Received December 10, 2021, accepted December 23, 2021, date of publication December 24, 2021,
date of current version January 6, 2022.

Digital Object Identifier 10.1109/ACCESS.2021.3138586

A Customizable Architecture for Application-Centric Management of Context-Aware Applications

UNAI GANGOITI[1], ALEJANDRO LÓPEZ[1], AINTZANE ARMENTIA[1], ELISABET ESTÉVEZ[2], OSKAR CASQUERO[1], AND MARGA MARCOS[1], (Senior Member, IEEE)

[1]Systems Engineering and Automatic Control Department, University of the Basque Country (UPV/EHU), 48013 Bilbao, Spain
[2]Electronics and Automation Engineering Department, University of Jaén, 23071 Jaén, Spain

Corresponding author: Aintzane Armentia (aintzane.armentia@ehu.eus)

This work was supported in part by the Ministerio de Ciencia, Innovación y Universidades (MCIU)/Agencia Estatal de Investigación (AEI)/Fondo Europeo de Desarrollo Regional (FEDER), Unión Europea (UE), under Grant RTI2018-096116-B-I00; and in part by the Gobierno Vasco (GV)/Eusko Jaurlaritza (EJ) under Grant IT1324-19.

ABSTRACT Context-aware applications present common requirements (e.g., heterogeneity, scalability, adaptability, availability) in a variety of domains (e.g., healthcare, natural disaster prevention, smart factories). Besides, they do also present domain specific requirements, among which the application concept itself is included. Therefore, a platform in charge of managing their execution must be generic enough to cover common requirements, but it must also be adaptable enough to consider the domain aspects to meet the demands at application-level. Several approaches in the literature tackle some of these demands, but not all of them, and without considering the applications concept and the customization demands in different domains. This work proposes a generic and customizable management architecture that covers both types of requirements based on multi-agent technology and model-driven development. Multi-agent technology is used to enable the distributed intelligence needed to address many common requirements, whereas model-driven development allows to address domain specific particularities. On top of that, a customization methodology to develop specific platforms from this generic architecture is also presented. This methodology is assessed by means of a case study in the domain of eHealthCare. Finally, the performance of MAS-RECON is compared with the most popular tool for the orchestration of containerized applications.

INDEX TERMS Application-centric management, application-driven adaptability, context-aware applications, customizable management architecture, multi-agent systems, stateful availability.

I. INTRODUCTION

Current advances on information and communication technologies have allowed the expansion of the Internet of Things (IoT) [1], [2] as well as of its industrial variation, Industrial IoT (IIoT) [3], [4]. These paradigms are based on the universal interconnection of ''objects'' or ''things'' endowed with digital entities with the ability to measure or to process data, which allows the development of context-aware applications. Context-aware applications monitor their context to capture data that can be used just for supervisory purposes or for detecting abnormal situations with the aim of preventing or reacting to them. All this without human intervention. These context-aware applications belong to very different application domains, ranging from remote monitoring for natural disaster prevention [5], [6] or medical supervision [7], [8], to smart agriculture [9], [10] or flexible manufacturing systems (FMS) [11], [12].

Such different applications have some common requirements, as illustrated in Table 1. Context data are usually captured by embedded devices close to the physical environment, whereas processing tasks may require high performance equipment that is usually located far away (*distribution and node heterogeneity*). Therefore, these applications consist of different pervasive components that must communicate with each other, sometimes with time constraints (*timing requirements*). These applications might need to evolve with context changes (*adaptability*). Furthermore, sometimes co-operation among different applications is necessary to monitor the environment and/or to react to changes in it.

The associate editor coordinating the review of this manuscript and approving it for publication was Renato Ferrero.

TABLE 1. Examples of common requirements of context-aware applications.

DISTRIBUTION AND NODE HETEROGENEITY
• Prediction of volcanic eruptions: Sensors placed in the crater. Machine learning algorithms for data analysis.
• Smart irrigation systems: Sensors for soil and air moisture monitoring. Drones with cameras. Processing within the farm.
TIMING REQUIREMENTS
• Healthcare monitoring systems: Biomedical signals-with different measurement rates.
• Fire detection systems: Temperature and humidity sensors with different dynamic properties must be jointly processed.
ADAPTABILITY
• Early warning systems: Temporal, spatial and numerical resolution of active sensors according to the current criticality level.
• Nursing homes: Activation of remote monitoring of vital signs to provide emergency teams with useful information.
SCALABILITY
• Nursing homes: Admission or discharge of the elderly. Variations in the health status of residents.
• Early warning systems: Variations in the number of sensors according to the criticality level.
SECURITY AND PRIVACY
• Healthcare monitoring systems: Private access to medical data.
• Smart factories: Management of confidential data. Injuries, deaths and money loss due to security lacks.
SERVICE AVAILABILITY
• Smart factories: Rescheduling of manufacturing plan in case of machine failure.
• Healthcare monitoring systems: Avoiding dangerous situations for the patient.

As a result, applications and resources may join or leave over time (*scalability*), changing resource demand and/or availability accordingly. Finally, due to the sensitive nature of the captured and processed information, apart from securing data (*security and privacy*), it is also essential to minimize service interruption and recovering application execution from the stop point, even in case of node failure (*service availability)*.

Context-aware applications also present particularities of their own application domain, starting from the application concept itself. In eHealthCare (eHC) systems, an application can be understood as all the medical monitoring tasks needed to supervise the health of a person. However, an application in FMS can be understood as tracking the manufacturing of a set of products. In both domains, it might be necessary to resolve unexpected events, such as detecting health deterioration or a malfunction of a manufacturing station. Application structure is also domain dependent, as applications are composed of a set of domain entities that collaborate to achieve the application functional goals. Additionally, applications can have non-functional requirements which apply to all application entities (e.g., in a wildfire detection system, the number of sensors and the reading frequency may vary if temperature readings in an area increase).

Resources in which services are performed may also depend on the domain. Healthcare monitoring usually demands variable processing capabilities, connection to biophysical sensors or more complex sensors such as cameras. However, in FMS, specific manufacturing assets are required, such as assembling robots, milling/drilling machines, or automated guided vehicles (AGVs).

From an implementation point of view, different distributed software architectures have been used to develop context-aware applications, such as component-based systems [13], multi-agent systems (MAS) [14], service-oriented architectures [15] or microservices [16]. Any implementation of a distributed software architecture meets distribution, heterogeneity, scalability and timing requirements, and can be extended with security features. They also support starting and stopping of applications, and communication among their distributed modules. Platforms built on these software architectures also offer dynamic reconfiguration mechanisms to cope with adaptability and availability requirements [17]–[26]. However, what is not so common in these platforms is the consideration of the application concept, although it is necessary when requirements affect a set of application entities. In this case, typical requirements are temporal (e.g., end to end deadline) or context-related when it is necessary to make decisions on other applications. For instance, when a patient is being remotely monitored and some biophysical measurements exceed the established thresholds, new applications must be started to measure new biophysical variables. There are several proposals in the literature that attempt to undertake domain dependent demands but, in general, they are ad-hoc solutions and can hardly be applied in other domains. As far as the authors know, no platform covers all the requirements identified for context-aware applications.

Previous works of the authors proposed ad-hoc management platforms for context-aware applications, initially in the eHC field [27] and subsequently in the FMS domain [28]. The first platform had to be mostly redesigned to achieve the second one, as the application structure, resources and application management were completely different.

This work goes a step further, proposing an architecture for managing the execution of context-aware applications. It is based on a generic core that can be customized to concrete domains based on modeling artifacts. Specifically, the architecture contributes:

- The management of the execution of the application modules is driven by the key concept of application, understood as a set of interrelated domain modules.

TABLE 2. Platform requirements.

OPERATIONAL	
R1	Distributed execution and communication.
R2	Efficient application deployment.
R3	Life-cycle management.
SECURITY (NON-OPERATIONAL)	
R4	Security.
FLEXIBILITY (NON-OPERATIONAL)	
R5	Self-adaptability.
R6	Traceability/Self-awareness.
R7	Self-healing.
R8	Domain variability

- The logic of adaptation to relevant context changes is independent of the application functionality, being possible to act on applications.

The use of multi-agent technology allows decentralized decision-making by introducing intelligence within the domain application modules Thus, the architecture supports decentralized deployment, and fast service recovery for stateful applications through negotiation mechanisms. The latter is based on multiple replica management and supports even node failure. A customization methodology is presented to cope with domain specific particularities. It bases on the use of model-driven development for application definition and on a set of provided agent templates for agent development.

The remainder of this paper is organized as follows. Section II presents the main requirements that a management platform for context-aware applications must meet as well as how they have been addressed in the literature. Section III is devoted to the core architecture, whose design is mainly focused on fulfilling flexibility requirements from an application-centric point of view. Section IV describes the methodology for adapting this core architecture to a specific domain, which is illustrated through a case study in the eHC field in Section V. Section VI assesses the performance of the proposal in comparison with the most popular tool for the orchestration of containerized applications and, finally, Section VII highlights some concluding remarks and future work.

II. RELATED WORK

The main objective of an application management platform is to ensure that applications execute as specified. As commented above, context-aware applications present common and domain specific demands from which the main requirements that a platform must meet can be derived. Table 2 collects these platform requirements which can be divided into two groups: operational (R1-R3), which tackle application execution; and non-operational, dealing with security (R4) and flexibility (R5-R8).

From an operational point of view, the applications, deployed in heterogeneous devices, perform acquisition, processing and actuation tasks. The platform must enable the distributed execution of these tasks as well as the communication among them (*R1: Distributed execution and communication*). Besides, support for efficient deployment is necessary, taking into account resource availability and application demands (*R2: Efficient application deployment*). Added to this, context-aware systems consist of a set of applications that are dynamic in number and size, each with its own timing requirements, whose startup, stop and normal operation must be controlled (*R3: Life-cycle management*).

Concerning non-operational requirements, system security requires mechanisms to assure the privacy, confidentiality, authentication and integrity of data (*R4: Security*). It is important to remark that context-aware applications are included within the so-called self-adaptive systems, so they also require "self-capabilities" to autonomously adapt to changes in their environment. This implies not only context-awareness but also self-awareness [29]. To achieve context-awareness, the platform must be endowed with mechanisms for application-driven dynamic reconfiguration that allow applications to react to relevant situations by changing their behavior (*R5: Self-adaptability*). Self-awareness implies being aware of dynamic resource availability. To that end, the platform must track both the state of the infrastructure resources and the state of applications (*R6: Traceability/Self-awareness*).

Regarding resource availability, the platform must minimize service interruptions, including failure detection and automatic service recovery, while maintaining the application state (*R7: Self-healing*).

Finally, every application domain has its particularities in terms of application specification (concepts that define applications and their relationships) and execution management, or even in terms of resource types. To draw on the great effort involved in the design and development of a management platform, it would be beneficial to have a platform customizable to different domains (*R8: Domain variability*).

The next subsections analyze the related work that addresses the requirements identified. Table 3 collects the analysis of the main management platforms related to the particular case of flexibility requirements identified in Table 2.

A. OPERATIONAL REQUIREMENTS

Distributed software architectures consider applications as a set of modules (computational units) that run on different nodes and interact to achieve application functionality. However, module definition and module composition differ from one architecture to another. For example, in Component-Based Software Engineering (CBSE) [13], components are developed as black boxes that offer services in an application independent way. Applications are compositions of components based on their interface or following a component model. Applications based on MAS consist of intelligent and loosely-coupled software components, named agents, which are autonomous (they make decisions without direct human intervention), proactive (they have goal-directed behavior), reactive (they react to context changes) and social

TABLE 3. Compliance with Flexibility (Non-Operational) requirements by execution management platforms.

PLATFORM	R5: SELF-ADAPTABILITY		R6: TRACEABILITY /SELF-AWARENESS		R7: SELF-HEALING			R8: DOMAIN VARIABILITY	
	App. driven	App. as target	Focus	Dynamic Model	App. unaware	State integrity	Node failure	App. concept	Generic
[25]	No	No	CC	No	---	---	---	No	Yes
ACCADA [17]	No	No	CC	No	---	---	---	No	Yes
MUSIC [18]	No	No	CC	No	No	Yes	No	No	Yes
DARE [19]	No	No	CC	Yes	Yes	Yes	No	No	Yes
[20]	No	No	CC	Yes	---	---	---	No	Yes
THOMAS/PANGEA [21]/[22]	Restricted	Yes	OC	Yes	Yes	No	No	No	Yes
iLAND [23]	No	No	CC	Yes	Yes	No	Yes	Yes	No
DAMP [24]	Yes	Yes	AC	Yes	Yes	Yes	Yes	Yes	No
EI4MS [26]	No	No	UQC	Yes	Yes	Yes	No	No	Yes
MAS-RECON	Yes	Yes	AC	Yes	Yes	Yes	Yes	Yes	Yes

CC = Component-Centric, OC = Organization-Centric, AC = Application-Centric, UQC = User-perceived QoS-centric

(they interact among them, by co-operating or competing with each other) [14]. In Service Oriented Computing (SOC) [15], computational units are called services. Services are published by providers at repositories such as black boxes which consumers can discover and use, or even compose, creating new services. In the last years, the advent of microservice architectural style has allowed small and loosely-coupled services communicating through light-weight protocols, to be developed and deployed independently to compose highly scalable distributed applications [16]. The use of containerization technologies that enable lightweight virtualization has become de facto standard for packaging microservices in the cloud [30].

There are approaches aimed at easing the development and/or management of applications based on distributed architectural styles. They usually provide mechanisms for deploying, communicating and managing the life-cycle of application modules. For example, this is the case of the Java Agent Development (JADE) [31], the most used implementation of the Foundation for Intelligent Physical Agents (FIPA) [32] standard for MAS; and Kubernetes [33], the most popular tool for the orchestration of containerized microservice-based applications. Kubernetes is usually combined with frameworks such as the Robot Operating System (ROS), which supports communications and allows orchestrating services among distributed nodes, mainly in the field of robotic applications [34], [35].

A management platform built over any of these approaches or built directly over a distributed software architecture, as those illustrated in Table 3, directly meet R1 requirement (Distributed execution and communication) and, at least, a basic version of R2 (Efficient application deployment) and R3 (Life-cycle management).

B. NON-OPERATIONAL REQUIREMENTS

Distributed platforms can be extended by mechanisms which allow non-operational requirements to be met. The following subsections discuss research done in this direction.

1) SECURITY (R4)

A complete survey on mechanisms for ensuring secure access, storage, processing and transmission of data is presented in [36]. The most common solutions are Public Key Infrastructures (PKI), encryption, Secure Socket Layer (SSL), authentication and authorization mechanisms, and blockchain. As all these can be included in a platform without affecting the application management, it has been considered out of the scope of this work.

2) SELF-ADAPTABILITY (R5)

Self-adaptation is usually based on the implementation of MAPE-K loop models (i.e., to apply feedback loops from control theory to autonomic computing) [37]. Self-adaptability is a complex task that can be divided into

four phases: (1) monitoring/collection of context parameters; (2) detection of relevant changes by means of the analysis of the collected data; (3) planning of appropriate adaptation actions to respond to the changes; (4) execution of the planned actions.

The first two phases are highly dependent on the specific context, which varies from one field to another [38]. Some research efforts focus on context specification [20], [39], while others, such as the one selected in this work, consider context monitoring as part of the application functionality.

Several techniques have been proposed to select adaptation actions in phase 3. Self-adaptability can be expressed through application variability [40], defined in terms of variation points (where a planned change can occur) and variants (options that can be selected). In [25] every component is considered as a variation point, and implementations as the corresponding variants, each related to a specific situation in the context. Other works apply rules, one of the most used solutions, since they provide an easy and automatable classification method. For example, [20] and [41] use rules to detect and classify relevant situations, reasoning the best response.

The actions to be executed at phase 4 range from fixed and ad-hoc proposals, such as simple warnings [41] or alarms triggering [42], to more flexible ones, based on dynamic reconfiguration. In the case of dynamic reconfiguration, an external entity automates and manages adaptation execution, separating adaptation logic from application logic. Dynamic reconfiguration has been applied at two levels: component and application. Most approaches work at component level, it being possible to add, remove, replace, and/or reconnect application modules. Sometimes, adaptation is restricted to external requests, as in the DARE framework [19], which limits the autonomy of applications. Other times, the platform itself is responsible for detecting context changes and selecting the component implementations that best fit a new context state, as in the MUSIC project [18], the platform in [25] and the ACCADA framework [17]. Similarly, the evolution-oriented EI4MS architecture [26] detects degradation on the user-perceived QoS and calculates an optimal evolution plan which is executed by the microservices themselves. To that end, the platform must be aware of the concrete context view. As adaptation at component-level does not cover the application concept, there have been attempts to extend dynamic reconfiguration to the application level. The THOMAS platform [21] and its successor PANGEA [22] combine agent and service oriented technologies and allow structural organizations of agents. In this case, dynamic reconfiguration involves either the incorporation of new organizational structures, or the addition or removal of members. However, these capabilities are restricted to certain agent roles. In the iLAND middleware [23], dynamic reconfiguration consists of time-bounded re-composition of running service-based applications, and it is initiated by the middleware when applications are started or stopped. Previous works of the authors [27] and [24] go a step further, allowing components to ask for adaptation actions targeted to the whole application. In [27], it is possible to start and stop already deployed applications. In [24], an ad-hoc solution for the eHC field deploys applications only when needed and allows modifying application configuration.

3) TRACEABILITY/SELF-AWARENESS (R6)

Some of the analyzed management platforms trace the system state to make the most suitable decisions at runtime. Most use a kind of repository, which varies from one platform to another. For instance, the DARE framework [19] maintains only the configuration map (i.e., the mapping of the components in execution to nodes), which is automatically discovered by means of gossiping techniques. Platforms in [25], [17] and [18] make runtime decisions based on adaptation models provided at design time. These models contain implementation alternatives according to different context values. The composition algorithms of the iLAND middleware [23] handle an application model annotated with QoS parameters that refer to data processing and resource needs of the application services. The system model used in EI4MS to elaborate evolution plans describes current deployment state of the system through information about the existing logical services, available cloud/edge nodes, user demands, and so on [26]. In [20], the context model is separated from the system model, but both include dynamic aspects that relate them at runtime. It is worth mentioning that all these works consider an application as a simple graph of interacting components with several realizations or implementations, except the latter work which allows a hierarchical component definition. In addition, they are all component-centric proposals that do not consider the application concept as a set of interrelated components managed as a whole. As a result, they cannot manage application-centric management to cope with application level demands.

There are approaches that attempt to define more complex application structures. In the context of MAS, agent-oriented engineering methodologies that take into account social concepts have been proposed for the so-called open MAS [43]: a dynamic set of agents, which may be provided by different developers, with self-interested behaviors. Specifically, these methodologies allow specifying agent-societies or agent-organizations composed of several agents, playing different roles, whose interactions are led through a set of rules, norms and constraints [44]. Based on the idea of agent-societies, the authors in [21] and [22] propose platforms for the runtime management of dynamic virtual organizations in open MAS. They provide facilities for agents to voluntarily enter or leave a virtual organization as well as for on-demand creation, deletion and modification of virtual organizations. However, it is precisely the self-interested nature of this type of agent which makes application-centric management impossible. The DAMP platform [24] considers an application as a unique entity in order to achieve application QoS enforcement. It provides a middleware service that allows application registration before its execution.

This information, together with monitoring of resource availability, is used at runtime to reconfigure applications when needed. Previous works of the authors also go beyond simple application graphs and consider applications as whole entities: [27] intended for reconfiguration driven by the application in the eHC domain, and [28], [45] aimed at fault tolerance in flexible manufacturing. However, the proposed application structures and, therefore, the corresponding system models, are fixed, which makes them non-customizable to domains with a different application concept.

4) SELF-HEALING (R7)

Self-healing is understood as the system's capacity to detect when a service becomes unavailable and to restore it. It can be proactive or reactive [46]. Proactive self-healing implies prevention, i.e., to detect service degradation before failure. Prevention tasks can be considered from the design phase, as a particular case of self-adaptability, and it is even possible to decide the optimal time to react. In contrast, reactive self-healing is a more challenging issue as it intervenes when the failure has already occurred, and allows recovery from sudden component or node failures that cannot be foreseen. Sometimes, it is even necessary to maintain the consistency of the application state.

The works in [18] and [47] support reactive self-healing by means of programming. However, although including specific code in application modules allows fast failure detection and recovery, the application logic must ensure its own availability. To avoid this dependency, replication strategies provided by the platform (transparent to the application) have been extensively applied [48].

Failure detection mechanisms are usually based on heartbeat messages. Two approaches are distinguished: 1) gossip: a component or a node informs of its liveliness; 2) probe: an entity requires components or nodes to confirm they are still alive [49]. Some works propose a centralized management of heartbeat messages through a platform module in charge of detecting node failures. For example, the DAMP platform [24] and the iLAND middleware [23] use gossip techniques, whereas [50] is based on probe mechanisms. Another centralized proposal is presented in [51] which makes use of the application programming interface (API) of Kubernetes to monitor server events that indicate pod failures. There are also decentralized proposals that improve systems' autonomy and detection capacity. In the DARE framework [19], every node hosts a module in charge of gossiping to report possible node failures, whereas in [49] all nodes probe others' failures on their own.

Regarding failure recovery, some works propose a central entity with a global view of the whole system. This approach enables the separation of recovery logic from application logic and the most suitable decisions can be made. For instance, [24] and [23] make use of re-composition algorithms to select the best replica, whereas the architectures in [50] and [19] have a specific module to determine whether a failure can be recovered or not. The consistency of the application state is a relevant point when recovering a failure, as it assures full-service continuity. Two main approaches can be found. On the one hand, check-pointing-based-recovery allows the rollback of the system to its most recent coherent state [49]. For this purpose, not only is it necessary to store the system state, but also the messages received between checkpoints. On the other hand, an easier and more flexible solution is to provide means to transfer and restore the application state. In [24] components periodically send their state to the platform, which stores it for its restoration in the new selected implementations. However, this causes an overhead on the platform and it is only possible for periodic components. In [51] the so-called State Controller component is integrated with Kubernetes to allow stateful service recovery of pods. Although it considers elasticity (i.e., multiple active pods offer the same service), state transfer is limited to concrete pairs of pods.

C. LITERATURE ANALYSIS CONCLUSIONS

In conclusion, to meet non-operational requirements, management platforms extend the implementation of a distributed software architecture. Dynamic reconfiguration is the best mechanism to accomplish self-adaptability and self-healing. In both cases, decentralized approaches improve system autonomy, decreasing platform overhead. However, a global vision of the whole system is needed to make decisions that best fit the needs of all running applications. As far as the authors know, management platforms usually focus on some, but not all, of the identified requirements. Additionally, there is a lack of application-centric management, as most proposals do not consider applications as an entity. And if they do, it is considering an ad-hoc and/or fixed structure. Achieving application-centric management requires the platform to be aware of the application concept of a specific domain. An ad-hoc definition of applications makes the corresponding management platform also an ad-hoc solution [24], [27], [28]. Adapting these platforms to other domains involves redesigning and/or re-implementing them, as in the case of the previous works of the authors [27], [28] or in the MASHA architecture (which was initially developed for web sites [52] and later adapted to eLearning systems [53]). Therefore, having a generic and customizable architecture would reduce or even avert the necessity for this hard work (R8: Domain variability). For this, it is essential to be able to define application structure in an abstract way.

III. MAS-RECON: A CUSTOMIZABLE AND APPLICATION-CENTRIC ARCHITECTURE

This section presents the MAS-RECON architecture (see Fig. 1): a generic and application-centric proposal which can be customized to specific domains.

In order to meet the operational requirements (R1-R3), the architecture is based on multi-agent technology, which has been widely used for the development of complex systems and allows the distribution of decision-making [14]. As depicted in Fig. 1, without loss of generality, it has been

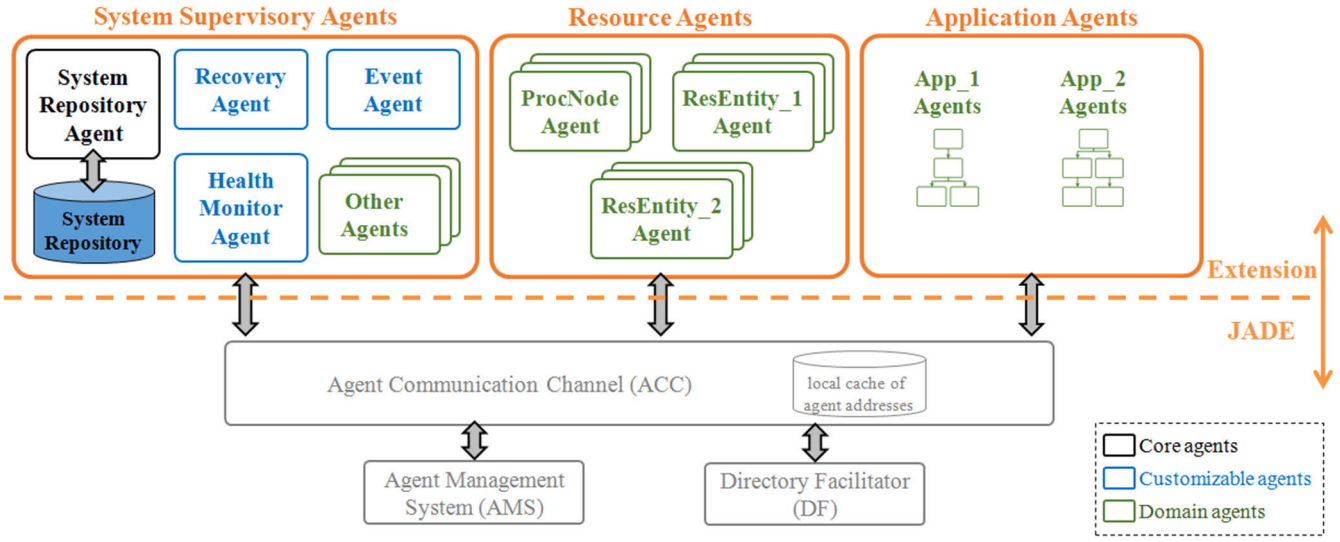


FIGURE 1. MAS-RECON architecture. It is based on an implementation of the FIPA specification to meet R1-R3 requirements. Decentralized decision-making is performed by domain entities: resources and applications, running as Resource Agents and Application Agents, respectively. System Supervisory Agents provide system-level supervision to meet different requirements: System Repository Agent focuses on R6 and R8, Event Agent on R5, and Health Monitor Agent jointly with Recovery Agent on R7. The System Repository Agent is the core of the architecture, common to all domains, whereas the other System Supervisory Agents can be customized to address domain particularities.

built upon the JADE framework [31], but any other implementation of the FIPA specification [32] could be adopted. MAS-RECON allows decentralized decision-making by introducing intelligence within the domain entities (*Resource Agents* and *Application Agents* in Fig. 1). At the same time, all these entities are supervised at system level (*System Supervisory Agents* in Fig. 1). The information needed to achieve system-level supervision is stored in the *System Repository* (SR), which is managed by the *System Repository Agent* (SRA) alone. These two entities are the basis for the application-centric management, contributing to meet R6 and R8. The *Event Agent* (EvA) focuses on R5, supporting application-driven reconfiguration. Finally, R7 is achieved through the *Health Monitor Agent* (HMA) and the *Recovery Agent* (ReA), which supervise resources and application agents for failure detection and recovery, including the case of stateful applications (those whose current execution state depends on previous ones). Finally, to tackle specific domain dependent requirements, such as elaborated mechanisms for admission control, new System Supervisory Agents might be included (*Other Agents* in Fig. 1).

The following subsections describe the MAS-RECON architecture as well as the mechanisms used to meet the requirements.

A. ARCHITECTURE CORE (R6 AND R8)

One of the contributions of this work is the architecture core common to all domains, which consists of the SR and the SRA, which are responsible for maintaining the state of the complete system, understood as the relevant information to tackle traceability/self-awareness (R6) and domain variability (R8).

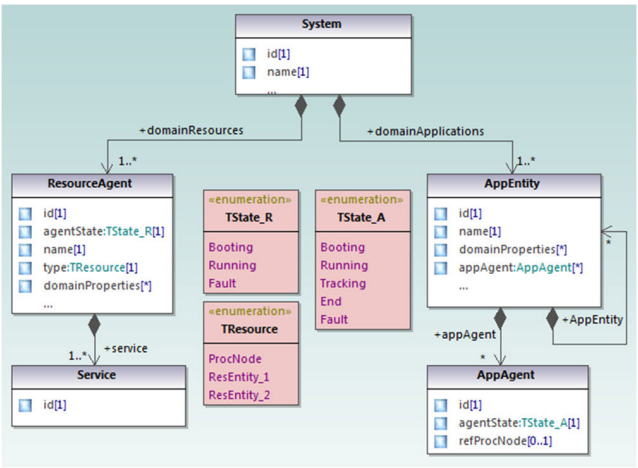


FIGURE 2. Generic structure of the System Repository. The meta-model identifies the domain entities that compose the SR (resources and applications), the relations among them, and their characterization. Hierarchical relationships among application entities are represented by the AppEntity composition links. All elements have a unique identifier assigned by the platform (id). In the case of the services offered by resource entities, their semantics are known only to domain entities, being transparent to the platform.

The SR is a model that represents the system state from an application-centric management point of view. The meta-model of the SR is depicted in Fig. 2. It is closely related to the core architecture presented in Fig. 1. It contains information related to the domain entities depicted in Fig. 1 (*Resource* and *Application Agents*). This information comprises properties needed for application management, common to all domains (e.g., *id* and *agentState*), and domain dependent properties (marked as *domainProperties* in Fig. 2). These

latter are determined when application concept is defined, as it is detailed in Section IV for the particular case of eHC.

The SR collects the set of Resource Agents (*Resource Agents* in Fig. 1 and *ResourceAgent* in Fig. 2), which represent available resources, and which are characterized by the services they offer to applications (*Service* in Fig. 2). Resource entity types are enumerated in the SR (*TResource* in Fig. 2): the processing node, which is the unique resource entity common to all domains as it hosts Application Agents and System Supervisory Agents (*ProcNode* in Fig. 2 and *ProcNode Agent* in Fig. 1), as well as those related to concrete domains (for instance, *ResEntity_1* and *ResEntity_2* in Fig. 2, and *ResEntity_1 Agent* and *ResEntity_2 Agent* in Fig. 1).

The SR also collects the set of applications that can be executed (*domainApplications* in Fig. 2). Applications are defined as a set of entities (*AppEntity* in Fig. 2) that are interrelated according to the application structure defined for the specific domain. Hierarchical and/or dependency relationships might exist among them. Applications must be registered before requesting their execution. At runtime, every registered application entity has at least one associated agent (*AppAgent* in Fig. 2 that corresponds to an *Application Agent* in Fig. 1). This will be explained in Section III.D, when discussing how MAS-RECON manages application availability. Thus, the state of the application is defined by the state of its corresponding Application Agents.

The SRA provides a unique access point to the SR through the generic API shown in Fig. 3.a. It allows the registration of resources and applications (*iRegAgent* and *iRegApplication* interfaces, respectively), the starting and stopping of applications (*iExecManagement* interface), and application information management (*iSystemInfo* interface).

B. OPERATIONAL REQUIREMENTS (R1-R3)

Meeting operational requirements covers the starting, stopping and normal operation of applications, taking into account that MAS-RECON relies on multi-agent technology and that the system state is stored at the SR.

Applications must be registered before being started (*Application Registration* in Fig. 3.b). The use of meta-modeling techniques to define the SR allows the SRA to handle a generic registration process that assures that applications conform to the application structure defined for the domain (*IRegApplication* interface in Fig. 3.a). Application registration is carried out in two phases. The initial phase consists of the iterative and unitary registration of all the entities that compose the application (*RegAppEntity* method in Fig. 3.a), one by one following a top-down order, according to the application hierarchy. The second phase involves the validation of the fully registered application (*AppValidation* method in Fig. 3.a). The so-called *Launcher Agent* in Fig. 3.b represents a domain-specific System Supervisory Agent that provides external users with access to application management (it belongs to the *Other Agents* group of Fig. 1).

Resource entities register their corresponding Resource Agents when they are booted (*Resources Startup* in Fig. 3.b) by means of the *RegResAgent* method of the *IRegAgent* interface. Fig. 4.a presents the state diagram of the finite state machine (FSM) that describes the generic behavior of Resource Agents. The architecture provides the Resource Agent code-skeleton that implements this FSM. Once started, Resource Agents perform two tasks. On the one hand, they supervise the related physical resources (e.g., processing nodes can monitor available free memory). On the other hand, they are provided with negotiation mechanisms to decide, in a distributed way, the most suitable resource to perform a task according to specific criteria.

The *IExecManagement* interface offered by the SRA (see Fig. 3.a) allows starting and stopping applications. Starting an application implies the registration, instantiation and deployment of all the Application Agents related to its registered entities. A top-down process is proposed, divided into two phases:

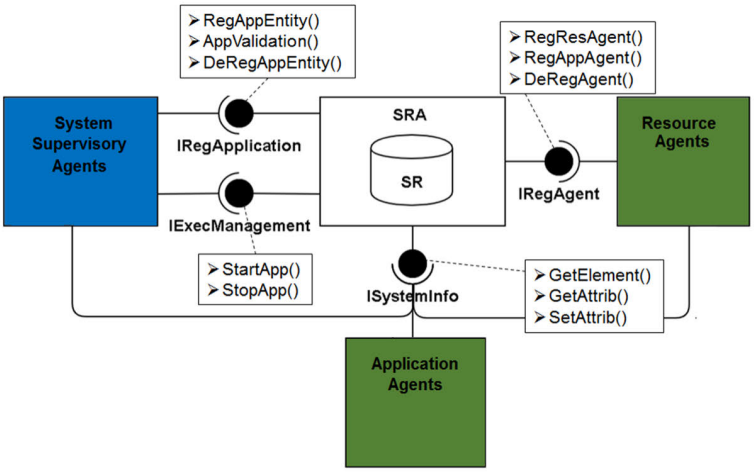
- Phase 1 (*Application Startup* in Fig. 3.b): the generic startup of first-level application entities. It is initiated upon the invocation of the *StartApp* method in Fig. 3.a. The SRA looks for the processing nodes offering the required services and launches a negotiation process among their corresponding Resource Agents (*Negotiation Process* in Fig. 3.b), including: agent data, negotiation criteria, and actions to be executed by the winner. In the example of Fig. 3.b: the memory required by the new agent and the class that implements it as agent data, the maximum free memory as negotiation criterion, and as winner actions: to register (*RegAppAgent* method), create and deploy the corresponding Application Agent (*createAgent* in Fig. 3.b).
- Phase 2 (*Domain dependent actions* in Fig. 3.b): the subsequent startup of lower-level entities in a decentralized way. Each Application Agent performs the startup of those at the next lower level. Being dependent on the concrete structure of the application, it is a domain-specific phase (see Section IV).

The FSM depicted in Fig. 4.b presents the generic behavior of Application Agents, implemented on the Application Agent code-skeleton, also provided by the architecture. At booting, they update the SR with the processing node in which they have been deployed (*refProcNode* property of the *AppAgent* element in Fig. 2) and start lower-level entities. Once booted, they execute their piece of application functionality until stopped.

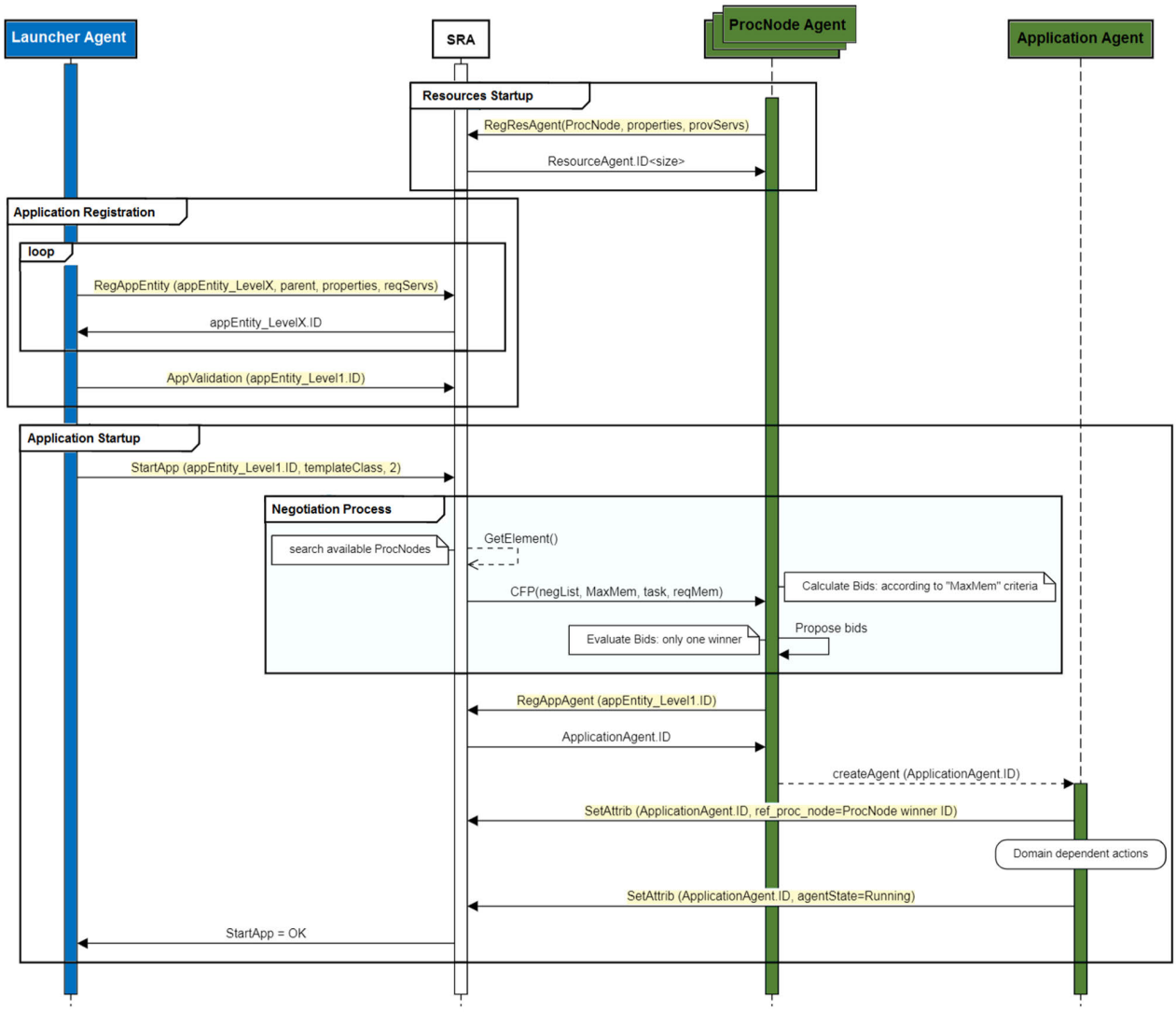
Application stopping is requested through the *StopApp* method of the *IExecManagement* interface and follows the reverse process. Lower-level Application Agents deregister themselves (*DeRegAgent* method in Fig. 3.a) and stop in a down-top sequence.

C. SELF-ADAPTABILITY (R5)

MAS-RECON contributes an application-driven reconfiguration approach, which is based on the notion of MAPE-K loop models [37] and handles two concepts: (1) *Event*: it identifies a relevant context change; and (2) *Action*: the reaction to



a) Generic API of the System Repository Agent



b) Sequence diagram for the startup of resource and application entities

FIGURE 3. Definition and use of the generic API of the System Repository Agent (color palette: core agent in white; customizable System Supervisory Agents in blue; Resource and Application Agents in green). The figure at the top presents the interfaces defined for registering application entities and Domain Agents, starting and stopping applications, and getting or updating the information collected at the SR. The figure at the bottom describes the startup process of resource and application entities, through the use of this API (highlighted in yellow).

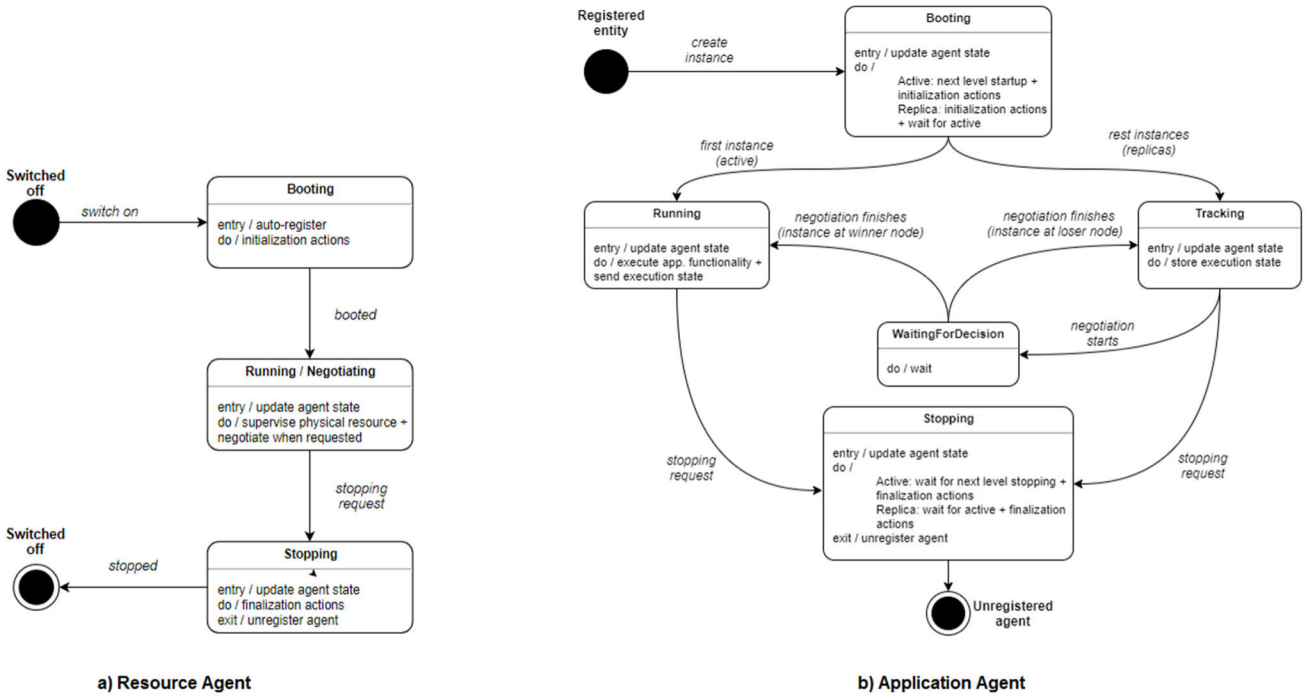


FIGURE 4. State diagram for the generic behavior of Domain Agents, implemented as the agent code-skeletons provided by the architecture: a) Resource Agents; b) Application Agents.

an event consists of executing a set of actions, each one targeted to an application (itself or another one), which might even follow a concrete execution order. To make application logic independent of adaptation logic, events and actions must be declared during application registration, so that the EvA (see Fig. 1) performs a centralized supervision of the adaptation process. For this, the *IEvent* and *IAction* interfaces depicted in Fig. 5 have been defined.

As events and actions are domain-specific, they are specified in the domain application meta-model. Section IV describes how to adapt the characterization of these elements to a specific domain.

MAS-RECON assumes that the first two phases of self-adaptation are part of the application functionality. Specifically, they are performed by the Application Agents in charge of acquiring and processing context data (*Event Trigger Application Agent* in Fig. 5). Therefore, context particularities are unknown by the platform. The EvA is provided with the *IEvent* interface, which allows these Application Agents to report on detected events.

Then, the EvA searches the SR for the corresponding actions, and is responsible for launching and supervising their execution, through the *IAction* interface implemented by other Application Agents (*Action Performer Application Agents* in Fig. 5).

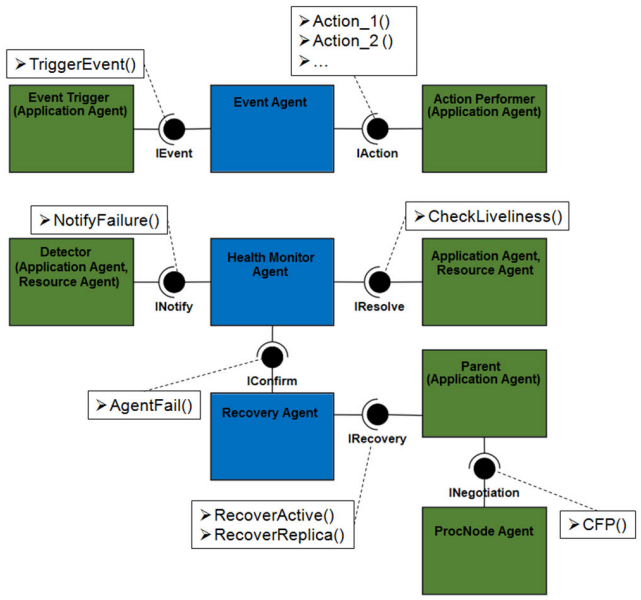


FIGURE 5. Interface definition for interactions among System Supervisory Agents and Resource and Application Agents, related to flexibility requirements: self-adaptability (R5) and self-healing (R7) (color palette: customizable System Supervisory Agents in blue; Resource and Application Agents in green).

D. REACTIVE SELF-HEALING (R7)

Following the idea of decentralized decision-making and system-level supervision, MAS-RECON supports reactive self-healing by distributed failure detection (Resource and Application Agents) with centralized verification and recovery supervision (HMA and ReA). For this, the *INotify*, *IResolve*, *IConfirm* and *IRecovery* interfaces depicted in Fig. 5

have been defined. It abstracts the design of application functionality from self-healing mechanisms. Besides, it covers isolated agent failures and crashes at processing resources which affect several application entities belonging to different applications. However, failures at domain specific resources are not covered by the architecture. MAS-RECON defines the sequence of messages from failure detection to recovery (summarized in Fig. 6), which determines the interactions among agents through the interfaces depicted in Fig. 5.

Failure detection relies on the potential of underlying MAS framework (e.g., JADE) to report on the non-delivery of a message (*Detect* in Fig. 6). It is undertaken by Resource and Applications Agents (identified as *Detector Agent* in Fig. 5 and Fig. 6), in the same way for all domains, through the *INotify* interface of HMA (see Fig. 5). As it is generic, the code for failure detection is included as part of the agent code-skeleton provided by MAS-RECON.

Considering that several domain agents can detect the same failure, the HMA is the sole receptor of all the notifications, and it performs a centralized verification of the failure (*Resolve & Confirm* in Fig. 6). This includes resolving whether the notification relates to a new failure (*CheckLiveliness* method of the *IResolve* interface in Fig. 5). Previously reported failures are ignored, whereas new ones are confirmed to the ReA (*AgentFail* method of the *IConfirm* interface in Fig. 5). The global state of the system contained in the SR allows the HMA to decide which Application Agents have to be recovered.

MAS-RECON proposes replica-based management of the application execution state to assure its consistency, essential in the case of unforeseen failures. Thus, as shown in Fig. 2, an application entity (*AppEntity* in Fig. 2) can have n+1 associated Application Agents (*AppAgent* in Fig. 2). One of them is the active instance that performs the piece of application functionality. The others are the replicas that just keep track of the execution state of the active. For this, Application Agents are provided with mechanisms that allow an indirect state transfer (i.e., the agent itself exports its execution state and restores it, when necessary), which is also implemented in the provided application agent code-skeleton.

The ReA supervises the failure recovery process (*Recover* in Fig. 6), which focuses on minimizing the unavailability of an active instance and maintaining the replication factor. It depends on whether the failed agent is the active or a replica:

a) Active agent failed: the ReA looks the SR for the failed agent's parent agent (*Parent Agent* in Fig. 6), according to the application hierarchy registered, and orders it to select the most suitable replica to be the next active agent, and to create a new replica agent (*IRecovery* interface of the *Parent Application Agent* in Fig. 5). The former implies launching a negotiation process among the processing nodes that hold the replicas (*INegotiation* interface in Fig. 5). During this negotiation, the replica agents wait at the *WaitingForDecision* FSM state (see Fig. 4). The replica located at the winner node will be the new active one. The latter is similar to the startup process. This contributes a straight stateful recovery of application execution, as the new active agent continues executing its piece of application functionality from the last known state just after negotiation process is finished.

b) Replica agent failed: the ReA looks the SR for the failed agent's parent agent (*Parent Agent* in Fig. 6), according to the application hierarchy registered, and orders it to start a new replica agent (*IRecovery* interface of the *Parent Application Agent* in Fig. 5).

IV. METHODOLOGY FOR DOMAIN-SPECIFIC PLATFORM DEVELOPMENT

This section presents a methodology to customize MAS-RECON to obtain a domain platform, which is divided into three steps:

1) Specification of the target domain, including the application concept. A domain meta-model is defined to identify, characterize and relate resource and application entities. The SR implements this domain meta-model, which allows a generic application registration that assures application correctness.
2) Definition of the templates to develop the domain agents that represent registered entities at runtime, based on the agent code-skeleton provided by MAS-RECON.
3) If needed, extension of customizable System Supervisory Agents to include domain particularities related to application-centric management.

The following subsections describe these steps, illustrated through the case study depicted in Fig. 7, which is targeted at the eHC field. The use of compact and portable health sensing components has allowed the development of distributed and person-centric eHC applications. They are context-aware applications which monitor their environment (the patient) and must react when an abnormal situation is detected. An in-depth description of the case study can be found in [27].

A. STEP 1: DOMAIN SPECIFICATION

The use of a model-based approach for domain specification allows the SRA to provide a generic registration process and a generic management of the SR. The proposed approach involves the definition of the domain meta-model that determines the structure, rules and restrictions that the SR must follow [54]. Specifically, the meta-model of the SR (see Fig. 2) is implemented as an eXtensible Markup Language (XML) Schema (XSD) [55] (SR.xsd), which is composed by two other schemas:

- The *Concepts* schema defines the domain concepts. It identifies the domain entity-types: resource entities (*TResource* enumeration in Fig. 2) and application entities (*AppEntity* in Fig. 2). It also identifies their relevant characteristics, from their management point of view (*domainProperties* of *ResourceAgent* and *AppEntity* in Fig. 2). For this, XML elements and attributes are

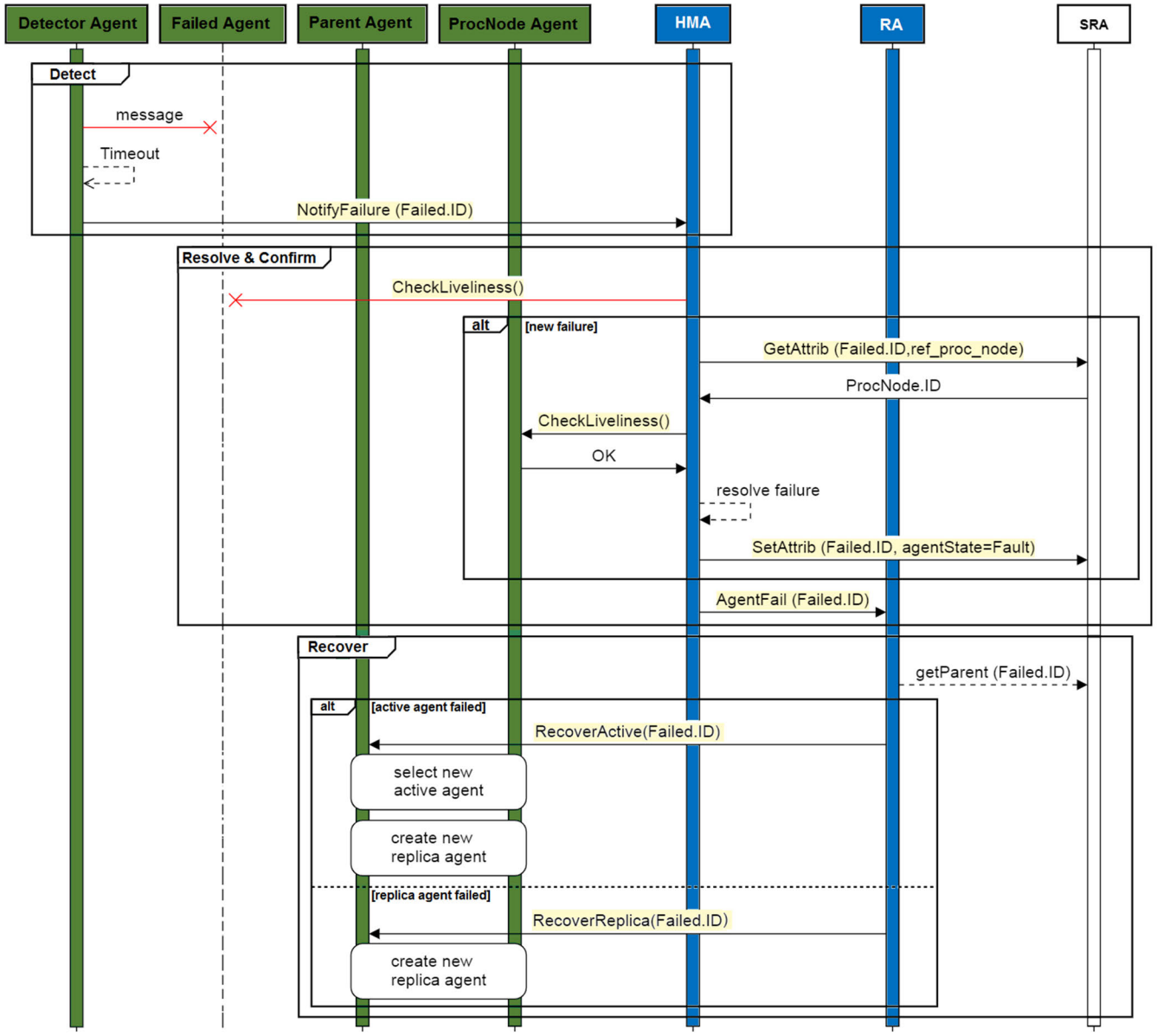


FIGURE 6. Reactive self-healing in MAS-RECON (color palette: core agent in white; customizable System Supervisory Agents in blue; Resource and Application Agents in green). The sequence diagram starts with the detection of the failure of an Application Agent (Failed Agent). It also describes the message sequence managed by the HMA to resolve and confirm the possible failure. Finally, the recovery process of confirmed failures is supervised by the ReA. The use of the API provided by the SRA is highlighted in yellow.

used, respectively. As adaptation actions are domain dependent, this schema also includes Event and Action concepts.

- The *Hierarchy* schema states the allowed relationships among concepts (relations between *AppEntity* elements in Fig. 2). Application hierarchy is defined through "parent-child" elements (composition of *AppEntity* elements in Fig. 2), whereas "Key/Keyref" constructs are used for dependency constraints.

The schema for SR (SR.xsd) extends the Hierarchy schema with properties common to all domains and used by MAS-RECON to fulfill all the requirements previously identified.

These schemas are used during the registration process of applications (*Application Registration* in Fig. 3.b). At the initial phase, every application entity is validated against the Concepts schema to assure its correctness, by means of the *RegAppEntity* method of Fig. 3.b. Then, the completely registered application is validated against the Hierarchy schema to ensure that it is well-formed, by means of the *AppValidation* method of Fig. 3.b. A detailed description of the registration process and its validation algorithm is found in [56].

Fig. 8 depicts the Concepts and Hierarchy schemas related to the person-centric eHC applications illustrated in Fig. 7. The *Patient* is considered the first-level application entity-type of the domain, which groups together a set of

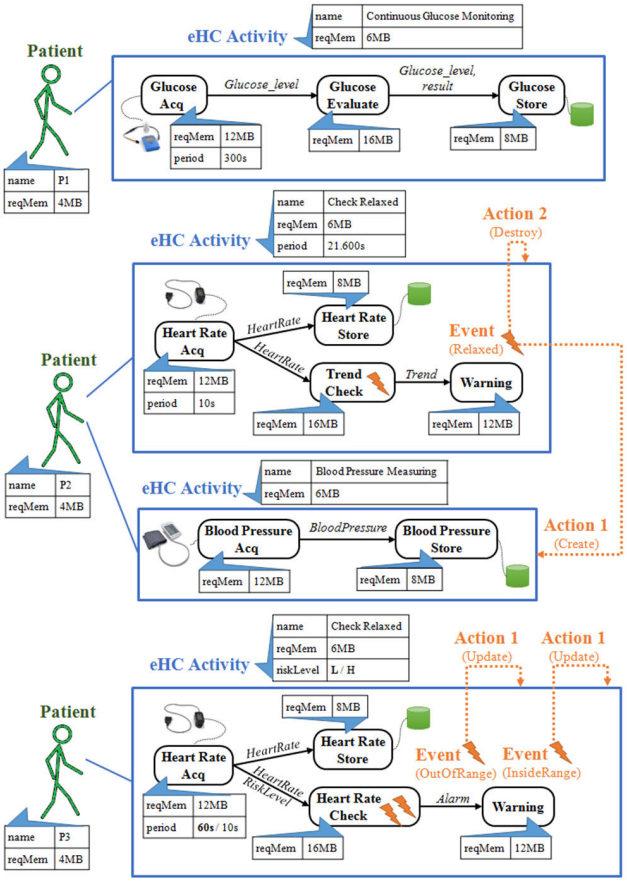


FIGURE 7. General scenario for the eHC case study that illustrates the lexicon of the domain. It depicts: 1) the concepts related to person-centric eHC applications (Patient, eHC Activity and Task); 2) the relationships among them (hierarchical relations as eHC Activities composed by Tasks, and dependency relationships as in the case of Actions targeted to eHC Activities); and 3) their technical characterization (e.g., required memory or period).

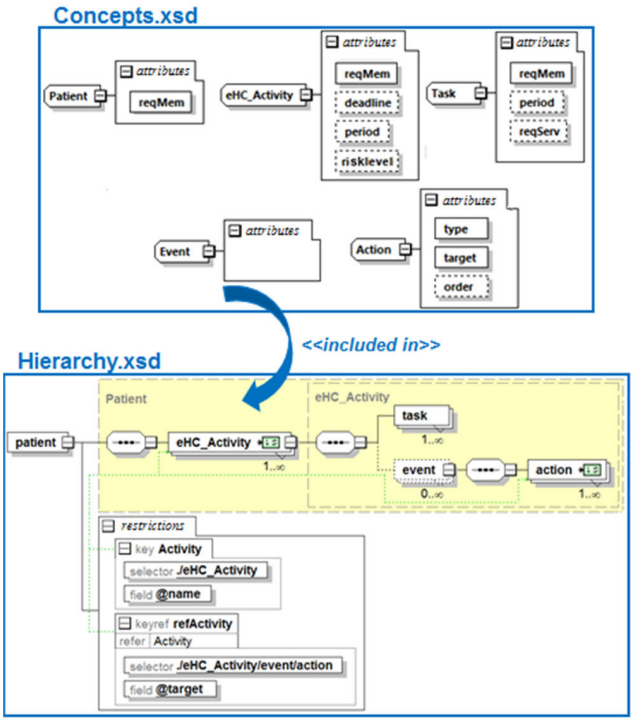


FIGURE 8. Concepts and Hierarchy XML schemas related to the person-centric applications of the eHC case study. Concepts.xsd identifies and characterizes application entities, in terms of XML elements and attributes, respectively. Hierarchy.xsd states their relationships, hierarchy (composition) and dependency (key-keyref construct).

eHC Activities for medical supervision and actuation, according to its health status. The eHC Activities may be divided into several *Tasks*, which cooperate among themselves through data exchange. Most Tasks are related to acquisition, processing and warning or storing assignments, as in the case of the *Continuous Glucose Monitoring* eHC Activity. Although very simple Tasks have been represented in the case study, they can refer to complex ones, such as interpreting medical images. Apart from these hierarchical relationships, dependency relationships also exist (highlighted in orange in Fig. 7), which represent the actions to perform because of an event trigger. Three action types are distinguished, all targeted to eHC Activities:

- Create: to initiate the execution of a new eHC Activity.
- Destroy: to finish the execution of an already running eHC Activity.
- Update: to change the properties of an eHC Activity (e.g., period or risk of level).

Regarding the characterization of application entities, all are described by the memory needed by their corresponding agent, as Application Agents are deployed according to the "maximum free memory" criterion (*reqMem* in Fig. 7 and Fig. 8). Some eHC Activities are periodic (e.g., checking if *P2* Patient has relaxed before measuring their blood pressure, carried out every six hours), whereas others can execute on different risk levels (e.g., when monitoring the heart rate of P3 Patient two risk levels can be distinguished: low (L) if acquired rate is inside its normal boundaries, or high (H) if it is out of range). Similarly, some Tasks run periodically. In this case, this period is different from that of eHC Activity. For instance, once *Check Relaxed* eHC Activity is activated, heart rate is acquired every 10 seconds.

Domain resource entities must also be considered. In this eHC system only processing resources, whose type is already defined in the core architecture, namely *ProcNode* type (see Fig. 2), are needed. Services offered represent the accessibility to a concrete biomedical sensor (e.g., pulsioximeter, glucometer...). Services may be required by Task entities (e.g., *Heart Rate Acq* or *Glucose Acq* in Fig. 7).

B. STEP 2: DOMAIN AGENT TEMPLATES

Resource or Application Agents related to the same domain entity-type share the same management particularities. Therefore, an agent template can be developed for each domain entity-type (application and resource) that has a

runtime representation, from which concrete Resource and Application Agents are derived.

The starting point is the agent skeleton-codes provided by MAS-RECON, which implements the FSMs of Fig. 4. Template development implies extending every FSM state according to the interactions defined among domain agents and System Supervisory Agents. The interactions established by the architecture must at least be considered (those interfaces depicted in Fig. 5 that domain agents use and/or provide). To that end, MAS-RECON also provides the code of those mechanisms for which the message sequence is fixed, namely:

- Failure detection and failure notification of non-delivered messages.
- Recovery of active instance or replicas.
- Negotiation process, including both launching a new negotiation process (*CFP* in Fig. 3), and sending and evaluating bids.

In the case of Resource Agents (see Fig. 4.a), required initialization and finalization actions will be considered at *Booting* and *Stopping* FSM states, respectively. *Running/Negotiation* state comprises the supervision of the concrete physical resource (e.g., the amount of free memory at processing nodes or the amount of remaining battery charge in AGVs). It is also the place to implement concrete negotiation mechanisms (e.g., the largest available memory in processing nodes or the time needed to cover a distance in AGVs).

Regarding Application Agents (see Fig. 4.b), *Booting* and *Stopping* FSM states are related to the application startup and stop, respectively. As MAS-RECON itself covers the startup of the first-level application entities (*Application Startup* in Fig. 3.b), application agent templates focus on the subsequent booting of the entities at lower levels, level by level until the last one is reached, according to the concrete application concept. This includes:

1) Looking for next-level entities (child), by means of queries to the SR through the SRA.
2) Launching a negotiation process among processing resources for every child, considering the required services.
3) Waiting for child agents to be started, except in the case of last-level entities.
4) Informing upper-level entity (parent) that booting has finished.

The implementation of the *Running* and *Tracking* FSM states focuses on the normal execution and on flexibility needs. For self-adaptability, it is necessary to identify, at least, which entities are in charge of detecting relevant context changes (*Event Trigger Application Agents* in Fig. 5), and which are responsible for executing adaptation actions (*Action Performer Application Agents* in Fig. 5). The former make use of the IEvent interface, whereas the latter implement the IAction interface. Similarly, for self-healing, at least, the following agents have to be identified: those in charge of failure detection (*Detector Resource* and *Application Agents* in Fig. 5), and those which execute recovery actions (*Parent Application Agents* in Fig. 5).

In the eHC case study, no resource templates are needed, since there are no domain specific resources (note that the ProcNode agent template is part of MAS-RECON). However, three templates are necessary for Patient, eHC Activity and Task agents. As an example, Fig. 9 represents the functions tackled by Task Agents at each FSM state, and which have to be implemented in the corresponding template. The code provided by MAS-RECON is marked in black whereas domain dependent code is highlighted in blue. According to the application structure depicted in Fig. 8, during application startup (Booting FSM state), Patient Agents supervise the creation of eHC Activity Agents and these latter do the same with Task Agents. As Task Agents exchange data messages among them, it is necessary to synchronize their start-up. Regarding self-adaptability, Task Agents are Trigger Application Agents because they process context data, being able to detect context changes. As all actions are targeted at eHC Activities, Patient Agents are the Action Performer Application Agents in charge of creating, destroying or updating eHC Activities. Finally, as far as self-healing is concerned, all Application Agents act as Detector Application and Resource Agents as they communicate through messages. Parent Application Agents are determined according to the specified application hierarchy.

C. STEP 3: EXTENSION OF SYSTEM SUPERVISORY AGENTS

MAS-RECON implements the interfaces provided by the System Supervisory Agents that represent application management needs common to all domains (see Fig. 5). Domain specific needs can be tackled in two ways. On the one hand, it is possible to extend these common interfaces to consider concrete adaptability actions or new ways of failure detection and/or recovery. On the other hand, new System Supervisory Agents could be included in the architecture. Their interfaces and the corresponding implementation derive from the analysis of interactions with other agents, which should also be included in the domain agent templates. In the eHC case study, a new System Supervisory Agent called Launcher Agent has been added to handle the external requests for application registration, start and stop.

V. CASE STUDY: A PLATFORM FOR THE eHC DOMAIN

This section validates that the management platform built for the eHC domain, based on the MAS-RECON architecture and following the proposed customization methodology covers the requirements in Table 2.

Two main tests have been carried out. The first one focuses on testing the fulfillment of operational requirements (R1-R3) and self-adaptability (R5) for a specific domain (R8). The second one evaluates self-healing on node failure (R7) and self-awareness mechanisms (R6).

From an infrastructure point of view, the test bed consists of the following resources:

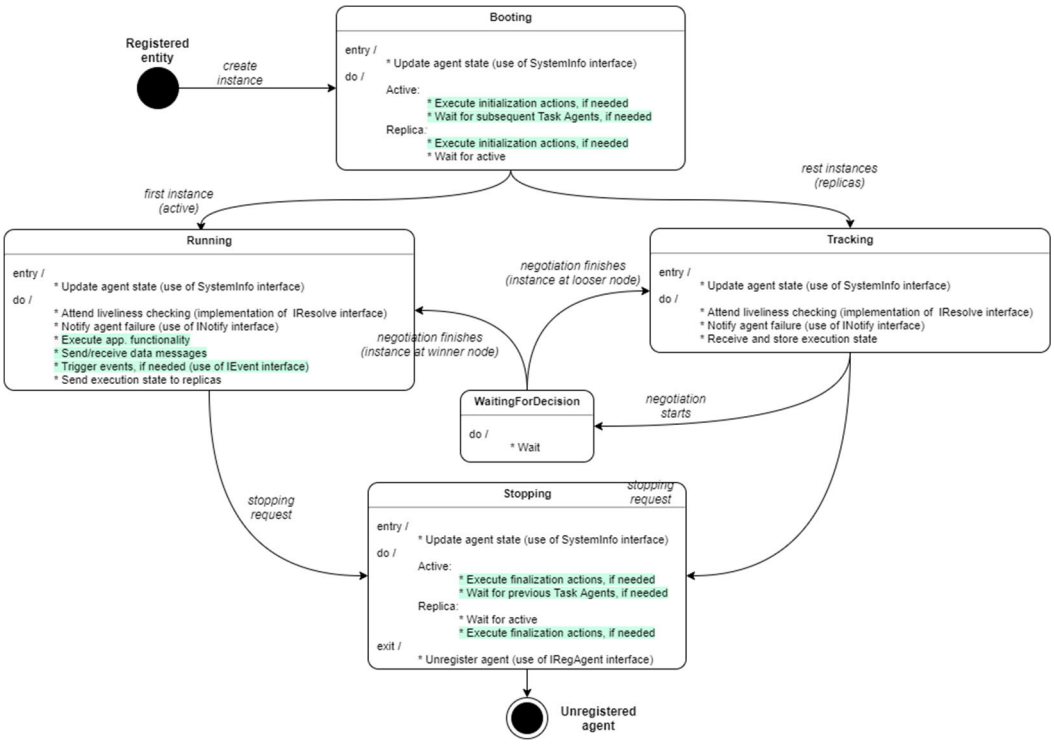


FIGURE 9. Customization needs for the development of the template of Task Agents in eHC. Functionalities tackled by Task Agents at every FSM state are depicted. Those implemented by the code provided by MAS-RECON are marked in black (e.g., notify agent failure). Those implemented in domain dependent code are highlighted in blue (e.g., implementation of the logic for triggering events related to relevant context changes, through the IEvent interface): Note that Tracking and Waiting for Decision FSM sates do not have domain dependent functionality.

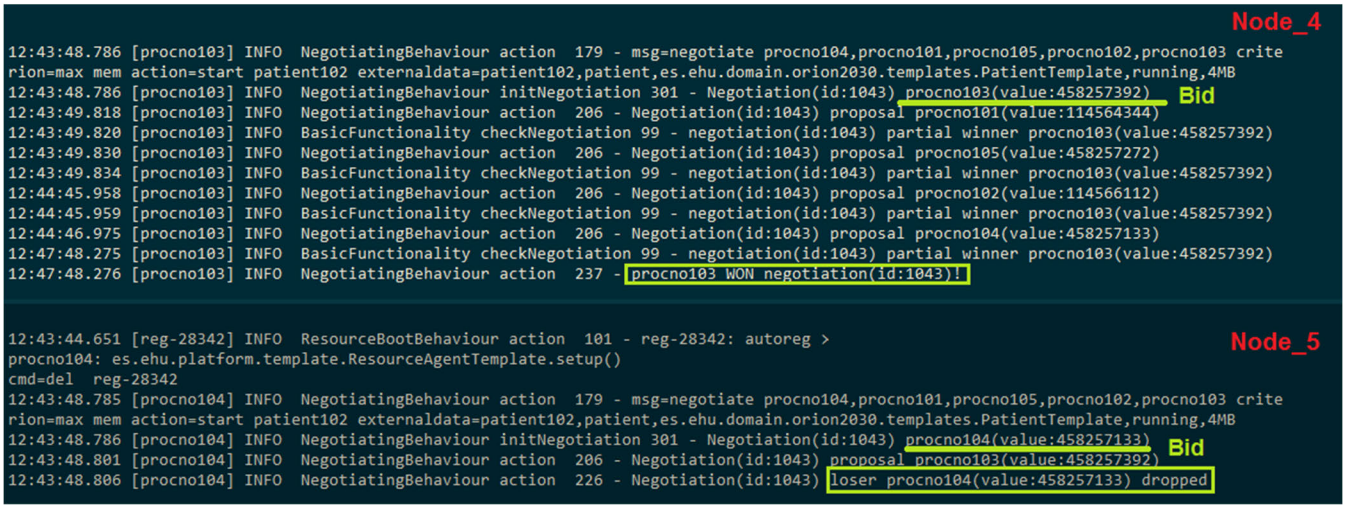


FIGURE 10. Example of a negotiation process carri0ed out during the start-up of P2 Patient application. It corresponds to the deployment of the active instance of P2 Patient entity. As it does not require any concrete service all available ProcNode Agents negotiate ('procno104' id is Node_5; 'procno101' id is Node_3; 'procno105' id is Node_6; 'procno102' id is Node_2; and 'procno103' id is Node_4). Node_4 is the winner as its bid is the best one. When a ProcNode Agent receives a better bid, it assumes that it cannot win and leaves the negotiation. This is the case of Node_5.

- 2 Raspberry Pi (Node_2 and Node_3), with access to biomedical sensors through the so-called "e-Health Sensor Platform V2.0." shield [57]. They measure the blood glucose level of P1 Patient (Gluc_P1 service of Node_2) and the heart rate and blood pressure of P2 Patient (Pulsioxy_P2 and Sphyg_P2 services of Node_3), respectively.
- 4 PC (Node_1, Node_4-Node_6), only to host agents.

From a software point of view, the functionality related to Tasks has been implemented as a Java library whose methods

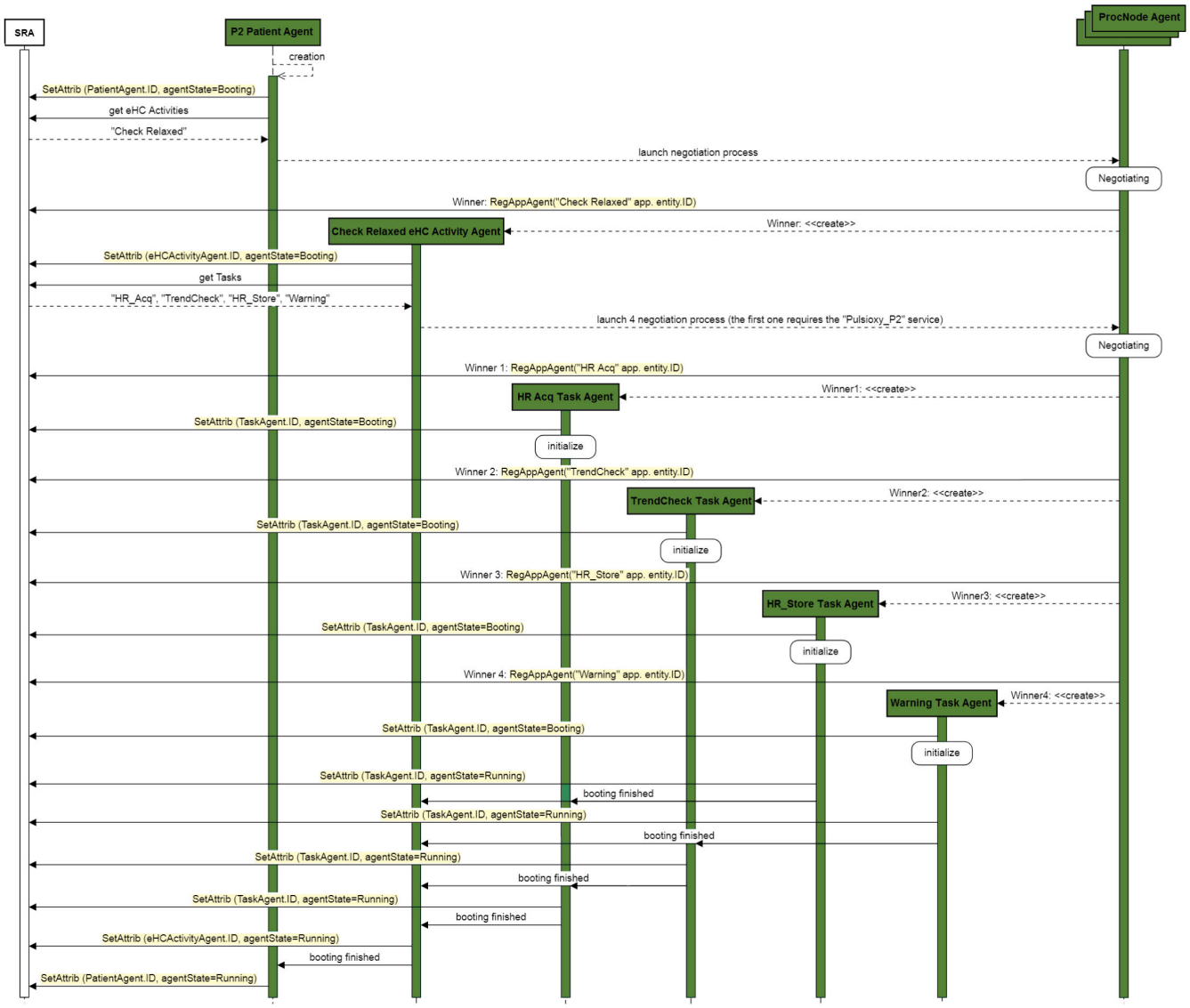


FIGURE 11. Startup sequence for P2 Patient application (color palette: core agent in white; Resource and Application Agents in green). For simplicity, redundancy level has not been considered and Heart Rate has been abbreviated as HR. Although P2 Patient consists of two eHC Activities, Blood Pressure Measuring is not initially deployed because it is event-triggered. To achieve synchronization among Task Agents, when their booting is finished, they warn the corresponding eHC Activity Agent and other Task Agents with which they communicate. The use of the API provided by the SRA is highlighted in yellow.

are invoked at the Running and Tracking FSM states of Task Agents.

Once the platform is launched on Node_1 (i.e., when all System Supervisory Agents are initiated), the SRA creates the SR, which is initially empty. Then, the other processing nodes are also booted, registering themselves at the SR as described in Section III.B.

A. APPLICATION REGISTRATION AND START-UP

The first test starts with the registration of the P2 Patient application. All application entities are registered one by one, following the application hierarchy depicted in Fig. 8: 1) Patients; 2) eHC Activities; 3) Tasks; 4) Events; 5) Actions. The SRA assigns a unique identifier to every registered entity (*id* attribute in Fig. 2). Finally, the correctness of the whole application is validated. This model-based registration process can prevent errors such as incorrect properties for application entities (e.g., registering a Patient entity without reqMem property, which does not match the Concepts schema in Fig. 8) or incorrect parent-child relationships (e.g., registering a Task entity as a Patient's child, which does not match the Hierarchy schema in Fig. 8).

The SRA initiates the startup of the P2 Patient application as in Fig. 3.b. An excerpt of the negotiation process carried out to deploy the active instance of P2 Patient Agent is presented in Fig. 10. The ProcNode Agent related to Node_4 is the negotiation winner, as its bid is the best one. On the contrary, when the ProcNode Agent related to Node_5 receives

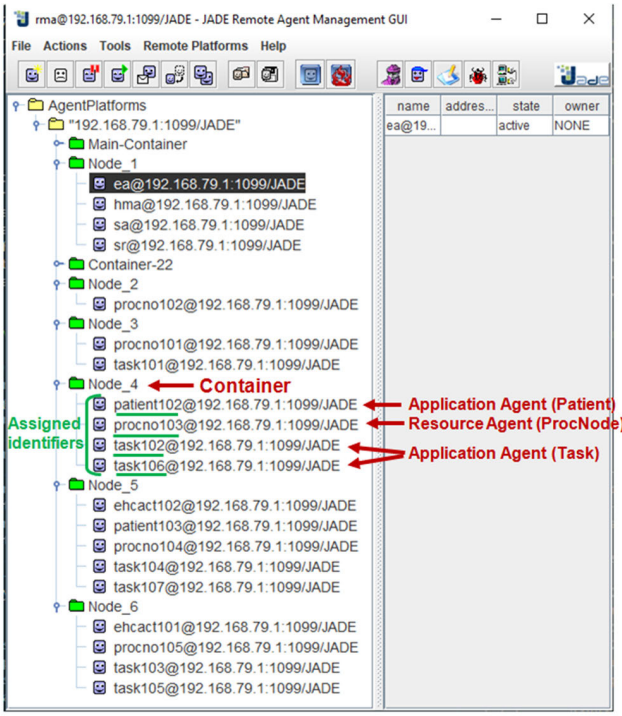


FIGURE 12. Initial deployment of P2 Patient application. Processing nodes are represented by containers. Each container hosts its corresponding Resource Agent and several Application Agents, according to the result of the negotiation processes carried out during the start-up.

a proposal better than its bid, it considers itself a loser, and leaves the negotiation process. Then, as represented in Fig. 11, the active instance of the P2 Patient Agent is responsible for the subsequent startup of its lower-level entities. Within its Boot FSM state, it reads, from the SR, the eHC Activities to create and start two agent instances for every one (replication factor is 1 in this case): one is the active instance and the other the replica. This process is repeated level by level, until Task Agents are created.

The start-up results in the deployment presented in Fig. 12, where containers, depicted by green directories, represent processing nodes. Each container hosts the corresponding Resource Agent and those Application Agent instances whose negotiation has won. For example, Node_4 contains its Resource Agent (*procno103* id in Fig. 12), a Patient Agent instance (*patien102* id in Fig. 12) and two Task Agent instances (*task102* and *task106* ids in Fig. 12).

Note that the container of Node_3 hosts only a Task Agent instance. The reason is twofold. On the one hand, Heart Rate Acq Task is the only application entity that requires the Pulsioxy_P2 service. Thus, it must be allocated on Node_3. On the other hand, Node_3 does not win negotiations for other instances (Patient Agents, eHC Activity Agents or other Task Agents) as its memory availability is less than that of PCs. This latter is also the reason why Node_2 does not host an application agent instance.

B. EVENT MANAGEMENT

Once started, P2 patient application is executed as illustrated in Fig. 13. The main objective of the application is to monitor the blood pressure of P2 patient. However, to avoid so-called ''white coat syndrome'', it is not measured until the patient is relaxed [58]. To that end, the P2 Patient application has been defined as two eHC Activities (Check Relaxed and Blood Pressure Measuring) which are related through an event that represents patient relaxation.

Patient relaxation is monitored by periodically acquiring heart rate values of P2 patient. (*Step 1* and *Step 2* in Fig. 13). These measurements are also stored for further processing (*Step 3* in Fig. 13). These steps are implemented by the Task entities that compose the Check Relaxed eHC Activity. When relaxation is detected (*Step 4* in Fig. 13), the Relaxed Event is triggered, leading to the interactions and message sequence depicted in *Step 5*. Each event triggers actions that may affect any registered application entity. In this case, one application entity (Check Relaxed eHC Activity) is stopped whereas another application entity (the Blood Pressure Measuring eHC Activity) is initiated. This latter captures systolic and diastolic blood pressure (*Step 6* in Fig. 13), which are also stored for further processing (*Step 7* in Fig. 13).

C. FAILURE RECOVERY

The second test assesses the ability of MAS-RECON to recover from node failures. In this test, 20 Patient applications similar to P1 Patient are registered and started, with replication factor stated as 1. When all the applications are running, the fail of Node_5 is forced (a PC that hosts Application Agents without required services). Fig. 14 depicts the evolution of memory use of the processing nodes described above, from the start-up of the applications to failure recovery.

As observed, initially (*Instant 1* in Fig. 14), those Task Agents that require the Glucometer_P1 service are deployed to Node_2. Node_3 does not hold an agent instance, since no application entity requires its services and has less memory than PCs. The rest of Application Agents are distributed in a balanced way, according to their memory needs.

When the HMA receives the notification of the failure of Node_5 (*Instant 2* in Fig. 14), it verifies it and identifies the affected agents. Finally, it reports the ReA that supervises the failure recovery as follows (by looking up the information stored at the SR):

- Firstly, instances of Patient Agents are re-instantiated by the ReA itself, also supervising the necessary negotiation processes for failed active instances.
- Secondly, the ReA asks the active instance of Patient Agents to tackle the recovery of failed eHC Activity Agents, including the supervision of negotiation processes for failed active instances.
- Finally, the ReA asks the active instance of eHC Activity Agents to deal with the recovery of failed Task Agents, including the supervision of negotiation processes for failed active instances

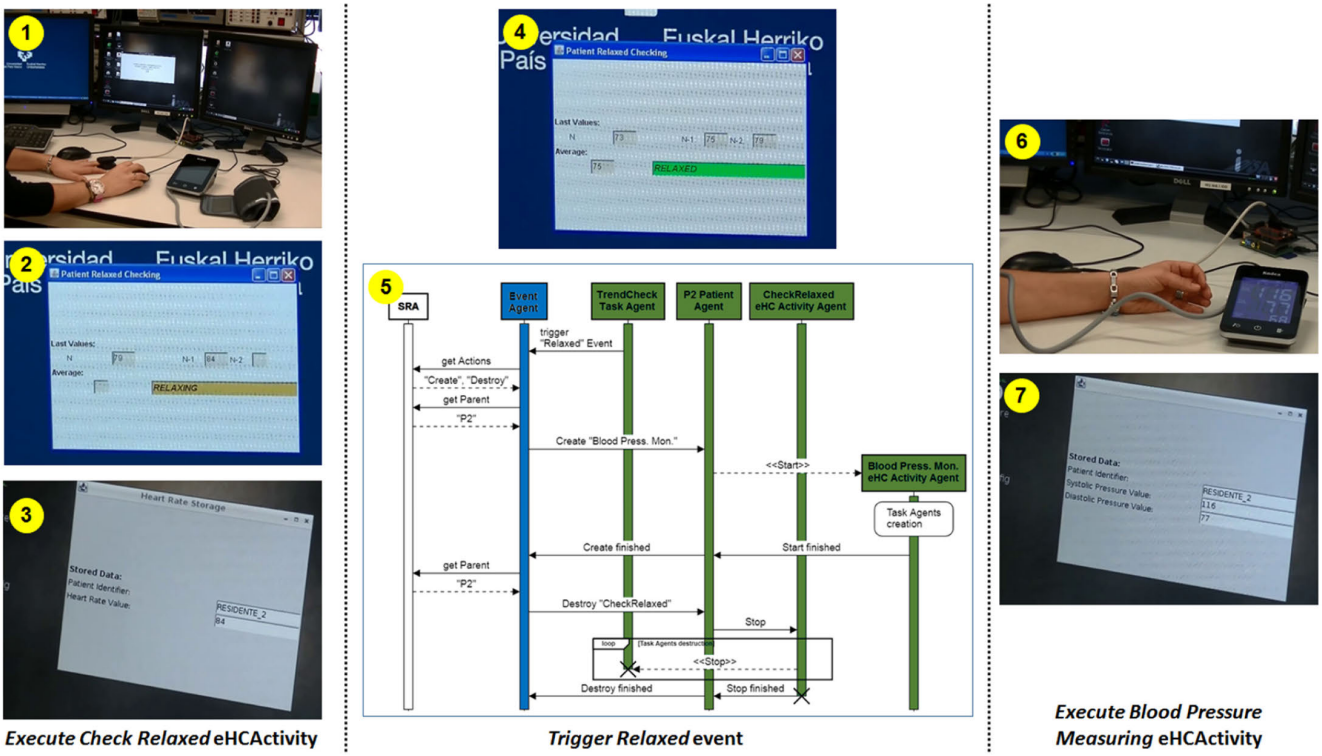


FIGURE 13. Execution steps of P2 Patient application. Initially, Check Relaxed eHC Activity is executed. When the relaxation of P2 Patient is detected by the TrendCheck Task Agent, the corresponding Relaxed Event is triggered, resulting in the measurement of its blood pressure (initialization of Blood Pressure Measuring eHC Activity) and finishing its heart rate supervision (stopping Check Relaxed eHC Activity).

After the recovery (*Instant 3* in Fig. 14), the memory use of Node_4 and Node_6 increases in a balanced way. However, Node_2 and Node_3 are not affected (their memory use does not change), because they have limited resources and do not win any negotiation processes.

VI. PERFORMANCE ANALYSIS

This section analyzes the performance of MAS-RECON, so that developers working with industry standard platforms for microservices can appreciate its benefits. The objective of this analysis is twofold: to benchmark deployment times of MAS-RECON against other application management architectures available on the market, and to extend the failure recovery analysis of Section V.C with response time measurements.

From an infrastructure point of view, the test bed for both performance analysis consists of 1 PC (Dell Precision 3551 with Intel Core i9-10885H and 64GB of RAM) that hosted a cluster of virtual machines (3 CPU and 3.5GB of RAM) created with multipass. Both the host and the virtual machines use Ubuntu 20.04 operative system.

A. BENCHMARK OF DEPLOYMENT TIMES: MAS-RECON VS. KUBERNETES

Kubernetes was selected as the industry standard implementation of microservices management platform against which to compare MAS-RECON. K3s, a lightweight and easy to install Kubernetes distribution, was selected to build the Kubernetes cluster.

Deployed applications were composed of three modules: a Generator that produces a pair of random numbers that are sent to a Processor that adds them up and sends the result to a Consumer that prints it in the standard output. The functionality of each module was programmed in Java and encapsulated in a Docker container in the case of Kubernetes, and in an agent in the case of MAS-RECON. In Kubernetes, each container was deployed using one-container-per-pod model, the three containers were related one to each other to form the application through a docker-compose file. In order for the comparison to be made on the same term, in MAS-RECON a very simple application structure was defined, consisting only of a first-level application entity-type called Component. Replication factor was stated to 0.

Two main tests were carried out. The first one focused on testing deployment times in MAS-RECON and Kubernetes for different workloads in a cluster made up of a fixed number of $N = 20$ nodes and over a maximum workload of $30*N = 600$ modules (components from now on). In the second test, the same measurements were taken for a 100% workload ($30*N$ components) in different cluster sizes. System Supervisory Agents of MAS-RECON and the control-plane of Kubernetes were deployed on the host machine, whereas processing nodes were installed on the virtual machines.

Measures were taken with a gateway agent that collected the timestamps of agent events in the case of MAS-RECON, and with a watcher used to listen to pod events in the case of Kubernetes. The initial timestamp in each test is the deployment request time.

Fig. 15 shows the mean scheduling, creation and startup times of both platforms for 10% (60 components), 25% (150 components), 50% (300 components), 75% (450 components) and 100% (600 components) workloads in a cluster made up of N = 20 nodes. The results show that: 1) Kubernetes schedules components ∼1.5 times faster than MAS-RECON; 2) MAS-RECON creates components ∼8.3 times faster than Kubernetes; and 3) Kubernetes starts components faster that MAS-RECON on low workloads, but startup times converge at 100% workload and the trendlines suggest that MAS-RECON starts components faster than Kubernetes on higher workloads.

Fig. 16 depicts the mean scheduling, creation and startup times of both platforms for N = 1, N = 5, N = 10, N = 15, N = 20 cluster sizes for a 100% workload (30*N components). The results related to scheduling and creation times resemble those obtained in the previous test. Regarding startup times, Kubernetes starts components faster that MAS-RECON at small cluster sizes, but startup times converge at N = 20 nodes and the trendlines suggest that MAS-RECON starts components faster than Kubernetes on bigger clusters.

The observed differences in planning times can be attributed to two reasons. On the one hand, the negotiation mechanism used in MAS-RECON to distribute the scheduling decision among the processing nodes is based on an adaptation of the Contract-Net protocol, which is not optimized for this task. On the other hand, in MAS-RECON the scheduling of Component agents was synchronized, as they exchange message data as in the case of Task Agents in eHC: first, the Generator is planned and, when it is created, the Processor is planned; then, when the latter is created, the Consumer is planned. Since the initial timestamp is the same for all the components of the deployment, this synchronized startup leads to higher scheduling times. It should be remarked that this synchronization cannot be achieved in Kubernetes without customizing it.

Regarding differences in creation times, the creation of an agent is faster than starting a container, since the former involves instantiation of a Java class, whereas the latter involves the instantiation of a virtual machine image.

B. RESPONSE TIME FOR FAILURE RECOVERY

This probe focused on testing agent recovery times in MAS-RECON for applications of different size in a case of a node failure. The test was performed in a cluster made up of 8 nodes where the same application structure described in the previous section was maintained. But in this case a Generator component, that produces a pair of random numbers, was

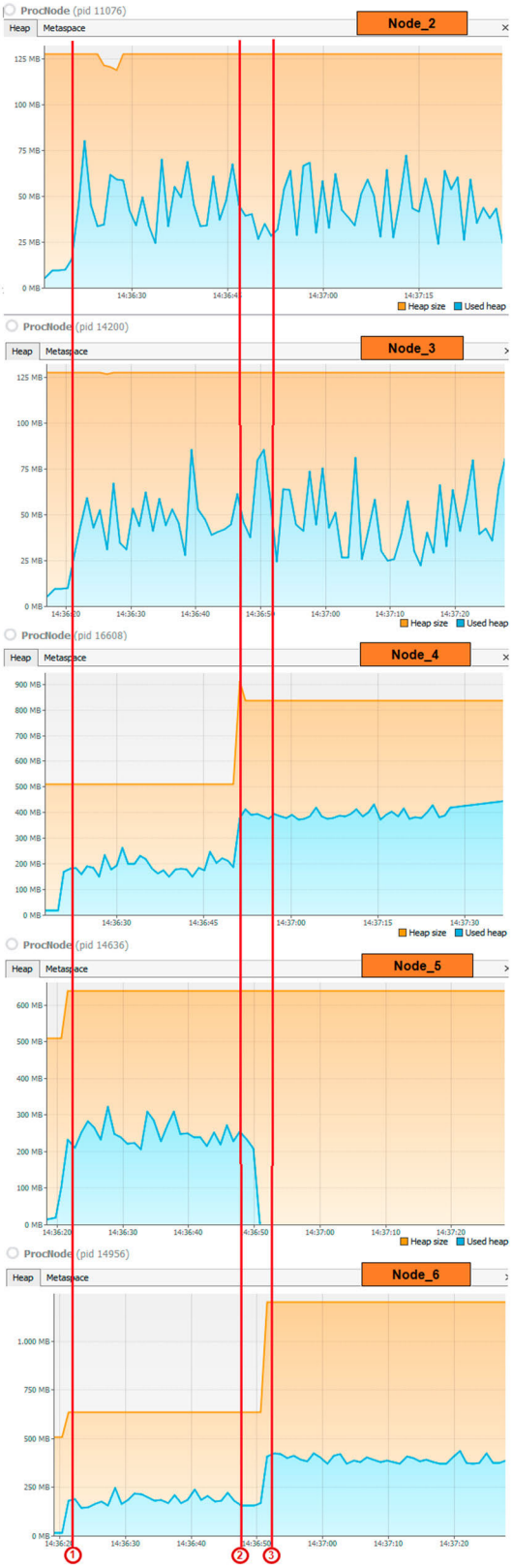


FIGURE 14. Recovery process of Node_5. The figure represents the evolution of memory use during the failure detection and recovery. Instant 1 represents the initial memory distribution after start-up. Instant 2 refers to the failure of Node_5. Finally, Instant 3 indicates the result of the recovery process.

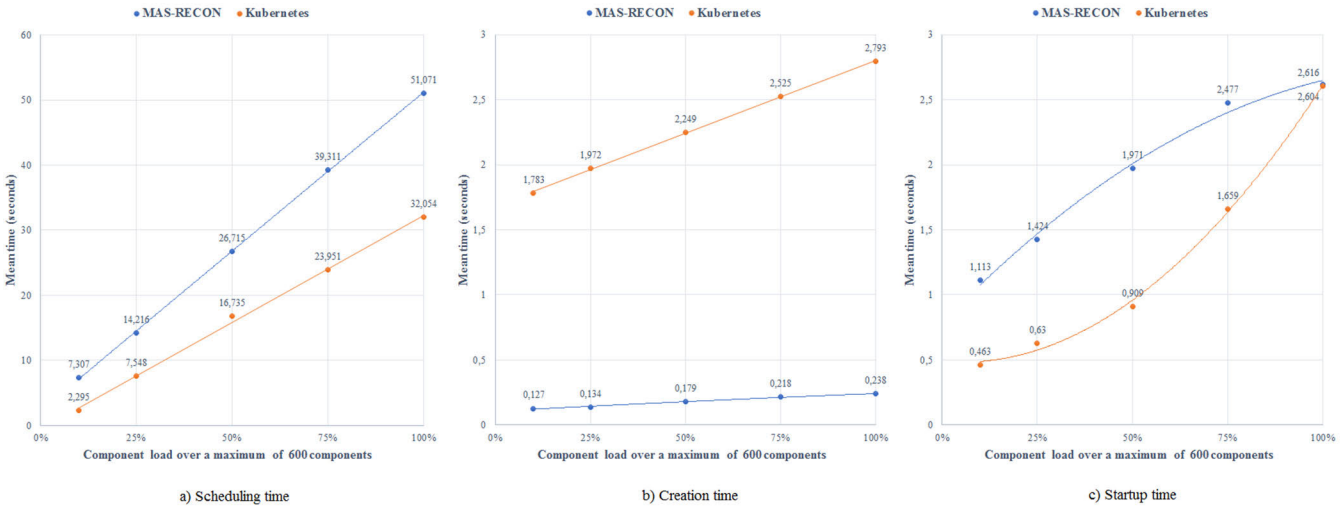


FIGURE 15. Component deployment times in MAS-RECON (blue color) and Kubernetes (orange color), for different workloads in a cluster made up of a fixed number of N = 20 nodes and over a maximum workload of 30*N (600) components: a) mean scheduling time; b) mean creation time; c) mean startup time.

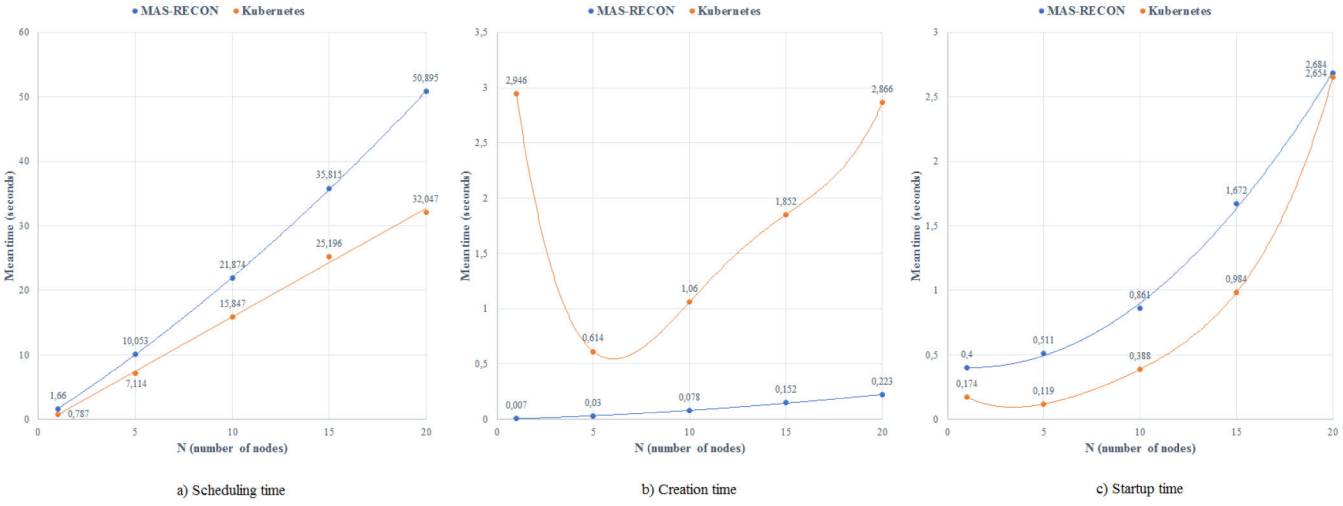


FIGURE 16. Component deployment times in MAS-RECON (blue color) and Kubernetes (orange color), for a 100% workload (30*N components) in different cluster sizes: a) mean scheduling time; b) mean creation time; c) mean startup time.

connected to N Processor components in serial that increase their input in one unit. The last Processor component was connected with a Consumer component that prints the result in the standard output. Again, connection among components was based on message exchange, and replication factor was stated to 0. Processor components were restricted to be deployed on nodes 2-7 (six nodes), whereas Generator and Consumer components were deployed on nodes 1 and 8. The failure involves the disconnection of one of the nodes and the recovery of all the failed components.

To reflect the different aspects to be considered in the recovery process, the following measurements were collected: reaction time, repair time and recovery time [51]. The reaction time measures the time elapsed from the failure until the platform starts to respond (i.e., until the HMA receives notification of the failure). The repair time measures the time elapsed from the detection of the failure until the first failed component is recovered. Finally, the recovery time measures the time elapsed from the detection of the failure until the application is restored (i.e., all failed components are recovered and the Consumer component prints a valid result again).

Table 4 shows the reaction, repair and recovery times for applications made up of N = 60 (recovery of 10 components), N = 150 (recovery of 25 components), N = 300 (recovery of 50 components), N = 450 (recovery of 75 components) and N = 600 (recovery of 100 components) Processor components.

The results show that the reaction and repair times remain approximately constant in order of magnitude (they suffer

TABLE 4. Results of response times for failure recovery with applications of different size (N). Failure recovery is based on the failure of one node (i.e., N/6 components are lost).

	N=60	N=150	N=300	N=450	N=600
Reaction	3,14s	3,14s	3,14s	3,54s	3,81s
Repair	5,28s	5,56s	5,33s	4,35s	5,97s
Recovery	8,38s	18,62s	52,87s	85,81s	96,16s

an increase of less than 0.70s between the lightest and the heaviest load). These results were expected, since the reaction time is a load-independent capability of the platform; in turn, the repair time is measured for the first component recovered, which makes it also independent of the number of components to be recovered. Finally, the recovery time increases, as expected, with the number of components to be recovered, but the ratio between the recovery time and the number of recovered components is ∼1 (i.e., it takes approximately ∼1s to recover a component).

VII. CONCLUSION

This paper has proposed MAS-RECON, a generic and customizable architecture for the management of context-aware applications. It is mainly focused on considering the domain application concept from its initial design, also fulfilling the operational and flexibility requirements of target applications from an application-centric point of view.

The formalization of the domain based on models facilitates platform customization, allowing a generic management of the system state. It has also been proven that distributed intelligence, (achieved through multi-agent technology) jointly with system-level supervision, makes it possible to face unexpected events: namely, relevant context changes or agent failures. The MAS-RECON architecture, together with the proposed customization methodology, allows domain specific platforms to be developed, which meet common and domain dependent requirements of context-aware applications.

From the analysis of deployment times of MAS-RECON against Kubernetes, it can be concluded that scheduling time in Kubernetes is better than in MAS-RECON. This is mainly due to the personalization facilities that MAS-RECON offers, which, among others, allow startup or deployment of agents customized to concrete domains. In fact, it can be said that MAS-RECON goes beyond other management platforms, being a kind of development framework that eases agent implementation through the definition of templates.

However, the approach still has limitations. Currently, MAS-RECON lacks an admission control, which assures that applications are accepted only if there are enough resources. Additionally, given the dynamism of resource availability, flexible QoS management is needed to adjust the QoS level of running applications to the available resources at any moment. Another serious drawback is the effort to develop a particular platform, as it requires an in-depth knowledge of the MAS-RECON architecture to integrate a new code. In this sense, future work is aimed at extending MAS-RECON architecture to facilitate the development of new domain platforms. Thus, by making the most of model-driven engineering in terms of model transformations, it will be possible to customize the architecture to different domains.

REFERENCES

[1] A. Čolaković and M. Hadžialić, "Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues," *Comput. Netw.*, vol. 144, pp. 17–39, Oct. 2018, doi: 10.1016/j.comnet.2018.07.017.

[2] C. Perera, C. H. Liu, S. Jayawardena, and M. Chen, "A survey on Internet of Things from industrial market perspective," *IEEE Access*, vol. 2, pp. 1660–1679, 2014, doi: 10.1109/ACCESS.2015.2389854.

[3] H. Boyes, B. Hallaq, J. Cunningham, and T. Watson, "The industrial Internet of Things (IIoT): An analysis framework," *Comput. Ind.*, vol. 101, pp. 1–12, Oct. 2018, doi: 10.1016/j.compind.2018.04.015.

[4] H. Xu, W. Yu, D. Griffith, and N. Golmie, "A survey on industrial Internet of Things: A cyber-physical systems perspective," *IEEE Access*, vol. 6, pp. 78238–78259, 2018, doi: 10.1109/ACCESS.2018.2884906.

[5] P. P. Ray, M. Mukherjee, and L. Shu, "Internet of Things for disaster management: State-of-the-art and prospects," *IEEE Access*, vol. 5, pp. 18818–18835, 2017, doi: 10.1109/ACCESS.2017.2752174.

[6] S. A. Shah, D. Z. Seker, M. M. Rathore, S. Hameed, S. B. Yahia, and D. Draheim, "Towards disaster resilient smart cities: Can Internet of Things and big data analytics be the game changers?" *IEEE Access*, vol. 7, pp. 91885–91903, 2019, doi: 10.1109/ACCESS.2019.2928233.

[7] S. B. Baker, W. Xiang, and I. Atkinson, "Internet of Things for smart healthcare: Technologies, challenges, and opportunities," *IEEE Access*, vol. 5, pp. 26521–26544, 2017, doi: 10.1109/access.2017.2775180.

[8] S. M. R. Islam, D. Kwak, M. H. Kabir, M. Hossain, and K.-S. Kwak, "The Internet of Things for health care: A comprehensive survey," *IEEE Access*, vol. 3, pp. 678–708, 2015, doi: 10.1109/ACCESS.2015.2437951.

[9] M. Ayaz, M. Ammad-Uddin, Z. Sharif, A. Mansour, and E. M. Aggoune, "Internet-of-Things (IoT)-based smart agriculture: Toward making the fields talk," *IEEE Access*, vol. 7, pp. 129551–129583, 2019, doi: 10.1109/ACCESS.2019.2932609.

[10] M. S. Farooq, S. Riaz, A. Abid, K. Abid, and M. A. Naeem, "A survey on the role of IoT in agriculture for the implementation of smart farming," *IEEE Access*, vol. 7, pp. 156237–156271, 2019, doi: 10.1109/ACCESS.2019.2949703.

[11] B. Chen, J. Wan, L. Shu, P. Li, M. Mukherjee, and B. Yin, "Smart factory of industry 4.0: Key technologies, application case, and challenges," *IEEE Access*, vol. 6, pp. 6505–6519, 2017, doi: 10.1109/ACCESS.2017.2783682.

[12] Q. Qi and F. Tao, "A smart manufacturing service system based on edge computing, fog computing, and cloud computing," *IEEE Access*, vol. 7, pp. 86769–86777, 2019, doi: 10.1109/ACCESS.2019.2923610.

[13] T. Vale, I. Crnkovic, E. De Almeida, and P. Neto, "Twenty-eight years of component-based software engineering," *J. Syst. Softw.*, vol. 111, pp. 128–148, Jan. 2016, doi: 10.1016/j.jss.2015.09.019.

[14] W. Michael, *An Introduction to MultiAgent Systems*, 2nd ed. Hoboken, NJ, USA: Wiley, 2009.

[15] J. Al-Jaroodi and N. Mohamed, "Service-oriented middleware: A survey," *J. Netw. Comput. Appl.*, vol. 35, no. 1, pp. 211–220, 2012, doi: 10.1016/j.jnca.2011.07.013.

[16] J. Lewis and M. Fowler. (2014). *Microservices*. Accessed: Jul. 20, 2021. [Online]. Available: https://Martinfowler.com/articles/microservices.html

[17] N. Gui, V. De Florio, H. Sun, and C. Blondia, "Toward architecture-based context-aware deployment and adaptation," *J. Syst. Softw.*, vol. 84, no. 2, pp. 185–197, Feb. 2011, doi: 10.1016/j.jss.2010.09.017.

[18] S. Hallsteinsen, K. Geihs, N. Paspallis, F. Eliassen, G. Horn, J. Lorenzo, A. Mamelli, and G. A. Papadopoulos, "A development framework and methodology for self-adapting applications in ubiquitous computing environments," *J. Syst. Softw.*, vol. 85, no. 12, pp. 2840–2859, Dec. 2012, doi: 10.1016/j.jss.2012.07.052.

[19] E. Albassam, J. Porter, H. Gomaa, and D. A. Menasce, "DARE: A distributed adaptation and failure recovery framework for software systems," in *Proc. IEEE Int. Conf. Auton. Comput. (ICAC)*, Columbus, OH, USA, Jan. 2017, pp. 203–208, doi: 10.1109/ICAC.2017.12.

[20] M. Hussein, J. Han, and A. Colman, "An approach to model-based development of context-aware adaptive systems," in *Proc. 35th Annu. Comput. Softw. Appl. Conf.*, Munich, Germany, 2011, pp. 205–214, doi: 10.1109/COMPSAC.2011.34.

[21] E. Argente, V. Botti, C. Carrascosa, A. Giret, V. Julian, and M. Rebollo, "An abstract architecture for virtual organizations: The Thomas approach," *Knowl. Inf. Syst.*, vol. 29, no. 2, pp. 379–403, Nov. 2011, doi: 10.1007/s10115-010-0349-1.

[22] G. Villarrubia, D. Hernández, J. F. De Paz, and J. Bajo, "Combination of multi-agent systems and embedded hardware for the monitoring and analysis of diuresis," *Int. J. Distrib. Sens. Netw.*, vol. 13, no. 7, pp. 1–17, 2017, doi: 10.1177/1550147717722154.

[23] M. G. Valls, I. R. Lopez, and L. F. Villar, "ILAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems," *IEEE Trans. Ind. Informat.*, vol. 9, no. 1, pp. 228–236, Feb. 2013, doi: 10.1109/TII.2012.2198662.

[24] A. Agirre, J. Parra, A. Armentia, E. Estvez, and M. Marcos, "QoS aware middleware support for dynamically reconfigurable component based IoT applications," *Int. J. Distrib. Sens. Netw.*, vol. 2016, Oct. 2016, Art. no. 2702789, doi: 10.1155/2016/2702789.

[25] M. U. Khan, R. Reichle, and K. Geihs, "Architectural constraints in the model-driven development of self-adaptive applications," *IEEE Distrib. Syst.*, vol. 9, no. 7, pp. 1–10, Jul. 2008, doi: 10.1109/MDSO.2008.19.

[26] X. He, Z. Tu, X. Xu, and Z. Wang, "Programming framework and infrastructure for self-adaptation and optimized evolution method for microservice systems in cloud–edge environments," *Future Gener. Comput. Syst.*, vol. 118, pp. 263–281, May 2021, doi: 10.1016/j.future.2021.01.008.

[27] A. Armentia, U. Gangoiti, R. Priego, E. Estévez, and M. Marcos, "Flexibility support for homecare applications based on models and multi-agent technology," *Sensors*, vol. 15, no. 12, pp. 31939–31964, Dec. 2015, doi: 10.3390/s151229899.

[28] M. L. Iglesias and J. M. Garechana, "Tolerancia a fallos en sistema de fabricación flexible basado en MAS," in *Proc. Conf. Automtica*, Badajoz, Spain, 2018, pp. 799–805, doi: 10.17979/spudc.9788497497565.

[29] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive Mobile Comput.*, vol. 17, pp. 184–206, Feb. 2015, doi: 10.1016/j.pmcj.2014.09.009.

[30] Y. Wang, H. Kadiyala, and J. Rubin, "Promises and challenges of microservices: An exploratory study," *Empirical Softw. Eng.*, vol. 26, no. 4, pp. 1–44, Jul. 2021, doi: 10.1007/s10664-020-09910-y.

[31] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, "JADE: A software framework for developing multi-agent applications. Lessons learned," *Inf. Softw. Technol.*, vol. 50, nos. 1–2, pp. 10–21, Jan. 2008, doi: 10.1016/j.infsof.2007.10.008.

[32] *Standard FIPA Specifications*, Foundation for Intelligent Physical Agents, 2002. [Online]. Available: http://www.fipa.org/repository/standardspecs.html

[33] (2020). *Kubernetes*. Accessed: Jul. 22, 2021. [Online]. Available: https://kubernetes.io/docs/home/

[34] P. González-Nalda, I. Etxeberria-Agiriano, I. Calvo, and M. C. Otero, "A modular CPS architecture design based on ROS and Docker," *Int. J. Interact. Design Manuf.*, vol. 11, no. 4, pp. 949–955, Nov. 2017, doi: 10.1007/s12008-016-0313-8.

[35] G. Toffetti, T. Látscher, S. Kenzhegulov, J. Spillner, and T. M. Bohnert, "Cloud robotics: SLAM and autonomous exploration on PaaS," in *Proc. 10th Int. Conf. Utility Cloud Comput.*, Austin, TX, USA, Dec. 2017, pp. 65–70, doi: 10.1145/3147234.3148100.

[36] V. Hassija, V. Chamola, V. Saxena, D. Jain, P. Goyal, and B. Sikdar, "A survey on IoT security: Application areas, security threats, and solution architectures," *IEEE Access*, vol. 7, pp. 82721–82743, 2019, doi: 10.1109/ACCESS.2019.2924045.

[37] P. Arcaini, E. Riccobene, and P. Scandurra, "Modeling and analyzing MAPE-K feedback loops for self-adaptation," in *Proc. IEEE/ACM 10th Int. Symp. Softw. Eng. Adapt. Self-Managing Syst.*, Florence, Italy, Apr. 2015, pp. 13–23, doi: 10.1109/SEAMS.2015.10.

[38] X. Li, M. Eckert, J.-F. Martinez, and G. Rubio, "Context aware middleware architectures: Survey and challenges," *Sensors*, vol. 15, no. 8, pp. 20570–20607, 2015, doi: 10.3390/s150820570.

[39] J. R. Hoyos, J. García-Molina, and J. A. Botía, "A domain-specific language for context modeling in context-aware systems," *J. Syst. Softw.*, vol. 86, no. 11, pp. 2890–2905, Nov. 2013, doi: 10.1016/j.jss.2013.07.008.

[40] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou, "Variability in software systems—A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 40, no. 3, pp. 282–306, Mar. 2014, doi: 10.1109/TSE.2013.56.

[41] A. Rocha, A. Martins, J. C. Freire, M. N. Kamel Boulos, M. E. Vicente, R. Feld, P. van de Ven, J. Nelson, A. Bourke, G. ÓLaighin, C. Sdogati, A. Jobes, L. Narvaiza, and A. Rodríguez-Molinero, "Innovations in health care services: The CAALYX system," *Int. J. Med. Informat.*, vol. 82, no. 11, pp. e307–e320, Nov. 2013, doi: 10.1016/j.ijmedinf.2011.03.003.

[42] T. Wu, F. Wu, J.-M. Redouté, and M. R. Yuce, "An autonomous wireless body area network implementation towards IoT connected healthcare applications," *IEEE Access*, vol. 5, pp. 11413–11422, 2017, doi: 10.1109/ACCESS.2017.2716344.

[43] L. R. Coutinho, A. A. F. Brandão, O. Boissier, and J. S. Sichman, "Towards agent organizations interoperability: A model driven engineering approach," *Appl. Sci.*, vol. 9, no. 12, pp. 1–38, 2019, doi: 10.3390/app9122420.

[44] J. J. Gómez-Sanz and R. Fuentes-Fernández, "Understanding agent-oriented software engineering methodologies," *Knowl. Eng. Rev.*, vol. 30, no. 4, pp. 375–393, Sep. 2015, doi: 10.1017/S0269888915000053.

[45] U. Gangoiti, "Model-driven design and development of flexible automated production control configurations for industry 4.0," *Appl. Sci.*, vol. 11, no. 5, pp. 1–27, Mar. 2021, doi: 10.3390/app11052319.

[46] H. Psaier and S. Dustdar, "A survey on self-healing systems: Approaches and systems," *Computing*, vol. 91, no. 1, pp. 43–73, Jan. 2011, doi: 10.1007/s00607-010-0107-y.

[47] I. García-Magariño and C. Gutiérrez, "Agent-oriented modeling and development of a system for crisis management," *Expert Syst. Appl.*, vol. 40, no. 16, pp. 6580–6592, Nov. 2013, doi: 10.1016/j.eswa.2013.06.012.

[48] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, Apr. 1997, doi: 10.1109/2.585156.

[49] W. Huan and N. Hidenori, "Failure detection in P2P-grid environments," in *Proc. 32nd Int. Conf. Distrib. Comput. Syst. Workshops*, Macau, China, 2012, pp. 369–374, doi: 10.1109/ICDCSW.2012.18.

[50] A. Ruiz, G. Juez, P. Schleiss, and G. Weiss, "A safe generic adaptation mechanism for smart cars," in *Proc. IEEE 26th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Gaithersbury, MD, USA, May 2015, pp. 161–171, doi: 10.1109/ISSRE.2015.7381810.

[51] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A Kubernetes controller for managing the availability of elastic microservice based stateful applications," *J. Syst. Softw.*, vol. 175, no. 110924, pp. 1–13, May 2021, doi: 10.1016/j.jss.2021.110924.

[52] D. Rosaci and G. M. L. Sarné, "MASHA: A multi-agent system handling user and device adaptivity of web sites," *User Model. User-Adapted Interact.*, vol. 16, no. 5, pp. 435–462, Nov. 2006, doi: 10.1007/s11257-006-9015-4.

[53] S. Garruzzo, D. Rosaci, and G. M. L. Sarne, "MASHA-EL: A multi-agent system for supporting adaptive E-learning," in *Proc. 19th IEEE Int. Conf. Tools Artif. Intell.*, Patras, Greece, vol. 2007, pp. 103–110, doi: 10.1109/ICTAI.2007.83.

[54] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003, doi: 10.1109/MS.2003.1231146.

[55] W3C. (2004). *XML Schema Part: Primer (Second Edition) W3C REC-xmlschema-0-20041028*. [Online]. Available: https://www.w3.org/TR/2004/REC-xmlschema-0-20041028/

[56] Casquero, O, Armentia, A, Estevez, E, Lápez, A, and M. Marcos, "Customization of agent-based manufacturing applications based on domain modelling," in *Proc. 21st IFAC World Congr.*, Berlin, Germany, 2020, pp. 1–4. Accessed: Nov. 18, 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0952197608001437

[57] C. Hacks. (2013). *E-Health Sensor Platform V1.0 for Arduino and Raspberry Pi [Biometric/ Medical Applications]*. [Online]. Available: https://www.cooking-hacks.com/documentation/tutorials/ehealth-v1-biometric-sensor-platform-arduino-raspberry-pi-medical.html

[58] Á. Jobbágy, P. Csordás, and A. Mersich, "Blood pressure measurement at home," in *Proc. World Congr. Med. Phys. Biomed. Eng.*, Seoul, South Korea, vol. 14, 2006, pp. 3453–3456, doi: 10.1007/978-3-540-36841-0_873.

UNAI GANGOITI received the B.S. degree in industrial engineering from the University of the Basque Country (UPV/EHU), Spain, in 2001.

In 2002, he started to work as an Assistant Professor with the Department of Automatic Control and Systems Engineering, UPV/EHU. As a Researcher, since 2002, he has been belonging with the Grupo de Control e Integración de Sistemas (GCIS) Research Group, Department of Systems Engineering and Automatic Control. Since 2005, he has been an Analyst with the IT Department, UPV/EHU. He is the author of three indexed journal articles and several electronic publications at international conferences, indexed in the Web of Science as well as Scopus databases. His research interests include the application of information and communication technologies to industrial systems.

Mr. Gangoiti has collaborated as a reviewer with several indexed international conferences.

ALEJANDRO LÓPEZ received the B.S. degree in industrial electronics and automation engineering and the M.S. degree in control, automation and robotics engineering from the University of the Basque Country (UPV/EHU), Spain, in 2016 and 2019, respectively. He is currently pursuing the Ph.D. degree in control, automation and robotics with the Faculty of Engineering in Bilbao, at the University of the Basque Country (UPV/EHU).

After almost two years as a Technician at the Automotive Smart Factory Competence Center in Advanced Manufacturing in AIC (Amorebieta), in November 2019. As a Researcher, he is a part of the Grupo de Control e Integración de Sistemas (GCIS) Research Group, Department of Systems Engineering and Automatic Control, where he is committed to the study of smart and flexible automation production systems. He is the coauthor of one indexed journal article, and the author or coauthor of four publications in conference proceedings.

AINTZANE ARMENTIA received the B.S. degree in telecommunications engineering the M.S. and Ph.D. degrees in control, automation and robotics engineering from the University of the Basque Country (UPV/EHU), Spain, in 2001, 2011, and 2016, respectively.

After six years of industrial experience as a Programmer and an Analyst Programmer (from 2002 to 2008), she started to work as a Researcher with the Department of Automatic Control and Systems Engineering, UPV/EHU. Since 2017, she has been an Assistant Professor with the Department of Automatic Control and Systems Engineering, initially in Bilbao, Spain, and currently in Vitoria, Spain. She is the author of six indexed journal articles and more than 12 electronic publications at international conferences, indexed in the Web of Science as well as Scopus databases. Her research interests include model-based software engineering, multi-agent systems, and flexible manufacturing systems.

Dr. Armentia was a recipient of the Best Paper Award in Computers and Control at the XXXIX Jornadas de Automática in 2018. She collaborates as a reviewer with several indexed journals as well as international conferences.

ELISABET ESTÉVEZ received the B.S. degree in telecommunications engineering and the Ph.D. degree in automatic control from the University of the Basque Country (UPV/EHU), Spain in 2002 and 2007, respectively.

In 2002, she started to work as a Researcher with the Department of Automatic Control and Systems Engineering, UPV/EHU. Since 2017, she has been an Assistant Professor, initially with the Department of Automatic Control and Systems Engineering, UPV/EHU, Bilbao, Spain, and since 2011, she has been with the Electronics and Automation Engineering Department, University of Jaén, Jaén, Spain. She is the author or coauthor of more than 80 technical papers in international journals and conference proceedings in the field of distributed industrial control systems. Her research interests include applying software engineering concepts to industrial control as well as smart and flexible automation production systems.

Dr. Estévez was a recipient of the Best Paper Award in Computers and Control at the XXXIX Jornadas de Automática in 2018. She collaborates as a reviewer for several conferences and technical journals.

OSKAR CASQUERO received the B.S. degree in telecommunication engineering and the Ph.D. degree in engineering, in 2003 and 2013, respectively.

From 2004 to 2007, he worked as an IT Architecture Analyst at the Virtual Campus of the University of the Basque Country. Since 2007, he has been working as an Assistant Professor with the Systems Engineering and Automatic Control Department, currently at the Faculty of Engineering, Bilbao. He investigates on smart and flexible manufacturing systems using digital twins, model-driven engineering, multi-agent systems and cloud computing technologies.

Dr. Casquero collaborates as a reviewer with several indexed journals as well as international conferences.

MARGA MARCOS (Senior Member, IEEE) received the B.Sc., M.S., and Ph.D. degrees in control engineering from the University of the Basque Country (UPV/EHU), Spain, in 1983, 1984, and 1988, respectively.

She is currently a Professor of control engineering at the University of the Basque Country, where she was the Vice-Dean of the Faculty of Engineering, from 1990 to 1993. She has authored or coauthored more than 200 technical papers in international journals and conference proceedings. She has acted as the Main Researcher of more than 80 research projects funded by the National and European research and development programs. Her main research interests include application of model driven engineering to automation systems and the application of industrial agents in manufacturing.

Prof. Marcos is serving as a member of the IFAC Council for the 2020–2023 triennium. She has served and already serves on TCs of IFAC (AARTC, WRTP, and CC) and IEEE (NBCS, ICPS, IA). She was the Chair of the IFAC TC Computer for Control (2014–2017), a member of EUCA, a NOC of IFAC Spain, a Publication Co-Chair for IEEE CDC2005 and a General Co-Chair for IEEE ETFA 2010. She has served as an Associate Editor for the IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING journal (2018–2020). She was the Chairperson of the Control Department (1995–2005).