Journal of Ambient Intelligence and Humanized Computing (2020) 11:2963–2986
https://doi.org/10.1007/s12652-019-01439-3

ORIGINAL RESEARCH

A RESTful middleware for AI controlled sensors, actuators and smart devices

Nikos Dipsis[1] · **Kostas Stathis**[1]

Received: 14 February 2019 / Accepted: 28 August 2019 / Published online: 12 September 2019

Abstract
The numerous applications of internet of things (IoT) and sensor networks combined with specialized devices used in each has led to a proliferation of domain specific middleware, which in turn creates interoperability issues between the corresponding architectures and the technologies used. But what if we wanted to use a machine learning algorithm to an IoT application so that it adapts intelligently to changes of the environment, or enable a software agent to enrich with artificial intelligence (AI) a smart home consisting of multiple and possibly incompatible technologies? In this work we answer these questions by studying a framework that explores how to simplify the incorporation of AI capabilities to existing sensor-actuator networks or IoT infrastructures making the services offered in such settings smarter. Towards this goal we present eVATAR+, a middleware that implements the interactions within the context of such integrations systematically and transparently from the developers' perspective. It also provides a simple and easy to use interface for developers to use. eVATAR+ uses JAVA server technologies enhanced by mediator functionality providing interoperability, maintainability and heterogeneity support. We exemplify eVATAR+ with a concrete case study and we evaluate the relative merits of our approach by comparing our work with the current state of the art.

Keywords Internet of things · Sensor networks · Smart devices · Artificial intelligence · Middleware · RESTful API

1 Introduction

One common requirement for the constituents of sensor-actuator networks and IoT infrastructures is that they should access and transform the environment in which they are situated. Consider, for example, the "smarter planet" vision where cognitive AI is applied on sensors and actuators embedded in physical objects found in every environment of human activity (IBM Watson IoT 2017). Such a vision to apply AI to a global network of sensors is further reinforced by analogous efforts [see Google DeepMind (2018), TensorFlow (2016)], who are showing an increasing interest in home automation technologies (see Nest Labs 2019), IoT platforms (see Google Cloud IoT 2018) and smart services (see Amazon Web Services IoT 2018). Similar ideas, for example Sundmaeker et al. (2010) and earlier works such as that of de Bruijn and Stathis (2003), require things to "interact and communicate among themselves and with the environment by exchanging information sensed about it while reacting autonomously to the physical world events and influencing it by running processes that trigger actions and create services with or without direct human intervention". The adoption of these ideas for a variety of popular applications that provide smart electronic services for domestic, healthcare and work environments suggest that their supporting technologies are here to stay.

However, the numerous application areas requiring IoT and sensor-actuator networks combined with the specialized devices used in each has led to the creation of countless specialized middleware. Zachariah et al. (2015) show that this problem has led to a multiplicity of problem specific middleware, creating interoperability issues between the architectures they enable due to the diversity in the technologies used and the architectural approaches to IoT. For example, Hydra [see Eisenhauer et al. (2009), LinkSmart (2018)] abstracts devices as services using semantic ontologies to implement discovery while Google Fit (2018) uses

✉ Nikos Dipsis
ndipsis@gmail.com

Kostas Stathis
kostas.stathis@cs.rhul.ac.uk

[1] Computer Science Department, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK

a representational state transfer (RESTful) application programming interface (API) as discussed in Fielding (2000) and does not use high level abstractions for incorporating new devices in its architecture. Google Cloud IoT (2018) and Amazon Web Services IoT (2018) on the other hand work with edge-based services (closer to the sensors/things, localised services) and cloud based web services. They both require from devices and settings to run their proprietary software to access their infrastructures and their cloud web-services that include powerful machine learning and analytics functionalities.

But what if we wanted to apply a machine learning algorithm to an existing IoT setting to add further intelligence to the environment (e.g. Mehmood et al. 2019), or enable a software agent to enrich with AI capabilities a smart home (e.g. Poncela et al. 2018) consisting of multiple and not necessarily interoperable technologies? We are trying to answer these questions through a framework that demonstrates how to simplify the incorporation of AI capabilities to existing sensor-actuator networks or IoT infrastructures making the services offered in such settings smarter. Yet another IoT or sensor network middleware directly connecting the low level sensors and physical devices with AI capabilities would only contribute to the existing mosaic of available middleware. As the current state of the art indicates, such attempt would inevitably be application/hardware specific and not very useful to most existing systems. Instead we want to integrate, when possible, with existing settings and make them smarter with the added benefit of interoperability between heterogeneous and diverse IoT architectures.

Working at this level would take away the specialized sensor and smart device integration complexities that has led to the multitude of IoT middleware approaches and would allow the middleware to work with existing settings instead of requiring their replacement. In this context, we argue that what will simplify a developer's task is a more customized middleware that takes into account the particularities of binding an AI to a sensor/actuator network to make their integration transparent and systematic. The reason why integration transparency is important is that it can abstract away the low-level details of how an AI discovers and interacts with a set of a sensors and actuators, as in Görgü et al. (2018). In other words, the system developer that uses eVATAR+ in an application would describe AIs and devices using abstractions and the middleware would bind them to each-other and route messages between them without the developer having to deal with or have knowledge of how this is done inside the middleware i.e. transparently. Systematization refers to providing developers with a standard way of implementing the specific type of integrations that involve AI platforms and sensor/actuator/smart device networks. We aspire to simplify the task of integrating agent AI with sensor, actuator and smart device networks and a way of achieving this is by using a familiar and easy to use API. Interoperability and heterogeneity are also important features of the discussed middleware because a middleware would not be of much use if it contradicted its intrinsic goal to interconnect heterogeneous software.

Our middleware is associated with an interaction paradigm for binding AI capabilities with sensor, actuator and smart devices; the capabilities will be part of intelligent agents using different agent models (Kakas et al. 2008) or architectures (Witkowski and Stathis 2004). According to Heim (2007), an interaction paradigm is a model or pattern of human–computer interaction encompassing all aspects of interaction, including physical, virtual, perceptual and cognitive. Our middleware's paradigm is inspired by the familiar concept of avatar as it has been popularised in virtual reality and computer games applications. However, instead of representing a user in a virtual environment, our avatar architecture explores the reverse arrangement, viz., where an AI agent running in an electronic environment is bound with an avatar body that is essentially comprised of sensors, actuators and smart devices deployed in a physical world. In this new view, the AI provides an invisible mind that controls a physical body, thus adding an anthropomorphic dimension to the integrated system. According to Epley et al. (2007) human-like qualities enable robotic and AI systems to become more familiar and comprehensible by both end users and developers. Thus, we used the notion of the avatar to conceptualize and develop a familiar, comprehensible and therefore intuitive interaction paradigm for the systematization of interactions between AI programs and heterogeneous sensor, actuator and smart device technologies.

The notion of a software component acting like a mind to control another software component representing a body with sensors and actuators is not new, for example see the agent architecture described in Stathis et al. (2004). We essentially augment that architecture to control physical sensors and actuators. Also, an important focus of our work is to produce middleware that implements such interaction transparently, i.e. a developer can bind AI agents with specific sensors and actuators for free and thus concentrate on other aspects, viz. the modelling of the application level interaction between components (e.g. Stathis and Sergot 1996), its specification and architecture (e.g. Stathis 2000), and their implications on a specific domain (e.g. Cohen and Stathis 2001). The resulting middleware is called eVATAR+, a play with the words electronic and avatar to denote that it enables an entity in an electronic environment to have an avatar through specific sensors and actuators situated in the physical environment (see Fig. 1 in the next section for an example use).

eVATAR+ is an evolution of our previous work with the EVATAR system (see Dipsis and Stathis 2010 and Dipsis and Stathis 2009). The older version featured a

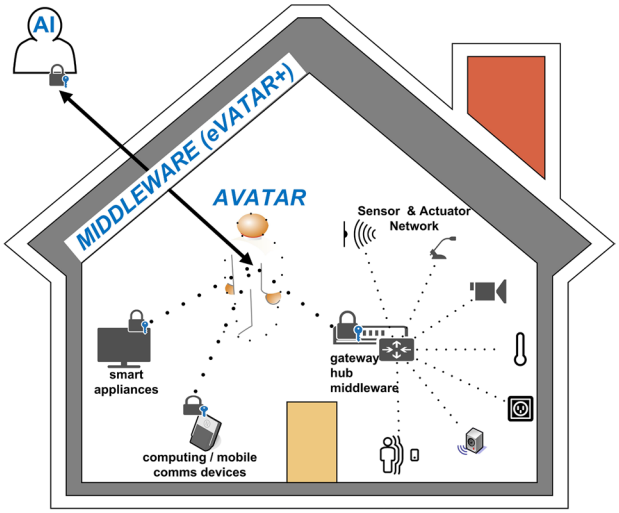

Fig. 1 Example of a smart home system that uses a middleware implemented following the avatar framework (e.g. eVATAR+). The locks indicate secure communications

message-oriented middleware with a centralized broker and XML based messaging that was enabled by adaptors that were running on the various communicating entities (agents and devices) implemented on a service-oriented architecture (see Dipsis and Stathis 2012). The new version discussed here, however, is a substantial reengineering of the previous system as a web server with enhanced mediation capabilities and a RESTful API for easier interaction with third party software (vs the cumbersome and complicated XML ontologies of the older version). In addition, we present here an approach that utilizes widely-used technologies integrated in such a way that can be easily replicated by developers.

To present our approach, the paper is structured as follows. In Sect. 2, we take a look at current state of the art of middleware that could enable the connection of AI capability to existing smart home and IoT settings and identify potential shortcomings. In Sect. 3, we describe our approach attempting to overcome these shortcomings where we present the architecture of eVATAR+ and its instantiation. Then in Sect. 4 we present a case study that illustrates the type of applications we envisage when using eVATAR+, particularly in the area of AI enabled smart homes. Finally, we summarise our contributions in Sect. 5 where we also present our plans for future work.

2 State of the art

A plethora of middleware for the IoT community is already available, see Ngu et al. (2016) for more details. We have selected here from the current state-of-the-art those middleware that could support AI to sensor, actuator and IoT device integration and we evaluate them against the following requirements. Particularly, we will be focusing on the following characteristics that could simplify the proposed integrations:

- API technology prevalence of use for the development of similar APIs and API level of complexity indicating ease of use;
- a systematic and familiar way for implementing the integrations between AI and IoT/Sensor networks;
- integration transparency.

Agent based middleware intrinsically support agent AI to sensor network integration. They are used to implement software agents on sensor and actuator networks. Specifically, an intelligent agent is commonly deployed on a single sensor/actuator/device node. Agents running on a node react to their environment in ways that support complex tasks requiring intelligence. However, key frameworks under this approach such as SensorWare in Boulis et al. (2007), Impala in Liu and Martonosi (2003), Agilla in Fok et al. (2009), Smart Messages in Kang et al. (2004), Ubiware in Michal et al. (2009) and UbiRoad in Terziyan et al. (2009) are usually implemented with an intrinsic support of a single hardware/software platform overlooking heterogeneity issues and also they are usually bound to a single AI platform.

More generally, deploying intelligent applications using this type of middleware that implements software agents can be a challenging task due to: (a) hardware limitations; (b) the complexity of programming decentralized nodes to exchange increased volumes of context data about the environment and coordinate their activities to support cooperative tasks; (c) proprietary APIs. In environments with fast and reliable networking connectivity such as in smart homes (our area of interest), there are commonly facilities for centralized processing and the sensors and actuators are usually either static (wired) or wireless. Decentralized (node level) computation is not as critical in such settings. Therefore we see that agent based middleware provide with a systematic way of integrating agents with sensor and actuator networks but they tend to be complex to use and application/domain/platform specific.

A more common approach follows a service oriented architecture (SOA) paradigm to implement middleware that provide with ways to interconnect sensor (and actuator) network nodes, IoT and smart home devices. Such middleware components focus on connectivity, interoperability and on low-level tasks such as gathering information from sensors or controlling actuators. The SOA paradigm enables middleware to also implement integrations with external applications through the representation of nodes in a sensor/actuator network as services and external systems as service consumers, for example in our case an AI agent. In this way such

middleware can support data integrations in which a sensor network produces data used by external systems.

SOA-based middleware components such as Hydra (LinkSmart) (Eisenhauer et al. 2009), TinySOA (Rezgui and Eltoweissy 2007), USEME (Cañete et al. 2009) and SIXTH (O'Hare et al. 2012), NOSA (Chu et al. 2006), OASiS (Kushwaha et al. 2007), SenseWrap (Evensen and Meling 2009), MUSIC (Rouvoy et al. 2009) and SOCRADES (Guinard et al. 2010) provide services for the integration with an AI program. Web-server based middleware exposing their services using RESTful APIs such as Kaa-IoT Technologies (2017), Konker Labs (2017) and DataArt Solutions, DeviceHive (2017) work in a similar way. However, they do not provide a systematic way for linking AI capabilities to a sensor and actuator network that abstracts away from the low-level details of how this is achieved, nor do they achieve this linking in a transparent way. In order to make this process systematic and transparent we would require a middleware that looks at the integration from an application perspective. SOA-based middleware for sensor actuator networks tend to focus on gathering information from sensors and they tend to ignore how to use this information effectively at a higher-level. SOA based middleware can achieve the integrations and they tend to offer industry standard API interactions that are familiar to a wide range of engineers and developers but their intrinsic focus is the interconnectivity of sensors, actuators and IoT devices because this is what they are designed to do. Therefore they do not offer a systematic and transparent way enabling developers to integrate AI agents to existing IoT and sensor/network ecosystems because they were not designed for this particular type of integrations.

Pervasive and Ambient Intelligence middleware such as SALSA in Favela et al. (2004), RoboCare in Bahadori (2005), the middle layer in Kim et al. (2007) and ReMMoC in Grace et al. (2005) are more flexible in terms of creating application specific solutions that use the data of low-level sensor/actuator network middleware. However, none of these middleware satisfies both required characteristics for: (a) a systematic integration (providing developers with a standard way for implementing the specific type of integrations that involve AI platforms and sensors and actuator networks) and (b) transparency (abstracting the low-level details of how an AI discovers and interacts with a set of a sensors and actuators). In general, state of the art IoT middleware such as the ones mentioned above and other notable examples such as PEIS (Saffiotti et al. 2008) and iCore (Giaffreda 2013) could also potentially enable architectures that integrate AI with IoT infrastructures, sensors and actuators. A downside in the above approaches is that there is usually a steep learning curve when attempting to integrate such middleware into a new system or use their APIs that tend to be complex, proprietary and targeting specialized audiences (due to the IoT middleware pluralism). Furthermore, being middleware, they intrinsically focus on connectivity and interoperability between different elements of sensor/actuator networks, smart home and IoT infrastructures as opposed to simplicity and integration transparency. We also considered middleware approaches for robotics such as MARIE (Cote et al. 2006) and Player (Gerkey et al. 2003) that can achieve integrations between AI programs and sensor and actuator networks, but they also do not satisfy the systematic integration and transparency requirements. Google Fit (2018) is another example of a body network middleware that is application specific and thus not suitable for the purposes of our research for a middleware capable of integrating various AI software with existing IoT and sensor network ecosystems.

Google Cloud IoT (2018) offers a solution for connecting, processing, storing and analysing data both at the edge and in the cloud. A similar approach is followed by Amazon Web Services IoT (2018). The downside of these approaches is that they require proprietary software to be run on the devices, sensors and actuators in order to participate in their infrastructures and they limit the AI to the services offered by their private clouds. Thus, they are not easily interoperable with different technologies.

Atmojo et al. (2015) propose an approach for designing AmI systems based on the use of a concurrent programming language called SystemJ. SystemJ programs control heterogeneous sensor/actuator nodes to implement distributed AmI systems. SystemJ runs on the Java Virtual Machine and provides high-level abstracted objects, signals and channels, for communications between different software entities and the nodes. It can be a complementary approach to eVATAR+ as it is generally designed to implement programmable distributed systems consisting of sensors and actuators while eVATAR+ is designed to make them more intelligent.

The following table summarizes the representative list of middleware that were considered.

Table 1 suggests that there is paucity of frameworks that enable the linking of the computation and functionality of AI programs to networked sensor/actuation devices in a way that fulfils all desired characteristics that were identified in the Introduction i.e. offering: a simple, familiar and easy to use API, a systematic and familiar way for implementing the integrations and integration transparency. We therefore found an opportunity to build upon experience gained from current research and proposed our own approach, eVATAR+. Our approach is tailor made to the particularities of integrating agent AI to sensor-actuator networks, IoT settings and smart homes offering a systematic and transparent way to achieve this (similarly to the agent based middleware) while at the same time offering a commonly used and familiar approach to API interactions. Furthermore, another goal of our middleware was independence of AI agent or IoT/sensor network platforms.

Table 1 Current state of the art evaluated against our researched application

Middleware type	For	Against
Agent based SensorWare in Boulis et al. (2007), Impala in Liu and Martonosi (2003), Agilla in Fok et al. (2009), SmartMessages in Kang et al. (2004), Ubiware in Michal et al. (2009) and UbiRoad in Terziyan et al. (2009)	A systematic way of integrating agents with sensor and actuator networks	They tend to be complex to use and application/domain/platform specific
SOA Hydra (LinkSmart) in Eisenhauer et al. (2009), TinySOA in Rezgui and Eltoweissy (2007), USEME in Cañete et al. (2009) and SIXTH in O'Hare et al. (2012), NOSA in Chu et al. (2006), OASiS in Kushwaha et al. (2007), SenseWrap in Evensen and Meling (2009), MUSIC in Rouvoy et al. (2009) and SOCRADES in Guinard et al. (2010) *RESTful web services* Kaa-IoT Technologies (2017), Konker Labs (2017), DataArt Solutions, DeviceHive (2017)	Easy to use, commonly used and familiar APIs	No systematic and transparent way enabling developers to integrate AI agents to existing IoT
Pervasive, ambient intelligence and robotics middleware SALSA in Favela et al. (2004), RoboCare in Bahadori (2005), the Middle Layer in Kim et al. (2007), ReMMoC in Grace et al. (2005), MARIE in Cote et al. (2006), Player in Gerkey et al. (2003), PEIS (Saffiotti et al. 2008), iCore (Giaffreda 2013), Google Fit (2018) and Atmojo et al. (2015).	Application specific solutions that use the data of low-level sensor/actuator network middleware	Generally complex to use, application specific and no systematic and transparent way enabling developers to integrate AI agents to existing IoT
Cloud based middleware Google Cloud IoT (2018) and Amazon Web Services IoT (2018)	Complete solutions for connecting, processing, storing and analysing sensor, actuator and IoT device data both at the edge and in the cloud	Proprietary software, needs to be run on the devices, sensors and actuators and AI limited to services offered by their private clouds. Not easily interoperable with other technologies

3 EVATAR+

In order to ground our discussion on eVATAR+ and exemplify its concepts we consider as our motivating application one that needs to use IoT integration for a smart home. The way we envisage the use of eVATAR+ middleware for this class of applications, is depicted in Fig. 1 below.

Figure 1 illustrates an AI agent interacting with smart appliances, smart phones and with a gateway that controls a smart home sensors and actuators. eVATAR+ is installed in the edge of the local network and interacts with an AI in the cloud. In other settings the AI software would typically run locally or on a smart phone.

The main idea behind the middleware is to enable AI programs/agents to register with it by sending abstract descriptions of the functionalities they require in the search of a new avatar. The avatar would be a set of physical devices that also register with the middleware by sending abstract descriptions of their functionalities. eVATAR+ performs discovery by matching the registered agents with suitable registered devices. It will then map them together in a way that the agent is able to send action requests to the devices or receive sensory data from them, thus enabling it to enhance existing sensor networks or IoT settings with AI capabilities. This way the devices will constitute the avatar body of the agent in the physical world.

Throughout the paper and for presentation simplicity whenever we speak about physical devices, we refer to physical devices as exposed to eVATAR+ via:

- a sensor and actuator network middleware
- an IoT infrastructure
- a smart home application
- directly in the case that they are "smart" and capable of supporting the calls required by the eVATAR+ API.

Only in the last case eVATAR+ would communicate with the devices directly. Still, whether the software that registers a device is run on the device itself or whether it is run in a controller gateway that exposes its functionality to eVATAR+, the functionality of the latter will remain the same. In other words, the way eVATAR+ will perform registration, discovery, binding and mediation of messages does not depend on the type of communication supported by the endpoint devices, as long as they are capable of using its API. Therefore, for simplicity we will be using the word device, sensor or actuator for all above cases as aliases to an exposed device, exposed sensor or an exposed actuator.

Having an idea of how a smart home application using eVATAR+ looks like (Fig. 1), we can now proceed with describing the middleware, starting with the architecture.

3.1 The architecture of eVATAR+

Figure 2 presents the building blocks of the reference architecture for eVATAR+. In this section we will refer to this architecture to present the implementation choices and functionality of the middleware.

The architecture assumes that any external resource such as sensors, actuators, smart devices and the AI software should call the middleware's API that uses REST (Fielding 2000) and JavaScript Object Notation (JSON) to exchange messages (JSON 2017). This allows a feedback loop between sensor data that trigger the AIs to select actions (using cognitive capabilities such as decision making, planning, learning and reactivity or social capabilities such as cooperation or negotiation) and then instruct the sensor and actuator network to perform these actions.

In order to simplify a developer's task we have selected implementation technologies that would enable deployment to the cloud, on a dedicated board computer (e.g. Raspberry Pi 2018), on an existing local server/pc or a smartphone. As a result, we have chosen the DropWizard framework (DropWizard 2017) that is essentially a collection of Java libraries glued together to enable RESTful server applications. The benefit of using DropWizard is that it enables the implementation of lightweight monolith servers and microservices. The latter can be deployed in the cloud as well as on already existing local server PC, dedicated board computers and smartphones. All options except the one of the dedicated board computer would alleviate the need for another device in the smart home. This way we can view eVATAR+ as a low footprint add-on to an existing architecture.

As we can see in Fig. 2 eVATAR+ uses a Jetty container (Jetty 2018) handled by Jersey that enables us to implement in Java a handler that supports the REST API (Jersey 2018).

The persistence layer of eVATAR+ supports the business logic with a relational database. We have chosen to use PostgreSQL because it is a powerful, open source object-relational database system. It supports multiple datatypes, scalability, a good online support community, it is being constantly improved and updated, it is cross-platform and has good administrative tools (pgAdmin, DBeaver). In our typical so far web server setting, the persistence layer uses the hibernate object relational mapping (ORM) framework that is responsible for saving the entity that is a Java object as a relational database record (Hibernate 2018). Most of the libraries that we use are part of the DropWizard framework. Furthermore, we implement the data access object (DAO) design pattern to provide an abstract interface and access to a database by using the ORM. It provides and includes all basic create, read, update, delete (CRUD) methods to interact with the database. The DAO is as light as possible and exists solely to provide a connection to the database.

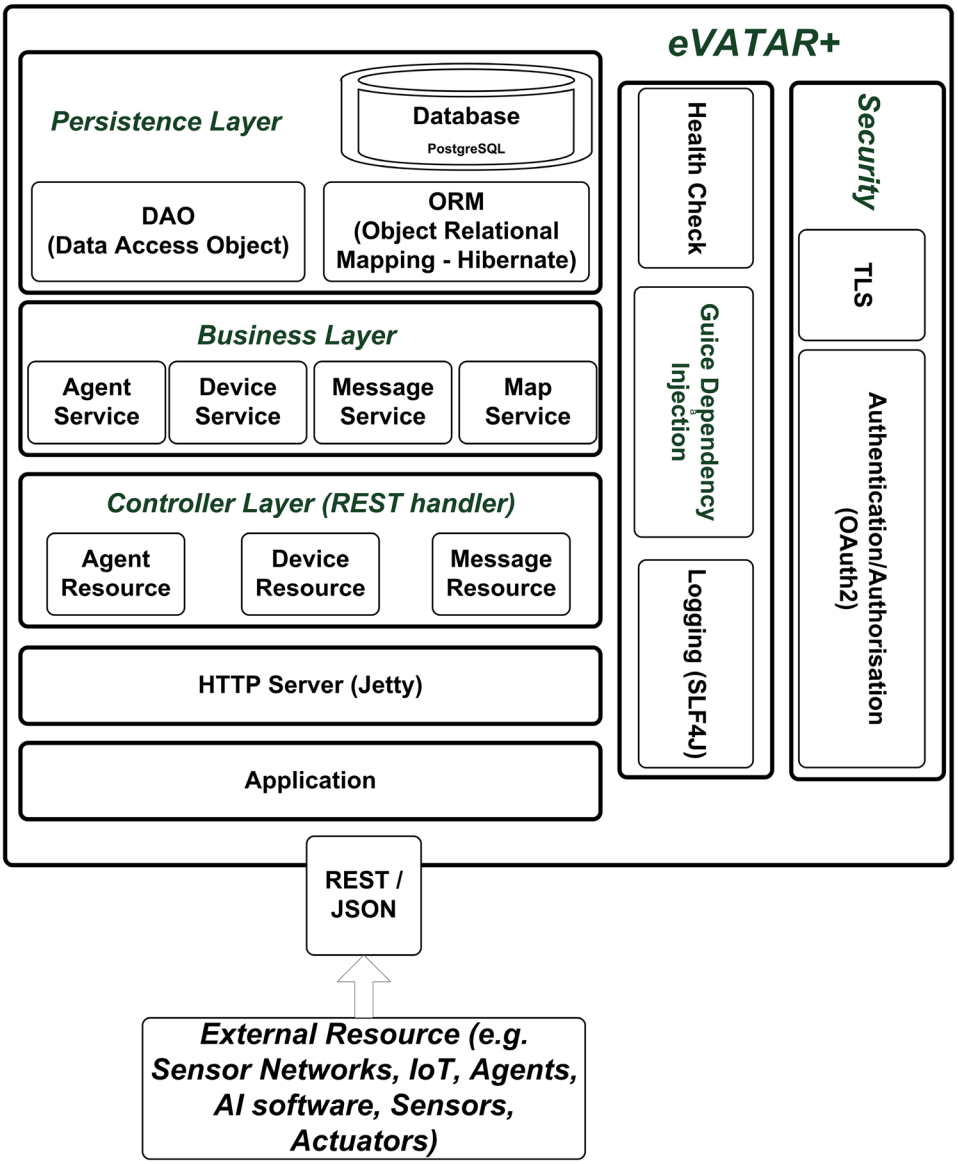

Fig. 2 The reference architecture of eVATAR+

Furthermore, in our web server architecture of Fig. 2 there is a service layer which can be described as a layer between the resources in the controller layer and the DAO class in the persistence layer. The service layer is called by the resource layer and makes use of the DAO to interact with the database. The service layer provides the business logic to operate on the data sent to and from the DAO and the client. This is why we all this layer as the business layer in the architecture of Fig. 2. Another reason for using an extra service layer to add business logic is security. A service layer that has no relation to the DB, makes it more difficult to access the DB from the client unless it goes through the service. If the DB cannot be accessed directly from the client then an attacker taking over the client will need to hack the service layer as well before gaining access to the data. In the service layer we implement mediator functionality, see Gamma (1995). To support the mediations we also implement discovery of suitable sensors and actuators to an agents' requirements and their binding i.e. the creation of exclusive communication relationships between agent components and physical sensor, actuators and devices (see below for more details and clarity).

In this architecture the resource layer is independent of the data storage engine. To further ensure layer independence we use dependency injection for simplifying testing and improving decoupling. Dependency injection is a practice where objects are designed in a manner where they receive instances of other objects instead of constructing them internally.

The OAuth2 protocol (OAuth2 2018) is used for authentication and authorization for the agents and the devices that connect to the middleware and the communications are

secured with TLS protocol (TLS 2018) as per standard. In an example application a user would login using a password and acquire an expiring session token. Agents and devices in the application would use the expiring token to authenticate their communications with eVATAR+. In addition, in our proof-of-concept prototype we strive to keep the system with the latest versions of the libraries and software used (such as DropWizard and Jetty). Further considerations regarding this issue are more relevant for a commercial deployment of the system and therefore are beyond the scope of this work.

The application block, refers to the term used in the DropWizard framework for the centralized piece of code that puts everything together and runs the server (or microservice in a different architectural context).

Having identified an appropriate technological framework for our middleware, we can now proceed with the description of the design of eVATAR+ within the particular framework. In the following, we will describe only the relevant functionality and classes that are needed to implement the integrations omitting details about configuration, authentication and parts of functionality that come as standard with the DropWizard framework.

3.2 The controller layer

Describing the eVATAR+ API would be a good starting point for providing an overview of what eVATAR+ does. The eVATAR+ API allows entities to interact with other entities via eVATAR+ by sending and receiving JSON objects. eVATAR+ provides a REST/JSON API for interoperability and easy integration. REST APIs dominate the Internet because they are easy to use and widely known. Similarly, JSON is lightweight and intrinsically designed for describing data in way that is easy to be read by humans. Thus, the proposed middleware becomes accessible to a wide audience of developers. Any existing AI software technology or IoT/sensor network middleware and gateway, sensor, actuator, sensor-actuator, smart appliance or IoT device that has the resources to make REST calls can interact with eVATAR+ and participate in the proposed architecture enabling interoperability and heterogeneity.

The API allows the integrated devices and software to establish loosely coupled, asynchronous coarse grain communications between them. Most devices in modern smart homes and especially IoT infrastructures tend to feature connectivity and computing capabilities and show autonomy on their own or as part of a local setting that uses a gateway that controls them. Consequently, there is no need for a fine grain communication between a software agent and a physical device. Instead a sensor can use the API to PUT sensory data to eVATAR+ while the agent would be polling to GET the sensory data or an agent can PUT a request to be temporarily stored in eVATAR+ while an actuator, or a sensor actuator or an IoT device would be polling to GET the request (or set of requests) and carry on the appropriate tasks in the physical world.

The API work in eVATAR+ is performed in the "controller layer" by resource classes which model the resources exposed in our RESTful API (i.e. the Jersey handlers of the http requests Fig. 2). The resource layer is essentially a handler to the Jetty server managing API calls. The UML diagram below shows the main resource classes of eVATAR+ (Diagram 1).

Agents and devices register by POSTing their descriptions in JSON format to eVATAR+ (see Table 2). The device description object in JSON always contains a type tag indicating whether it is a sensor, actuator or smart appliance (including IoT devices). There is a "description" element where we can describe the device and what it does and this description can be used (future work) for display purposes in a UI. The metadata section is important as it uses an array describing the particulars of the device such as the status of the sensor (e.g. 1 for sensed motion 0 for the opposite) and its location. There is no limitation in what we can put in the metadata section. The "name" element should contain a unique name for the device.

The agent description object in JSON similarly contains its type and description as well as an array of device descriptions. Its device description in the agent object has the exact same structure as the standalone JSON descriptions of the physical devices. eVATAR+ then performs discovery by matching the required by the agent device descriptions to already registered descriptions of sensors and actuators and maps the compatible ones. A required by an agent device and a physical device are compatible if their metadata arrays have the same values. For example in Table 2 shown below the required motion sensor and the physical motion sensor have the same values in their metada arrays ("metadata":["status"]).

When we POST a description, eVATAR+ returns JSON objects containing unique identifying (Ids) Long integer numbers for the entities being described. As we can expect the device will receive a unique Id that it will be using for every future communication with eVATAR+ and the agent will receive a unique Id for itself and an array of identifiers for every required device that it describes.

Agent registration messages like the one of Table 2 are handled by the AgentResource class in Fig. 3 and in particular by the registerAgent handler function that deals with the agent registration. Similarly, device registration are handled by the DeviceResource class and the registerDevice handler function. The locality element allows us to determine a particular sensor/device and comes handy in settings where we have the same type of sensors/devices. It takes a string that can describe a location or an identifier.

Table 2 API for registration by POSTing JSON descriptions of agents and devices

Agent Registration	*Device Registration*

`POST /api/v1/agent`
 	
Agents POST JSON objects containing registration metadata that describes the functionality they require. For example:
 	
`JSON Object`
`{`
`"type": "agent",`
`"description": "Jade agent",`
`"devices":`
`[`
`{`
`"type": "sensor",`
`"name": "requiredsensor1",`
`"description": "motion",`
`"locality": "livingroom",`
`"metadata":`
`["status"]`
`},`
`{`
`"type": "actuator",`
`"name": "requiredswitch1",`
`"description": "lightswitch",`
`"locality": "089e8eejd",`
`"metadata":`
`["set"]`
`}`
`]`
`}`
 	
Returns JSON Object:
 	
`{"agentId": "123",`
`"devices":`
`[{"name":"requiredsensor1",`
`"deviceId": "232"},`
`{"name": "requiredswitch1",`
`"deviceId": "552"}]}`
	`POST /api/v1/device`
	
	Sensors, actuators, smart and IoT devices, gateways controlling multiple POST JSON objects containing registration metadata that describes the functionality they provide. For example:
	
	`JSON Object`
	`{`
	`"type": "sensor",`
	`"name": "motionsensor1",`
	`"description": "motion",`
	`"locality": "livingroom",`
	`"metadata":["status"]`
	`}`
	
	Returns JSON Object:
	
	`{`
	`"name": "motionsensor1",`
	`"deviceId": "123"`
	`}`

Other API functions handled by the DeviceResource and the AgentResource handlers:

The register devices PUT sensory data with messages like the ones in Table 4 below and the agent is polling the middleware to GET this sensory data. In general, when a device or an agent make a PUT rest call with an action request or to send sensory data, they include JSON objects like the ones of Table 4. These calls include the Id (e.g. 334) that was returned to them by eVATAR+ when they registered by POSTing their descriptions.

We notice that the message JSON objects contain a metadata section that has elements with names that match the String values in the metadata array of the corresponding device registration description JSON object (see Table 2).

3.3 The Business Layer

The business layer in eVATAR+ implements the business logic that supports the resource (controller) layer. As we have seen in the architecture the classes in the business layer are called services and they enable the REST API handlers in the resources layer to interact with the persistence layer indirectly while also providing the business logic to operate on the data sent to and from the DAO and the client.

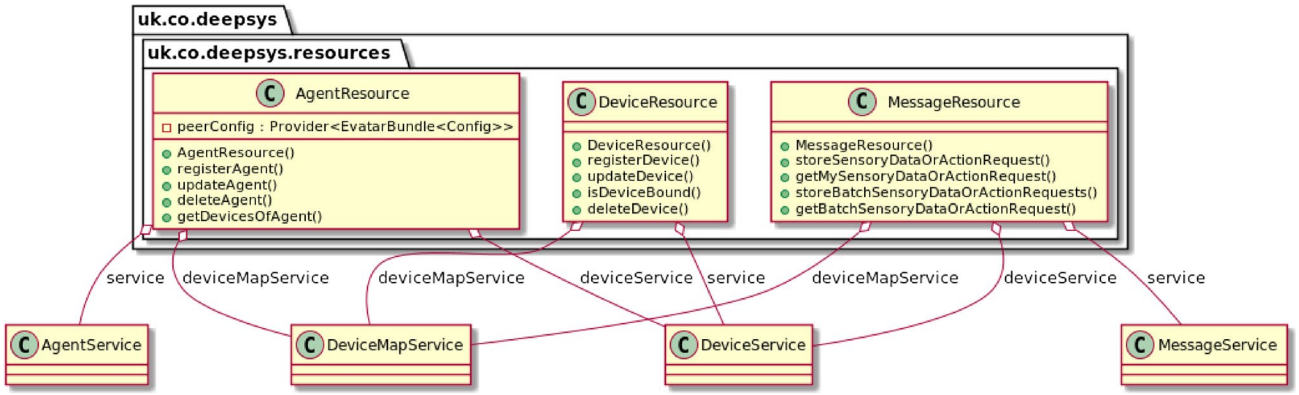

Fig. 3 UML diagram of key classes in the resource layer (we show only the important aspects)

Figure 4 shows a UML class diagram of the most important Service classes in the business layer. We can see that they all have the same superclass the "Service" class that calls the DAO (data access object) functionality enabling them to interact with the persistence layer and the database. We use a separate service layer not having the resource classes calling the DAO directly because it enables the controller layer to be independent from the data storage and we can add extra business logic here.

With regards to the business logic, eVATAR+ implements in the services layer the mediator design pattern, as described in Gamma (1995). The agents/AI software and the sensor networks/IoT settings do not exchange messages directly but via eVATAR+'s mediator functionality. The mediator pattern reduces communication complexity between multiple endpoints and it supports loose coupling (this way a change in the code of one participant would not require a change in the code of the other and thus the code is easier to maintain). The mediator behavioural pattern in eVATAR+ is supported by the processes of discovery and binding. In eVATAR+ discovery is the process by which an agent finds (discovers) the set of sensors, actuators and devices that it requires in order to sense and act in the physical world. We saw that during the registration stage, an agent will POST a description JSON object. This JSON object also includes a set of descriptions of required sensors, actuators and devices. Discovery in eVATAR+ is the process of finding sensors/actuators/device records in the database that are compatible to the ones described in the agent description. The compatibility is determined by comparing their metadata elements. For example, in Table 2 we see the motion sensor required by the agent and the device description (of the motion sensor in the right) have identical strings in their metadata arrays of strings ("metadata":["status"]). eVATAR+ would consider them as compatible. Discovery is implemented in the DeviceService class of Fig. 4 ("discoverCompatibleDevice()").

Binding is the result of the discovery and it essentially means that eVATAR+ maps required by an agent devices to physical devices and when a message from a required device is received, eVATAR+ will make it available to the mapped (mapped) physical device and the opposite. The mapping logic is implemented in the DeviceMapService class.

If an agent/AI software upon registration does not find a suitable physical device for all the required device in its description it will poll eVATAR+ to check if one is found later on (see Table 3). After this point agents can interact with the physical devices by sending messages such as the ones in Table 4. Agents and devices use their own device Ids when making GET REST calls and eVATAR+ uses these Ids to identify their mapped counterparts in the persistence layer (see below). The mapped device will access the message when it polls eVATAR+ for its messages (Table 4). This way the different communication endpoints are loosely coupled, do not communicate directly with each other and thus eVATAR+ implements the mediator functionality.

3.4 Persistence layer

The business logic of eVATAR+ is supported by the persistence mechanism that uses a relational database (PostgreSQL). The services in the business layer use the DAO object to interact with the database of Fig. 5 that shows the most important to the described framework database tables.

The posted JSON descriptions of agents and devices that are handled by the Jersey REST API handler (in the controller layer) of eVATAR+ are represented in the software as JPA entities (Java Persistence API—https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html) and persisted in the database. JPA entities are used for mapping Java objects to relational database tables and in particular they are Java objects whose non-transient fields should be persisted to a relational database (according to Oracle). An agent entity Java object in other words contains all the data fields of the JSON object

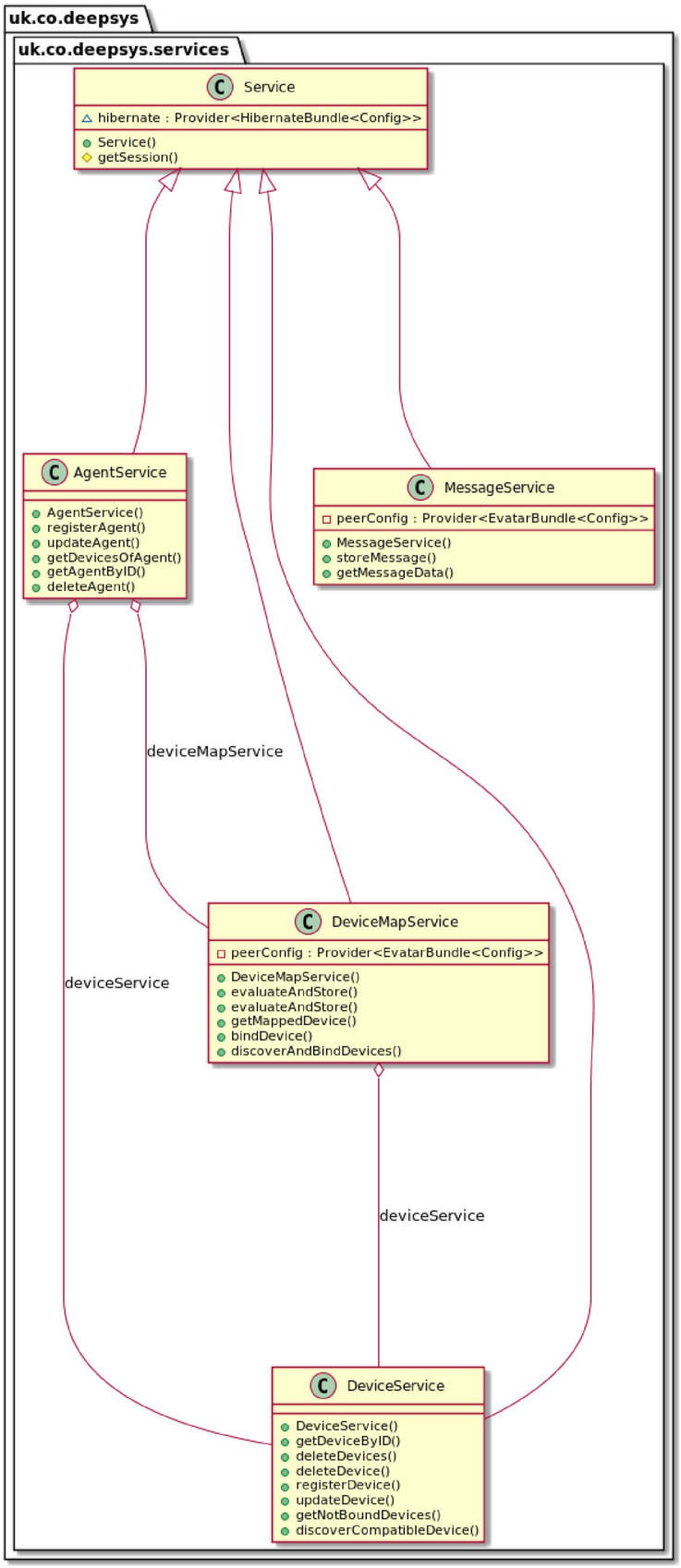

Fig. 4 UML class diagram of the most important service classes in the business layer

Table 3 API calls for updating and deleting agent and devices registrations in eVATAR+

Handler resource	API
AgentResource	GET/api/v1/agent/devices/{agentId} Agents poll eVATAR+ for new device bindings in the case that they have not found a match to all the required devices upon initial registration. The agentId is the identifier returned by eVATAR+ upon registration
AgentResource	PUT/api/v1/agent/{agentId} Agents update their registration metadata by sending JSON objects with the same structure as the ones in the registration in Table 2. The agentId is the identifier returned by eVATAR+ upon registration
DeviceResource	PUT/api/v1/device/{deviceId} Sensors, actuators, smart and IoT devices, gateways controlling multiple devices update their registration metadata by sending JSON objects with the same structure as the ones in the registration in Table 2. The deviceId is the identifier returned by eVATAR+ upon registration
AgentResource	DELETE/api/v1/agent/{agentId} Agents can delete their data from the middleware and deregister (agentId is above)
DeviceResource	DELETE/api/v1/device/{deviceId} Sensors, actuators, smart and IoT devices, gateways controlling multiple devices can delete their data from the middleware and deregister (deviceId as above)

Table 4 JSON objects for PUT calls to send action requests/store sensory data and GET calls for polling eVATAR+ for action requests and stored data

Agent	*Devices*
GET /api/v1/message/{232} *Agents poll the middleware and download sensory data if there is any. For example: poll message from agent for data from motion sensor that is associated with the agent device Id 232. According to the mapping table the physical motion sensor associated with it has Id 334. eVATAR+ will return from the database what is stored for 334.*	**PUT /api/v1/message/{334}** *Devices with sensing capabilities (sensors, actuators, sensor/actuators, IoT devices) send sensory data to the middleware in encoded within series of JSON objects. For example: motion sensor with id 334 storing sensory data.* **JSON Object:** **{** **"msgType": "sensory_data"** **"metadata":** **{** **"status": "1",** **}** **}**
PUT /api/v1/message/{552} *Agents send action requests to the middleware encoded within JSON objects. For example: agent sends an action request via agent device Id 552.* **JSON Object:** **{** **"msgType": "action_request"** **"metadata":** **{** **"set": "ON"** **}** **}**	**GET /api/v1/message/{156}** *Sensors, actuators, smart and IoT devices, gateways controlling multiple devices poll the middleware and download any requests for them encoded as JSON objects.* ***Poll** message from device with Id 156 for action requests from agent via mapped agent deviceId 552.*

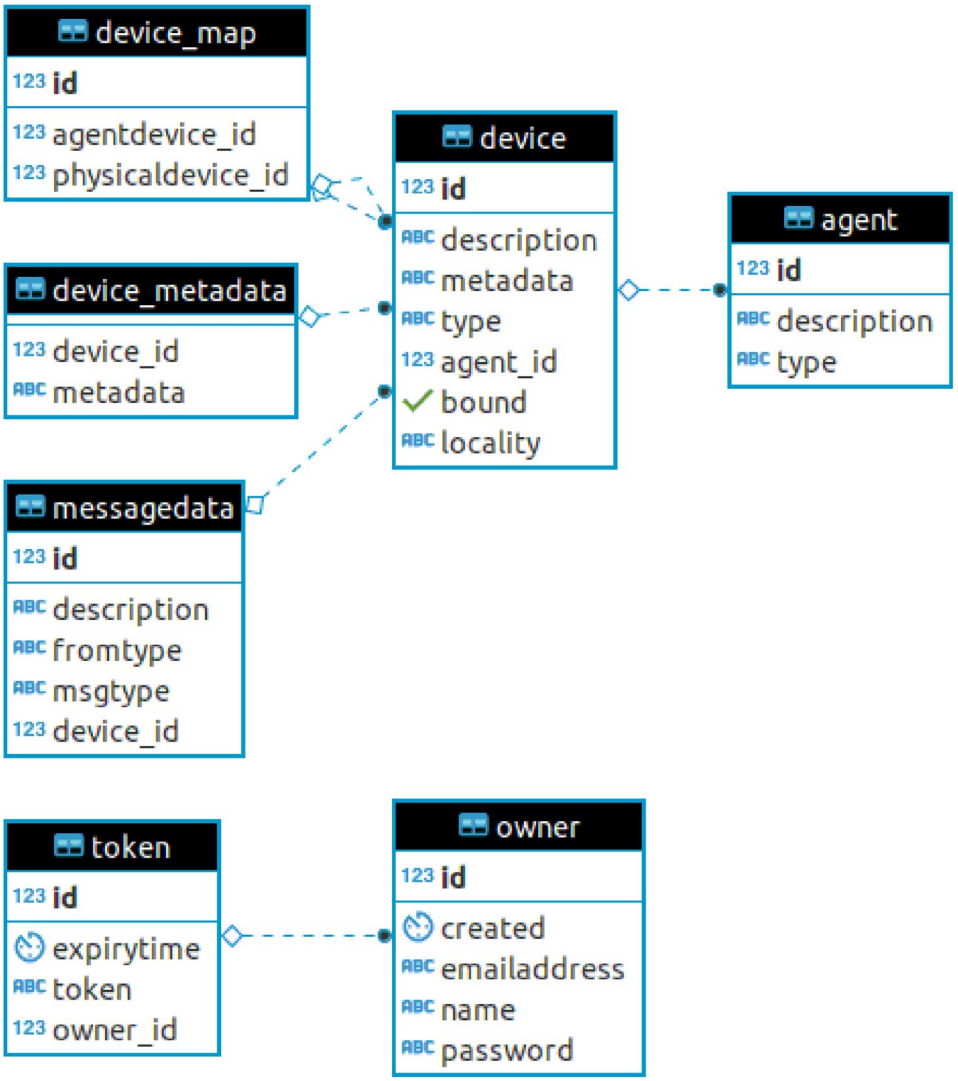

Fig. 5 Entity relationship diagram with the main tables required by eVATAR+

for the agent. The hibernate ORM framework persists such entities as relational database records (agent records in Fig. 5). Similarly, a device (sensor, actuator, IoT device) description is sent in JSON object format, converted into a device JPA entity and persisted in the database as device records (Fig. 5). We saw that the POSTed agent description contained an array of required device descriptions. These will also be stored in the relational database just like their real counterparts. The JPA entities that will be stored as PostgreSQL records (Listing 1).

Every agent and device record in the database has a unique identifier (Id). The database stores agent records as referencing multiple required device records (this is an "one-to-many" relationship Fig. 5). This means that when we store an agent, besides the agent record the database stores new records for every sensor/actuator/device that it requires and all these records have also their own unique Ids. These are the Ids that are returned to the agent (receives its own agent Id and the Ids of all its required device database records) as a response to a POST description call when it registers. The registerAgent function in the business layer returns the agent and device records that include the Ids making them available to the API. The Ids are used for further communication with eVATAR+. Similarly, registerDevice will return the deviceId.

The persistence layer also the business layer functionality that implements discovery, binding and the mediator pattern. When a required device by the agent is compatible to a physical device that has been registered and has a record stored in the database, eVATAR+ updates a database table that stores a mapping of their identifiers (in our example device Id 232 with device Id 334). This is the "device_map" table in Fig. 5 (Listing 2).

Now we say that the required device by the agent is bound to the real device and this enables an exclusive communication between the agent and that particular device. When the agent polls (GET) for sensory data for deviceId (232 in our example) using the Id of the requested motion sensor,

eVATAR+ uses the "device_map" table in the database that has it mapped with 334. Then it will use the mapped Id (334) to retrieve all stored messages from the device with Id 334 if any and return them to the agent that contains the required device with Id 232. The aforementioned process illustrates how the persistence layer supports our implementation of the Mediator pattern. The agent and the devices do not talk directly to each other. Instead they communicate via a shared memory space in eVATAR+ thus achieving loose coupling. For fine grain communication, if needed, Jersey supports streaming and it is similar to uploading a file one end and downloading it on the other end. On our setting though the costly streaming should be rarely needed.

After describing all key components of eVATAR+ the following UML activity diagram would provide the reader with an overview of what happens eVATAR+ receives a message.

Having described the most important aspects of the eVATAR+ architecture and design we can exemplify its use with a case study.

4 Case study: agent capabilities in Google NEST

In this section we discuss a case study exemplifying how eVATAR+ (and its associated architectural framework) binds AIs with sensors and actuators in a systematic, secure and transparent manner. The presented case study intends to provide insight about the type of applications eVATAR+ is designed to support. In our case study our goal is to show how we can use eVATAR+ to enable a Jade (Bellifemine et al. 2007) MAS agent to integrate with an Nest application in order to become an actor within a Nest smart home setting by reading sensory events within it and also creating events. We have selected NEST because it offers a well-documented API, a simulator and is also a widely used in modern households allowing us to demonstrate how eVATAR+ would be applied in such a setting.

4.1 The scenario

Let us consider that we have a Nest application controlling a Nest smart home setting. We are going to show how a developer can use eVATAR+ to enrich the smart home setting with software agent capabilities. We are going to show how the Nest application uses the eVATAR+ API to register the smart home devices and how an agent registers its interest for smart home devices by using the same API. eVATAR+ will perform discovery and bind the agent to a set of smart home devices enabling it to apply its functionalities to the smart home.

Initially we used the Nest Home Simulator (https://developers.nest.com/guides/home-simulator) that allows us to easily simulate events in the smart home that were also made available to the Jade agent via eVATAR+. We will show how to enable an agent to sense the simulation environment by receiving e.g. motion detection events from a camera (Fig. 6) and perform actions in it e.g. control a thermostat. The simulation uses exactly the same API as the real sensors and actuators and we could switch our application to a real environment without making any changes to our application. In the second part we exemplify this point by replacing the Nest simulator with a Nest thermostat (as the simulation cannot coexist with the physical Nest devices in the same setting). Figure 7 illustrates an overview of the architecture that would enable a Jade agent (Bellifemine et al. 2007) to control Google Nest devices.

Google Nest and Google IoT infrastructures offer a complete solution for connecting, processing, storing and analysing data both at the edge and in the cloud. Their infrastructure software is not accessible to developers in any way other than via using their public APIs. Nest offers a RESTful API, therefore a Nest application making REST calls to Nest should also be capable of making REST calls to eVATAR+ and use its API. In our scenario we will integrate a Nest smart home application with a Jade agent via eVATAR+.

4.2 The smart home setting

The Nest home simulator is a self-contained application for creating virtual versions of the Nest physical devices. Interaction with the Nest home simulator is identical to the interaction with a similar setting consisting of physical devices including authentication and identical REST API calls to communicate with the devices, whether virtual or physical. The added benefit of the simulator is that it simulates conditions that would be expensive and time-consuming to replicate in a real world setting. Our Nest application connects to simulations of:

- smart thermostats that can read and set current temperature, set target temperature, read humidity levels.
- smart cameras that also perform motion detection cloud storage and send notifications.
- Smoke and carbon monoxide alarms that trigger as expected a smoke and CO alarm.

The following snapshot of the Nest simulator shows a virtual camera and how we can set events that will be sensed by it.

The simulation enables us to generate different types of events such as sound, motion, smoke, carbon monoxide leak and temperature related events to name a few. These events

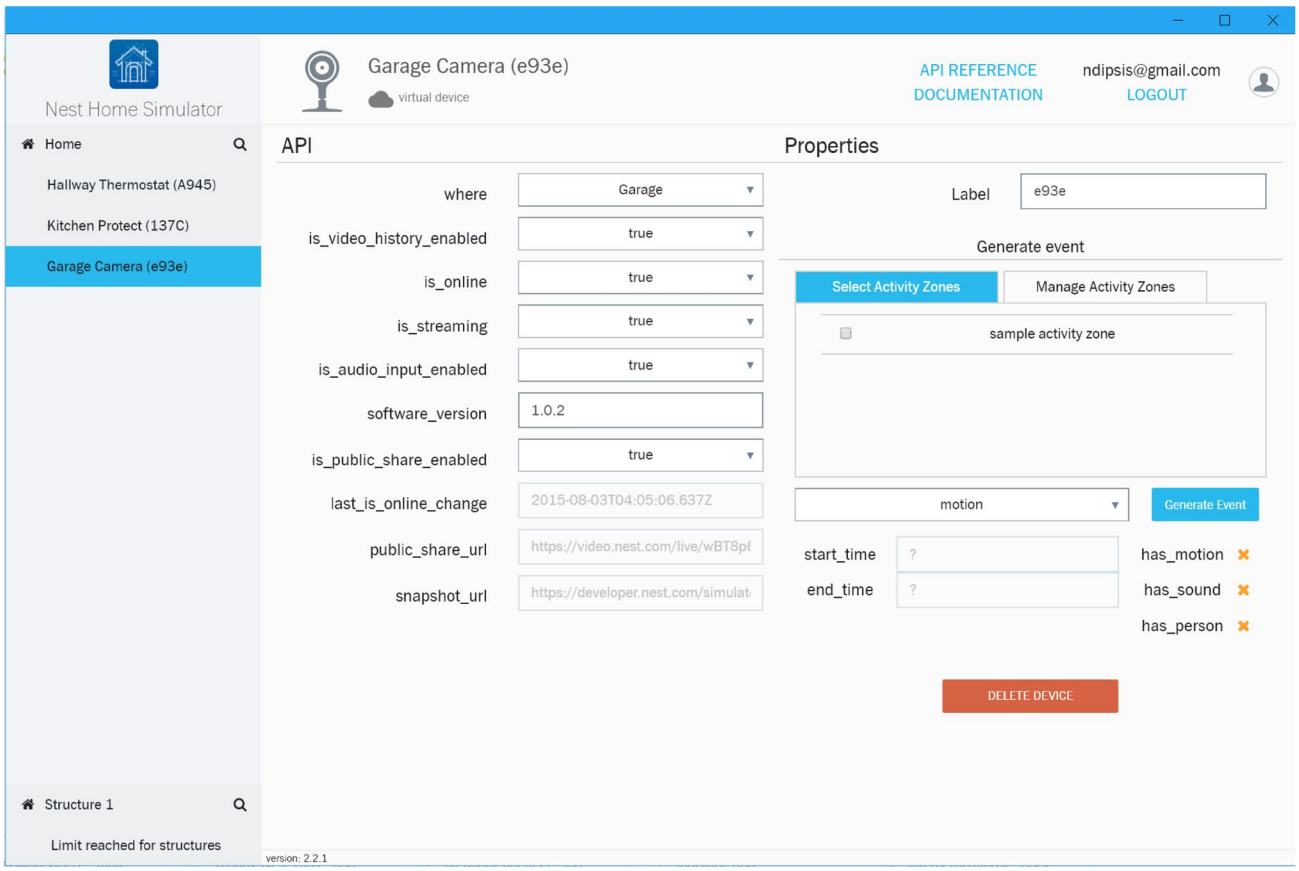

Fig. 6 The Google Nest simulation. Here we see a camera sensor. Notice that we can set motion and sound events that will be sensed by the camera, their duration, even a streaming video status

are sensed by the virtual devices and we can access them via the Nest REST API. Also, we can use the same API to alter the state of the devices in the simulation, for example to set the target temperature of the thermostat.

We implemented our Nest application in Java. An example call to Nest for setting the target temperature of a Nest Thermostat would look like shown in Table 5.

In order for an application to interact with the NEST infrastructure, it needs to authenticate itself using OAuth 2.0 after which it receives an access token (long alphanumerical token that will be used by the API calls to verify that the application is authorised to control the devices). Having a Nest application that interacts with the Nest devices (in the simulation) using the REST API of Nest we can pursue the goal of this case study which is to show how we can enrich an application in the Nest environment with agent capabilities. To achieve this, the Nest application should use the eVATAR+ API to register the simulation devices with eVATAR+. We remind that in order to register the devices, the Nest application will need to POST to eVATAR+ their descriptions using JSON. In order to create the JSON description of e.g. a smoke and CO sensor we would need to extract the useful features of the particular sensor. In our case, the place to look at is at the JSON objects that are already defining its interaction with Nest.

As we can see in Listing 3 we can encapsulate the description of the Nest API JSON object into the metadata section of eVATAR+ and this way all features of the sensor are potentially accessible to the agent. In practice, when we integrate AI functionality into an existing setting we normally do not intend to replace all native control and sensing functions with new ones stemming from the AI (e.g. agent) component. Instead, we select those useful to the agents' goals and capabilities.

In Listing 4 we see a more compact description that only contains information that would be useful to an agent. We also see how an API call by the Nest application to eVATAR+ would look like. In particular this call sends the state of the sensor for it to be read by the agent (see chapter 3 for more information).

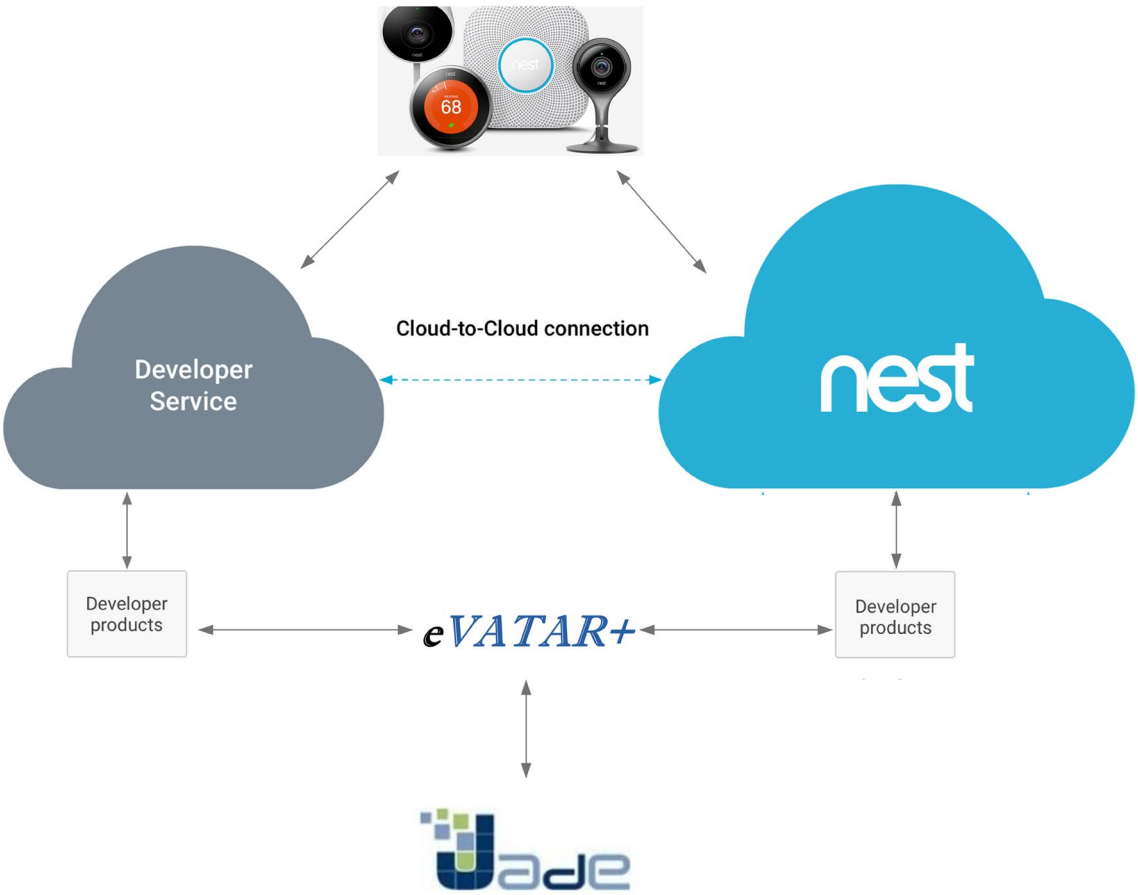

Fig. 7 Jade agents applied to a Google Nest smart home setting. The setting is inspired by the example architecture in https://developers.google.com/nest/guides/architecture

Table 5 API reference to POST and GET calls to Nest for reading and setting the temperature of a Nest thermostat

API	*Description*
POST https://developer-api.nest.com/devices/thermostats/device_id/ Java – Jersey client call **client.target("https://developer-api.nest.com/devices/thermostats/").** **path(your_thermostat_device_id).** **request().** **header("Content-Type", "application/json;charset=UTF-8").** **header("Authorization", "Bearer "+ authorisationToken.trim()).** **accept(MediaType.*APPLICATION_JSON*).** **post(Entity.*json*(theromstatRequest) , Integer.class);**	POST: set the target temperature of the thermostat The ORM object theromstatRequest corresponds to the following JSON: **{target_temperature_c: 22}** Returns: number Example: 21.5 Range: 9-32

4.3 The multi-agent system

Jade applications (Bellifemine et al. 2007) are implemented in Java. We implemented for the purposes of our case study a single Jade agent that would interact via eVATAR+ with the Nest smart home. We saw that the Nest application would POST three devices to eVATAR+: a smart camera, a smart thermostat and a smoke and CO sensor. In Listing 4 we saw the JSON description for eVATAR+ of the smoke_co_alarm. The smart camera and smart thermostat were similarly described. On the other end our Jade agent sent a JSON object describing three required devices (matching the physical devices of the Nest smart home) (Listing 5).

Our agent features cyclic behaviours (atomic behaviours that must be executed forever). There are cyclic behaviours polling the state of sensors, e.g. periodically requesting the last change in the state of the camera sensor and update the internal state of the agent with the acquired information. Other cyclic behaviours consult the current internal state of the agent that is essentially a collection of data structures reflecting the sensory data describing the physical environment and send action requests via eVATAR+. The following JAVA code sample illustrates an example of an implementation of the sensing behaviour in the Jade agent (we notice that it uses the JAVA API to eVATAR+) (Listing 6).

Similarly, cyclic agent behaviours check variables like "motion_sensed" and send action requests to Nest devices via eVATAR+ and the application.

4.4 Completing the Picture

The Nest application registers the Nest devices and the MAS registers the descriptions of the devices it requires. eVATAR+ performs discovery as described in 3.3 and binds the required devices to the physical ones. This way Jade has access to the existing Nest application and can add intelligent and interoperable behaviours. In our simulation we can create events e.g. set the smoke alarm go and the Jade will be polling the smoke and CO sensor and as soon as it receive a smoke event it will send a notification (for the purposes of our test it sends an email). In general for the purposes of our integration capability evaluation we had Jade sending emails of describing events that we set in motion in the Nest simulation that were sensed by the sensors and sent to eVATAR+. Also, Jade was able to change the state thermostat and this way we show how the agent becomes another actor in the Nest environment capable of reading and creating events.

We also implemented an integration with a real thermostat simply by replacing the simulation with the real smart home. No code changes where required apart from the

Agent

```
@Table(name="agent")
public class Agent {

@Id
@Column(name = "id", columnDefinition = "serial")
@GeneratedValue(strategy=GenerationType.IDENTITY
)
private Long id; //A unique identifier for the interactor.

@Column(name = "type")
private String type;

@Column(name = "description")
private String description;

//The required devices
@OneToMany(mappedBy="agent")
private List<Device> devices = new ArrayList<>();

   //Getters and setters
}
```

Device

```
@Table(name="device")
public class Device {
 @Id
 @Column(name = "id", columnDefinition = "serial")

@GeneratedValue(strategy=GenerationType.IDENTITY
)
 private Long id; //A unique identifier for the interactor.

 @Column(name = "type")
 private  String type;

 @Column(name = "bound")
 private Boolean isBound;

 @Column (name = "description")
 private String description;

 @ElementCollection
 private List<String> metadata = new ArrayList<>();

 @ManyToOne
 private Agent agent;

 //Getters and setters
}
```

Listing 1 JPA entities representing an agent and a device. This is how they will be stored in the relational database. Note that the required devices of the agent use the same type of device records as the physical devices

```
@Table(name="device_map")
public class DeviceMap {
    @Id
    @Column(name = "id", columnDefinition = "serial")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    @OneToOne (fetch=FetchType.LAZY)
    private Device agentDevice;

    @OneToOne(fetch=FetchType.LAZY)
    private Device physicalDevice;

    //Getters and setters
}
```

Listing 2 JPA entity representing the mapping between a requested agent device database record that has been registered via the agent registration process and a physical device record that was stored via the physical device registration. This mapping is stored in the relational database. Note: (i) the required devices of the agent use the same type of device records as the physical devices. (ii) this table enables the routing of messages from the agent with the requested device Id to the physical device with the mapped device Id and the opposite

Nest (smoke_co_alarms)

```
{
"device_id" : "aDevId",
"locale" : "en-UK",
"software_version" : "1.01",
"structure_id" : "aStructId",
"name" : "Hallway (upstairs)",
"name_long" : "Hallway Protect
(upstairs)",
"last_connection" : "aTime",
"is_online" : true,
"battery_health" : "ok",
"co_alarm_state" : "ok",
"smoke_alarm_state" : "ok",
"is_manual_test_active" : true,
"last_manual_test_time" : "aTime2",
"ui_color_state" : "gray",
"where_id" : "aWhereId...",
"where_name" : "Hallway"
}
```

eVATAR+ (smoke_co_alarms)

```
{
"type": "sensor",
"name": " smoke_co_alarms1",
"description":"smoke_co_alarms",
"metadata":[
"device_id",
"locale","software_version",
"structure_id","name",
"name_long","last_connection",
"is_online", "battery_health",
"co_alarm_state",
"smoke_alarm_state",
"is_manual_test_active",
"last_manual_test_time",
"ui_color_state","where_id",
"where_name"]
}
```

Listing 3 Encapsulating the native description to an eVATAR+ JSON description

configuration to target the different setting (Listings 1, 2, 3, 4, 5, 6, Diagram 1).

5 Conclusions and future work

We have presented eVATAR+, a framework with an associated middleware that binds systematically and transparently interactions between AI capabilities and existing sensor-actuator networks or IoT infrastructures, thus making the services offered in such settings smarter the context of such integrations while providing a simple and easy to use interface for developers to use. Our evaluation of current state of the art middleware with regards to the integration of sensor actuator networks and IoT settings with AI agents resulted in a set of characteristics that were used in the design of eVATAR+. We exemplified eVATAR+ with a concrete case study that illustrated a possible use of eVATAR+ and demonstrated a systematic and transparent integration of AI platform functionality (implemented in the Jade agent platform)

Useful Description

```
{
    "type": "sensor",
    "name": "smoke_co_alarms1",
   "description":"smoke_co_alarms",
    "metadata":[
    "device_id",
    "locale",
    "structure_id","name",
    "name_long","last_connection",
    "is_online",
    "co_alarm_state",
    "smoke_alarm_state",
    "where_id",
    "where_name"]
}
```

Example eVATAR+ API call

```
PUT /api/v1/data/sensor/{334}
{
 "msgType": "sensory_data"
 "metadata":
 {
  "device_id" : "aDevId",
  "locale" : "en-UK",
  "structure_id" : "aStructId",
  "name" : "Hallway (upstairs)",
  "name_long" : "Hallway Protect
(upstairs)",
  "last_connection" : "aTime",
  "is_online" : true,
  "co_alarm_state" : "ok",
  "smoke_alarm_state" : "ok",
  "where_id" : "aWhereId...",
  "where_name" : "Hallway"
 }
}
```

Listing 4 Useful JSON description to the smoke_co_alarm. The agent does not need to know about "software_version", "battery_health", "is_manual_test_active", "last_manual_test_time", "ui_color_state" etc

with a smart home setting that contains physical sensors and actuators (Nest Simulation, Nest smart home). More specifically, we have shown that eVATAR+ features a standard and systematic way of achieving the integrations by:

- abstracting sensor, actuator and AI program functionality by describing it using simple and readable JSON documents;
- using standard RESTful API calls to enable sensors to PUT sensory data in eVATAR+ that was read by polling agents (GET calls) as well as to enable agents to PUT action requests in eVATAR+ that were read by polling actuators (there was no need to write extra code for dealing with how the integrations between agents).

In addition, we illustrated how agents via eVATAR+ performed transparently dynamic discovery and binding to the physical sensors and actuators without the developers having to deal with the low level implementation of how the discovery and the binding are achieved within the middleware.

Systematic and transparent integrations already point to a simpler task for integrating AI with sensor networks/IoT environments. Furthermore, our choice of JSON/REST style integrations that are abundant in today's Internet technologies due to their simplicity and ease of use further enhances our goal to simplify developers' tasks when attempting the integrations described in this paper.

As part of our future work plans we will explore the possibility of providing a qualitative analysis of the followed approach and start a discussion as part of a research paper about the merits of providing systematic and transparent middleware solutions not just in the area of IoT. We will also explore the possibilities of implementing an application that integrates AI (e.g. Jade agents or use a machine learning component) with existing sensor/actuator technologies such as Google Nest, the Zigbee wireless standard (ZigBee 2019), Arduino (2017), Google Assistant (2017), Amazon Alexa (2017), and IFTTT (2019) among others. The proprietary nature of these technologies and the competing standards tend to lead to interoperability issues between them and the lack of a systematic way for implementing integrations. There are not many systems allowing the control of different competing technologies from a single user interface. We intend to overcome this problem by integrating their APIs with eVATAR+ enabling a centralized control unit that uses a learning AI and a UI (User Interface) for user input. We intend to investigate the possibilities and the advantages/disadvantages of deploying the application as part of an Android app (DropWizard that implements eVATAR+ can run on Android devices) and/or on the cloud or a dedicated low-cost device deployed in the edge.

We also plan to look into deploying eVATAR+ in a variety of settings and applications domains. For example we intend to investigate the possibility of deploying eVATAR+ as part of ecosystems integrating sensor networks with cloud based architectures that provide semantic world knowledge in the form of linked open data. We will then evaluate our approach in conjunction with approaches like

Agent Description

```
{
  "type": "agent",
  "description": "Jade agent",
  "devices":
  [
    {
      "type": "sensor",
      "name": "required_smoke_co_alarms_1",
      "description": "smoke_co_alarms",
      "metadata":["device_id", "locale", "structure_id","name",
                  "name_long", "is_online", "co_alarm_state",
                  "smoke_alarm_state","where_id", "where_name"]
    },
    {
      "type": "sensor-actuator",
      "name": "thermostats_1",
      "description": "thermostats",
      "metadata":["device_id","locale","structure_id", "name",
                  "name_long","last_connection", "is_online",
                  "target_temperature_c","target_temperature_high_c",
                  "target_temperature_low_c","ambient_temperature_c",
                  "humidity","where_id","where_name" ]
 } ,
 {
    "type": "sensor_actuator",
    "name": "cameras_1",
    "description": "cameras",
    "metadata":["device_id","software_version","structure_id",
                "where_id","where_name","name",
                "name_long","is_online","is_streaming",
                "web_url","app_url","activity_zones",
                "public_share_url","snapshot_url",
                {"last_event":["has_sound","has_motion","has_person",

                               "start_time","end_time","urls_expire_time",
                               "web_url","app_url","image_url",
                               "animated_image_url","activity_zone_ids"]}}
                          ]
      }
  ]
}
```

Listing 5 JSON description of an agent requiring a smoke_co_alarm, a smart thermostat and a smart camera

SPITFIRE (Pfisterer et al. 2011; Chatzigiannakis et al. 2012) that provides vocabularies to integrate descriptions of sensors and things with the "linked open data" cloud, describes their high-level states and provides search for sensors and things based on their states. Similarly to eVATAR+, they also claim ease of use due to the fact that they use commonly used and familiar technologies. In their case any application experts who are able to publish web pages should also be able to use SPITFIRE. They also provide a qualitative evaluation of their approach.

In view of extending the functionality of eVATAR+ and potentially adopting more flexible deployment possibilities we will look at deploying it as part of a microservices architecture. DropWizard is a common framework for developing microservices as well as web servers. Every DropWizard microservice would have the exact same layers and components within the DropWizard framework i.e. a Jetty container, Jersey REST API controller, a services business logic layer, ORM framework and its own database. Spring Boot (2019) offers a similar architecture with a difference that

```
/* Add the CyclicBehaviour for sensing using the Camera Sensor. */
addBehaviour(new CyclicBehaviour(this) {
  /* The motion sensor of the Jade agent */
  public void action()  {
    if(motion_sensed == false)
    {
       LOG("Sensing motion");

        /* The motion sensor s/w using the eVATAR+ API poll for Sensory data (GET). */
        CameraEntityRepresentation cameraEntityRepresentation
                                  = (CameraEntity) api.poll(“cameras_1”);

       /* If the received message indicates motion detection... */

        if (cameraEntityRepresentation.getLast_event().getHas_motion() == true)
       {
       /* update internal variable that a motion was sensed */
         motion_sensed = true;
       }
      }
    }
 } );
```

Listing 6 A sensing behaviour of the Jade agent

it provides a variety of choices for particular technologies used e.g. tomcat as an alternative to jetty. A transition from a DropWizard web server to a microservices architecture would involve using the exact same architecture and dividing the business logic and distributing the overall functionality into different microservices by: (a) dividing the database tables, (b) dividing the REST API and (c) dividing the business logic in the services layer.

The current version of EVATAR+ could be viewed as a monolithic server. At this stage there is no justification for implementing eVATAR+ using microservices as the business logic revolves around a specific task i.e. the integration of AI with sensor networks. Furthermore, we could add eVATAR+ as it is to existing microservices architectures as a separate microservice. In the future we would like to see eVATAR+ presenting more intelligent functionality e.g. to support operations on historic events, that clearly constitute big data, needed to train an AI model.

When we add this extended functionality it would be logical to migrate to a microservices architecture where one microservice would be responsible for integrations of AI with sensor/actuator networks, another for dealing with data analytics and possibly a third one for authentication and user management. The fact that every DropWizard server and microservice has the same layered architecture would make the transition easier as it would involve splitting the code and the database but keeping the same structure. Furthermore, we will also investigate ways to improve operations at a streaming level for example for anomaly detection. This would fit in with a new microservices architecture and architecturally deployed before a load balancing server that routes the data to the different microservices.

In terms of currently proposed deployment of eVATAR+ the data is not anticipated to reach high enough volumes that would significantly affect performance especially with the addition of in memory caching such as memory caching of Redis (2019). However, storage based message switching using a faster database technology and potentially with a smaller footprint such as a NoSQL/key-value database is a possible direction to explore if performance is affected by high data volumes. In this context, we will need to weigh the benefits of selecting such a technology instead of using a relational database that would support more complex business logic as we add new features and possibly data analytics.

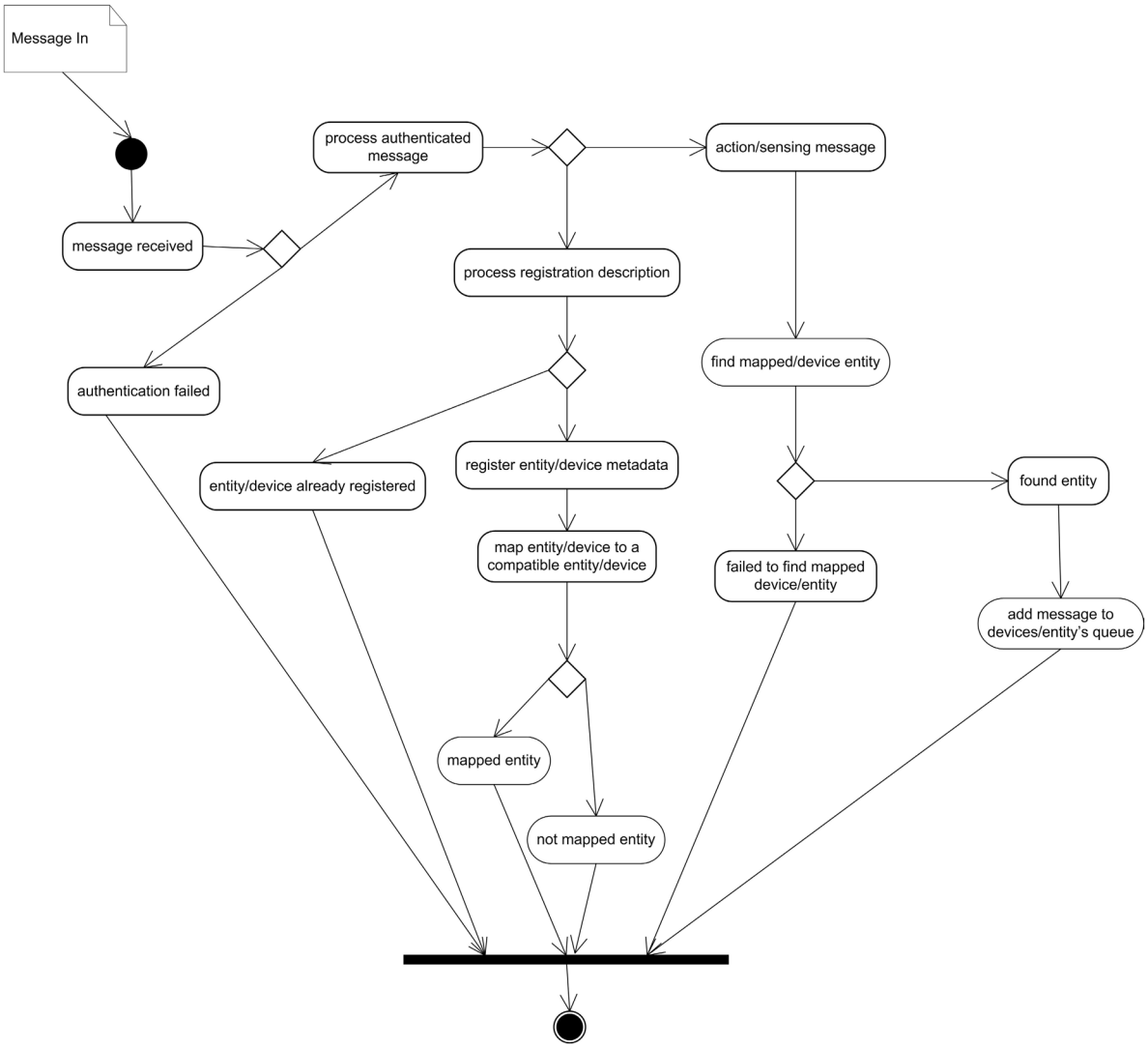

Diagram 1 UML activity diagram describing what happens when eVATAR+ receives a message

Acknowledgements We wish to thank the anonymous reviewers for their constructive comments on a previous version of this work.

References

Amazon Alexa (2017). http://alexa.amazon.com/. Accessed 30 Sep 2017

Amazon Web Services IoT (2018) https://aws.amazon.com/iot/. Accessed 28 Sep 2018

Arduino (2017). http://www.arduino.cc. Accessed 30 Sep 2017

Atmojo UD, Salcic Z, Wang KIK et al (2015) System-level approach to the design of ambient intelligence systems based on wireless sensor and actuator networks. J Ambient Intell Hum Comput 6:153. https://doi.org/10.1007/s12652-014-0221-3

Bahadori S et al (2005) Towards ambient intelligence for the domestic care of the elderly. Ambient Intelligence, pp 15–38

Bellifemine FL, Caire G, Greenwood D (2007) Developing multi-agent systems with Jade. Wiley, Hoboken

Boulis A, Han C, Shea R, Srivastava M (2007) SensorWare: programming sensor networks beyond code update and querying. Pervasive Mob Comput 3(4):386–412

Cañete E, Chen J, Díaz M, Llopis L and Rubio B (2009) A service-oriented middleware for wireless sensor and actor networks. In: Proceedings of 6th Int. Conf. on information technology: new generations, pp 575–580

Chatzigiannakis I, Hasemann H, Karnstedt M, Kleine O, Kröller A, Leggieri, Pfisterer D, Römer K, Truong C (2012) True self-configuration for the IoT. In: Proceedings of the 3rd international conference on the internet of things (IOT), pp 9–15

Chu X, Kobialka T, Durnota B and Buyya R (2006) Open sensor web architecture: core services. In: Proc. 4th ICISIP, pp 98–103

Cohen M, Stathis K (2001) Strategic change stemming from E-commerce: implications of multi-agent systems in the supply chain. J Strateg Change 10:139–149. https://doi.org/10.1002/jsc.524
Cote C, Brosseau Y, Létourneau D, Raievsky C, Michau F (2006) Robotic software integration using MARIE. Int J Adv Robot Syst 3(1):55–60
DataArt Solutions, DeviceHive (2017) Open source IoT data platform with the wide range of integration options. https://devicehive.com/. Accessed 20 Nov 2017
de Bruijn O, Stathis K (2003) Socio-cognitive grids: the net as a universal human resource. Socio-cognitive grids: the net as a universal human resource. In: Kameas A, Streitz N (eds) Proceedings of the conference of "tales of the disappearing computer", CTI Press, Santorini, pp 211–218
Dipsis N, Stathis K (2009) Internalizing unknown objects by means of perception and communication in multi-agent systems. In: Proceedings of the 5th international conference on intelligent environments (IE'09), Barcelona, Spain, IOS Press, pp 499–509
Dipsis N, Stathis K (2010) EVATAR—a prototyping middleware embodying virtual agents to autonomous robots. In: Augusto JC, Corchado JM, Novais P, Analide C (eds) Ambient intelligence and future trends-international symposium on ambient intelligence (ISAm I 2010). Advances in soft computing, vol 72. Springer, Berlin
Dipsis N, Stathis K (2012) Ubiquitous agents for ambient ecologies. Pervasive Mob Comput 8(4):562–574
DropWizard (2017). https://www.dropwizard.io/1.3.5/docs/. Accessed 30 Sep 2017
Eisenhauer M, Rosengren P, Antolin P (2009) A development platform for integrating wireless devices and sensors into ambient intelligence systems. In: Proceedings of SECON 2009, communication society conference on IEEE, Rome
JPA Entities (2018) Java persistence API–entity. https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html. Accessed 2 Mar 2018
Epley N, Waytz A, Cacioppo JT (2007) On seeing human: a three-factor theory of anthropomorphism. Psychol Rev 114:864–886
Evensen PL, Meling H (2009) A service oriented middleware with sensor virtualization and self-configuration. In: Proc. int. conf. intelligent sensors, sensor networks and information processing (ISSNIP)
Favela J, Rodriguez M, Preciado A, Gonzalez V (2004) Integrating context-aware public displays into a mobile hospital information system. IEEE Trans Inf Technol Biomed 8(3):279–286
Fielding RT (2000) Architectural styles and design of network-based software architectures. www.ics.uci.edu/fielding/pubs/dissertation/top.htm
Fok CL, Roman GC, Lu C (2009) Agilla: a mobile agent middleware for self-adaptive wireless sensor networks. ACM Trans Auton Adapt Syst (TAAS) 4(3):16
Gamma E (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading
Gerkey B, Vaughan R, Howard A (2003) The player/stage project: tools for multi-robot and distributed sensor systems. In: Proceedings of the 11th int. conf. advanced robot, pp 317–323
Giaffreda R (2013) iCore: a cognitive management framework for the internet of things. The future internet. Lecture notes in computer science, vol 7858. Springer, Berlin
Google Assistant (2017). https://assistant.google.com. Accessed 30 Sep 2017
Google Cloud IoT (2018). https://cloud.google.com/solutions/iot/. Accessed 28 Sep 2018
Google DeepMind (2018). https://deepmind.com/. Accessed 2 Mar 2018
Google Fit (2018). https://developers.google.com/fit/. Accessed 27 Sep 2018
Görgü L, Kroon B, O'Grady MJ et al (2018) Sensor discovery in ambient IoT ecosystems. J Ambient Intell Hum Comput 9:447. https://doi.org/10.1007/s12652-017-0623-0
Grace P, Blair GS, Samuel S (2005) A reflective framework for discovery and interaction in heterogeneous mobile environments. ACM SIGMOBILE Mob Comput Commun Rev 9(1):2
Guinard D, Trifa V, Karnouskos S, Spiess P, Savio D (2010) Interacting with the SOA-based internet of things: discovery, query, selection, and on-demand provisioning of web services. IEEE Trans Serv Comput 3(3):223–235
Heim S (2007) The resonant interface HCI foundations for interaction design. Addison Wesley, Boston
Hibernate (2018) Hibernate relational mapping—ORM. http://hibernate.org/orm/. Accessed 19 Aug 2018
IBM Watson IoT (2017) IBM watson internet of things platform (IoT). http://www.ibm.com/internet-of-things/. Accessed 29 Jan 2017
IFTTT (2019) If this then that. https://ifttt.com/. Accessed 7 May 2019
Jersey (2018) RESTful web services in Java. https://jersey.github.io/documentation/latest/index.html. Accessed 5 Apr 2018
Jetty (2018) Eclipse Jetty Web server and javax.servlet container. http://www.eclipse.org/jetty/. Accessed 15 Jun 2018
JSON (2017). https://www.json.org/. Accessed 30 Sep 2017
Kaa-IoT Technologies (2017) Kaa open-source IoT platform—IoT cloud platform the internet of things solutions and applications that set the standard. https://www.kaaproject.org/. Accessed 10 Nov 2017
Kakas A, Mancarella P, Sadri F, Stathis K, Toni F (2008) Computational logic foundations of KGP agents. J Artif Intell Res 33(1):285–348
Kang P, Borcea C, Xu G, Saxena A, Kremer U, Iftode L (2004) Smart messages: a distributed computing platform for networks of embedded systems. Comput J Special Focus Mob Pervasive Comput 47:475–494
Kim T, Choi S, Kim J (2007) Incorporation of a software robot and a mobile robot using a middle layer. IEEE Trans Syst Man Cybern Part C 37(6):1342–1348
Konker Labs (2017) Konker—your solutions connected in a fast and simple way. http://www.konkerlabs.com/. Accessed 11 Nov 2017
Kushwaha M, Amundson I, Koutsoukos X, Neema S, Sztipanovits J (2007) OASiS: A programming framework for service-oriented sensor networks. Communication systems software and middleware (COMSWARE 2007), pp 1–8
LinkSmart (2018). https://www.linksmart.eu/. Accessed 5 Apr 2018
Liu T, Martonosi M (2003) Impala: a middleware system for managing autonomic, parallel sensor systems. ACM SIGPLAN Notices 38(10):107–118
Mehmood F, Ullah I, Ahmad S et al (2019) Object detection mechanism based on deep learning algorithm using embedded IoT devices for smart home appliances control in CoT. J Ambient Intell Hum Comput. https://doi.org/10.1007/s12652-019-01272-8
Michal N, Artem K, Oleksiy K, Sergiy N, Michal S, Vagan T (2009) Challenges of middleware for the internet of things. automation control—theory and practice. InTech
NEST (2018) Nest Home Simulator. https://developers.nest.com/guides/home-simulator. Accessed 28 Nov 2018
Nest architecture (2019). https://developers.google.com/nest/guides/architecture. Accessed 10 Jul 2019
Nest Labs (2019) Home Automation Nest Labs. https://store.google.com/us/category/connected_home. Accessed 10 Jul 2019
Ngu A, Gutierrez M, Metsis V, Nepal S, Sheng Q (2016) IoT middleware: a survey on issues and enabling technologies. IEEE Internet Things J. https://doi.org/10.1109/jiot.2016.2615180
OAuth2 (2018) OAuth2 Protocol authorization. https://oauth.net/2/. Accessed 22 Sep 2018

O'Hare GMP, Muldoon C, O'Grady MJ, Collier RW, Murdoch O, Carr D (2012) Sensor web interaction. Int J Artif Intell Tools 21(02):1240006

Pfisterer D, Römer K, Bimschas D, Kleine O, Mietz R, Truong C, Hasemann H, Kröller A, Pagel M, Hauswirth M, Karnstedt M, Leggieri M, Passant A, Richardson R (2011) SPITFIRE: toward a semantic web of things. IEEE Commun Mag 49(11):40–48

Poncela A, Coslado F, García B et al (2018) Smart care home system: a platform for eAssistance. J Ambient Intell Hum Comput. https://doi.org/10.1007/s12652-018-0979-9

Raspberry Pi (2018). https://www.raspberrypi.org/. Accessed 5 Apr 2018

Redis (2019). In-memory data structure project. https://redis.io/. Accessed 07 May 2019

Nest API reference (2019). https://codelabs.developers.google.com/codelabs/wwn-api-quickstart/. Accessed 10 Jul 2019

Rezgui A, Eltoweissy M (2007) Service-oriented sensor–actuator networks: promises, challenges, and the road ahead. Comput Commun 30(13):2627–2648

Rouvoy R et al (2009) Middleware support for self-adaptation in ubiquitous and service-oriented environments. Software engineering for self-adaptive systems. Springer, New York, pp 164–182

Saffiotti A, Broxvall M, Gritti M, LeBlanc K, Lundh R, Rashid J, Seo B, Cho Y (2008) The PEIS-ecology project: vision and results. In: Proceedings of EEE/RSJ international conference on intelligent robots and systems, pp 2329–2335

SmartThings (2018) SmartThings developer documentation. http://docs.smartthings.com/. Accessed 2 Dec 2018

Spring Boot (2019) An application framework and inversion of control container for the Java platform. https://spring.io/projects/spring-boot. Accessed 7 May 2019

Stathis K (2000) A game-based architecture for developing interactive components in computational logic. J Funct Logic Progr (5)

Stathis K, Sergot M (1996) Games as a metaphor for interactive systems. In: Sasse MA, Cunningham RJ, Winder RL (eds) People and computers XI. Springer, London

Stathis K, Kakas AC, Lu W, Demetriou N, Endriss U, Bracciali A (2004) PROSOCS: a platform for programming software agents in computational logic. In: Proceedings of the 4th international symposium AT2AI-4—EMCSR 2004 Session M, pp 523–528

Sundmaeker H, Guillemin P, Friess P, Woelfflé S (2010) Vision and challenges for realising the Internet of things. Publications Office of the European Union, Luxembourg

TensorFlow (2016) TensorFlow—an open source software library for machine intelligence. https://www.tensorflow.org/. Accessed 2 Mar 2016

Terziyan V, Kaykova O, Zhovtobryukh D (2009) UbiRoad: semantic middleware for context-aware smart road environments. In: Proc. of fifth international conference on internet and web applications and services (ICIW), pp 295–302

TLS (2018) Transport layer security. https://www.gov.uk/government/publications/email-security-standards/transport-layer-security-tls. Accessed 5 Apr 2018

Witkowski M, Stathis K (2004) A dialectic architecture for computational autonomy. In: Nickles M, Rovatsos M, Weiss G (eds) Agents and computational autonomy. AUTONOMY. Lecture notes in computer science, vol 2969. Springer, Berlin

Zachariah T, Klugman M, Campbell B, Adkins J, Jackson N, Dutta P (2015) The internet of things has a gateway problem. In: Proceedings of the 16th international workshop on mobile computing systems and applications (HotMobile '15). ACM, New York, NY, USA, pp 27–32

ZigBee (2019) ZigBee wireless standard. https://www.zigbee.org/. Accessed 10 Jul 2019