Autonomous and self controlling smart objects for the future internet

Marco E. Pérez Hernández, Stephan Reiff-Marganiec
Department of Computer Science
University of Leicester, UK
Email: {meph1,srm13}@le.ac.uk

***Abstract*—Development of IoT applications relies heavily on middleware architectures. Multiple solutions have been proposed using service and agent paradigms, that consequently imposes an specific application development model. We observe a trend towards a data-feeder approach in which smart objects are simple data gatherers and senders. This approach reduce the autonomy of smart objects and brings privacy concerns about the data and service manipulation by remote applications and services. We propose a new approach for middleware and application development for smart objects based on web agents and services. Specifically, we integrate services as part of activities included in roles of a web agent allowing us to keep control of the service and data generated on-object, in contrast to delegating to remote applications or services. In addition, our P2P-inspired method for service discovery gives autonomy to smart objects since a central directory is not required to interact with others. We completed an initial implementation of our architecture and demonstrate its feasibility in a case study in the smart home domain.**

***Keywords*-IoT Middleware; Web Agents; P2P ; Service Discovery;**

I. Introduction

Since the early stages of the IoT paradigm, middleware has been identified as key enabler. It aims to facilitate the development of IoT applications and provide useful abstractions to IoT developers. As it is natural, IoT middleware is being built on top of successful existing approaches, technologies and concepts.

Two fundamental approaches in IoT Development are Service orientation and the Agent Paradigm. Service orientation has enabled the creation of IoT applications assembling data gathered from the physical environment while leveraging the almost unrestricted cloud computing power. In the same way, agents have also been a useful paradigm to endow autonomy and near-human features —e.g. reasoning, sensing, etc.— to IoT applications. Robust agent platforms have grown in controlled environments imposing particular models of integration with open environments such as the Internet. Consequently several Agent-based IoT middleware solutions have inherited the same approaches.

In regards to proposed IoT middleware, a common approach we observe is that they are useful in developing applications when the data control and "smartness" of the smart object resides outside it. This of course, ensures intensive use of powerful infrastructures but compromises on the ideal IoT requirements for privacy and autonomy. We think that another approach is possible. We seek for a balanced development of IoT applications, driven by a middleware that allows to use Smart Object (SO) capabilities to keep control of data and services offered, while exploiting remote services under the direction of the SO.

To achieve this, we see the need of using an open approach for agent and service integration for the design of IoT middleware architectures. It should allow transparent use of services and the Internet infrastructure while taking advantage of abstractions provided by the agent paradigm. It implies also the definition of new approaches for search and selection of the IoT and SO services.

The main contributions of this paper are: (1) a new approach to the use of web agents and services to develop IoT SO Applications. (2) A new approach to middleware architecture oriented to keep the control of the data generated and the services offered in the SO. (3) A new method for IoT service search and selection based in the Gnutella P2P protocol and using a map-reduce ranking-based selection.

The remaining is organized as follows: Firstly, we provide some background to understand our contributions. Secondly, we introduce our IoT middleware architecture and its components. Then, we explain the novel method for search and selection of services in the SO context. Later, we give details about our implementation. Next, we describe the case study which lead our initial implementation effort. We present then some of the related work, and finally we offer relevant conclusions based on our findings.

II. Background

A. Beyond Data Feeders

The Smart Objects are central entities of the IoT [1]. They are able to overcome the digital and physical frontiers by not only gathering data from their surroundings but also altering the existing situations and working together and autonomously with other Smart Objects.

Despite potential capabilities of Smart Objects, popular IoT middleware and Platforms still present smart things as mere data feeders. They systematically send raw data to powerful nodes, which are in charge of running the logic and use the data in multiple applications or services. These objects usually enclose platform-specific and basic functionality for interacting with the attached devices which is offered to third parties via pre-programmed interfaces.

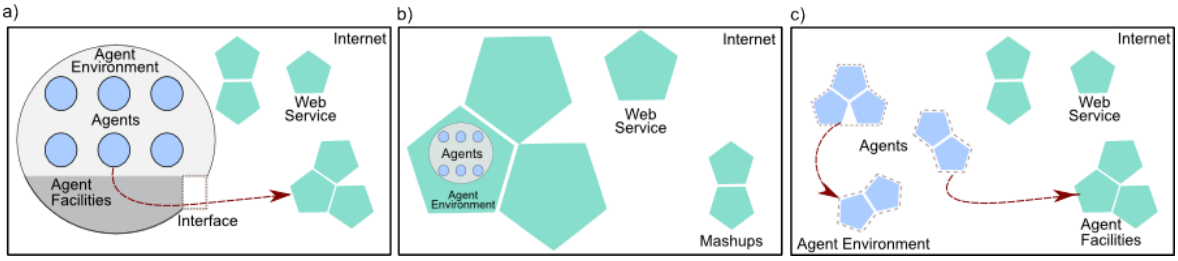

Figure 1. Approaches to Agent/Service Integration: a)Enclosed agent interface approach b) Wrapped agent service c) Open integration

One problem of this approach is the displacement of the data control away from the Smart Object. This raises concerns about privacy, since the practices for handling the data and the final receiver are unknown to the Smart Object (and its owner). Besides, the coupling of the Smart Object to a specific application or remote platform, constraint the true potential of connecting to others autonomously.

The lack of resources in the SO push design decisions towards a data-feeder model. Under the idea of leveraging in cloud resources and services, things become "silly" nodes. However, hardware platforms for SO are increasing resources everyday, and now there are devices with processing power even better than some multi-purpose computers. This makes possible, an "intelligent leverage" in which core and minimal processes are held in the SO, while the use of cloud resources is lead by the SO Accordingly, IoT middleware can drive adaptation of SO processes to its resources and seek for a proper formula for cloud leveraging. For this, endowing SO with agent characteristics is a suitable technique that should coexist with service-orientation.

B. Agent and Services Integration

Service-Orientation and Agent Paradigms have proven to be powerful and useful abstractions in multiple contexts. In the IoT middleware arena there has been a variety of works using these models in the solution of existing challenges in the field. Detailed reviews of existing solutions can be found in [2] and [3], and we consider pertinent work in section VII.

The use of these popular paradigms is characterized by an imaginary frontier between one world and the other. Web services, the key element in Service-oriented middleware are usually seen as external to the agent environments. As a result, integration efforts have been oriented to enable the use of deployed services through interfaces provided by the agent platform, as shown in figure 1 (a). We call it the *enclosed agent interface* approach. Alternatively, agent platforms, MAS —Multi Agent Systems— or agent processes are wrapped as services that can be composed or orchestrated into more complex applications, as shown in figure 1(b). We call this the *wrapped agent service* approach. Since agent environments are usually provided by heavy platforms, a problem here is that SOs require a more powerful node for hosting the agent platform facilities.

A more service-oriented approach, yet less exploited, is to expose the agents as a collection of services and so they use the Internet as the agent environment, as shown in figure 1 (c). We call this *open integration* approach. Services define the behavior of the agent, they allow to benefit from the resources and infrastructure already existing on the Internet. It gives agents more independence to consume services since they are no centrally controlled by an isolated platform.

C. Challenges for Smart Object Middleware

We consider the open integration, the most suitable when designing middleware for Smart Objects. The closer integration between agent and service oriented paradigms and technologies boosts the autonomy of the SO and enable easy access to Internet resources while keeping control of data generated. However, there is a lack of solutions following this approach. Possibly, due to the fact that most popular and robust agent platforms drive naturally towards an *enclosed agent interface* approach and there being also a lack of robust web agent platforms.

Given the advantages of the mentioned approach, our first challenge is to define a new approach to an IoT middleware architecture, based on web agents using the Internet as environment and platform. IoT middleware requirements have been widely identified in previous works such as [4] and [3]. With the approach chosen, multiple requirements are covered transparently by the agent and service paradigms, however there are also multiple derived challenges. In this paper, we particularly concentrate on providing Smart Object application development based on web-agents and services and a method for discovery and location of SO services, using P2P protocols and agents over the Internet.

III. Proposed Approach

A. Middleware Architecture

The main drivers of the architecture are to exploit on-object capabilities while keeping control on the object and maintaining autonomy. Therefore, middleware provides minimal agent facilities on-object and gives access to delegable facilities consumed from cloud services or also hosted in the SO, according to available local resources. This is possible, due to a flexible and lightweight layer of agent facilities.

For autonomy, our solution keeps control of the use of cloud services, relying on a SO Controller for that purpose.

A multi-layered view of the proposed architecture is shown in Figure 2. Components in yellow are required to be in the SO, while components in blue are delegable to cloud services. The components are described below:

1) Communication Facilities: This is a basic component required and present in most agent-based middleware solutions. It is based on application protocols over IP and UDP. It incorporates mostly, web protocols as they are the mean for service consumption. More capable devices could run on HTTP while others on protocols such as COAP. Agent/Agent communication and Agent/Service communication is performed using these facilities.

2) Service Assembler: This is a new component introduced in our architecture. It is responsible for allocating the services required by the Smart Object and assembling them, according to the SO Controller and the role activities. The SO Controller drives the use of architectural modules such as the Reasoning Engine and the Knowledge base; while each defined role, indicates to Service Assembler th way of assemble the SO-specific services. This component is the basis for allowing adaptation, since it allows for different configurations of on-object and cloud services, according to SO resources.
The Service Assembler includes methods to search, select and use the needed services. It also updates the SO address repository and searches for services, available in other SOs, following the method introduced in section IV.

3) Lightweight Agent Platform: It offers the minimum agent facilities, enabling use and adaptation of additional elements in the upper layers according to real needs and host resources. According to the Smart Object infrastructure, agent facilities are delagable, except for:

- Basic agent abstractions such as agent, activities, goals and roles
- Services for management of the agent life cycle
- An agent communication protocol

The agent communication protocol relies on the supported *communication facilities* avoiding overloading, e.g. by requiring separate infrastructures for communication with other agents and for communication with the "external world". It also allows transparent communication of the agents, regardless of their location, and uses the existing network addressing schemes. Unlike FIPA-compliant platforms, a centralized Directory Facilitator is not required at this level, since the *Service Assembler* identifies agent, services and addresses to contact. We rely on an existing web-agent platform to provide most of this functionality.

4) Device Management Facilities: This is again a common component required and proposed in other Smart Object middleware. In our proposal, it provides interfaces for configuration and use of the attached sensors and actuators. Methods for adding, removing, restarting, reading, activating and deactivating these devices are invoked by the SO Controller. These interfaces are standard, hiding specific functions provided by Hardware manufacturers. This component call Operating System commands to adjust sensor and actuators according to Controller instructions.

5) Smart Object Controller: It is the main component, responsible for keeping the SO up and running according to goals specified and stored in the knowledge base. It uses the Agent Facilities to send and receive messages to/from other SO components and to/from local or remote agents. It performs the cycle depicted in Algorithm 1 which is an adaptation of the abstract agent loop proposed in [5].

Algorithm 1 High-level SO Controller Loop

```
while pendingGoals do
  if setUpRequired then
    checkDeviceChanges(devices);
    checkCapabilityChanges(localCapabilities);
    updateKnowledgeBase();
    performKBMaintenance();
  end if
  observe(propertiesOfInterest);
  updateKnowledgeBase();
  activities = reason(roles, observations);
  for all activityinactivities do
    serviceContract =
      searchServiceHosts(actionToExecute);
    callService(serviceContract, serviceArgs);
  end for
end while
```

6) Reasoning Engine: It enables the SO to develop beyond a purely reactive behavior. It performs the decision-making process, choosing activities and services to use, selecting the providers and scheduling its execution. Although there might be multiple implementations, in a basic approach, this component receives from the controller the set of rules defined for the roles. Then it evaluates these rules according to pre-existing and performed observations and finally it obtains the set of activities to invoke. Since reasoning processes are processing-intensive, this component works mainly as a proxy delegating the most of the decision-making process to cloud services.

7) Capability Loader: It is responsible for on-demand loading of the instances required to run the services demanded either by the local controller or by other

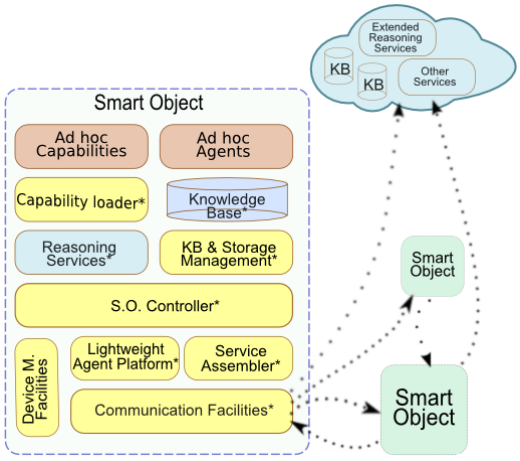

Figure 2. Multi-layer view of a Smart Object middleware architecture

SOs. We define execution instances which contain the information relevant to dynamically —at runtime— instantiate or reuse the stateless objects having service implementation. The instantiation process is performed with three elements stored in the KB: the service contracts, the service parameters data and the location of the capability enclosing the service. The service contract and the known locations are defined as part of the capability definition. SOs store known capabilities they can use either locally or remotely. For each one, one or many URIs can be specified. Capabilities are stored in the KB when the controller is set up. The specific data to use when invoking the service comes from the data gathered from the SO or it could be pre-defined by the SO Developer. In either case it is linked to actions compounding the activities associated to each role.

8) Knowledge Base (KB): A distributed repository, based on semi-structured data storages, that keeps the domain model including, goals, roles, activities, known service contracts, devices and properties of interest, among others. In order to take advantage of existing standards and developments, the initial data model is inspired by the agents domain model [5] and OGC's Sensor Things data model [6]. This repository also stores the data resulting from SO observations and those received from third-objects.

A design decision was to follow a document-oriented approach, that is considered suitable for a number of reasons: Primarily because of the heterogeneous nature of the SO and the data generated it makes no sense to force a single fixed model; this approach allows for flexibility of the structure even at runtime. Secondly, the amount of data generated by the SO imposes strong scalability requirements that are covered by these storage solutions.

One of the advantage of our approach is to have control over generated data and services offered. It is critical for generating trust in the SO applications that will consequently

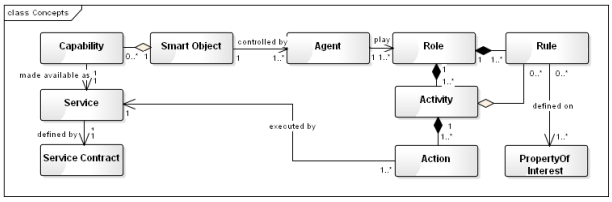

Figure 3. Main concepts and relations for SO programming

boost the propagation of the IoT paradigm. We ensure control through determination of *access policies* for the KB, mainly applied on the data generated for the properties-of-interest and the offered services. We define three levels of access: *private* —only available for the SO—, *public* — available for any object or application— and *specific* — allows to define a set of authorized URL.

9) KB and Storage Management: It provides access to the KB to the rest of the components. It encloses CRUD methods altering the contents and the structure of the KB. According to a pre-defined configuration based on the SO resources, it ensures a periodic replication and cleanup of the KB. This is the only component to access the KB, when required by the Controller.

B. Programming model

The middleware supports the developer in programming the SO, by enabling the definition of goals, roles, activities and rules following a semi-declarative model. Where definition of roles, activities and actions is declarative, although internal service programming do follow the control flow.

Figure 3 shows the main concepts and relations of the model. Each SO can have one or multiple roles. Each role includes a set of conditions and activities to execute when the conditions are met. Each activity has at least one action, which is associated to one service and the corresponding parameters to use in its execution. Every SO has a set of know capabilities, which are composed by a set of services. So, the service is the link between the capability and the action. Multiple agents and services can be added on top of the architecture, according to the SO resources.

IV. Search, Selection and Use of SO Services

Activities are categorized according to the actions included. Similarly, services use the same category model; e.g. an "air-related" category, is shared by activities and services related to 'sense chemical properties' or 'measure pollution level'. When the SO Controller needs a particular service it invokes the Service Assembler, which aims to locate the most suitable service for the SO, based on the known service providers defined in the KB and the role activities.

A. Selection based in Map-Reduce

In a IoT scenario, multiple Smart Objects can provide common capabilities. Additionally, the service location is

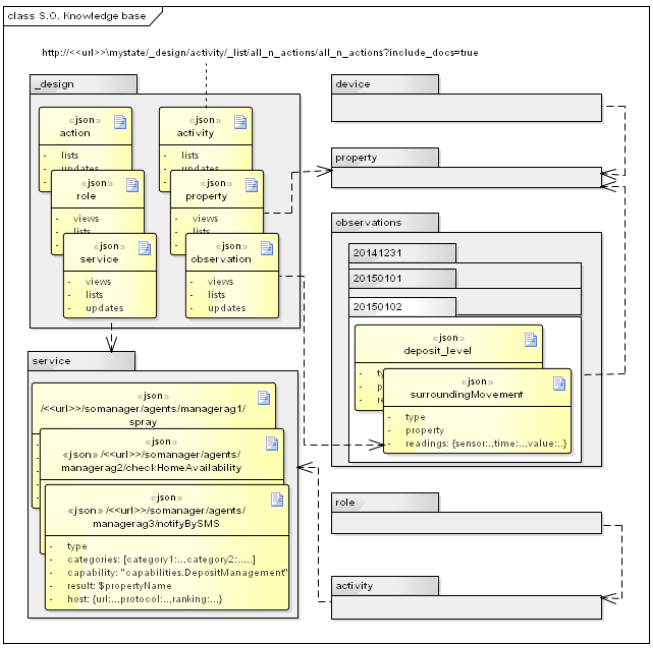

Figure 4. SO Knowledge base fragment

stored in distributed repositories —SO Knowledge bases— that have to be analyzed in order to obtain the demanded service. Performing an efficient search and selection over huge data sets becomes a key aspect. The MapReduce programming model [7], is a simple but efficient way to search in data repositories with these characteristics. It requires the definition of the map and reduce functions, that we approached as described below:

1) **Map phase** The search is based in the Service Contract stored in the KB; an example is shown in Figure 5. When the SO is executing an activity, it searches for services to carry out activity's actions. If the service is known, it performs a simple search with the service contract as filter. However, more extensive models, for example including *soft* and *hard* requirements on non-functional properties as proposed in [8], are also possible. The map function should emit one intermediate value for all the hosts (URI) where the demanded service contract is present —according to KB.
2) **Reduce phase** Since there could be multiple implementations of a single service, the selection of the right services is based on further action attributes; e.g. the last date the service was successfully used, or a combination of QoS parameters. The proposed solution does not enforce a particular selection criteria, on the contrary it is up to the SO Developer to define it. The reduce function, simply ranks the results of the map function using the selected criteria and returns the location of the demanded service.


```
{
  "_id": "service/127.0.0.1:8080/somanager/agents/managerag3/notifyBySMS",
  "_rev": "8-6507733c216940f8d869e88167c1164a",
  "type": "service",
  "categories": [
      "alert-management"
  ],
  "name": "notifyBySMS",
  "args": {
      "target": "java.lang.String",
      "message": "java.lang.String"
  },
  "result": "java.lang.String",
  "capability": "capabilities.AlertManagement",
  "local": "true",
  "host": {
      "url": "http://127.0.0.1:8080/somanager/agents/managerag3/",
      "protocol": "http",
      "ranking": "1",
      "lastSuccess": "2015-01-16T22:32:42.943"
  }
}
```


Figure 5. JSON Service Contract example

B. P2P inspired service discovery

If the service is unknown by the SO, the strategy is to find providers to ask for the demanded service. Initially, providers are searched for using the category and working under the assumption that providers, having capabilities in a category, are more likely to know other hosts in the same category. The category matching can be syntactic, however, the reasoning engine could be extended to a semantic mapping based on parametrized rules. If there are no hosts under the same category, arbitrary known hosts are taken.

We use some of the primitives from the Gnutella protocols [9], such as query and query-hint, for conducting the search. It allowed us to define a fully decentralized model not requiring super nodes. A *query* method performs the search described in section IV-A, locally and then asking other SOs.

Having the addresses (URI) of the identified providers, the controller makes a *query* call to the identified SO through the agent protocol and communication facilities. When service is found, the SO replies to the *query* call with the address of the provider. At the end, the requester's KB is updated to include the new provider for the service.

When replying to *query* calls, the SO also checks the access policy configured for the known services. If services are public, they can be included in every *query-hint*, otherwise they are only included if the requester object has the access granted by the provider of the service (it could be granted through a subscription process). Service providers are the only ones entitled to modify these restrictions.

V. Implementation

A prototype has been developed following an incremental and iterative approach.We concluded an initial release lead by the case study explained in section VI. It was focused on the core functionality of the architecture modules marked with a star (*) in the Figure 2. These modules were identified as mandatory for the case study. The implementation is based on the Java Platform, the *EVE* Agent Platform and *CouchDB* as distributed repository.

EVE [10], is a web-based agent platform that works on top of the Jetty Web Server. Jetty is a "small, fast, embeddable web server and servlet container" [11]. EVE promotes a model in which agents reside in an open environment, it does not rely in a central directory facilitator unlike most FIPA implementations. The framework is also lighter than other platforms since it offers only an essential agent model focused on communication, task-scheduling and agent memory management. These are captive features since it gives us flexibility to enhance the platform using our own agent/service search and selection method.

We extended the EVE platform by introducing a base agent with pre-defined features and actions common to all agents (e.g. loading the KB reference). We also included the implementation of the agent domain model with the elements mentioned in section III-A3.

CouchDB implements a document oriented repository using JSON documents. It allows for the definition of map-reduce functions in javascript and offer a REST interface for operations over the structure and the contents. Although the EVE platform offers a simple key/value storage interface to *CouchDB*, it is restricted. So we had to implement a custom client to enable more flexibility of the model for storing the status of the SO. The initial design of the agent's KB is presented in Figure 4. The documents in the *design* folder contain the map-reduce views, lists and onupdate triggers, that allow us to get the KB data in a desired format, regardless of a particular structure.

For the inter-smart object communication, we rely on EVE capabilities. It wraps agent messages as JSON-RPCs over the offered transports. We worked with HTTP since it seemed to be the most robust choice[1]. The Service Assembler deals with service-oriented communication. It allows the use of cloud services by the SO components (e.g. KB, reasoning, etc.) and the role activities. We assume cloud services offer a REST interface and we implement a client supporting the main HTTP primitives. Both, agent communication platform and Service Assembler use the same HTTP transport.

VI. Evaluation

We describe now the case study, that led our initial implementation. It is based in a smart home scenario, where a Smart Air Freshener (SAF) cooperates with a Cleaning Cupboard (SCC) and with the Smart Home Controller (SHC). Goals for every participant are present in Table I. For this case, we focused on the SAF's behavior and goals. Mdidleware allows to enhance what SAF can do —considering its hardware or knowledge restrictions— by locating and enabling the use of others' capabilities available as services. For SAF we use a Raspberry Pi 1 Model B, and for SCC a Model B+, for SHC a Laptop with a Core i5.

[1]Our proposal does not fundamentally depend on HTTP, so if desired other transport protocols could be employed.

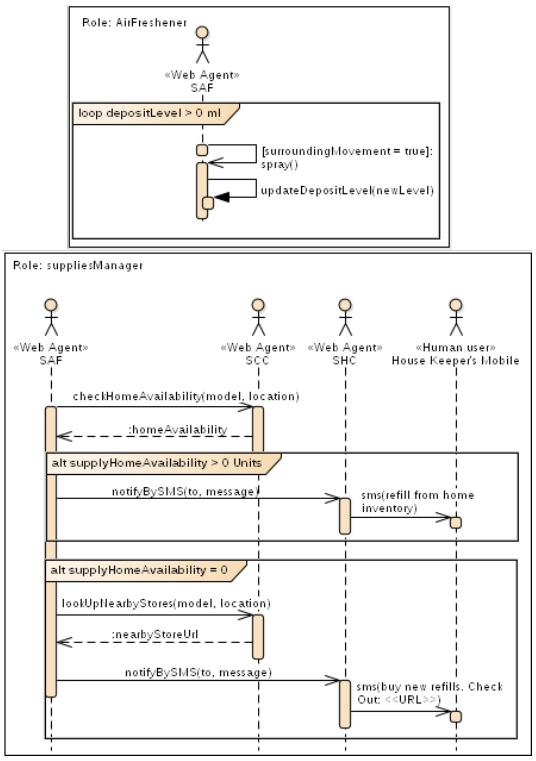

Figure 6. Case Study: SAF's roles and interactions

	Roles	Goal	Observable Properties	Condition	Relevant Actions
SAF	airFreshener	keep air fresh	surroundingMovement	$= true$	spray
	suppliesManager	keep freshener deposit full	depositLevel	$> 0\ ml$	
				$\leq 1\ ml$	checkHomeAvailability
			knownSuppliesAvailability	$> 0\ units$	notifyBySMS
				$= 0\ units$	lookUpNearbyStores
					notifyBySMS
SCC	cleaningSuppliesManager	keep control of inventory of cleaning products in home	incomingRFID	$\neq null$	addProductToInventory
			outgoingRFID	$\neq null$	removeProductFromInventory
SHC	homeManager	Notify about status of home objects	airFreshenerStatus	$= failed$	notifyBySMS
			cleaningCupboardStatus	$= failed$	notifyByEmail

Table I
Case Study: Participants and Roles

1) Smart Object Programming: We installed the first build of the middleware implementation. Middleware ensures that every object runs a web agent with an URL (i.e. IP + port + webcontext + agentId). SAF was programmed with two roles: *airFreshener* and *suppliesManager*, defined as JSON documents in SAF's KB. A sequence diagram of SAF's agent interactions per each role is shown in Figure 6. The first role ensures that fragrance is sprayed when surrounding movement is detected. The second role, observes the fragrance deposit to, proactively, ask for a refill when the specified threshold is reached. To achieve the goals, SAF discovers and uses services in SCC and SHC. Movement was simulated randomly and for the deposit level observation, we assumed 1 spray spent approximately 0.1 ml of fragrance, and a refill is 250 ml. So, an environment property in the KB was updated with every spray.

2) Service Search, Selection and Use: The three participants are part of an overlay network and have different knowledge about others' services as shown in Table II. When SAF joins the network, it just knows its private services and the pre-defined service *notifyByEmail* provided by SHC. SAF does not consume this service but uses it to find the other services it requires as shown in Figure 6. The scenario starts with SAF spraying until reaching the deposit threshold when the *suppliesManager* role is triggered. Then, the agent sends a *query* request to SHC asking for the *checkHomeAvailability* service. SHC receives the *query* request and looks for the services locally in its Knowledge Base. Service contracts initially included in the KB of every participant are also showed in Table II. So, SHC initially knows *notifyByEmail*, *notifyBySMS* and various *checkHomeAvailability* provided by SCC and by another out-of-home object. Some of these services are unavailable to SAF's. SHC keeps control of services to share with SAF and replies with the execution details of the best ranked *checkHomeAvailability* service. In this case, ranking is based in the location provided by SAF, it also consider access rights according to SHC's policy. It means, that private services are excluded from the recommendations.

SAF stores the execution details of the received service in its KB and call directly the provider. Service are consumed asynchronously and when a service provider is unknown or unavailable, there is a pre-configured number of attempts. If provider is available, SAF calls the SHC's *notifyBySMS* service, so the house keeper is notified to replace SAF's refill. After it, SAF goes back to the *airFreshener* role until the refill is fully consumed. Then, it asks again to SHC, but this time it replies with 0 available refills. At this point, SAF decides how to deal with this situation, based on the *suppliesManager* role rules. It now searches for the service *lookUpSupplyInMarket*, sending a *query* message to known providers. When SHC receives the *query* and after checking that it has nothing to reply to SAF, it forwards the *query* to SCC. SCC then reply back to SHC and this to SAF. Note that SAF sends also the query directly to SCC since it already knows it. However, when registering a new service provider in the KB, it first check that it is not already there. Knowing providers for the services used in the role activities, SAF searches for new providers only if known ones are unavailable. The services discovered by each participant are also included in the Table II.

The implemented middleware allowed us to program the SO in a high-level declarative way. Once capabilities are implemented, these can be easy shared by other objects and included as actions in multiple role definitions. Both actuating and sensing capabilities could be exposed as services, while the access policies provide an intuitive and simple way of control what to share with others. Regarding the service search method, the map-reduce query performs well when there are various services available, however our method had to be adapted to deal with cases when just few service are found. JSON-RPC over HTTP worked effectively, however other choices will be explored since the message could become heavy for complex service contract searching.

				SAF		SHC		SCC
Service	**Arguments**	**Provider**	**Access Policy**	**KB's Initial Stage**	**Discovered Service**	**KB's Initial Stage**	**Discovered Service**	**KB's Initial Stage**
spray	None	SAF	private	X				
updateDepositLevel	newLevel	SAF	private	X				
notifyByEmail	to, subject, message	SHC	specific (home)	X		X		
notifyBySMS	to, message	SHC	specific (home)		X	X		
lookUpNearbyStores	model, location	SCC	public		X		X	X
checkHomeAvailability	model, location	SCC	specific (home)		X			X
checkHomeAvailability	model, location	External Object	public			X		
checkHomeAvailability	model, location	External Object	specific (no SAF)			X		
checkHomeAvailability	model, location	SHC	private			X		

Table II
CASE STUDY: S.O. AGENT'S KNOWLEDGE BASE STAGES

VII. RELATED WORK

Various approaches to IoT middleware have adopted agent and service-oriented paradigms. The work in [12] presents a vision of the IoT middleware where agents manage IoT resources. It elaborates on the concepts of repository of roles and reusable behaviors to allow flexibility and adaptation in runtime. The agents reside in a semi-closed and controlled platform, although interoperability between platforms is offered. It also proposes the use of P2P approaches, but as complement to central directory facilities, in which agent communication depends on.

[13], [14] and [15] present a CSO Architecture based on the popular JADE-Platform and a Discovery Service providing a REST interface for integration with the agent middleware. The proposed architecture follows a master/slave model with a powerful node hosting the agent platform. It is based on events and tasks, while it offers flexibility by the use of adapters for communication, device and KB management. Agent communication relies in ACL and the central JADE Directory Facilitator.

In [16], the authors present an Agent Service Platform, mainly addressing the management of heterogeneity in devices. Resource-constrained devices based on Arduino, delegate control to agents placed outside them, but able to communicate with other agents and devices through a message bus. Runtime adaptation is possible using Portable-Service abstractions and dependency-injection patterns. Agents in this proposal are located away from the object they represent and depend on an external agent manager.

Another lightweight approach is presented in [17]. They aim to distribute the processing of data among distributed IoT devices. A resource directory is the basis for localization of services and communication relies on HTTP and COAP. The rely in a gateway to interact with "external" services.

Additional middleware solutions for Smart Environments and Smart Objects are surveyed and compared in [3].

The works reviewed in this section present the control of the SO and its data, as a task which is fully or partially carried out away from the SO, in contrast to our approach. Regarding the SO autonomy, it is is constrained by the dependency on either a (central) directory or a message bus. In addition, for agent-service integration the most popular approach is the enclosed agent interface one, while the open integration is not clearly used.

VIII. Conclusions and Future Work

Multiple approaches have been used to agent/service integration for development of IoT middleware. We identified three basic approaches: (1) enclosed agent interface, (2) wrapped agent service and (3) open integration. We consider the latter one the most suitable for a SO middleware in order to boost autonomy and keeping control of data and services.

We presented a novel approach for IoT middleware architecture, following an *open integration* approach. Our architecture is based in lightweight agent facilities and services. It allows to perform essential processes in the SO, while using external services for process-intensive activities such as reasoning. This approach allows us to exploit SO resources and delegate work to third-party providers when required. We establish policies for gathered data and services in order to keep control with the SO.

The challenge of using an *open integration* approach implied the use of non-traditional directory facilities trying to leverage the existing Internet infrastructure. We proposed a method for service search, selection and use based on map-reduce functions and inspired by Gnutella protocols for discovery of service providers.

We demonstrated the feasibility of our approach by implementing a middleware and IoT application using it, solving a problem in the Smart Home domain. A Smart Air Freshener, was programmed with high-level roles and activities, can rely on other SO's capabilities to achieve its own goals. It does not need a central directory to find other SOs.

We are extending the initial implementation of our architecture with the aim of establishing a testbed, that allow us to improve our design. We will also add to the case study seeking to implement other participant SO and expose SO's capabilities with other purposes. e.g. SAF movement detection can be used by the SHC for alarm triggering.

References

[1] G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy, "Smart objects as building blocks for the internet of things," *Internet Computing, IEEE*, vol. 14, no. 1, pp. 44–51, 2010.

[2] F. C. Delicato, P. F. Pires, and T. Batista, *Middleware Solutions for the Internet of Things*. Springer, 2013.

[3] G. Fortino, A. Guerrieri, W. Russo, and C. Savaglio, "Middlewares for smart objects and smart environments: overview and comparison," in *Internet of Things Based on Smart Objects*. Springer, 2014, pp. 1–27.

[4] S. Bandyopadhyay, M. Sengupta, S. Maiti, and S. Dutta, "Role of middleware for internet of things: A study," *International Journal of Computer Science & Engineering Survey*, vol. 2, no. 3, pp. 94–105, Aug. 2011. [Online]. Available: http://www.airccse.org/journal/ijcses/papers/0811cses07.pdf

[5] L. Sterling and K. Taveter, *The Art of Agent-Oriented Modeling*. The MIT Press, 2009, vol. 47, no. 06.

[6] OGC, "SensorThings Data Model." [Online]. Available: http://ogc-iot.github.io/ogc-iot-api/datamodel.html

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] S. Reiff-Marganiec, H. Q. Yu, and M. Tilly, "Service selection based on non-functional properties," in *Service-Oriented Computing-ICSOC 2007 Workshops*. Springer, 2009, pp. 128–138.

[9] I. Ivkovic, "Improving gnutella protocol: Protocol analysis and research proposals," *Prize-Winning Paper for LimeWire Gnutella Research Contest*, 2001.

[10] J. D. Jong, L. Stellingwerff, and G. E. Pazienza, "Eve: A Novel Open-Source Web-Based Agent Platform," *2013 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 1537–1541, Oct. 2013. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm? arnumber=6722018

[11] "Jetty - Servlet Engine and Http Server." [Online]. Available: http://www.eclipse.org/jetty/

[12] A. Katasonov, O. Kaykova, O. Khriyenko, S. Nikitin, and V. Terziyan, "Smart Semantic Middleware for the Internet of Things," in *Fifth International Conference on Informatics in Control, Automation and Robotics,*, 2008, pp. 169–178.

[13] G. Fortino, A. Guerrieri, and W. Russo, "Agent-oriented smart objects development," in *Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on*. IEEE, 2012, pp. 907–912.

[14] G. Fortino, A. Guerrieri, M. Lacopo, M. Lucia, and W. Russo, "An agent-based middleware for cooperating smart objects," in *Highlights on Practical Applications of Agents and Multi-Agent Systems*. Springer, 2013, pp. 387–398.

[15] G. Fortino, M. Lackovic, W. Russo, and P. Trunfio, "A discovery service for smart objects over an agent-based middleware," in *Internet and Distributed Computing Systems*. Springer, 2013, pp. 281–293.

[16] E. Jung, I. Cho, and S. M. Kang, "iotSilo: The Agent Service Platform Supporting Dynamic Behavior Assembly for Resolving the Heterogeneity of IoT," *International Journal of Distributed Sensor Networks*, vol. 2014, pp. 1–11, 2014. [Online]. Available: http://www.hindawi.com/journals/ijdsn/2014/608972/

[17] T. Leppnen and J. Riekki, "A lightweight agent-based architecture for the Internet of Things."