastructure to achieve high availability, scalability and fault-tolerance. In that sense, simply moving a monolithic application to cloud infrastructure does not comply with the idea of cloud-native applications. We adopt the cloud-native design pattern for cloneMAP as described in Sect. 3.

Kubernetes is a container orchestration tool developed open-source by a large community around the Cloud Native Computing Foundation. Thus, it is well-established in industry but also object of many research activities related to cloud computing over the last years [6, 32], especially regarding orchestration improvements [26, 30]. Kubernetes is developed in the *Go* programming language and supports the widely-used *Docker Engine*. Kubernetes and Docker are used for the deployment of cloneMAP.

2.4 Combination of cloud computing and MAS

The weaknesses of existing MAPs with respect to fault-tolerance and scalability call for new implementation paradigms. As outlined in Sect. 2.3, cloud-computing techniques offer measures to achieve scalability and fault-tolerance. Therefore, a combination of MAS and cloud-computing is identified as a promising solution [42]. MASs running on a cloud system can make use of its capabilities, i. e., dynamic scalability and fault-tolerance. This matches the idea of cloneMAP.

In [23] a MAS is used for distributed control in smart microgrids in form of a cloud service. The solution is evaluated by a cloud service providing control for voltage stability with the aid of the MAS executed on JADE. The importance of scalability is addressed but the evaluation is performed on a single node. Hence, no conclusion can be made on scalability beyond one computing node. Similarly, in [39] the Elastic-JADE approach is presented which is based on so-called scalable multi agents using cloud resources but with no study on the scalability of JADE itself. Elastic-JADE deploys JADE to a cloud environment and enables dynamic provision of new machines that extend the JADE platform. However, just moving an application to the cloud does not increase its scalability. Architectural problems remain the same, whether the application is executed on local hardware or in the cloud. That is why we decided to implement a novel MAP that is based on a cloud-native design and hence, fully exploits the advantages of a cloud environment.

3 cloneMAP concept

Based on the literature review cloneMAP's major design goals are high scalability and fault-tolerance. It follows a modular design and builds upon the container orchestration system Kubernetes. First we motivate the choice of the programming language and the use of Kubernetes. Subsequently the architecture of the platform is described in detail.

3.1 Go programming language

Go is a compiled language which was developed especially for networked servers [10] and therefore is predestined for cloud computing. It provides native support for concurrency. Concurrency in Go is realized with *green threads*, or *goroutines* in Go terminology. Green threads are not managed by the OS but by a scheduler which is part of the Go runtime library and operates on application level. This scheduler takes care of scheduling the green threads to OS threads whose number typically equals the number of available CPUs. They are more lightweight in stack size and scheduling time compared to OS level threads and hence, introduce less overhead. Comparisons to other programming languages, like Java [44], indicate the benefits.

One main advantage of MASs is the fact that agents can be executed individually and therefore concurrently. This enables the exploitation of multi-core and distributed hardware. However, especially in large-scale MASs the number of agents typically exceeds the number of available CPUs drastically. Hence, with a large number of agents a parallel execution of all agents is not possible. Therefore, using OS level threads for agent concurrency, i. e., one thread per agent, leads to considerable overhead for scheduling. Since green threads are more lightweight their use for agent concurrency introduces less overhead.

The widespread use of Go in the area of cloud computing and the available concurrency feature motivate its use for cloneMAP.

3.2 Kubernetes and Docker

Besides an appropriate programming language for the development also the deployment of cloneMAP is of importance in order to achieve its design goals. As already outlined in Sect. 2.3 container technology is one of the main trends in cloud computing. A container does not only contain the application it is built for but also all its dependencies. This way the use of containers offers high portability and reduces the dependency on specific infrastructure. We use Docker as the most widely used container engine.

Kubernetes is a container orchestration tool. It enables its user to combine several computing nodes to one cluster and to execute containers on that cluster. The deployment unit in Kubernetes are so-called *Pods*, which consist of one or multiple containers. Pods are assigned a network name by means of a DNS and can be addressed by other Pods using that network name. The user can execute multiple instances of the same Pod, e.g., in order to handle more requests simultaneously. A Kubernetes *Service* is used to cluster these Pods and to load-balance incoming requests among them. The Kubernetes user specifies a desired state of the cluster, regarding the different resource types like Pods and Services. Kubernetes takes action to drive the actual cluster state to the desired state and to maintain it. For example Kubernetes automatically restarts failed Pods.

The choice to use Kubernetes for the deployment of cloneMAP is motivated by its features that support the execution of microservice-based applications. The different parts of

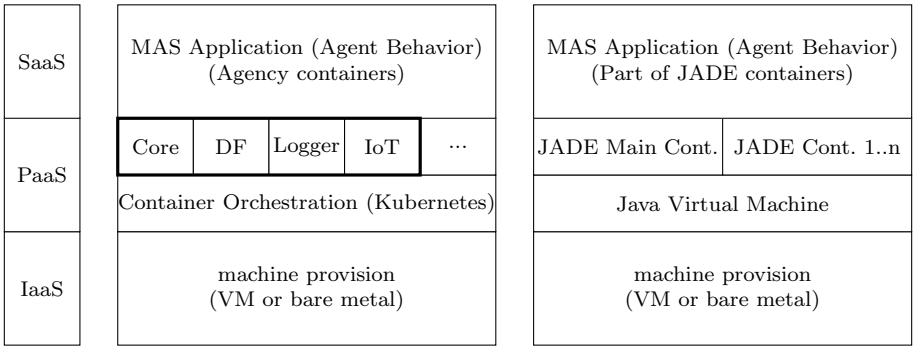

Fig. 2 Cloud stack of cloneMAP and comparison to JADE

the MAP are mapped to Kubernetes Pods as described below. The concept of Kubernetes Services is used to scale single MAP parts horizontally.

3.3 Overview

The conceptualization of cloneMAP is based on the FIPA reference architecture introduced in Sect. 2.1. That means that cloneMAP implements the AMS, the DF and a MTS. In addition further components extend the functionality of cloneMAP. Agents use the FIPA Agent Communication Language to construct their messages. While the concept of cloneMAP follows the FIPA specifications, it is not fully FIPA compliant. The reason for this is that cloneMAP uses the JSON format for the encoding of messages which is not included in FIPA specifications. JSON is chosen over other formats like XML because it is lightweight and popular in web applications.

The FIPA reference architecture is complemented by a cloud-native design of cloneMAP to fulfill the identified requirements. That means that the above listed components of the MAP are developed and deployed as distinct applications, or microservices. Each microservice offers a representational state transfer (REST) Application Programming Interface (API) for interaction with other services. Further, these microservices are designed to be stateless. In order to achieve statelessness we utilize distributed storages (etcd and Cassandra DB; see Fig. 3). Microservices store their state in these storages. Therefore, microservices can be scaled horizontally. The statelessness of the microservices eliminates the need for synchronization among the single instances. This is completely handled by the distributed storages which use well established algorithms. Moreover, the microservices are fault-tolerant as a stateless microservice can simply be restarted in case of a failure.

In order to manage the microservice Kubernetes is used. Each microservice is deployed as a Pod with a single Docker container. Kubernetes enables the horizontal scaling, i.e., the execution of multiple instances of a Pod, and the automatic restart in case of a failure. Figure 2 shows the resulting cloud stack for cloneMAP. Figure 2 also shows the architecture of JADE in the same model. While JADE can also be distributed over several computing nodes, it remains a single monolithic application on each node. Central components like the AMS and the DF are available on the main JADE container.

Figure 3 shows cloneMAP's overall architecture. The microservice approach leads to a modular architecture. The *core* module implements the AMS and the MTS, while the *DF* module implements the DF. Two additional modules are provided by cloneMAP: *Logging*

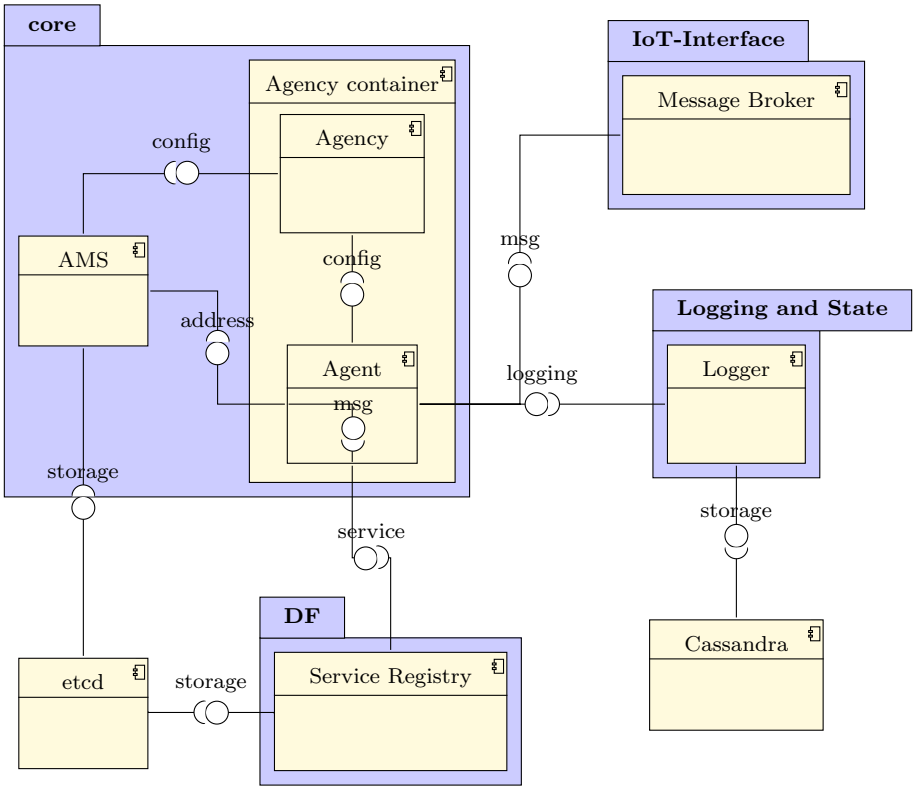

Fig. 3 Modular architecture of cloneMAP

and State and *IoT-Interface* The four modules and their components are explained in more detail in the following. Some important aspects are compared to the architecture of JADE.

3.4 Core

The *core* module comprises two components: the AMS and a variable number of agencies containing the agents.

The AMS has the task to manage the lifecycle of a MAS and its agents. Hence, it is responsible for MAS and agent creation, manipulation and termination. In order to allow the AMS to be stateless, an etcd cluster is used to store the configuration of the MAS and the agents, i. e., the AMS's state. Etcd[2] is a distributed key-value store suitable for storing such configuration data. The AMS provides an API to create and terminate a MAS and to retrieve configuration information.

Agencies are the deployment units for agents. An agency is a container with one agency process and a predefined but configurable number of agents, executed in separate green

[2] https://etcd.io/.

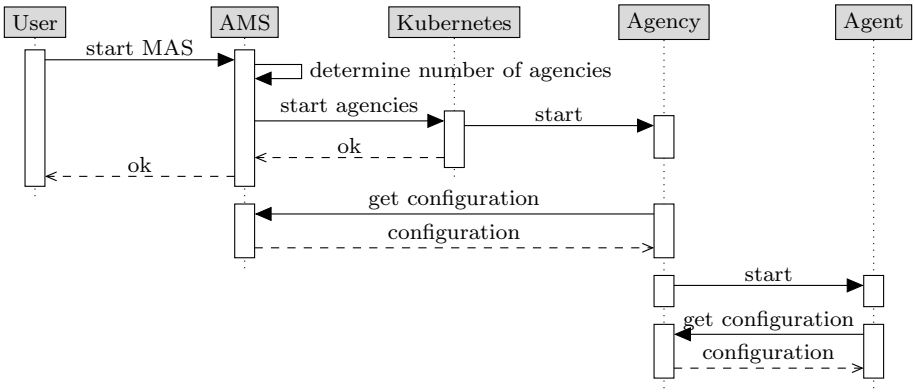

Fig. 4 MAS starting sequence

threads. The agency is responsible for starting and terminating agents. Moreover, it periodically monitors the agents' status and restarts them in case of errors.

When the AMS is requested by the user to create a new MAS, the procedure depicted in Fig. 4 is executed. The number of agencies to be created results from the total number of agents in the MAS and the number of agents per agency, as defined by the user. By choosing the number of agents per agency, the user can control the level of distribution. If more agents are located in a single agency the effect of a fault of one agency will be more severe as a large number of agents is affected. However, locating only a small number of agents in an agency leads to a large number of agencies and therefore, to more computational overhead. The AMS invokes the Kubernetes API to deploy a *StatefulSet* of agencies with the calculated number of instances. Pods in StatefulSets have a stable network name which consists of the specified name of the set and a unique ID. Even if the agency has to be restarted by Kubernetes, in case of a failure, the same network name is assigned to the agency Pod again. This results in a unique and stable network address for each agency. The AMS assigns agents to the executed agencies and stores the respective address of each agent. The assignment to agencies happens based on the ID of agents. This can be further improved in the future by locating agents, that frequently interact with each other, within the same agency in order to minimize message exchange among agencies. After their creation, the agencies request their configuration, i. e., a list of agents they should contain, from the AMS and start the corresponding agents.

Messaging among agents is handled using *Go channels*. Each agent has one channel for incoming messages. For the communication among agents within the same agency, agents append messages directly to their peer's inbox. Messages directed to agents in other agencies are added to an agency's outbox which is also implemented as a Go channel. The AMS is used as white pages to retrieve the addresses of other agencies. Subsequently the sending agency takes care of sending this message to the remote agency which then routes the message to the receiver agent. The handling of incoming and outgoing messages between agencies is handled by multiple green threads. Their number is adjusted dynamically according to the number of agents and the total number of agencies in the MAS.

In JADE agents are deployed in so-called JADE containers which are one JAVA application. The AMS is only present on the main container and not horizontally scalable. JADE uses one thread per agent. For a large MASs where the number of agents massively

exceed the number of available CPUs this leads to considerable overhead compared to green threads, as discussed before. Moreover, JADE dedicates five threads per container (so-called *deliverer*) to sending of messages [4]. For local agents these threads add the message directly to the receiver's inbox. For agents within another container the message is added to the inbox via a remote procedure call. While this mechanism for message sending is similar as for cloneMAP, the fixed number of deliverer threads might impose scalability issues for applications that involve massive communication among agents.

3.5 Directory facilitator

The *DF* module provides yellow pages functionality. Agents use the *DF* to register, deregister and request services. Each service contains a description revealing information about the kind of service that is offered. Agents can register one, multiple or no services with the DF. Requests to the DF can be used to search for agents which offer the requested service. For this purpose the search request is compared to the description of registered services in order to find suitable agents.

In case agents represent objects, which are connected in a single connected graph structure, the cloneMAP DF offers a topological search. The MAS developer can specify a MAS topology, i. e., a graph consisting of nodes and edges. Agents are attached to a node and edges indicate a direct connection between nodes with a weight. An agent can then request all services within a certain distance. The distance of two agents is calculated as the sum of all edge weights for the shortest path (i. e. the minimum sum of its edge weights) between the agents in the topology graph. The topological search allows agents to search for local agents with a small distance. This feature is important for the use case SwarmGrid-X as described in Sect. 4.1.

Similar to the *core* module, the *DF* consists of a stateless frontend, the service registry, that offers a REST API for the agents and a distributed backend. The same etcd cluster, as for the AMS, is reused as backend for the registered services. The service registry can be scaled horizontally due to its statelessness. Kubernetes load-balances incoming requests among all instances of the DF.

Requests to the DF result in interaction between the service registry and the etcd backend. Hence, the answer time is increased compared to an implementation without a backend, e. g., as for JADE. In order to cope with this challenge, the service registry contains a local cache, i. e., a copy, of all registered services. The etcd watcher feature is used to update the local cache whenever a key in the etcd cluster changes, i. e., a service is registered, altered or deregistered. In these cases, the service registry forwards the update to the etcd backend which then informs all instances of the service registry about the changes. Subsequently, these update their local cache. In case services are requested, the service registry uses its local cache to search for services that match the request. Hence, no direct interaction with etcd is necessary at request time. Since registration and deregistration are typically expected to happen less often than search requests for services this procedure increases the overall performance of the DF as the interaction between etcd and the service registry is minimized.

Similar to the AMS the DF in JADE is usually only present in the JADE main container. However, JADE offers a functionality for a distributed DF: federation. The user can start multiple DFs and federate them. DFs forward requests to the DFs they are federated with. However, each DF still only manages the services of those agents which registered with it. Hence, the DF remains a single-point of failure at least for a part of the MAS. Moreover,

forwarding requests to multiple DF instances produces more traffic and as a result limits scalability. An additional feature of JADE is the possibility to deploy a fault-tolerant platform. In this setup multiple main containers are started and the DF and AMS are replicated in each main container instance. However, only one main container is active while the others are kept updated but remain dormant. In case the active main container terminates with an error, another replica becomes the active main container. While this depicts a fault-tolerance mechanism on platform level, it does not increase the scalability of JADE since only one main container's components, i.e., AMS and DF, are active at a time. A topological search as offered by cloneMAP is not available in JADE.

3.6 Logging and state

The *logging and state* module has two tasks: offer logging capabilities for agents and offer backup capability for agents' states. Again, a stateless frontend providing a REST API is combined with a distributed storage. In contrast to the *core* and the *DF* modules, which use etcd, a Cassandra database.[3] It is deployed to a variable number of separate Pods. Both, logging and state information, are time series data. NoSQL databases, such as Cassandra, have proven to be convenient for storing this kind of data [47].

The logging functionality provides an easy way of debugging during the MAS development and analyzing during its operation. Agents can add log messages to different topics such as messaging, errors or status. The MAS developer or user can request and filter these messages for debugging or to analyze the agents' behaviors.

The state storage functionality can be used to store the state of agents. cloneMAP is able to automatically restart agents that were terminated due to errors, e. g., hardware failures. In case of a failure, Kubernetes takes care of restarting the affected agencies. These would then repeat the sequence shown in Fig. 4 and start the agents. The agency itself has no other state information except for its configuration which is retrieved from the AMS. However, for an agent to continue working as intended, also the state of the agent before termination is required. The state of an agent is defined by the MAS developer. It might contain the most recent sensor values obtained from a device or information retrieved from other agents. As agents store their state, either periodically or manually triggered by the MAS developer, it can be restored after the restart and agents can continue to work from that point.

The best strategy for updating the state stored in the Cassandra database highly depends on the concrete MAS application. It is not restricted by cloneMAP itself. In case an agent's state changes directly before a failure happens, the stored state might not be the most recent one. Additional recovery strategies may be necessary that ensure a consistent agent state in the MAS application layer. It should be noted that, typically, one main motivation for using MAS is the ability to implement fault-tolerant behavior, i. e., to implement an application that can continue to operate even if single agents fail. Automatically restarting failed agents is an additional measure for fault-tolerance introduced by cloneMAP. In that sense cloneMAP offers fault-tolerance on platform level by means of Kubernetes' functionality of restarting failed Pods. In addition it offers a mechanism that supports the implementation of fault-tolerance also on agent level. The developer of a concrete MAS application can then utilize this mechanism, i.e., the state storage functionality.

[3] http://cassandra.apache.org/.

Also JADE provides a mechanism for fault-tolerance on agent level, namely virtual replicated agents. In this concept, agents can be replicated, i. e., executed several times, and a virtual agent takes care of routing messages to the instances. The state of all instances has to be synchronized which causes overhead. As the target of cloneMAP are large-scale MAS this approach is not suitable. The replication of a large number of agents and the necessary synchronization among instances requires additional effort and therefore, imposes performance issues. In cloneMAP each agent is only executed once and restarted only if necessary, i. e., an error occurred. JADE's fault-tolerance mechanism depends on the synchronization of an agent's state among all its instances. Hence, the same considerations as for cloneMAP apply in case an agent fails before it is able to perform the synchronization after the state changes.

3.7 IoT-interface

The fourth module of cloneMAP is the *IoT-Interface*. It represents the connection of the agents to the outside world. As explained before the *core* module enables inter-agent communication. However, typical MAS applications involve the monitoring and/or control of (distributed) devices. Hence, agents also need an interface for agent-device communication.

The IoT-Interface consists of a message broker implementing a communication protocol. We chose Message Queuing Telemetry Transport (MQTT) for this purpose since it is widely used in the area of IoT-devices. It offers a publish-subscribe mechanism for communication. cloneMAP uses the Eclipse Mosquitto implementation.[4] Due to the modular approach also other messaging mechanisms for interaction with devices can be used with cloneMAP. cloneMAP offers an interface to MQTT for the implementation of agents. For other messaging mechanisms a corresponding interface has to be implemented.

Using a publish-subscribe message broker allows to model different communication patterns between IoT-devices and agents in cloneMAP. A `1:1` communication between agents and devices as well as any `n:m` pattern are possible. As described in the literature review we see agents as another abstraction layer for the IoT. Flexible communication patterns allow for an effective communication of a single agent with multiple devices or vice versa. Agents can also use the message broker to communicate among each other in cases where peer-to-peer messaging is not suitable.

4 Evaluation methodology

We define four performance indicators and respective benchmarks to demonstrate cloneMAP's scalability and compare it to JADE. All benchmarks are executed on the same multi-node cluster as specified in Sect. 4.6. For both platforms, the benchmarks are implemented with no specific optimizations but as little code as possible to execute the same behavior. In the case of JADE, the procedures found in the documentation were used to implement the agent behavior. The choices achieve a comparison that is as fair as possible. In addition to these benchmarks, assessing general platform features, we analyze the performance of cloneMAP for the specific use case SwarmGrid-X in the area of electrical power grids.

[4] https://mosquitto.org/.

4.1 Use case SwarmGrid-X

The application of MASs for the control of power grids is currently an active field of research. The increase of renewable power resources and the coupling with other sectors, e. g., heat and mobility, poses challenges that may be resolved through the flexibility of some of the new components via intelligent EMS. MASs are proposed in literature as a framework for this task. In the distribution grid control concept SwarmGrid-X, agents are assigned to flexible components and decide how to use the available flexibility concerning different objectives. This decision making is achieved by negotiations among power consumer agents and producer agents trying to balance power as locally as possible. Negotiation partners find each other by means of a swarm concept, utilizing the DF. For this purpose the topological search as introduced before is used. Consumer agents request producer agents close to their location from the DF. The local agents one agent communicates with are referred to as its swarm. The inherent structure of the power grid, based on the different voltage levels, is mirrored in the MAS by a holonic architecture. SwarmGrid-X, is able to mitigate voltage band violations and to provide additional flexibility to superordinate levels, i. e., the transmission system. A more detailed description and evaluation of SwarmGrid-X can be found in [11]. Given the scale of SwarmGrid-X comprising possibly several thousand agents, it is an appropriate IoT use case to assess cloneMAP.

In [11] SwarmGrid-X is studied as part of a simulation model using the simulator DistAIX [19]. Besides its functionality [11] also proves SwarmGrid-X's scalability. In this work the SwarmGrid-X concept is implemented utilizing cloneMAP. For this purpose the described behavior of producer and consumer agents is transferred to cloneMAP. Agents are executed within agencies as described in Sect. 3.4. The cloneMAP *DF* is utilized for the introduced swarm forming, the *logging and state* module is used by agents to store their state and the *IoT-Interface* is the connection to the controllable grid components, i. e., the IoT devices. The implementation of agent behaviors is done as close as possible to the implementation presented in [11]. The main difference between the DistAIX and the cloneMAP implementations is that agents are executed with a fixed time step in the simulation but continuously and asynchronously in cloneMAP. As a result the outcome of SwarmGrid-X implemented with cloneMAP is non-deterministic. The actions of agents in a MAS obviously depend on interaction (i. e. messaging) with other agents which is one source of non-deterministic behavior: due to the IP-based communication, messages might be received in different order for multiple executions of the same scenario. Also the scheduling of agents to the CPU can differ. Despite the differences among multiple executions, SwarmGrid-X leads to a similar overall behavior in all executions. Agents try to balance power as locally as possible. Even if non-deterministic effects affect the negotiations their outcome is always similar.

We use DistAIX for the simulation of the agents' physical environment, i. e., the electrical grid and components. Agents communicate with their components via MQTT. DistAIX is executed in real-time for this purpose. The setup enables the assessment of cloneMAP and SwarmGrid-X under real-world conditions. The physical environment is replaced by a simulation. Such a setup is referred to as a Software-in-the-Loop setup.

SwarmGrid-X is evaluated for different scenario sizes. We use the electrical grid introduced in [11] as base scenario. It consists of two low voltage grids connected to a common medium voltage grid. N copies of the grid are connected at the point of common coupling with the overlaying grid to scale the scenario size. A single copy of the grid comprises 588 agents. The number of agents for N copies is

$$A(N) = N \cdot 585 + N \cdot 2 + 1. \tag{1}$$

4.2 Benchmark 1: round-trip time

The first benchmark aims to assess the performance of the messaging among agents. Similar tests have been defined in [9]. An even number of agents is executed and distributed evenly among all computing nodes. Pairs of two agents are generated in a random way. The two agents of a pair sequentially exchange messages. The agent initiating the communication, measures the round-trip time (RTT), i. e., the time span between the sending of a message and receiving of its peer's answer. This is done for 1000 messages in a row. Since the creation and termination of agents takes additional time, messages are already exchanged before and also after the actual measurement is performed. This ensures that the measurement of RTTs is performed under full load condition for every agent pair.

The decision making process of agents generally involves different steps. First agents have to learn about possible communication partners with the help of the DF. After that they exchange information with other agents and reason about it. Depending on the application this might happen multiple times iteratively until a decision is made. While the reasoning of the agents is implemented by the MAS developer, the messaging and the DF are part of the platform.

Hence, the scalability of messaging is crucial for any large-scale MAS application as it is a core functionality needed for the interaction between the agents and therefore needs to scale with their number. The time required for decision making strongly depends on the speed of the communication. The scalability of messaging is evaluated by performing multiple measurements with an increasing number of agents, and therefore, communication effort. For all these measurements the avgRTT is analyzed as a metric for the messaging performance which indicates the latency in the interaction between agents. This can be pivotal for the performance of the whole MAS, when the interactions are time-sensitive. The avgRTT for the SwarmGrid-X use case with different scenario sizes is measured, too. This enables the reader to set the results of the generic benchmark into the context of a realistic use case.

4.3 Benchmark 2: directory facilitator

As outlined in the literature review, the DF has been identified as bottleneck and therefore, a scalability limitation for large-scale MAS. In the described use case SwarmGrid-X, the DF has to serve requests from possibly several thousand agents. Similar to the messaging also the DF answer time influences the required time for decision making.

We analyze the performance of the DF by putting it under heavy load with a varying number of agents. Each agent performs the same sequence of requests: first a service is registered, then a search request is sent for eight times and finally the registered service is deregistered again. To ensure that the DF is under full load during the entire time, agents send search requests before and after the actual measurement. Every agent measures the total time that is required for the described sequence. The number of search requests is higher than the number of registrations and deregistrations since for most MAS applications it can be assumed that agents register a limited amount of services but repeatly search for services of other agents. In the case of SwarmGrid-X we measure the average answer time for search requests and relate this to the results of the benchmark.

4.4 Benchmark 3: CPU utilization

Besides the performance of the messaging, also the performance of the MAP for a given number of agents and a given agent task is of importance. Hence, we analyze how much a certain computation exhausts the available computing resources, i. e., the CPU. In order to simulate an agent behavior from a computational point of view, every agent performs a summation of float values for a predefined portion of the execution time and idles for the rest of it. The period T is divided in two phases T_{run} and $T - T_{run}$. For T_{run} the agent performs the float summation looping through two arrays of floating point values. Subsequently, for $T - T_{run}$ it sleeps. This results in the relative load factor $\alpha = \frac{T_{run}}{T}$ which ensures that the applications generate the same load even if the required time for one float summation differs between Java and Go applications. The motivation behind our measuring method (instead of letting the applications execute a fixed number of float summations between each message exchange) is that we do not want to determine the FLOPS performance of the different programming languages but compare the performance of the different MAPs implementations. Additionally, each agent pair as defined in Sect. 4.2 exchanges one message every second.

The average CPU utilization of all computing nodes is measured over time for a fixed number of agents. The factor α is varied to achieve different load scenarios.

In the context of cloud computing, CPU utilization is an important metric as more CPU power translates to higher costs in case of public cloud infrastructure. An application which requires less resources can be executed on smaller and therefore, cheaper (virtual) machines. But also for private or on-premise infrastructure reduced hardware requirements are beneficial.

4.5 Benchmark 4: fault-tolerance

The last benchmark aims at analyzing the fault-tolerance capability of cloneMAP and comparing its performance with the one of JADE. For this purpose benchmark 1 is repeated. Within an agent pair the agent initializing communication counts the number of received messages. This is defined as the agent's state. This state is stored every 25 ms. For cloneMAP this is done by means of the module *Logging and State* (see Sect. 3.6). In the case of JADE each agent creates one replicated agent. Storing the state means that it is synchronized among the original agent instance and its replica. We evaluate the avgRTT for varying MAS sizes. The factor f_{avg} by which the avgRTT obtained in benchmark 1 is increased reveals the additional overhead that is generated by the fault-tolerance functionality.

In addition we demonstrate the importance of the fault-tolerance capability with the aid of the SwarmGrid-X use case. According to the SwarmGrid-X concept, power consumers and producers negotiate the amount of consumed and produced power with the goal to balance power as locally as possible. These negotiations result in contracts between agents which define their behavior. The list of all contracts is defined as the agent's state. Whenever the state of an agent changes it is stored in the Cassandra DB.

During a simulation of the single-copy scenario we artificially introduce a failure of three of the agencies, which leads to a restart of the agency containers and hence, the agents executed in these agencies. To simulate the failure we manually terminate the agency containers. This might correspond to a software failure leading to a termination of

the containers or a hardware failure of the machines the agencies are executed on. Note that Kubernetes would also restart the containers if the cluster was overloaded and the containers slow to respond. In that case the Kubernetes cluster needs to be extended by adding more nodes to it. Kubernetes can then schedule containers that are not responding to the new nodes.

The three terminated agencies comprise about 27% of all agents in the scenario. We compare the overall behavior of the power grid with and without the failure. For this purpose the active power exchange at the point of common coupling is evaluated. The active power depicts the overall behavior of the entire system. This allows to conclude implications of the fault-tolerance capability for the overall functionality of SwarmGrid-X. Additionally, the average time between the failure and the restart of the agents is analyzed for varying scenario sizes.

4.6 Hardware configuration

All presented benchmarks are executed on the same hardware setup for both cloneMAP and JADE. It consists of three physical machines with two Intel Xeon E5-2650 v2 each, leading to a total number of 16 cores per machine each with 64 GB of main memory. Hyper-Threading is disabled to ensure an equal number of physical cores and total numbers of vCPUs (virtual CPUs of all VMs). On every physical machine 4 VMs are executed, each with 4 vCPUs and 12 GB of main memory. This leads to a virtual twelve node cluster. All three physical machines are connected by GBit-Ethernet. All host and guest machines use Ubuntu 16.04.6 with a 4.4.0 Linux Kernel as OS.

We used the tool kubeadm to set up a single master Kubernetes cluster. Calico is used as network plugin for Kubernetes. The used Kubernetes version is 1.13.4 and Go version 1.11.2 was used to build all cloneMAP components. The Kubernetes master node is not used for hosting cloneMAP components. The two used distributed storages, etcd and Cassandra DB, are executed with three instances each in all benchmarks. In case of JADE all twelve nodes are used for the benchmarking MASs. We use JADE 4.5.0 and Java SE Runtime Environment 1.8.0. The executed agents are distributed equally among all nodes for all benchmarks. Each virtual node executes exactly one container, either for a cloneMAP agency or for JADE.

5 Evaluation

5.1 Benchmark 1: round-trip time

Figure 5 shows the avgRTT and the 95th percentile of all messages for the execution of benchmark 1. JADE performs slightly better for small use cases with less than 100 agents. However, the difference is below 1 ms and therefore negligible. For both, JADE and cloneMAP, the slope of the curves increases starting from a MAS size of fifty agents. The reason for this is, that the test setup has a total number of 48 CPU cores. As a result, up to this number the execution of agents can be completely parallelized.

For larger MASs, cloneMAP shows much better performance compared to JADE. The avgRTT at 10,000 agents is 11 ms in the case of cloneMAP. In the case of JADE it is more than 20 times higher (222 ms). Moreover, the factor between the avgRTT of cloneMAP and JADE increases with the number of agents. This is different for the other MAPs claiming

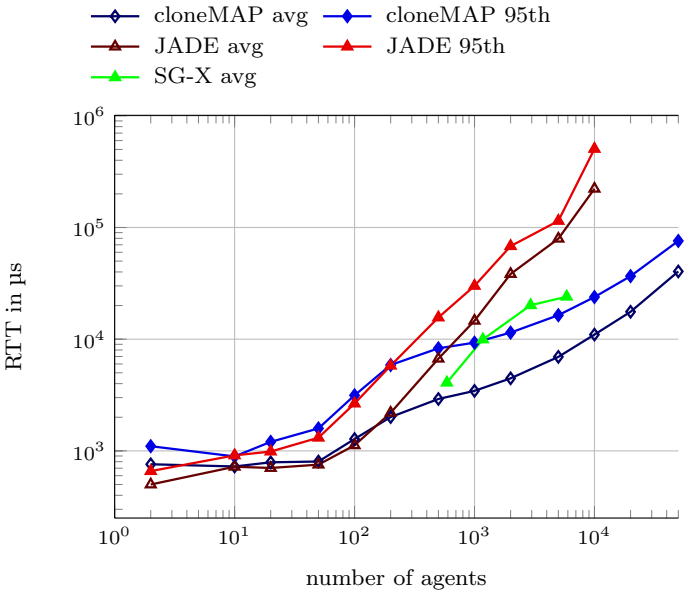

Fig. 5 Benchmark 1: RTT for varying number of agents

scalability, where the factors remain roughly constant for different MAS sizes. As a result the scaling factor of cloneMAP is much larger compared to the one for Magentix and SAGE, based on the literature review, for large MASs containing several thousand agents. One reason for the constant scaling factors in other MAPs could be a similar execution model for agents. As in JADE agents are executed in separate OS threads. In contrast to this agents in cloneMAP are executed in green threads which are much more lightweight and reduce the overhead especially for large MASs. An execution of JADE with more than 10,000 agents failed for 10 consecutive executions.[5] The results show that the messaging of cloneMAP is much more scalable to higher numbers of agents.

Figure 5 also shows the avgRTT for SwarmGrid-X (SG-X) implemented with cloneMAP. They are slightly increased compared to the RTTs in the dedicated benchmark. For example the average RTT for the scenario with 1,175 agents is 10 ms while it is only 3.5 ms for the RTT benchmark with 1000 agents. Two reasons can be identified for this. First, in the RTT benchmark no sophisticated agent behavior as for SwarmGrid-X is executed. Hence, the overall CPU utilization for SwarmGrid-X is much higher for a similar scenario size. Second, the measured RTT for SwarmGrid-X also includes computations an agent has to perform before sending an answer. Despite the increased RTTs the results remain one order of magnitude below 1second. Therefore, the functionality of SwarmGrid-X is not affected by message delays for all scenario sizes.

[5] Exemplary error message: `Error serving H-Commandjade.core.messaging.Messaging/3: jade.core.NotFoundException: getContainerID()failed to find agent ag3824@192.168.0.151:1099/JADE`.

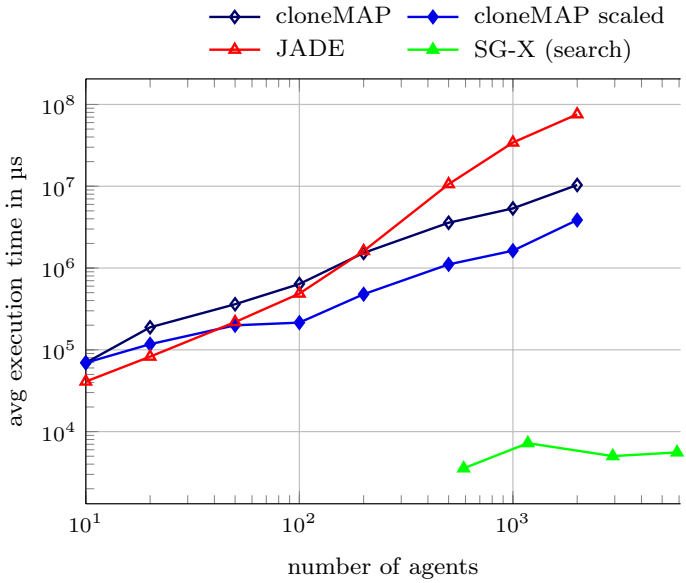

Fig. 6 Benchmark 2: execution time of DF request sequence and answer time of search requests in the case of SG-X

5.2 Benchmark 2: directory facilitator

The results for the DF benchmark can be seen in Fig. 6. For cloneMAP the results are shown for a single instance of the DF and for a certain number of instances leading to ideal results. In the case of 10 agents the JADE DF performs better compared to cloneMAP. The cloneMAP DF has to perform two requests for registration and deregistration, one from the agent to the service registry and one from the service registry to the etcd backend. This leads to an overhead that increases the execution time. For larger MASs this overhead is less critical as the backend is distributed and can serve more requests. As a result, the execution time increases much more for the JADE DF with a growing number of agents.

In addition to the backend, also the service registry can be scaled horizontally. A comparison of execution times for a single instance and for multiple instances reveals that scaling the DF further decreases the execution time. However, the different kinds of requests scale differently. While the registration and deregistration requests invoke also the etcd backend, scaling is limited by its performance. Searching for services uses only the local cache of a service registry. Hence, the parallel execution of this action within different instances does not interfere. For 1000 agents the ideal number of service registries is five. The average execution time for a search request with a single instance is 441 ms while the execution time with five instances is 110 ms (JADE: 3808 ms). For registration requests, the average execution time with one instance is 915 ms and for five instances 382 ms (JADE: 1268 ms). These examples show that using a local cache in every service registry enables a better scalability. As search requests are more frequent in typical MAS application, the overhead for registration and deregistration is acceptable and the optimization for search requests is beneficial.

The answer times for search requests in SwarmGrid-X are also depicted in Fig. 6. Note that these are the times of only one single operation while in the benchmark a sequence of

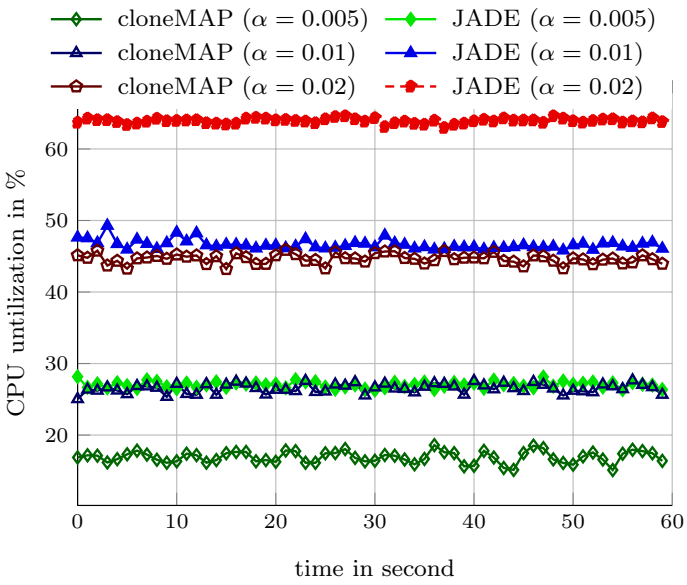

Fig. 7 Benchmark 3: CPU utilization for different load scenarios

ten operations is executed. The answer times are in a similar range for all scenario sizes and do not increase significantly. In contrast to the dedicated DF benchmark the DF is not fully utilized in the use case. The benchmark represents the worst case behavior since all agents are requesting the DF simultaneously. Requests to the DF occur occasionally in the case of SwarmGrid-X. Hence, it is able to maintain small reaction times for all scenario sizes. Similar to the results for the message RTTs, the DF answer times do not violate the requirements of SwarmGrid-X and therefore, do not disturb its operation.

5.3 Benchmark 3: CPU utilization

The CPU utilization for three cases of α is depicted in Fig. 7. In all cases the utilization is substantially lower for cloneMAP. We identified two main reasons for this. First, JADE is written in Java and hence, is compiled to byte code which requires a Java Virtual Machine to be executed. Go on the other side produces native machine code. Second, the use of green threads for concurrent execution of agents proves to be more efficient. Instead, JADE uses one thread per agent. For large MASs this adds a substantial overhead, as the OS has to take care of scheduling with the corresponding context switching.

5.4 Benchmark 4: fault-tolerance

Figure 8 shows the factor f_{avg} by which the avgRTT of benchmark 1 is increased if agents store, or synchronize, their state. For JADE f_{avg} decreases until a MAS size of 20 agents. Up to this point the total number of agents, i.e., the original agents and their replicas, is smaller than the total number of available CPUs. Hence, JADE can parallelize all agents and the relative overhead introduced by the fault-tolerance mechanism decreases. For

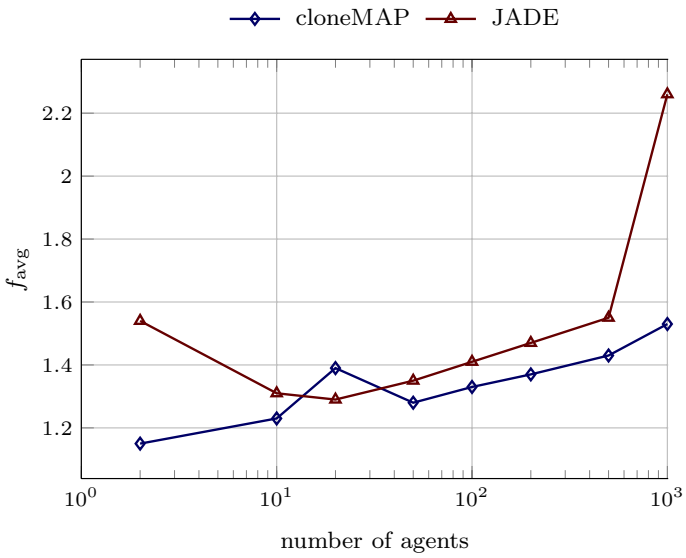

Fig. 8 Benchmark 4: factor between avgRTT in benchmark 4 and benchmark 1

greater MAS sizes the factor increases. For a MAS size of 1000 agents, the avgRTT exceeds 25 ms which is the state synchronization rate. As a result the MAP is overloaded and the increase of f_{avg} is stronger.

For cloneMAP f_{avg} increases with increasing MAS size. More agents increase the pressure on the *Logging and State* module what worsens the overall performance. However, the rate by which f_{avg} increases is smaller for MAS sizes larger than 20 compared to JADE. Note that f_{avg} as depicted in Fig. 8 reveals the relative increase of the avgRTT. Since the absolute value of the avgRTT of cloneMAP in benchmark 1 is much smaller for large MASs compared to JADE, also the absolute increase of the avgRTT in this benchmark is smaller. For example, for a MAS size of 500 agents the avgRTT in benchmark 1 is 2.92 ms for cloneMAP. With fault-tolerance enabled it increases to 4.16 ms which equals a f_{avg} of 1.43. The avgRTT of JADE for the same scenario is 6.69 ms without and 10.37 ms with fault-tolerance leading to a f_{avg} of 1.55. While the relative increase factors of cloneMAP and JADE seem similar, the absolute increase is 1.24 ms for cloneMAP and 3.68 ms for JADE.

For the demonstration of the fault-tolerance capability of cloneMAP three simulations are executed. Figure 9 shows the active power exchange P at the point of common coupling with the overlaying grid for the execution without a failure, with a failure and no recovery and with a failure but with recovery of failed agents. The vertical dashed line highlights the point in time when the failure of the three agencies is introduced.

Comparing the results for a simulation without failure against those obtained with failure and without recovery, yields that at the beginning the overall system behavior is similar in both cases. However, after the failure both curves for active power begin to deviate. Because SwarmGrid-X aims at balancing power as locally as possible, if agents fail and are not executed anymore their energy resources cannot adapt their behavior according to the rest of the system. In the simulated scenario the load, i. e., power consumption, decreases over time while the power generation increases due to photovoltaic (PV) systems. As a

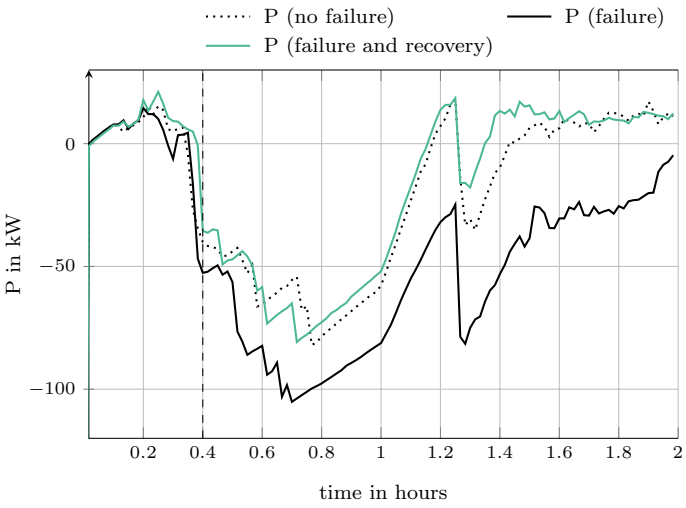

Fig. 9 Demonstration of fault-tolerance with SwarmGrid-X with agencies failure at 0.4 h

Table 1 SwarmGrid-X restart time in milliseconds for different scenario sizes *N*

N	Agents	Logger instances	Avg restart time	Avg state loading time
1	588	2	1452	68
2	1175	3	1919	52
5	2936	5	2483	80
10	5871	8	2770	107

result storages have to change their behavior from discharging in the beginning, to support the loads, to charging in order to store the renewable power exceed. The inability of flexibility providers to react in the described manner leads to the offset between both cases.

In the case of a recovery from the introduced failure, agents are restarted and hence, can continue to interact with the rest of the MAS. As a result the flexibility providers can react to the changes in the system and adapt their behavior. Therefore, Fig. 9 shows that the system behavior in case of a recovery is much closer to the one without the introduced failure.

However, still some deviations between the two curves exist. One reason for this is, that multiple executions of the exact same scenario always yield slightly different results due to the mentioned non-deterministic effects. Another reason for small deviations, even in case of the recovery, is that minor inconsistencies between the state of an agent before and after the restart can occur. If the agency fails during the manipulation of an agent's state, e. g., while adding or changing a contract with another agent, that state is not yet stored with the *logging and state* module. After the restart the information about that manipulation is missing. SwarmGrid-X is not optimized for the use of the described recovery mechanisms of cloneMAP. Further measures on application level are necessary to prevent such situations. However, the goal of this paper is not the demonstration of fault-tolerance of a specific MAS application, i. e., SwarmGrid-X, but the demonstration of fault-tolerance mechanisms provided by the platform cloneMAP. The results

indicate that even for an application, which does not ensure state consistency, the recovery mechanisms of cloneMAP improve fault-tolerance substantially.

Table 1 shows the average time required for restarting the agents for different scenario sizes N. The restart time is the time between the failure and a proper operation of the agents. The state loading time reveals the time needed to load the previously stored state and apply it to the agent's knowledge.

The time required for the restoring of the state depicts only a small portion of the overall restart time. Both times increase for increasing scenario sizes. However, the total restart time remains in the order of a few seconds and hence, is sufficient for the use case SwarmGrid-X. The largest part of the restart time is consumed by the detection of the failure by Kubernetes and the subsequent restart of the failed container. This time can not be influenced by cloneMAP.

6 Conclusions

This work presents cloneMAP as a MAP that is based on cloud computing techniques that achieve scalability and fault-tolerance. These features are deemed crucial for the adoption of the MAS paradigm in real world and large-scale applications which are common, e. g., in the IoT.

cloneMAP is implemented on top of the widely used container orchestration system Kubernetes using Go as a modern programming language especially designed for scalable network and cloud services. The cloud-native design of cloneMAP eliminates single-points of failure. The use of distributed data storage and stateless frontends enables horizontal scalability and fault-tolerance. Modern features offered by Go such as green threads allow an efficient execution of concurrent agent tasks. The benchmarks show cloneMAP's improved scalability and fault-tolerance features compared to JADE. The performance assessment of the implementation of the use case SwarmGrid-X indicates the suitability of cloneMAP in real world applications.

Hence, we regard cloneMAP as the next step towards a broader adoption of MASs also outside scientific research, i. e., in real-life and industrial applications. In combination with the IoT many possible fields of application exist. cloneMAP's presented features enable this combination. Moreover, the modular microservice architecture allows for a comparatively easy extension of cloneMAP. New modules adding further functionality can be included by adopting their REST API. Therefore, also existing software can be reused, e.g., other message brokers connecting agents to IoT devices. Further, this flexibility allows users of cloneMAP to adjust the platform to their needs, deploying only those modules which provide the required functionality. In the future, more real world use cases have to be implemented using cloneMAP to demonstrate its potentials.

Acknowledgements We gratefully acknowledge the financial support provided by the German Federal Ministry of Economic Affairs and Energy in the research Project AGENT under Grant Number 03ET1495A and the German Federal Ministry of Education and Research in the research project ENSURE under Grant Number 03SFK1CO.

Funding Open Access funding enabled and organized by Projekt DEAL.

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Ahmad, H., Suguri, H., Ali, A., Malik, S., Mughal, M., Shafiq, M., et al. (2005). *Scalable fault tolerant Agent Grooming Environment: SAGE,*. https://doi.org/10.1145/1082473.1082816.
2. Alami-Kamouri, S., Orhanou, G., & Elhajji, S (2016) Overview of mobile agents and security. In 2016 international conference on engineering MIS (ICEMIS) (pp. 1–5). https://doi.org/10.1109/ICEMIS.2016.7745371.
3. Alberola, J., Such, J., Botti, V., Espinosa, A., & García-Fornes, A. (2013). A scalable multiagent platform for large systems. *Computer Science and Information Systems*, *10*(1), 51–77. https://doi.org/10.2298/CSIS111029039A.
4. Alberola, J. M., Such, J. M., Garcia-Fornes, A., Espinosa, A., & Botti, V. (2010). A performance evaluation of three multiagent platforms. *Artificial Intelligence Review*, *34*(2), 145–176. https://doi.org/10.1007/s10462-010-9167-9.
5. Bellifemine, F. L., Caire, G., & Greenwood, D. (2007). *Developing multi-agent systems with JADE* (Vol. 7). New York: Wiley.
6. Bernstein, D. (2014). Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, *1*(3), 81–84.
7. Brunner, S., Blöchlinger, M., Toffetti, G., Spillner, J., & Bohnert, T. M. (2015). Experimental evaluation of the cloud-native application design. In 2015 IEEE/ACM 8th international conference on utility and cloud computing (UCC) (pp. 488–493). https://doi.org/10.1109/UCC.2015.87.
8. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, omega, and kubernetes. *ACM Queue*, *14*(1), 10:70–10:93. https://doi.org/10.1145/2898442.2898444.
9. Cortese, E., Quarta, F., & Vitaglione, G. (2002) Scalability and performance of JADE message transport system. In AAMAS workshop on AgentCities. Bologna
10. Donovan, A. A., & Kernighan, B. W. (2015). *The go programming language*. Boston: Addison-Wesley Professional.
11. Dähling, S., Kolen, S., & Monti, A. (2018). Swarm-based automation of electrical power distribution and transmission system support. *IET Cyber-Physical Systems: Theory and Applications*, *3*(4), 212–223. https://doi.org/10.1049/iet-cps.2018.5001.
12. Fortino, G. (2016). Agents meet the IoT: Toward ecosystems of networked smart objects. *IEEE Systems, Man, and Cybernetics Magazine*, *2*(2), 43–47. https://doi.org/10.1109/MSMC.2016.2557483.
13. Fortino, G., Guerrieri, A., Russo, W., & Savaglio, C. (2014) Integration of agent-based and cloud computing for the smart objects-oriented IoT. In Proceedings of the 2014 IEEE 18th international conference on computer supported cooperative work in design (CSCWD) (pp. 493–498). https://doi.org/10.1109/CSCWD.2014.6846894
14. Foundation for Intelligent Physical Agents: Standard FIPA Specifications. http://www.fipa.org/repository/standardspecs.html. Accessed on 23 October 2020
15. Gannon, D., Barga, R., & Sundaresan, N. (2017). Cloud-native applications. *IEEE Cloud Computing*, *4*(5), 16–21. https://doi.org/10.1109/MCC.2017.4250939.
16. Giordano, A., Spezzano, G., & Vinci, A. (2016) Smart agents and fog computing for smart city applications. In: E. Alba, F. Chicano, G. Luque (Eds.), Smart cities, Lecture Notes in Computer Science (pp. 137–146). Springer
17. Gutknecht, O., & Ferber, J. (2000) MadKit: A generic multi-agent platform. In Proceedings of the fourth international conference on Autonomous agents, AGENTS '00 (pp. 78–79). Barcelona: Association for Computing Machinery. https://doi.org/10.1145/336595.337048.
18. Iglesias, C. A., Garijo, M., & González, J. C. (1998). A survey of agent-oriented methodologies. In: International workshop on agent theories, architectures, and languages (pp. 317–330). Springer.

19. Kolen, S., Dähling, S., Isermann, T., & Monti, A. (2018). Enabling the analysis of emergent behavior in future electrical distribution systems using agent-based modeling and simulation. *Complexity*,. https://doi.org/10.1155/2018/3469325.
20. Kravari, K., & Bassiliades, N. (2015). A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, *18*(1), 11.
21. Leitão, P., Karnouskos, S., Ribeiro, L., Lee, J., Strasser, T., & Colombo, A. W. (2016). Smart agents in industrial cyber-physical systems. *Proceedings of the IEEE*, *104*(5), 1086–1101. https://doi.org/10.1109/JPROC.2016.2521931.
22. Lin, J., Yu, W., Zhang, N., Yang, X., Zhang, H., & Zhao, W. (2017). A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet of Things Journal*, *4*(5), 1125–1142.
23. Marchi, S. B. D., Ponci, F., & Monti, A. (2013). Design of a MAS as cloud computing service to control smart micro grid. *IEEE PES ISGT Europe*, *2013*, 1–5. https://doi.org/10.1109/ISGTEurope.2013.6695381.
24. McArthur, S. D. J., Davidson, E. M., Catterson, V. M., Dimeas, A. L., Hatziargyriou, N. D., Ponci, F., et al. (2007). Multi-agent systems for power engineering applications-part I: Concepts, approaches, and technical challenges. *IEEE Transactions on Power Systems*, *22*(4), 1743–1752. https://doi.org/10.1109/TPWRS.2007.908471.
25. McArthur, S. D. J., Davidson, E. M., Catterson, V. M., Dimeas, A. L., Hatziargyriou, N. D., Ponci, F., et al. (2007). Multi-agent systems for power engineering applications-Part II: Technologies, standards, and tools for building multi-agent systems. *IEEE Transactions on Power Systems*, *22*(4), 1753–1759. https://doi.org/10.1109/TPWRS.2007.908472.
26. Medel, V., Rana, O., Bañares, J. Á., & Arronategui, U. (2016). Modelling performance and resource management in Kubernetes. In 2016 IEEE/ACM 9th international conference on utility and cloud computing (UCC) (pp. 257–262). IEEE.
27. Mell, P., & Grance, T. (2011). *The NIST definition of cloud computing*. USA: NIST, Gaithersburg.
28. Mengistu, D., Tröger, P., Lundberg, L., & Davidsson, P. (2008). Scalability in distributed multi-agent based simulations: The JADE case. In 2008 Second international conference on future generation communication and networking symposia (Vol. 5, pp. 93–99). https://doi.org/10.1109/FGCNS.2008.158.
29. Mzahm, A. M., Ahmad, M. S., & Tang, A. Y. C. (2013). Agents of things (AoT): an intelligent operational concept of the internet of things (IoT). In 2013 13th international conference on intellient systems design and applications (pp. 159–164). https://doi.org/10.1109/ISDA.2013.6920728.
30. Netto, H. V., Lung, L. C., Correia, M., Luiz, A. F., & de Souza, L. M. S. (2017). State machine replication in containers managed by Kubernetes. *Journal of Systems Architecture*, *73*, 53–59.
31. Oey, M., van Splunter, S., Ogston, E., Warnier, M., & Brazier, F. M. (2010) A framework for developing agent-based distributed applications. In 2010 IEEE/WIC/ACM international conference on web intelligence and intelligent agent technology (Vol. 2, pp. 470–474). https://doi.org/10.1109/WI-IAT.2010.134.
32. Pahl, C. (2015). Containerization and the PaaS Cloud. *IEEE Cloud Computing*, *2*(3), 24–31.
33. Pico-Valencia, P., Holgado-Terriza, J. A., & Senso, J. A. (2019). Towards an internet of agents model based on linked open data approach. *Autonomous Agents and Multi-Agent Systems*, *33*(1), 84–131. https://doi.org/10.1007/s10458-018-9399-7.
34. Quinn, M. J. (2003). *Parallel programming in C with MPI and OpenMP*. New York: McGraw-Hill Education Group.
35. Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2018) IoT applications: From mobile agents to microservices architecture. In 2018 International conference on innovations in information technology (IIT) (pp. 117–122). https://doi.org/10.1109/INNOVATIONS.2018.8605967.
36. Savaglio, C., Ganzha, M., Paprzycki, M., Bădică, C., Ivanović, M., & Fortino, G. (2020). Agent-based internet of things: State-of-the-art and research challenges. *Future Generation Computer Systems*, *102*, 1038–1053. https://doi.org/10.1016/j.future.2019.09.016.
37. Schatten, M., Ševa, J., & Tomičić, I. (2016). A roadmap for scalable agent organizations in the internet of everything. *Journal of Systems and Software*, *115*, 31–41. https://doi.org/10.1016/j.jss.2016.01.022.
38. Shakshuki, E. (2005). A methodology for evaluating agent toolkits. In International conference on information technology: Coding and computing (ITCC'05)—Volume II (Vol. 1, pp. 391–396). https://doi.org/10.1109/ITCC.2005.15.
39. Siddiqui, U., Tahir, G. A., Rehman, A. U., Ali, Z., Rasool, R. U., & Bloodsworth, P. (2012). Elastic JADE: Dynamically scalable multi agents using cloud resources. In 2012 Second international conference on cloud and green computing (pp. 167–172). https://doi.org/10.1109/CGC.2012.60.

40. Stanković, R., Štula, M., & Maras, J. (2017). Evaluating fault tolerance approaches in multi-agent systems. *Autonomous Agents and Multi-agent Systems*, *31*(1), 151–177. https://doi.org/10.1007/s10458-015-9320-6.
41. Strasser, T. I., Andrén, F. P., Vrba, P., Šuhada, R., Moulis, V., Farid, A. M., & Rohjans, S. (2018) An overview of trends and developments of internet of things applied to industrial systems. In IECON 2018—44th annual conference of the IEEE industrial electronics society (pp. 2853–2860).
42. Talia, D. (2012). Clouds meet agents: Toward intelligent cloud services. *IEEE Internet Computing*, *16*(2), 78–81. https://doi.org/10.1109/MIC.2012.28.
43. Tanenbaum, A. S., & Bos, H. (2014). *Modern operating systems* (4th ed.). Upper Saddle River, NJ: Prentice Hall Press.
44. Togashi, N., & Klyuev, V. (2014) Concurrency in Go and Java: Performance analysis. In 2014 4th IEEE international conference on information science and technology (pp. 213–216). https://doi.org/10.1109/ICIST.2014.6920368.
45. Tosatto, A., Ruiu, P., & Attanasio, A. (2015) Container-based orchestration in Cloud: State of the art and challenges. In 2015 Ninth international conference on complex, intelligent, and software intensive systems (pp. 70–75). IEEE
46. van Moergestel, L., van den Berg, M., Knol, M., van der Paauw, R., van Voorst, K., Puik, E., Telgen, D., & Meyer, J. (2016). Internet of smart things, a study on embedding agents and information as a service. In ICAART
47. Veen, J. S. V. D., Waaij, B. V. D., & Meijer, R. J. (2012) Sensor data storage performance: SQL or NoSQL, physical or virtual. In 2012 IEEE Fifth international conference on cloud computing (pp. 431–438). https://doi.org/10.1109/CLOUD.2012.18.
48. Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, *10*(2), 115–152. https://doi.org/10.1017/S0269888900008122.