ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Logic Programming as a Service in Multi-Agent Systems for the Internet of Things

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Logic Programming as a Service in Multi-Agent Systems for the Internet of Things / Roberta Calegari; Enrico Denti; Stefano Mariani; Andrea Omicini. - In: INTERNATIONAL JOURNAL OF GRID AND UTILITY COMPUTING. - ISSN 1741-847X. - STAMPA. - 10:4(2019), pp. 344-360. [10.1504/IJGUC.2019.10022135]

Availability:
This version is available at: https://hdl.handle.net/11585/657394 since: 2019-08-12

Published:
DOI: http://doi.org/10.1504/IJGUC.2019.10022135

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

17 April 2024

This is the final peer-reviewed accepted manuscript of:

Calegari, R. (2019) Logic programming as a service in multi-agent systems for the internet of things. International Journal of Grid and Utility Computing. [Online] 10 (4), 344–360.

The final published version is available online at: http://dx.doi.org/10.1504/IJGUC.2019.10022135

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)

When citing, please refer to the published version.

Logic Programming as a Service in Multi-Agent Systems for the Internet of Things

Roberta Calegari

Dipartimento di Informatica – Scienza e Ingegneria (DISI),
Alma Mater Studiorum – Università di Bologna
E-mail: roberta.calegari@unibo.it

Enrico Denti

Dipartimento di Informatica – Scienza e Ingegneria (DISI),
Alma Mater Studiorum – Università di Bologna
E-mail: enrico.denti@unibo.it

Stefano Mariani

Dipartimento di Scienze e Metodi dell'Ingegneria (DISMI),
Università di Modena e Reggio Emilia
E-mail: stefano.mariani@unimore.it

Andrea Omicini

Dipartimento di Informatica – Scienza e Ingegneria (DISI),
Alma Mater Studiorum – Università di Bologna
E-mail: andrea.omicini@unibo.it

Abstract: The widespread diffusion of low-cost computing devices, along with improvements of cloud computing platforms, is paving the way towards a whole new set of opportunities for Internet of Things (IoT) applications and services. Varying degrees of intelligence are required for supporting adaptation and self-management: yet, they should be provided in a light-weight, easy to use and customise, highly-interoperable way. In this paper we explore *Logic Programming as a Service* (LPaaS) as a novel and promising re-interpretation of distributed logic programming in the IoT era. After introducing the reference context and motivating scenarios of LPaaS as an effective enabling technology for intelligent IoT, we define the LPaaS general architecture, and discuss two different prototype implementations—as a web service and as an agent in a multi-agent system (MAS), both built on top of the tuProlog system, which provides the required interoperability and customisation. We finally showcase the LPaaS potential through two case studies, designed as simple examples of the motivating scenarios.

Keywords: IoT, logic programming, multi-agent systems, pervasive computing, LPaaS, artificial intelligence, interoperability.

1 Introduction

The widespread adoption of the IoT perspective, according to which sensor networks, actuator devices, and computational resources seamlessly interact with people, is going to transform urban environments into *smart environments* – that is, physical environments enriched with sensing, actuating, communication, and computation skills – capable of acquiring and exploiting *contextual knowledge* to adapt to inhabitants' preferences, habits, and requirements [CFG+17].

People are thus continuously connected together and with their surrounding entities, in a *situation-aware* and *socially-aware* way: this is increasingly shaping a dense ecosystem, where ICT devices and people collaborate as they were a *superorganism* [Zam15]—in particular, for complex urban services, such as intelligent transportation systems, environmental sustainability, and participatory governance [Zam15, BFMZ13].

Similarly to living organisms, which are based on innervation as the fundamental "infrastructural" support for delivering their functionalities, socio-technical superorganisms require an adequate software infrastructure to enable and support the notion of smart environment. In particular, infrastructures should *(i)* be easily customisable, both statically and dynamically, so as to match the application needs; *(ii)* be possibly self-managing; *(iii)* govern components and applications interaction; *(iv) encapsulate intelligence* in forms that are suitable for their exploitation by the

applications. Here, connectivity and *interoperability* are just the basic, yet fundamental, bricks [ACF+17]. An essential infrastructural feature to build customised, variously-situated services and applications is to provide *distributed situated intelligence* on demand—that is, the ability to spread light-weight, context-aware, and effective intelligence chunks where and when needed, to *locally* satisfy the specific reasoning needs of the application at hand.

The aforementioned scenario opens up novel and challenging opportunities for *logic-based languages*, which are a natural choice as the *intelligence providers* in the IoT area [OZ04]. However, traditional logic programming (LP henceforth) techniques fall short in IoT scenarios, where the mobility/cloud ecosystem grounded upon the service-oriented computing paradigm delivers infrastructure, platform, and software *as a service* with the promise of ubiquitous information access and on-demand computation. This is why in this paper, as the natural evolution of distributed logic programming under a fresh IoT perspective, we propose *Logic Programming as a Service* (LPaaS) as an effective way to cope with the increasing demand for distributed situated intelligence coming from today's pervasive systems. In particular, with respect to [CDMO17], we explore more in depth the benefits of combining LPaaS and IoT, discuss a concrete architecture, and present two prototype implementations—as a Prolog-based RESTful web service and as a multi-agent system (MAS henceforth).

Accordingly, after outlining the application scenarios in the context of current research, Section 2 defines the LPaaS architecture, focussing on the most relevant features for the IoT domain. Then, Sections 3 and 4 present the two prototype implementations—namely, *LPaaS as a RESTful web service*, to emphasise how LPaaS can be effective in supporting one of the most typical IoT paradigms, and *LPaaS as an agent* in an agent society, to highlight its effectiveness in supporting and promoting distributed situated intelligence. As a concrete case, Section 5 shows how LPaaS can be implemented in JADE [JAD]. Benefits and open issues are discussed in Section 6, related work in Section 7, while conclusions are drawn in Section 8.

1.1 Application Scenarios

LPaaS moves from the idea of providing an engine for logic inference in the form of a *service* – library service, middleware service, network service, etc. – leveraging the power of LP resolution. Application scenarios are not limited to the IoT landscape: for instance, a Prolog-based web service with fuzzy search capabilities on a collection of XML documents representing publications is presented in [HLS05]—which could be easily built in LPaaS.

Other more complex scenarios could be devised i.e., in the field of health-care infrastructure, whose purpose is the continuous monitoring of patients affected by some disease. In [BSS10], pregnant women with gestational diabetes mellitus are assisted through an e-health infrastructure: patients are equipped with a body-area network to monitor blood pressure and glucose levels. Sensors are connected to the patients' smartphone, working as a hub to collect the data. Abductive agents perform reasoning on data to provide a diagnosis – a task that could be well-suited for LPaaS using abductive LP [KKT92] – contacting health-care professionals if necessary.

Another intriguing application for LPaaS is *on-demand reasoning* in sensor networks, which offers the possibility to locally inject chunks of situated intelligence. In fact, as discussed in [FGG+13], implementing real-time, power-efficient, distributed signal-processing algorithms on wireless nodes that are severely resource-limited and have to meet stringent requirements in terms of wearability (including battery duration) is still extremely challenging and complex. In such situations, LPaaS offers the possibility of exploiting a light-weight inference engine to perform data reasoning on demand in a light-weight, efficient, and decentralised way.

Other research works aim at making the next generation IoT smarter—among them, agent-oriented and event-based frameworks for the development of cooperating smart objects [LMMZ17, FGRS14, FGL+13]: they all share the idea of moving from connecting things to generating intelligence by linking things in the real world with information in the digital world.

2 The LPaaS Architecture

Moving from the above considerations, the LPaaS architecture is designed so that LP can act as a source of *distributed intelligence* for the IoT world, by providing an abstract view of LP inference engines in terms of *service*. This is because service-oriented architectures (SOA) promote *interoperability*, *encapsulation*, and *situatedness*, thus reducing the need for integration and coupling while promoting *context-awareness*.

On the one hand, in this architectural re-interpretation, LP can deal with local knowledge within physical devices, taking into account the domain specificity of each environment, and making it possible to reason effectively on data that are local to situated components. Furthermore, this allows in principle diverse, specific computational models to be tailored to the local needs of each situated component, by exploiting LP extensions explicitly aimed at pervasive systems such as *labelled variables systems* [CDDO16, CDO15].

On the other hand, since interoperability mandates for standards, LPaaS defines a standard *interface* for client applications and relies on standard *representation* formats (i.e., JSON [JSO]) and *interaction* protocols (i.e., REST over HTTP, or MQTT [MQT]), versatile enough to fit a wide variety of application needs.

Table 1 LPaaS Configurator Interface. `ConfigList` is `[IsStateful, IsStateless, IsDynamic]`.

`setConfiguration(+ConfigList)`
`getConfiguration(-ConfigList)`
`resetConfiguration()`
`setTheory(+Theory)`
`getTheory(-Theory)`
`setGoals(+GoalList)`
`getGoals(-GoalList)`

Along this line, each LP server node exposes its services concurrently to multiple clients, via interfaces. The inference engine is expected to implement the SLD resolution [Rob65]: as in classical LP, it is configured with a *theory* of axioms—its Knowledge Base (KB); unlike classical LP, however, it is also configured with a set of goals that the client can ask to be proven—the *admissible goals*. This approach captures the idea that intelligence is deployed to fit specific needs that correspond to specific possible queries (in a specific location), not to provide a general-purpose, non specific LP inferential engine as in classical console-based LP systems.

The LP service is initialised and configured on the server at deployment-time: once started, it can be used by any number of *Client* agents in its current configuration—that is, a Client can ask the LP service to prove any of the admissible goals based on the logic theory constituting the current KB. Also, the LP service can be dynamically re-configured at run-time whenever needed by a privileged agent with adequate rights—the *Configurator* agent. Overall, generally speaking, applications can access the service as either Clients or Configurators via the corresponding interfaces: the *Client Interface* exposes methods for service *observation* and *usage*, while the *Configurator Interface* provides for service *configuration*.

2.1 Configurator Interface

A Configurator (agent) interacts with the server via the Configurator Interface. Configurator methods are detailed in Table 1, with the standard Prolog notation for input/output arguments [DDC96]: they provide for defining the service configuration, its KB, and the list of *admissible goals*.

Moreover, two key features of the LP service are the possibility of managing *stateful* and/or *stateless* requests, and of choosing between *dynamic* or *static* knowledge bases.

Stateful requests are needed to make it possible for the service to mimic a classical logic programming engine with SLD resolution [Rob65], where clients can ask for any number of solutions, and – possibly, then – ask for other solutions one by one, iteratively. To this end, the server must keep track of the state of each individual client request, i.e., of each client resolution process. Since this might be not resource-effective, and given that some application scenarios might even not require this – i.e., a temperature sensor that always needs to deal with just its latest measurement –, stateless requests are provided as an alternative, to be used according to the specific application needs. There, no session state is maintained on the server side, so each request needs to contain all the required information—see Subsection 2.2 below.

As far as the logic theory is concerned, a static KB is immutable (from the Client viewpoint), whereas a dynamic KB can evolve during the service lifetime by means of *assertion* and *retraction* of logic clauses: while the model is designed thinking about assertion/retraction of facts, it technically works also for clauses. This implies that clauses in a dynamic KB have a lifetime, too, since they are asserted and retracted as they represent mutable knowledge about the world—as in the case, for instance, of a clause representing the current temperature in a room.

It is worth noting that the LP service can be simultaneously stateful and stateless, as it can manage multiple kinds of request concurrently; the knowledge base, instead, can only be either static or dynamic—which is why `ConfigList` in Table 1 has just three parameters.

2.2 Client Interface

In the LPaaS Client Service Architecture the server provides clients with the *inference service* via the Client Interface detailed in Table 2 on page 4.

The LP service offers *observational methods* to provide configuration and contextual information about the service, and *usage methods* to query the service for logic computations and reasoning. Observational methods make it possible to ask the service for its configuration parameters (stateful/stateless, static/dynamic), the state of the KB, and its admissible goals. Usage predicates (i.e., logic predicates for usage methods) allow the service to be asked for solutions—one solution, n solutions, or all solutions. Usage predicates are slightly different in the two cases of stateless / stateful requests: in the former case, the `solve` operation is conceptually atomic and self-contained – it always has the `Goal` as argument – whereas in the latter case self-containment is not necessary, since the server keeps track of the client state and the goal can be set only once before the first `solve` request is issued.

The `reset` primitive simply resets the resolution process, with no need to reconfigure the service usage (i.e., to re-select the goal); instead, the `close` primitive actually ends the communication with the server, so the goal must be re-set to restart querying the server.

More details on the methods of the Client Interface are provided in Table 2 on page 4.

Table 2 LPaaS Client Interface.

	Stateless	Stateful
Static KB	getServiceConfiguration(-ConfigList)	
	getTheory(-Theory)	
	getGoals(-GoalList)	
	isGoal(+Goal)	
		setGoal(template(+Template))
		setGoal(index(+Index))
	solve(+Goal, -Solution)	solve(-Solution)
	solveN(+Goal, +NSol, -SolutionList)	solveN(+N, -SolutionList)
	solveAll(+Goal, -SolutionList)	solveAll(-SolutionList)
	solve(+Goal, -Solution, within(+Timeout))	solve(-Solution, within(+Timeout))
	solveN(+Goal, +NSol, -SolutionList, within(+Timeout))	solveN(+NSol, -SolutionList, within(+Timeout))
	solveAll(+Goal, -SolutionList, within(+Timeout))	solveAll(-SolutionList, within(+Timeout))
	solveAfter(+Goal, +AfterN, -Solution)	
	solveNAfter(+Goal, +AfterN, +NSol, -SolutionList)	
	solveAllAfter(+Goal, +AfterN, -SolutionList)	
		solve(-Solution, every(@Time))
		solveN(+N, -SolutionList, every(@Time))
		solveAll(-SolutionList, every(@Time))
	reset()	
	close()	
Dynamic KB	**Stateless**	**Stateful**
	getServiceConfiguration(-ConfigList)	
	getTheory(-Theory, ?Timestamp)	
	getGoals(-GoalList)	
	isGoal(+Goal)	
		setGoal(template(+Template))
		setGoal(index(+Index))
	solve(+Goal, -Solution, ?Timestamp)	solve(-Solution, ?Timestamp)
	solveN(+Goal, +NSol, -SolutionList, ?Timestamp)	solveN(+N, -SolutionList, ?Timestamp)
	solveAll(+Goal, -SolutionList, ?Timestamp)	solveAll(-SolutionList, ?Timestamp)
	solve(+Goal, -Solution, within(+Timeout), ?Timestamp)	solve(-Solution, within(+Timeout), ?Timestamp)
	solveN(+Goal, +NSol, -SolutionList, within(+Timeout), ?Timestamp)	solveN(+NSol, -SolutionList, within(+Timeout), ?Timestamp)
	solveAll(+Goal, -SolutionList, within(+Timeout), ?Timestamp)	solveAll(-SolutionList, within(+Timeout), ?Timestamp)
	solveAfter(+Goal, +AfterN, -Solution, ?Timestamp)	
	solveNAfter(+Goal, +AfterN, +NSol, -SolutionList, ?Timestamp)	
	solveAllAfter(+Goal, +AfterN, -SolutionList, ?Timestamp)	
		solve(-Solution, every(@Time), ?Timestamp)
		solveN(+N, -SolutionList, every(@Time), ?Timestamp)
		solveAll(-SolutionList, every(@Time), ?Timestamp)
	reset()	
	close()	

The service returns its currently configured properties when method `getServiceConfiguration(-ConfigList)` is invoked, where `ConfigList` is `[IsStateful, IsStateless, IsDynamic]`. Observational methods are `getServiceConfiguration`, `getTheory`, `getGoals`, `isGoal`, which return, respectively, configuration parameters, the KB the service relies on, the admissible goals, and whether the given input term is an admissible goal (true/false). Usage predicates vary depending on whether the service is stateless or stateful. In the former case, `solve` operation always needs the `Goal` as input parameter, whereas in the latter case `solve` is replaced by two distinct methods to be chained together: `setGoal` first, `solve` next—without specification of the goal. Furthermore, for stateful requests the returned solutions are always sequential, whereas for stateless ones the resolution process always restarts from the beginning. Accordingly, `solveAfter` methods have been introduced to enable fast-forwarding to the N+1 solution `AfterN`.

2.3 Time Awareness

Being aimed at dealing with real-world application environments, LPaaS requires the LP service to be *time-aware*—that is, capable of dealing with the fact that computation takes time, and that time flows regardless of whether it is computing or idle: this is why time-sensitive methods are included in LPaaS. Time-awareness of the service enables *time-situatedness* on the clients' side: should they need to perform time-related computations or inferences, they could just rely on the LP server.

Time may come into play at three levels: *(a)* through a timeout argument in queries, to prevent blocking the server; *(b)* in case of stateful requests, to ask for solutions periodically, every n milliseconds; and *(c)* in dynamic KBs, to capture the fact that information can expire after some time. Accordingly, `solve` operations can contain a `Timeout` parameter, specifying the maximum time that resolution should take (on the server side): if the resolution process does not complete within that time, the request is cancelled, and a negative response is returned.

In stateful requests, a client can ask to `solve` queries every `Time` milliseconds (server time), actually creating a *stream* of solutions: this is particularly useful in IoT scenarios, i.e., exploiting sensor devices or monitoring processes. Finally, all methods operating on a dynamic KB take an additional `Timestamp` argument, expressing the required time validity: only clauses valid at the given `Timestamp` are taken into consideration during SLD resolution process.

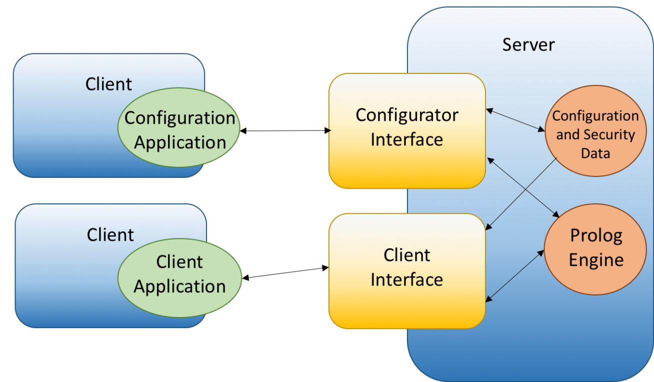

Figure 1 The LPaaS RESTful WS.

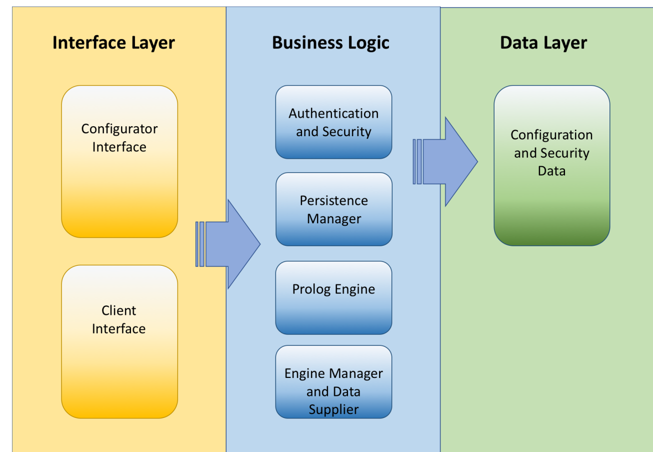

Figure 2 The LPaaS WS server architecture.

3 LPaaS as a RESTful Web Service

In order to test the effectiveness of the proposed architecture, we implement a first prototype of LPaaS as a RESTful web service (WS) [FT02]: we reuse and adapt patterns commonly used for the REST architectural style, and introduce a novel architecture supporting embedding Prolog engines into WS. Figure 1 shows the general architecture focussing on the server side and its components (access interfaces, Prolog engine, and data store), as well as some exemplary client applications interacting via HTTP requests and JSON objects.

The server-side inner architecture (Figure 2) is composed of three logical units: the *interface* layer, the *business logic* layer, and the *data store* layer. The interface layer encapsulates the Configurator and Client Interfaces. The Business Logic wraps the Prolog engine with the aim of managing incoming requests consistently. The Data Layer is responsible for managing the data store tracking, i.e., all the configuration options necessary to restore the service in case of unpredictable shutdown (i.e., operating parameters and security metadata such as clients' role, username, password, etc.).

Since these data are expected to be limited in size for most scenarios, we choose to keep them in the server application to offer a light-weight, self-contained service: however, they could be easily moved to a separate persistence layer on, i.e., an external DB application, if necessary.

The server implementation is built by exploiting a number of technologies commonly available in the field: in particular, the Business Logic is built using the J2EE framework [J2E], exploiting EJB [EJB], whereas the database interaction is implemented on top of JPA [Jav].

The Prolog engine is implemented on top of the tuProlog system [DOR01], which provides not only a light-weight engine, particularly well-suited for this kind of applications, but also a multi-paradigm and multi-language working environment, paving the way towards further forms of interaction and expressiveness. Since version 3.2, tuProlog also natively supports JSON serialisation, ensuring the interoperability required by a WS. The tuProlog engine, distributed as a Java JAR (or, as Microsoft .NET DLL, or, as Android app), is easily deployable and exploitable by applications as a *library service*—that is, from a software engineering standpoint, a suitably-encapsulated collection of related functionalities.

The service interfaces exploit the EJB architecture, but can also be accessed as a RESTful WS, realised using JAX-RS Java Standard (Jersey) [Jer]. Security is based on jose.4.j [jos], an open source (Apache 2.0)

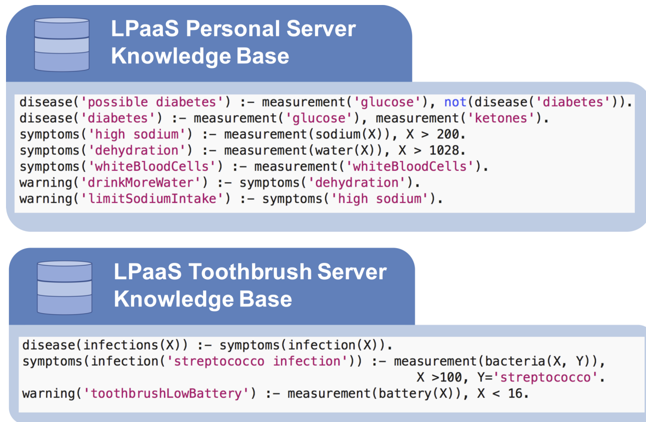

Figure 3 KB of two different LPaaS server: namely, Personal Server and Toothbrush Server.

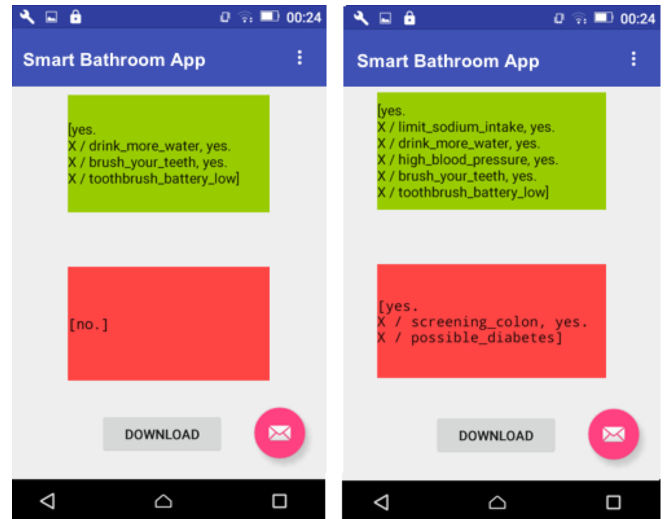

Figure 4 The Android application exploiting LPaaS: non-urgent messages are shown in green, urgent ones in red. The left screenshot shows three non urgent messages (drink more water, brush your teeth, and toothbrush battery low), while the right one shows two non-urgent (limit sodium intake, high blood pressure) and two urgent messages (the possibility of diabetes, and the suggestion of a colon screening).

implementation of JWT and the JOSE specification suite [jos]. The application is deployed using the Payara Application Server [Pay], a Glassfish open source fork, and its source code is freely available on Bitbucket [tuP].

3.1 Example Application

As a testbed scenario, let us consider a Smart Bathroom to monitor physiological functions to deduce symptoms and diseases, and properly alert the user. Sensors collect data and undertake reasoning based on LPaaS provided by tuProlog, to come up with solutions made available to the user through a dedicated Android application. The Smart Bathroom system is composed of three different tuProlog-enabled LPaaS services processing data collected by

Toilet Server toilet sensors analysing biological products, such as temperature, volume or glucose sensors: an example of such a circuit is reported in [KM15], where a sensor node for sampling water and checking for the presence of harmful bacteria such as E. coli in water sources was developed

Toothbrush Server nano sensors integrated into the toothbrush

Personal Server ultrasonic bathtubs, pressure sensing toilet seats and other devices to monitor people's cardiovascular health

Collected data may trigger different alerts: urgent ones, such as presence of Streptococcus infection, positive Diabetes Tests, etc. and normal ones, such as the need to drink more water, recharge batteries, and so on. An excerpt of the knowledge base of the services is shown in Figure 3.

The system is built on the following hardware configuration:

- Toilet Server: Raspberry Pi 3 (Ubuntu Mate Arm)
- Toothbrush Server: Lubuntu laptop
- Personal Server: Windows 10 laptop (stand alone machine)
- Client 1: Lenovo A10 tablet with Android 5.0.1
- Client 2: Windows 10 laptop running a desktop application.

Since tuProlog systems are Java-based, they can run on different platforms (*nix, Windows, iOS, Android). Windows 10 (on a stand-alone machine) is used for the personal server whereas the client2 app just as a Java host, in order to demonstrate the ability to inter-operate among different platforms [CD16].

Currently, all the data collected by sensors are simulated: Figure 4 shows some screenshots of the Android application. Urgent messages are in red boxes, minor warnings in green boxes.

Despite its simplicity, the case study is meant to show the potential of the LPaaS approach: local sensors can perform *situated reasoning*, applying their *local knowledge* to aggregate the raw data and synthesise higher-level information. Such higher-order data can then enable the creation of new computing services that autonomously respond to a user, and provide more accurate predictions based on situatedness—in this case, provided by the Android application.

4 LPaaS and Multi-Agent Systems

In this section we discuss how LPaaS can fit a multi-agent system (MAS), with the twofold aim of showing *why* merging LPaaS and MAS could be useful, especially in the IoT landscape, and *how* LPaaS and MAS could be successfully integrated.

4.1 Motivation

MAS for IoT. Agent-oriented engineering and MAS have been already recognised as a promising way of developing IoT applications and Cyber-Physical Systems (CPS), since they are well suited for supporting *decentralised*, *loosely-coupled* and *highly dynamic*, *heterogeneous* and *open* systems, in which components should *cooperate* opportunistically [CMS+17, AK15, ZO04]. They also offer a higher level of abstraction to system designers and developers than, i.e., RESTful approaches, as they replace low-level notions such as HTTP requests / responses with messages and interaction protocols [ON98].

Adopting the thing-oriented definition of IoT – that is, the so-called Smart Object (SO) IoT vision in which SOs are the basic IoT building blocks –, it is quite natural to map the sensing and actuating capabilities of SOs onto the perception and action capabilities of *situated agents* [HBKR10]. Also, SOs are meant to be autonomous in acting on behalf of their owner in the most common everyday activities such as, for a smart home scenario, adjusting temperature, tuning lights intensity, lock the doors, and so on. Not by chance, *autonomy* is also the core feature of agents [HCF03].

Another straightforward mapping may be drawn between the need for cooperation amongst SOs in complex IoT scenarios and the *social dimension* of agency [Cas98]—and, consequently, of MAS. In fact, SOs can be expected to interact with each other in order to perform even simple tasks, such as those related to situation recognition, and the usual means to do so in current IoT practice is either *(i)* to let the Cloud handle dependencies between tasks, for instance by monitoring a given sensor perceptions to trigger a given actuator when a threshold is met, or *(ii)* to exchange very simple messages (i.e., JSON structures) in a peer-to-peer way. Agents, instead, are naturally capable of diverse forms of social interactions, by exchanging rich messages in compliance with well-defined protocols having a clear and well-understood semantics—i.e., FIPA protocols and ACL messages [ON98].

Finally, featuring a *goal-oriented/driven* behaviour [Cas12], agents can plan and act based on the specific contingencies of the environment in which they operate. This deeply contrasts the imperative way of commanding SOs in current IoT practice, where actuator devices are usually only able to react to precise and direct instructions about what/how *to do*, not what *to achieve*, as a more declarative approach would suggest.

In spite of the above benefits, a relevant issue may hinder adoption of the agent abstraction, thus of MAS, in the IoT landscape: the computational limitations of SOs, which can be too severely resource-constrained to embed a full-fledged software agent. Here is where LPaaS comes into play, as explained in the next section.

LPaaS for MAS. Besides *autonomy*, *situatedness*, and *sociality*, agents may have other features that could map onto SOs: for instance, mobility – intended as code mobility – can be easily implemented even on resource-constrained devices, whereas *intelligence* – a hot topic in current IoT research – is considerably a more challenging issue. The fact is that providing a reasonable perception of intelligence for a given SO (agent) requires many different technologies, such as machine learning, common-sense reasoning, natural language processing, advanced situation recognition and context awareness—which are all typically computationally expensive *per se*, let alone in conjunction with the others. This is why the concept of LPaaS may actually improve the state of art in engineering intelligent IoT systems: with LPaaS, just the "required amount" of situated intelligence can be seamlessly spread where needed, and/or where the available resources are able to bear the computational effort, with no need to have a full-fledged intelligent agent embedded in every SO.

In this new perspective, whenever local intelligence cannot be available for any reason – i.e., memory constraints hindering the opportunity to have a local KB, CPU constraints limiting efficiency of reasoning, etc. – a given agent (SO) may simply request to another, "more intelligent" one, to perform some inferences on its behalf. Moreover, the LPaaS functionality may also be charged upon the infrastructure, instead of the agents. In this scenario, agents are always computationally efficient and responsive, since they delegate reasoning-related tasks – such as situation recognition, planning, inference of novel information, etc. – to dedicated infrastructural services—either hosted in the Cloud, as it currently happens for most IoT platforms, or spread amongst a distributed set of devices working as gateways for SOs.

As a last remark, traditional LP has well proven valid over the time both as a knowledge representation (KR) language and as an inference platform for rational agents. Logic agents may interact with an external environment by means of a suitably defined observe–think–act cycle. Significant attempts were made in the last decade to integrate rationality with reactivity and proactivity in logic programming [KS96, DST98, KS99]: for KR purposes bottom-up approaches – based on the stable model semantics – were successfully proposed; yet, multi-agent reasoning in this area were also exploited—e.g., for planning with action languages [Son17, DFP13, DFP10]. The re-interpretation of LP under the LPaaS approach in MAS could be seen as the evolution of these research threads for modern pervasive and distributed systems.

4.2 Fitting LPaaS in MAS

Engineering a MAS generally requires three orthogonal yet complementary dimensions [ORV08] to be considered: the *agent* dimension, where the internal structure of agents is designed and their behaviour programmed; the *social* dimension, where the focus is on the space of interaction [OOR04], thus in designing how agents interact; the *environment* dimension, where

the representation of anything in the physical or computational world relevant to the MAS itself lives. Integrating LPaaS with MAS thus requires first of all to carefully decide where the integration should take place—that is, along which dimension, and involving which abstractions.

agent Integrating along the agent dimension is probably the most natural way to proceed, as it directly injects intelligence more closely to the agents' business logic, or even deeply in their inner reasoning workflow—likewise for BDI agents [Rao96]. Nevertheless, it may also be the least suitable way for several IoT scenarios, where devices are usually severely resource-constrained, hence unable to host a software agent together with a LP engine. LPaaS is conceived precisely to enable such a sort of integration with no need to host the LPaaS engine within the agent itself: the engine can be hosted anywhere, and made accessible to agents through a dedicated service layer—be it implemented according to REST, as exemplified in Section 3, or according to any other architectural style / technology.

society Integrating LPaaS and MAS along the social dimension amounts to embedding LP functionalities in the *coordination artefacts* devoted to manage the interaction space [ORV06]. This approach is similar to the one adopted in TuCSoN [OZ99] with the ReSpecT coordination language [Omi07], where coordination rules enacted by enhanced tuple spaces [OD01] are expressed in a Prolog-like language (actually interpreted by a tuProlog engine, the same used for LPaaS).

environment The last alternative is to consider LPaaS services as part of the MAS environment. Usually this means deploying LPaaS as a *middleware* service, provided by the infrastructure hosting agents and enabling them to access the network and devices' capabilities. This approach could be implemented by exploiting LPaaS as a RESTful WS, as described in Section 3, and is complementary to the first one, in the case agents do not embed an LP inference engine but exploit LPaaS opportunistically.

It should be noticed that the three aforementioned approaches are not to be taken as mutually exclusive: in fact, our choice in the prototype system is to exploit the first and last one together, leaving MAS designers free to choose whether to embed intelligence within agents, or, instead, have it provided as an infrastructural service.

Figure 5 illustrates the model of the LPaaS approach depicting the whole picture in the hybrid case where *(1)* some agents are kept more lightweight and rely on infrastructural services (or other more "intelligent" agents) to get LPaaS functionalities, *(2)* some agents embed the LPaaS functionalities, and *(3)* some LP functionalities are embedded in some services provided by the middleware (namely by the containers).

The traditional MAS architecture is enriched with the notion of LPaaS agent / service, which allows for situated reasoning on locally-available data by design. In this vision, agents can be split in two groups: *local* agents and *global* agents. The former includes all the agents embedded in sensor and actuator devices, and in charge of generating the *local knowledge*: they represent the local view of the IoT system. The latter group includes agents with a higher-level view of the system, not necessarily embedded in SO devices, which act and coordinate their activities to properly pursue the system's goal. In the service layer, LPaaS and other typical middleware / application services are supplied. These services can be provided by the devices located in the physical world, by the MAS agents, or by dedicated infrastructural components.

Figure 6 (left) illustrates in detail the case where SO agents embed the LPaaS service, and are therefore able to both perform their own reasoning and offer their capabilities to others. In this case, each SO has a representative autonomous software agent, which is capable of monitoring the state of the device, make decisions on behalf of the device, and discover and exploit external help if necessary.

4.3 Revisiting Agents' Inner Architecture

Figure 6 (right) shows the inner architecture of a SO modelled as an LPaaS agent:

- The *Perception Handler* takes care of measurements coming from sensors. The new data feed the knowledge base of the SO, may alter existing knowledge, and contribute to the inference of novel knowledge

- The *KB Update* component evaluates incoming data, and updates the knowledge base accordingly. A trivial implementation of the LPaaS agent can simply insert novel information with no modifications or restrictions (actually acting as an information repository), whereas a more sophisticated one can interact with the *KB Query* component to perform consistent updates of the KB (actually supporting the interpretation of the KB as a logic theory of the local world)

- The *KB Query* component receives queries from the agent specific behaviour implementation, from the Perception Handler, and from the KB Update component. In the first case, it supports the agent's internal decision-making; in the second, it helps the KB to remain consistent while updating itself in reaction to external stimuli. In the latter case, it helps selecting which information will be inserted in the KB

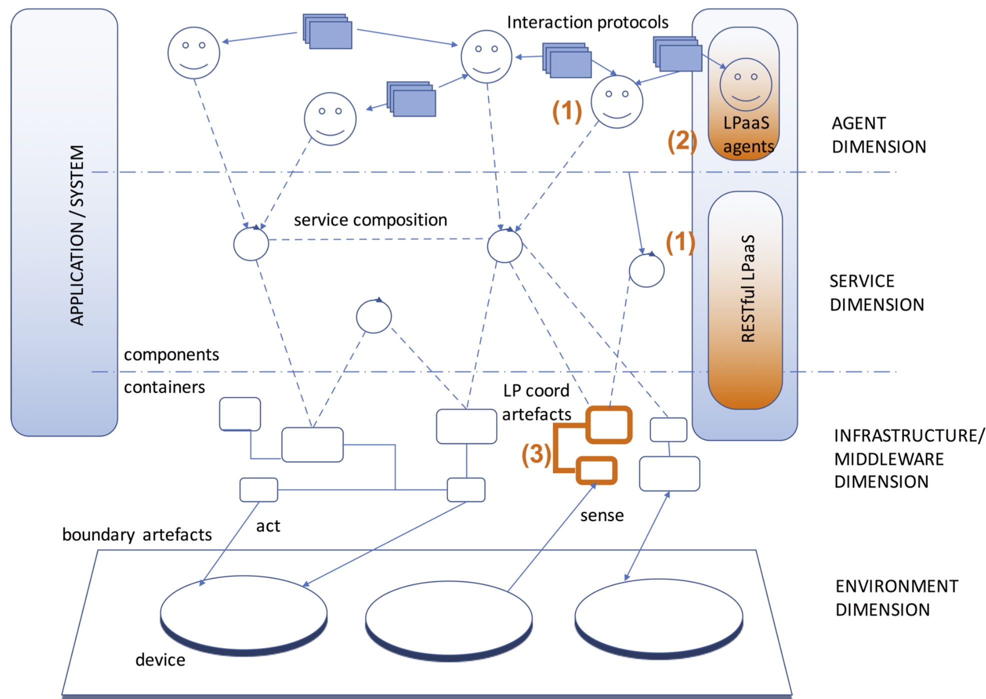

Figure 5 Overview of a LPaaS multi-agent system. At the bottom layer, the physical / computational environment lives, with *boundary artefacts* [ORV06] taking care of its representation and interactions with the rest of the MAS. Then, typically, some middleware infrastructure provides common API and services to application-level software – i.e., the containers where service components live – there including the *coordination artefacts* [ORV06] governing the interaction space. Finally, on top of the middleware, the application / system as a whole lives, in LPaaS MAS view as a mixture of services – possibly RESTful, as for LPaaS as a WS – and agents.

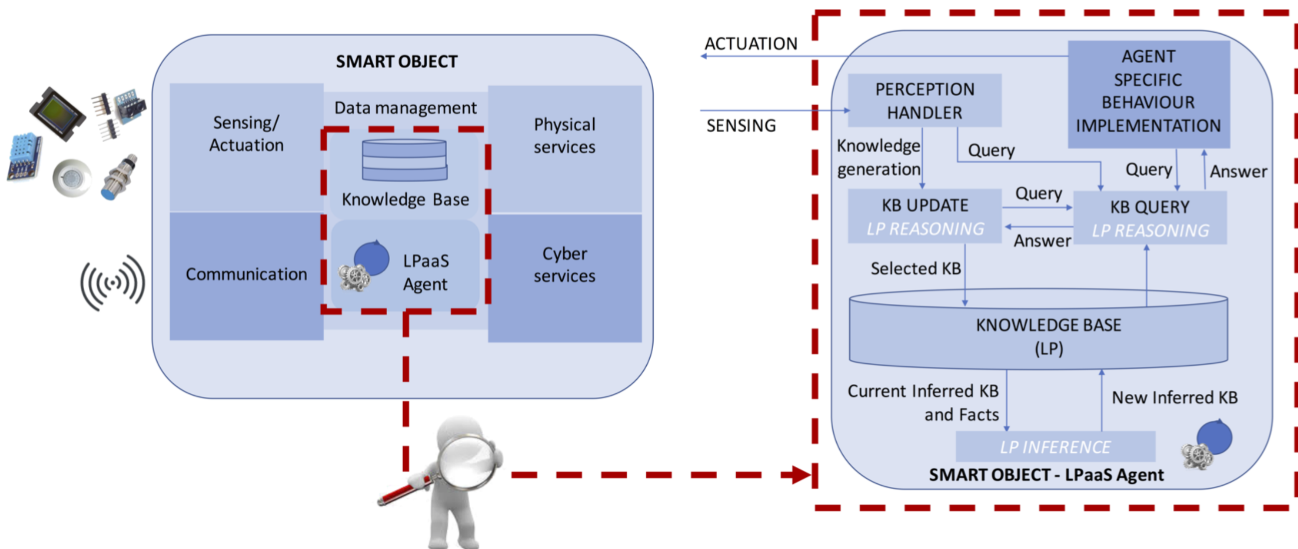

Figure 6 The Smart Object as an LPaaS Agent: typical SO conceptual architecture [FGRS17] enriched with LPaaS service (left) and inner architecture (right).

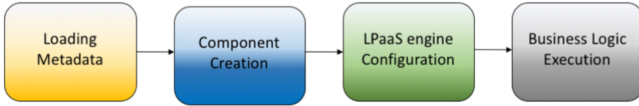

Figure 7 The LPaaS container in a MAS.

- The *Knowledge Base* is a logic theory, as described in Section 2. In principle, any modification of the surrounding environment perceived by sensors can produce an update
- The *LP Inference* component performs deduction from the existing KB. Its inputs is the current KB, which is a mixture of background knowledge, measured metrics, and deduced facts; its output is the novel inferred knowledge.

4.4 On LPaaS Agents' Lifecycle

In order to implement the LPaaS MAS architecture, a *container-component* programming model can be adopted: the LPaaS agent is a component living inside a container that manages its life cycle. The container-component model simplifies the configuration and usage of the LPaaS inferential engine in a distributed system, by *separating concerns*: on the one hand, the component is responsible for the LPaaS inferential engine and its business logic; on the other, the container is a portion of the middleware that manages the number of instances, configuration, and life cycle of the handled components.

Accordingly, an LPaaS agent is handled by an LPaaS container which manages the service's core. The agent is characterised by a cyclic behaviour: at each iteration, it receives a service request from a client, synthesises the response, and communicates the operation result. Figure 7 shows the operations performed by the container for creating and configuring an LPaaS component: the container, after loading the component's metadata, creates the component, configures the inferential engine, and runs the automatic methods (for self-configuration).

5 LPaaS MAS Prototype: LPaaS in Jade

The proposed approach has been implemented on top of the Jade middleware [JAD], which facilitates the development of interoperable, open, and heterogeneous multi-agent systems by relying on the FIPA standard [ON98], and exploiting tuProlog [DOR01, tuP] as the LPaaS Prolog engine. We choose tuProlog because of its peculiar blend of imperative, object-oriented, and logic programming styles: apart from being Java-based, light-weight, and easy deployable, it also enables and promotes a *multi-paradigm* programming style, where the Prolog code can invoke Java code and viceversa, yet keeping the two computational models clearly separate [DOR05].

Following the Jade approach to openness, the LPaaS agent must register with the Jade Directory Facilitator

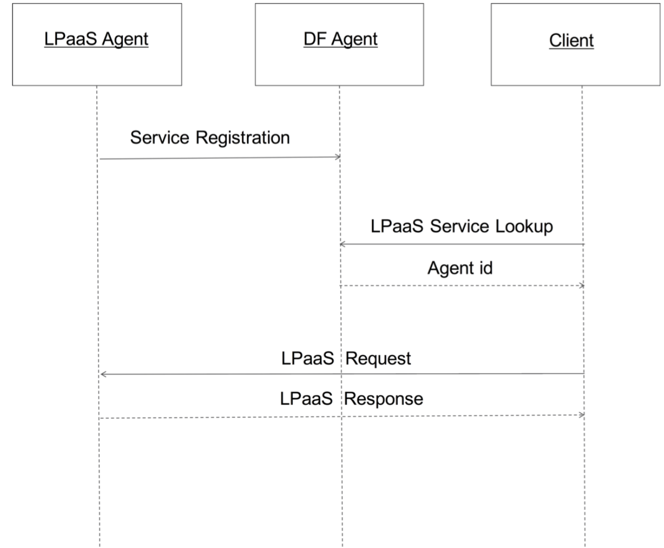

Figure 8 The LPaaS Service-DF-Client interaction.

(DF) – Jade yellow pages for services offered by agents in a MAS – by providing a logical identifier and indicating the sort of service offered (i.e., LPaaS). Thus, clients can dynamically perform a discovery operation and identify which LPaaS agents live in the system—since many may provide the LPaaS service simultaneously, i.e., for resiliency and performance reasons. In its turn, a client willing to use a LPaaS service must, as a first action, perform a discovery via the DF, requesting either the list of all agents offering LPaaS services, or a specific service, given its logical identifier.

The communication between Jade agents occurs via ACL (Agent Communication Language) messages [Fou02], that is, well-structured messages with a clear semantics and interoperable encoding. The request ACL message for an LPaaS service is always a FIPA Request, and should contain both the logical identifier of the LPaaS agent and the identifier of the operation to be run (a string that uniquely identifies a method). The request message may also contain the goal to demonstrate, possibly the number of solutions to be scanned, and the maximum service running time (i.e., timeout), depending on the nature of the LPaaS configuration (i.e. stateless vs. stateful, dynamic vs. static). The response ACL message is always a FIPA Inform [JAD], notifying the customer of the service result: the response message contains either the requested solutions or an error message if the service could not be run.

The client agent, once obtained the Jade AID (Agent IDentifier) of one or more agents from the DF as a result of the discovery phase, sends an LPaaS Request ACL message to the selected agent: in turn, the LPaaS agent replies with an LPaaS response message, which contains the service outcome (Figure 8). Interaction always adheres to the request-response pattern: the LPaaS agent is supposed to reply to the client in all cases, possibly with a failure message in case of errors.

The reception of the message may be either blocking or not blocking, both for the LPaaS agent and the client,

depending on the configuration parameter: blocking mode is the default for the LPaaS agent (so, with no service requests the agent is suspended), while the client fully depends on the application logic.

As far as security is concerned, two aspects are currently supported: *(i)* authentication and confidentiality of the communication; *(ii)* identification of client's permissions to access the service. The first is ensured using JADE-S (Secure JADE) [PRT01]: each LPaaS message is signed and encrypted. In particular, when an LPaaS agent receives a service request message, it first checks whether the message is signed by a known agent (via JADE-S) and only in this case proceeds by decrypting the message; otherwise, an error message is returned. The second level implies that LPaaS agents can distinguish privileged, configurator agents which can start, stop, and reconfigure (admissible goals, KB, etc.) the service.

Figure 9 The Smart Kitchen Architecture in a LPaaS tuProlog in JADE.

5.1 *Example Application*

As a first testbed for LPaaS in JADE, a Smart Kitchen IoT scenario is implemented. Four IoT devices – namely, a fridge, a pantry, a mixer and an oven – supply information to clients, exploiting the LPaaS approach, about the food supply and users' preferences.

The fridge and the pantry are capable of monitoring the quantity of food, and of collecting historical data on user's habits, i.e., the most commonly eaten food and preferred meals. The oven aims at supporting the user's cooking experience by relying on any available technology to identify and cook food. The user profile is supposed to include information about his/her dietary requirements. The mixer manages the recipe instructions, interacting with both the fridge – to check that the ingredients for the selected recipe are actually available – and with the oven—to check its ability to cook that food, and potentially synthesise the proper control instructions. Each device is supposed to have a limited computational capability, such as a Raspberry Pi, or of an Arduino board.

The application scenario requires that each device does not merely provide raw data, but is instead capable of producing higher-level knowledge and simultaneously of coordinating and collaborating with other entities in the system. In particular, each node (identified by a device) must maintain local knowledge about its status, be aware of the surrounding environment, and be able to communicate with the control device to share information about the kitchen state. Moreover, it should be possible to migrate the software from a device to another (i.e., if the device needs to be replaced) and to add / remove IoT devices to the system without shutting the system down first, i.e., in a "plug and play" fashion.

Assuming that all IoT and control devices are connected to a single home subnet, we choose to adopt a single JADE platform: the JADE Main Container is located on the controller node and is in charge of interacting / retrieving information from the smart kitchen devices. Each IoT device is designed as an LPaaS component, and is supposed to manage the node knowledge base and expose the goals. In this case, the available goals make it possible to query devices about available food and the user's habits—i.e., trace the available products, quantities, expiration date for perishables, purchase price and retailer, origin, users' preferred products, etc.

Figure 9 gives an overview of the corresponding JADE system, while Figure 10 shows a few key interactions. The Smart Kitchen Agent is the *global* agent, in our terminology, in charge of ensuring a coherent behaviour of the overall system based on the overall knowledge gathered. The interaction between the Smart Kitchen Agent and the LPaaS *local* agents – that is, those responsible of gathering local information and providing situated reasoning – occurs via ACL messages: the Smart Kitchen can thus obtain high-level information from the devices, process it, and decide the action(s) to be taken. For instance, if a given kind of food in the fridge is running out, the Smart Kitchen Agent may place an online order.

In its simplicity, the example scenario outlined here showcases how easy it is to spread situated intelligence in a IoT deployment by merging LPaaS with MAS. The Smart Kitchen agent, for instance, needs not bother tracking low-level data such as the amount and kind of perishables, recipes requirements, etc. altogether, so as to have all the system knowledge available and plan action accordingly. This is what typically happens in Cloud-based IoT deployments. In LPaaS MAS instead, the Smart Kitchen agent may directly ask its peers the higher-level information it needs for decision making – with queries such as "May I start cooking a lasagna?" or "Does Lisa like broccoli?" – expecting an informative reply—instead of a raw measurement, such as "We are missing besciamella for lasagna".

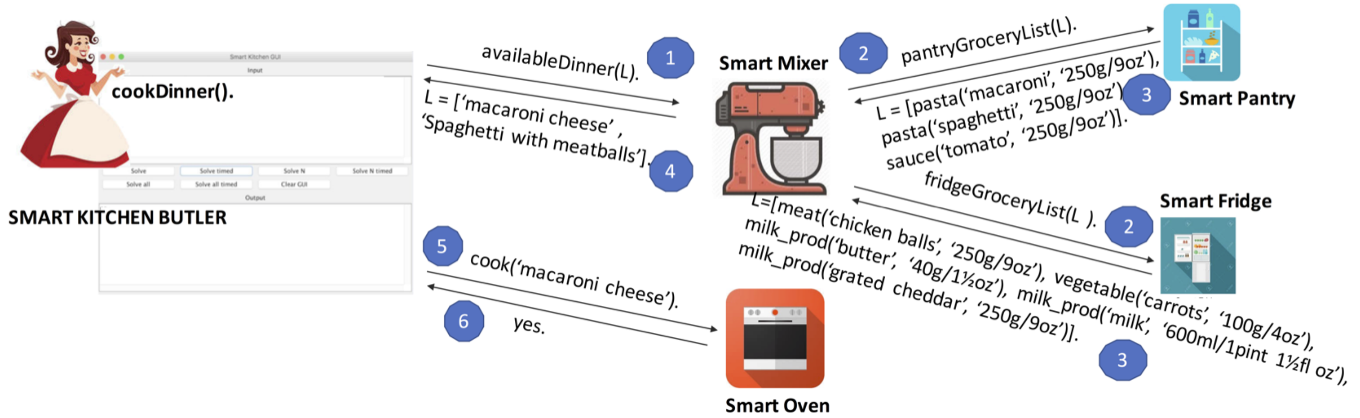

Figure 10 The Smart Kitchen Prototype: example of possible inferences.

6 Benefits & Open Issues

This section discusses the envisioned benefits of the LPaaS approach, both *per se* and merged with MAS, especially considering the IoT landscape, and along with the open issues to be dealt with for fully realising LPaaS, and LPaaS in MAS.

The first benefit, discussed throughout the paper, is due to LPaaS alone: *ubiquitous intelligence* for pervasive scenarios. LPaaS enables system designers to distribute reasoning and inference capabilities amongst the components they have, and let them balance the computational requirements to best suit the deployment scenario at hand—for instance, embedding LPaaS in more powerful components and letting ask their services by need. To the best of our knowledge, this is something unprecedented in the current IoT landscape, where most approaches either assume a fully-distributed network of smart objects capable of performing general-purpose computations, or resort to a Cloud-based setting where the whole system intelligence resides on the Cloud side.

The second benefit naturally follows: *situated reasoning*. LPaaS enables reasoning and inferential processes to be context-aware w.r.t. the (possibly ever-changing) environment where the process takes place. For instance, a sensor augmented with LPaaS capabilities – or, able to interact meaningfully with an LPaaS service – can reason on the locally-gathered data and provide to other system components not just raw measures, but high-level, inferred information about the sensed situation. In turn, this enables actuators to carry out a situated decision-making process, where the course of actions to undertake is the result of a situated planning (inference) process. It is worth noting that resorting mostly to locally available information reduces both the bandwidth consumption and the need for reliable communications between the distributed components, which are especially desirable features in IoT scenarios.

Furthermore, other benefits can be envisioned when coupling LPaaS with MAS: first of all, *goal-orientedness*. LPaaS agents may in fact exploit LPaaS to reason about their own goals, the plans and actions needed to achieve them, and the effects brought by—which is something only rational agents – such as BDI ones [Rao96] – usually do. Also, even non-LPaaS agents and non-agent components may do so, by simply interacting with the available LPaaS services. This is a simple and effective way to inject goal-orientedness within components of any kind, regardless of their inner architecture and implementation logic.

Lastly, when compared with the RESTful WS approach, the MAS-based one has some notable advantages: *(i)* complex interaction protocols built upon semantically rich messages (i.e., FIPA protocols on ACL messages); *(ii)* an interaction model particularly suitable for decentralised computations, based on the peer-to-peer model, yet not imposing a strict separation between client and server roles; and *(iii)* potential *mobility* of the service through the agent's own mobility.

In order to fully exploit the potential of the LPaaS approach, a few open issues are yet to be addressed. First of all, deploying LPaaS in a real IoT scenario is likely to require integration with databases, possibly distributed, which should work as the distributed knowledge base of the system. Then, the issue arises of handling replication and consistency of data scattered in connected devices, in particular when a coherent, logical interpretation of the data in LP terms is required—that is, as a logic theory of the state of the world.

Strict integration with sensor devices is also desirable to have LPaaS always working the most up-to-date perception of the environment properties of interest for the application at hand. In this respect, devising out ways to automatically embed the process of gathering sensors' perceptions into the LPaaS working cycle could prove to be extremely useful in facilitating adoption of the LPaaS approach and embedding of LPaaS within devices.

On the opposite side of the IoT spectrum – that is, looking at actuator devices – deep integration with their operation API is welcome, so as to have the LPaaS distributed engine automatically command

devices whenever some reasoning process results in the need of interacting with the physical world.

Another open issue is how to deal with situatedness in space, and mobility. Many modern applications would in fact benefit from having logic theories and inference processes somehow bound to spatial aspects of the application domain—similarly, and complementarily, to what LPaaS does with time. For instance, resolution of a query may consider also logic theories and LP engines in the neighbourhood of the queried one, and LPaaS engines may be able to move from node to node in search for better computational resources.

All the above issues are presently under consideration and will be subject of future research.

7 Related Work

Related work can be roughly divided into three main areas of interest: SOA in IoT, MAS in IoT, and standard knowledge representation in IoT. In spite of their differences, they all share the underlying idea that intelligence in the IoT comes from a large population of intelligent, small, networked, embedded devices at a high level of granularity—unlike more traditional approach where intelligence is typically concentrated in a smaller number of larger, monolithic entities (and applications) [HTM+14].

SOA in IoT. The SOA paradigm is widely used in IoT scenarios [MMM17, KCB+12, CKT10, GTW10, GTK+10]: moreover, communication via REST enables the direct integration of SOA-ready devices (i.e., devices hosting native web services).

MobIoT [HPI14] provides efficient service discovery, composition, and access in heterogeneous, dynamic, mobile IoT contexts, revisiting the standard SOA approach by providing probabilistic registration, look-up and thing-based composition based on comprehensive ontologies. However, it does not support runtime interaction with users to let them specify their goals, and still needs proper validation from the scalability viewpoint when the number of registered services is very large.

SIA [SKG+09] is a SOA which supports abstracting objects with embedded software as services. The architecture supports heterogeneous hardware, software and communication protocols among embedded systems. On the other side, it does not support self-adaptation due to runtime changes of the user's requirements, nor does it support runtime interaction with users to better understand their goals and preferences.

A novel approach for engineering IoT systems is proposed in [ASD17], where a set of things with their functionalities and services is connected and led to cooperate temporarily so as to achieve a given goal. The IoT-A project [IoT] investigates an architectural reference model based on a service-oriented approach where devices, data, and interfaces are abstracted and implemented as services: these services are then composed into complex processes to support the business needs. The composition is based on processes modelled by experts, and enacted by execution engines.

The HYDRA project [ERA09] defines a unique combination of SOA and a semantic-based Model Driven Architecture that creates a middleware for networked embedded systems enabling the development of generic services based on open standards. Developers can then create ambient intelligence applications based on wireless devices and sensors.

Many aspects developed in these works constitute source of inspiration for LPaaS, in particular in how the model and the architecture is conceived and designed. Following the SOA principles, we model ubiquitous intelligence in a dynamic context promoting portability and interoperable interaction over a network (gained through standards) and emphasising the separation of the service interface from its implementation. The LPaaS goes beyond the state of the art especially as concerns context-awareness: LPaaS facilitate the ability of injecting intelligence in existing services/agents exploiting the awareness of the context, thus promoting their adaptivity.

MAS in IoT. In the wide area of mediated interaction, MAS share with middleware-based approaches the idea of moving intelligence and decision-making closer to the actual infrastructure. Since agents are reactive, proactive and exhibit an intelligent and autonomous behaviour, MAS emerges as a natural approach to develop IoT systems [SFG+18, VZ17, SC17, MPS+16, MAT13].

MAIoT [NAG17] is a MAS-based architecture to coordinate IoT devices, based on the idea of enabling dialogues between IoT devices: to do so, IoT devices are wrapped into rational agents with reasoning and dialogue capabilities. A MAS architecture for healthcare AmI systems is presented in [RRLJ+17], where a MAS able to control the performance of all the tasks that a patient is doing during a rehabilitation therapy is discussed, showing how agents can be effective in reacting to humans based on information obtained from sensors and their knowledge.

In [For17], a multi-agent recommendation system for the IoT environment is proposed that exploits a decentralised and self-organising strategy. The recommendable things are described through metadata obtained by exploiting of a locality preserving hash function able to map similar things into similar metadata. Cyber agents manage the thing descriptors and autonomously decide to delivery/keep them by exploiting of tailored probability functions.

However, the above approaches do not really focus on the intelligence engineering and issues, nor do they seem to consider different paradigms integration by design. In fact, a fundamental principle of the LPaaS approach is to enrich the traditional MAS architecture with the notion of LPaaS agent / service, which allows for situated reasoning on locally available data by design.

In particular, orthogonal yet complementary dimensions (see Subsection 4.2) are taken into account in the design and implementation of LPaaS enabling the exploitation of the paradigm embedded in an agent, as a middleware service or as a boundary artefact.

Knowledge representation in IoT. Recent research works, like [FOM+17], propose standardisation models for sharing knowledge and reasoning across different IoT contexts, highlighting how the IoT knowledge is strongly related to the environment and how knowledge sharing is a requirement in order to reach the system intelligence. There, authors show how a multi-context knowledge base can be structured on the basis of modular ontologies and integrated with a distributed rule-based inference engine in multiple smart-building environments, in order to enable scalable contextual reasoning for intelligent assistance.

The LPaaS approach moves from similar roots, by proposing an LP approach for the standardisation and formalisation of knowledge: the basic assumption is that LP-based approaches are particularly suitable for data-intensive retrieval tasks with rule-based inference and promoting the automated reasoning and deduction on data. Moreover, the LP approach has obvious advantages like observability and understandability: developers suddenly have insight concerning on how a system can use its domain knowledge to achieve the task goals.

8 Conclusions

Pervasive and situated systems of any sort are increasingly demanding intelligence to be scattered throughout the computational devices populating the physical environment, as clearly demonstrated by IoT scenarios where varying degrees of intelligence are being used to support adaptation and self-management. The LPaaS architecture aims at fitting such a challenging context by introducing a standard interface that is general enough to account for both stateful and stateless services, with both static and dynamic knowledge bases, in a completely configurable and customisable way. Our implementation is designed on the top of tuProlog, a light-weight, multi-platform, and multi-language engine that is well suited for the purpose. We discuss and implement two different architectures: the first is based on the usual SOA infrastructure—namely, RESTful web services [FT02], while the second is based on multi-agent system [Fer99]. In addition, we discuss the integration of these different paradigm and solutions, highlighting the advantages of such a hybrid approach.

Of course, this is not the end of the story: many improvements can be devised in several direction, starting from a specialised LP-oriented middleware, dealing with heterogeneity of platforms as well as with distribution, life-cycle, interoperability, and coordination of multiple situated Prolog engines – possibly based on the existing tuProlog technology and TuCSoN middleware – so as to explore the full potential of logic-based technologies in IoT scenarios and applications. Moreover, the LPaaS interface can be extended with specific space-awareness methods to take the space around either the client or the server into account, exploring the chance to opportunistically federate LP engines by need as a form of dynamic service composition. Space-awareness and situatedness will be investigated, exploring the idea to opportunistically federate LP engines by need as a form of dynamic service composition.

References

[ACF+17] Gianluca Aloi, Giuseppe Caliciuri, Giancarlo Fortino, Raffaele Gravina, Pasquale Pace, Wilma Russo, and Claudio Savaglio. Enabling IoT interoperability through opportunistic smartphone-based mobile gateways. *Journal of Network and Computer Applications*, 81:74–84, 2017.

[AK15] Christos Alexakos and Athanasios P. Kalogeras. Internet of Things integration to a Multi Agent System based manufacturing environment. In *IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA 2015)*, Luxembourg, 8–11 September 2015.

[ASD17] Fahed Alkhabbas, Romina Spalazzese, and Paul Davidsson. Architecting emergent configurations in the Internet of Things. In *2017 IEEE International Conference on Software Architecture (ICSA 2017)*, pages 221–224, Gothenburg, Sweden, 3–7 April 2017.

[BFMZ13] Nicola Bicocchi, Damiano Fontana, Marco Mamei, and Franco Zambonelli. Collective awareness and action in urban superorganisms. In *2013 IEEE International Conference on Communications Workshops (ICC)*, pages 194–198, June 2013.

[BSS10] Stefano Bromuri, Michael Ignaz Schumacher, and Kostas Stathis. Towards distributed agent environments for pervasive healthcare. In Jürgen Dix and Cees Witteveen, editors, *Multiagent System Technologies*, volume 6251 of *LNCS*, pages 125–137. Springer, 2010.

[Cas98] Cristiano Castelfranchi. Modelling social action for AI agents. *Artificial Intelligence*, 103(1-2):157–182, August 1998.

[Cas12] Cristiano Castelfranchi. Goals, the true center of cognition. In Fabio Paglieri,

Luca Tummolini, Rino Falcone, and Maria Miceli, editors, *The Goals of Cognition. Essays in Honor of Cristiano Castelfranchi*, Tributes, chapter 41, pages 837–882. College Publications, December 2012.

[CD16] Roberta Calegari and Enrico Denti. Building Smart Spaces on the Home Manager platform. *ALP Newsletter*, December 2016.

[CDDO16] Roberta Calegari, Enrico Denti, Agostino Dovier, and Andrea Omicini. Labelled variables in logic programming: Foundations. In Camillo Fiorentini and Alberto Momigliano, editors, *CILC 2016 – Italian Conference on Computational Logic*, volume 1645 of *CEUR Workshop Proceedings*, pages 5–20, Milano, Italy, 20-22 June 2016. CEUR-WS.

[CDMO17] Roberta Calegari, Enrico Denti, Stefano Mariani, and Andrea Omicini. Logic Programming as a Service (LPaaS): Intelligence for the IoT. In Giancarlo Fortino, MengChu Zhou, Zofia Lukszo, Athanasios V. Vasilakos, Francesco Basile, Carlos Palau, Antonio Liotta, Maria Pia Fanti, Antonio Guerrieri, and Andrea Vinci, editors, *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC 2017)*, pages 72–77. IEEE, May 2017.

[CDO15] Roberta Calegari, Enrico Denti, and Andrea Omicini. Labelled variables in logic programming: A first prototype in tuProlog. In Elena Bellodi and Alessio Bonfietti, editors, *Proceedings of the Doctoral Consortium of the 14th Symposium of the Italian Association for Artificial Intelligence (AI*IA 2015 DC)*, volume 1485 of *CEUR Workshop Proceedings*, pages 25–30, Ferrara, Italy, 23–24 September 2015. CEUR-WS.

[CFG+17] Franco Cicirelli, Giancarlo Fortino, Antonio Guerrieri, Giandomenico Spezzano, and Andrea Vinci. Metamodeling of smart environments: from design to implementation. *Advanced Engineering Informatics*, 33(Supplement C):274–284, 2017.

[CKT10] Alessandro Cannata, Stamatis Karnouskos, and Marco Taisch. Evaluating the potential of a service oriented infrastructure for the factory of the future. In *8th IEEE International Conference on Industrial Informatics (INDIN 2010)*, pages 592–597, July 2010.

[CMS+17] Davide Calvaresi, Mauro Marinoni, Arnon Sturm, Michael Schumacher, and Giorgio Buttazzo. The challenge of real-time multi-agent systems for enabling IoT and CPS. In *2017 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2017)*, pages 356–364, New York, NY, USA, 2017. ACM.

[DDC96] Pierre Deransart, AbdelAli Ed Dbali, and Laurent Cervoni. *Prolog: The Standard. Reference Manual.* Springer, 1996.

[DFP10] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. An investigation of multi-agent planning in CLP. *Fundamenta Informaticae*, 105(1-2):79–103, 2010.

[DFP13] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Autonomous agents coordination: Action languages meet clp($\mathcal{FD}$) and Linda. *Theory and Practice of Logic Programming*, 13(2):149–173, 2013.

[DOR01] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 184–198. Springer, 2001.

[DOR05] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Science of Computer Programming*, 57(2):217–250, August 2005.

[DST98] Pierangelo Dell'Acqua, Fariba Sadri, and Francesca Toni. Combining introspection and communication with rationality and reactivity in agents. In Jürgen Dix, Luís Fariñas del Cerro, and Ulrich Furbach, editors, *Logics in Artificial Intelligence*, volume 1489 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 1998.

[EJB] EJB. Home Page. http://www.oracle.com/technetwork/java/javaee/ejb/.

[ERA09] Markus Eisenhauer, Peter Rosengren, and Pablo Antolin. A development platform for integrating wireless devices and sensors into ambient intelligence systems. In *6th IEEE Annual Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON 2009) Workshops*, June 2009.

[Fer99] Jacques Ferber. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999.

[FGG+13] Giancarlo Fortino, Roberta Giannantonio, Raffaele Gravina, Philip Kuryloski, and Roozbeh Jafari. Enabling effective programming and flexible management of efficient body sensor network applications. *IEEE Transactions on Human-Machine Systems*, 43(1):115–133, January 2013.

[FGL+13] Giancarlo Fortino, Antonio Guerrieri, Michelangelo Lacopo, Matteo Lucia, and Wilma Russo. An agent-based middleware for cooperating smart objects. In Juan M. Corchado, Javier Bajo, Jaroslaw Kozlak, Pawel Pawlewski, Jose M. Molina, Vicente Julian, Ricardo Azambuja Silveira, Rainer Unland, and Sylvain Giroux, editors, *Highlights on Practical Applications of Agents and Multi-Agent Systems*, volume 365 of *Communications in Computer and Information Science*, pages 387–398. Springer, 2013.

[FGRS14] Giancarlo Fortino, Antonio Guerrieri, Wilma Russo, and Claudio Savaglio. Integration of agent-based and Cloud Computing for the smart objects-oriented IoT. In *IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2014)*, pages 493–498, May 2014.

[FGRS17] Giancarlo Fortino, Raffaele Gravina, Wilma Russo, and Claudio Savaglio. Modeling and simulating internet-of-things systems: A hybrid agent-oriented approach. *Computing in Science Engineering*, 19(5):68–76, 2017.

[FOM+17] Jonathan Francis, Alessandro Oltramari, Sirajum Munir, Charles Shelton, and Anthony Rowe. Context intelligence in pervasive environments: Poster abstract. In *2nd International Conference on Internet-of-Things Design and Implementation (IoTDI'17)*, pages 315–316, New York, NY, USA, 2017. ACM.

[For17] Agostino Forestiero. Multi-agent recommendation system in Internet of Things. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '17)*, pages 772–775, Piscataway, NJ, USA, 2017. IEEE Press.

[Fou02] Foundation for Intelligent Physical Agents (FIPA). Agent communication language specifications. `http://www.fipa.org/repository/aclspecs.html`, 2002.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, May 2002.

[GTK+10] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. Interacting with the SOA-based Internet of Things: Discovery, query, selection, and on-demand provisioning of web services. *IEEE Transactions on Services Computing*, 3(3):223–235, July 2010.

[GTW10] Dominique Guinard, Vlad Trifa, and Erik Wilde. A resource oriented architecture for the web of things. In *2010 Internet of Things (IOT)*, 29 November–1 December 2010.

[HBKR10] Jomi F. Hübner, Olivier Boissier, Rosine Kitio, and Alessandro Ricci. Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous Agents and Multi-Agent Systems*, 20(3):369–400, May 2010.

[HCF03] Henry Hexmoor, Cristiano Castelfranchi, and Rino Falcone, editors. *Agent Autonomy*, volume 7 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer, Boston, MA, USA, 2003.

[HLS05] Bernd D. Heumesser, Andreas Ludwig, and Dietmar Seipel. Web Services based on Prolog and XML. In Dietmar Seipel, Michael Hanus, Ulrich Geske, and Oskar Bartenstein, editors, *Applications of Declarative Programming and Knowledge Management*, pages 245–257. Springer, 2005.

[HPI14] Sara Hachem, Animesh Pathak, and Valerie Issarny. Service-oriented middleware for large-scale mobile participatory sensing. *Pervasive and Mobile Computing*, 10:66–82, 2014. Selected Papers from the Eleventh Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2013).

[HTM+14] Jan Holler, Vlasios Tsiatsis, Catherine Mulligan, Stefan Avesand, Stamatis Karnouskos, and David Boyle. *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. Academic Press, 2014.

[IoT] IoT-A project. IoT-A architectural reference model. `http://open-platforms.eu/standard_protocol/iot-a-architectural-reference-model/`.

[J2E] J2EE. Home Page. `http://docs.spring.io/autorepo/docs/spring-framework/1.2.x/reference/`.

[JAD] JADE API. Home Page. http://jade.tilab.com/doc/api/jade/lang/acl/ACLMessage.html.

[Jav] Java Persistence API. Home Page. http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html.

[Jer] Jersey. Home Page. http://jersey.java.net.

[jos] jose.4.j. Home Page. http://bitbucket.org/b_c/jose4j/.

[JSO] JSON. Home Page. http://www.json.org.

[KCB+12] Stamatis Karnouskos, Armando Walter Colombo, Thomas Bangemann, Keijo Manninen, Roberto Camp, Marcel Tilly, Petr Stluka, Francois Jammes, Jerker Delsing, and Jens Eliasson. A SOA-based architecture for empowering future collaborative cloud-based industrial automation. In *38th Annual Conference on IEEE Industrial Electronics Society (IECON 2012)*, pages 5766–5772, October 2012.

[KKT92] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.

[KM15] Kyukwang Kim and Hyun Myung. Sensor node for remote monitoring of waterborne disease-causing bacteria. *Sensors*, 15(5):10569–10579, 2015.

[KS96] Robert Kowalski and Fariba Sadri. Towards a unified agent architecture that combines rationality with reactivity. In Dino Pedreschi and Carlo Zaniolo, editors, *Logic in Databases*, volume 1154 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 1996.

[KS99] Robert Kowalski and Fariba Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3):391–419, November 1999.

[LMMZ17] Marco Lippi, Marco Mamei, Stefano Mariani, and Franco Zambonelli. Coordinating distributed speaking objects. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS 2017)*, pages 1949–1960, June 2017.

[MAT13] Anas M. Mzahm, Mohd Sharifuddin Ahmad, and Alicia Y. C. Tang. Agents of Things (AoT): An intelligent operational concept of the Internet of Things (IoT). In *13th International Conference on Intellient Systems Design and Applications (ISDA 2013)*, pages 159–164, December 2013.

[MMM17] Fabrizio Messina, Rao Mikkilineni, and Giovanni Morana. Middleware, framework and novel computing models for grid and cloud service orchestration. *International Journal of Grid and Utility Computing*, 8(2):71–73, January 2017.

[MPS+16] Fabrizio Messina, Giuseppe Pappalardo, Corrado Santoro, Domenico Rosaci, and Giuseppe M. L. Sarné. A multi-agent protocol for service level agreement negotiation in cloud federations. *International Journal of Grid and Utility Computing*, 7(2):101–112, 2016.

[MQT] MQTT. Home Page. http://mqtt.org.

[NAG17] Juan Carlos Nieves, Daniel Andrade, and Esteban Guerrero. MAIoT – an IoT architecture with reasoning and dialogue capability. In Enrique Sucar, Oscar Mayora, and Enrique Munoz de Cote, editors, *Applications for Future Internet: International Summit, AFI 2016, Puebla, Mexico, May 25-28, 2016, Revised Selected Papers*, pages 109–113. Springer International Publishing, 2017.

[OD01] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, November 2001.

[Omi07] Andrea Omicini. Formal ReSpecT in the A&A perspective. *Electronic Notes in Theoretical Computer Science*, 175(2):97–117, June 2007.

[ON98] Paul D. O'Brien and Richard C. Nicol. FIPA — towards a standard for software agents. *BT Technology Journal*, 16(3):51–59, July 1998.

[OOR04] Andrea Omicini, Sascha Ossowski, and Alessandro Ricci. Coordination infrastructures in the engineering of multiagent systems. In Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors, *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter 14, pages 273–296. Kluwer Academic Publishers, June 2004.

[ORV06] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. *Agens Faber*: Toward a theory

of artefacts for MAS. *Electronic Notes in Theoretical Computer Science*, 150(3):21–36, 29 May 2006.

[ORV08] Andrea Omicini, Alessandro Ricci, and Giuseppe Vizzari. Smart environments as agent workspaces. *Ubiquitous Computing and Communication Journal*, CPE - Special Issue:95–104, June 2008.

[OZ99] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999.

[OZ04] Andrea Omicini and Franco Zambonelli. MAS as complex systems: A view on the role of declarative approaches. In João Alexandre Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies*, volume 2990 of *LNCS*, pages 1–17. Springer, May 2004.

[Pay] Payara. Home Page. http://www.payara.fish.

[PRT01] Agostino Poggi, Giovanni Rimassa, and Michele Tomaiuolo. Multi-user and security support for multi-agent systems. In Andrea Omicini and Mirko Viroli, editors, *WOA 2001 – Dagli oggetti agli agenti: tendenze evolutive dei sistemi software*, Modena, Italy, 4–5 September 2001. Pitagora Editrice Bologna.

[Rao96] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John W. Perram, editors, *Agents Breaking Away*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[RRLJ+17] Cristina Roda, Arturo C. Rodríguez, Víctor López-Jaquero, Elena Navarro, and Pascual González. A multi-agent system for acquired brain injury rehabilitation in ambient intelligence environments. *Neurocomputing*, 231(Supplement C):11–18, 2017.

[SC17] Munindar P. Singh and Amit K. Chopra. The Internet of Things and multiagent systems: Decentralized intelligence in distributed computing. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS 2017)*, pages 1738–1747, June 2017.

[SFG+18] Claudio Savaglio, Giancarlo Fortino, Maria Ganzha, Marcin Paprzycki, Costin Bădică, and Mirjana Ivanović. Agent-based computing in the Internet of Things: A survey. In Mirjana Ivanović, Costin Bădică, Jürgen Dix, Zoran Jovanović, Michele Malgeri, and Miloš Savić, editors, *Intelligent Distributed Computing XI*, pages 307–320. Springer, 2018.

[SKG+09] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, Luciana Moreira Sá de Souza, and Vlad Trifa. Soa-based integration of the internet of things in enterprise services. In *2009 IEEE International Conference on Web Services (ICWS 2009)*, pages 968–975, Washington, DC, USA, 2009. IEEE Computer Society.

[Son17] Tran Cao Son. Answer set programming and its applications in planning and multi-agent systems. In Marcello Balduccini and Tomi Janhunen, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 10377 of *Lecture Notes in Artificial Intelligence*, pages 23–35. Springer, 2017.

[tuP] tuProlog. Home page. http://tuprolog.unibo.it.

[VZ17] Eloisa Vargiu and Franco Zambonelli. Agent abstractions for engineering IoT systems: A case study in smart healthcare. In *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*, pages 667–672, May 2017.

[Zam15] Franco Zambonelli. Engineering self-organizing urban superorganisms. *Engineering Applications of Artificial Intelligence*, 41(C):325–332, May 2015.

[ZO04] Franco Zambonelli and Andrea Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, November 2004.