Logic Programming as a Service (LPaaS): Intelligence for the IoT

Roberta Calegari*, Enrico Denti*, Stefano Mariani† and Andrea Omicini*
*Dipartimento di Informatica, Scienza e Ingegneria (DISI) - ALMA MATER STUDIORUM–Università di Bologna
Email: {roberta.calegari, enrico.denti, andrea.omicini}@unibo.it
†Department of Sciences and Methods for Engineering – Università di Modena e Reggio Emilia
Email: stefano.mariani@unimore.it

***Abstract*—The widespread diffusion of low-cost computing devices, such as Arduino boards and Raspberry Pi, along with improvements on Cloud computing platforms, are paving the way towards a whole new set of opportunities for Internet of Things (IoT) applications and services. Varying degrees of intelligence are often required for supporting adaptation and self-management—yet, they should be provided in a light-weight, easy to use and customise, highly-interoperable way. Accordingly, in this paper we explore the idea of *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 a key enabling technology for intelligent IoT, we define the LPaaS general system architecture. Then, we present a prototype implementation built on top of the tuProlog system, which provides the required interoperability and customisation. We showcase the LPaaS potential through a case study designed as a simplification of the motivating scenarios.**

***Index Terms*—IoT, logic programming, LPaaS, pervasive computing, artificial intelligence, interoperability.**

I. 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* so as to adapt to inhabitants' preferences, habits, and requirements [1]. People are thus continuously connected together and with their surrounding entities, in a *situation-aware* and socially-aware way that is increasingly shaping a dense ecosystem where ICT devices and people collaborate as they were a *superorganism* [2]—in particular for complex urban services, such as intelligent transportation systems, environmental sustainability, and participatory governance [2], [3].

Similarly to living organisms, then, 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 suitable forms for their exploitation by the applications. In this context, connectivity and *interoperability* are just the basic – yet fundamental – bricks [4]. In order to build customised, variously-situated services and applications, a key infrastructural feature 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, so as 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, where software engineering, programming languages, and distributed artificial intelligence need to meet [5]. However, traditional logic programming (LP henceforth) techniques might be not enough for 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, as the natural evolution of distributed logic programming under a fresh IoT perspective, we here define *Logic Programming as a Service* (LPaaS) as an answer to the increasingly complex demand for distributed situated intelligence posed by nowadays pervasive systems.

Accordingly, Section II outlines the application scenarios envisioned for LPaaS and motivating the research effort as well; Section III defines the LPaaS architecture focussing on its most relevant features for the IoT business domain; Section IV reports on the first prototype implementation emphasising tuProlog effectiveness in enabling and supporting LPaaS; Section IV-A discusses an exemplary use case showcasing LPaaS potential; finally Section V concludes the paper and looks forward to future development of LPaaS vision.

II. APPLICATION SCENARIOS

LPaaS moves from the idea of providing an inference engine 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, in [6] a Prolog-based Web Service providing fuzzy search functionality on a collection of XML documents representing publications is discussed, which could be easily

978-1- 5090-4429- 0/17/$31.00

built as a LPaaS engine, resulting in a very compact and elegant program easy to maintain and deploy.

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 [7] 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 so as to provide a diagnosis – a task that could be well-suited for LPaaS using abductive LP [8] – contacting health care professionals if necessary.

Another intriguing application for LPaaS is *on-demand reasoning* in sensor networks, since it offers the possibility to inject chunks of situated intelligence locally. In fact, as discussed in [9], 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. There, 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.

There are several research works in the direction of making the next generation IoT smarter, such as agent-oriented and event-based frameworks for the development of cooperating smart objects [10], [11]. The baseline of works in this area is the idea of moving from connecting things to generating intelligence by the linking things in the real world with information in the digital world.

III. The LPaaS architecture

LPaaS provides an abstract view of an LP inference engine in terms of *service*, with the goal of promoting *interoperability*, *encapsulation*, and *situatedness*, thus reducing the need for integration and coupling while promoting *context-awareness*. Interoperability requires standards, which is why LPaaS defines a standard *interface* for client applications and relies on standard *representation* formats (i.e., JSON[1]) and *interaction* protocols (i.e., REST over HTTP, or MQTT[2]), versatile enough to fit a wide variety of application needs—especially in the IoT landscape. In particular, LPaaS is designed to enable situatedness, offering the chance to reason efficiently over data local to situated components. Diverse computational models can be tailored to the local needs of situated components, exploiting LP extensions explicitly aimed at pervasive systems such as *labelled variables systems* [12], [13].

In this perspective, each LP server node exposes its services concurrently to multiple clients, via the following interfaces. The inference engine is expected to implement SLD resolution [14], and is configured both with a *theory* of axioms – its Knowledge Base (KB) – and a set of *goals* – like in classic LP – that the client can ask to be proven.

[1]http://www.json.org
[2]http://mqtt.org

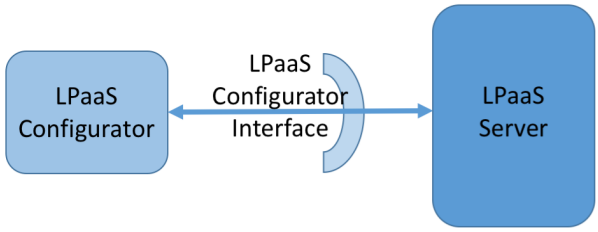

Fig. 1. The LPaaS Configurator Service Architecture

The service is initialised at deployment-time on the server machine; then, once started, it can be dynamically reconfigured at run-time whenever needed, but only by the Configurator (a privileged agent). Applications can access the service as either Clients or Configurators – in the latter case, access credentials have to be provided and checked – through the corresponding interfaces: the *Client Interface* exposes methods for observation and usage, while the *Configurator Interface* enables service configuration.

A. Configurator Interface

The Configurator interacts with the server via the Configurator Interface (Fig. 1). Configurator methods are detailed in Table I – with standard Prolog notation for input/output [15] – making it possible to set the service configuration, its KB, and the list of admissible goals.

Two main features characterise the LP service: the possibility of managing *stateful* and/or *stateless* requests, and *dynamic* vs. *static* KB.

Since the service aims at mimicking a classical logic programming engine using the SLD resolution [14], which means to ask for any number of solutions, and – only then – ask for each next solution iteratively, management of stateful requests is required. However, since this may not be resource-effective, and some application scenarios may not even need the feature – for instance a temperature sensor that always needs to reason on its latest measurement –, stateless requests are provided as an alternative option. In the latter case no session state is tracked by the server component, so each request must contain all the required information—whereas for stateful requests the server keeps track of the state of each individual client request (i.e., of each client resolution process).

As far as the logic theory is concerned, a *static* KB is immutable, while a *dynamic* KB can evolve during the server

TABLE I
LPaaS Configurator Interface

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

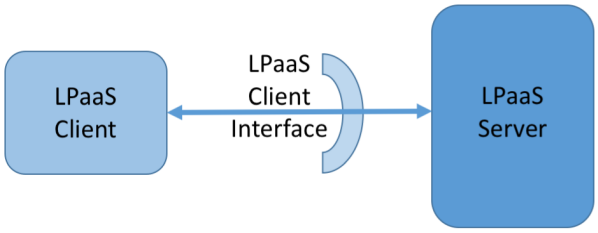

Fig. 2. The LPaaS Client Service Architecture

lifetime, thus implying that clauses have a lifetime too and can be asserted and retracted upon need—such as a clause representing the current temperature in a room.

It is worth noting that the service can be simultaneously stateful and stateless, as it can manage multiple kinds of request concurrently; the knowledge base, instead, can be either dynamic or static.

B. Client Interface

In LPaaS Client Service Architecture (Fig. 2) the server component provides the inference service to one or more clients via the Client Interface detailed in Table II.

The LP service offers *observational methods* to provide configuration and contextual information about the service, and *usage methods* to query the service for triggering computations and reasoning, and for asking solutions. Observational methods allow querying the service about its configuration parameters (stateful/stateless and static/dynamic), state of the KB, and admissible goals. Usage predicates (i.e., logic predicates for usage methods) allow the service to be asked for one or more solutions—one solution, N solutions, or all solutions available. Usage predicates are slightly different in case of stateless or stateful requests: in the former case, the `solve` operation is conceptually atomic and self-contained – always has the `Goal` as its parameter – whereas in the latter case self-containment is not necessary given that 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 resets the resolution process, effectively restarting resolution, with no need to reconfigure the service (i.e., select the goal); in turn, the `close` primitive actually ends communication with the server—so that the goal must be re-set in order to restart querying the server.

Further details about methods of the Client Interface are discussed in Table II.

C. Time Awareness

All the operations of the kind `solve` can also contain a `Timeout` parameter – that specifies the maximum time (server time) resolution should take – to avoid blocking the server: if the resolution process does not complete within the specified time, the request is cancelled and a negative response is returned to the client. In case of stateful requests, the client could also perform `solve` queries asking for one or more solutions every `time` milliseconds (server time), actually creating a *stream* of solutions—particularly useful in IoT scenarios exploiting sensor devices or monitoring processes. Furthermore, when the KB is dynamic all the methods take a `Timestamp` as an additional parameter, so that each theory has a *time-bounded validity* that can be used during the proof of a goal: only clauses valid at the given `Timestamp` are taken into consideration during SLD resolution process.

The service is then required to be *time-aware*, that is, conscious that computation takes time, and that regardless of whether it is computing or idle, time flows. This is why time-sensitive methods are included. The 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 ask the LP server.

IV. LPaaS in tuProlog

To test the effectiveness of the proposed architecture, we implemented a first prototype of LPaaS as a RESTful Web Service (WS) [16]: we reused and adapted patterns commonly used for the REST architectural style and introduced a novel architecture supporting embedding Prolog engines into WS. Fig. 3 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 (Fig. 4) is composed by 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, ...).

Since these data are expected to be rather limited in size for most scenarios, we choose to keep them in the server application so as 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.

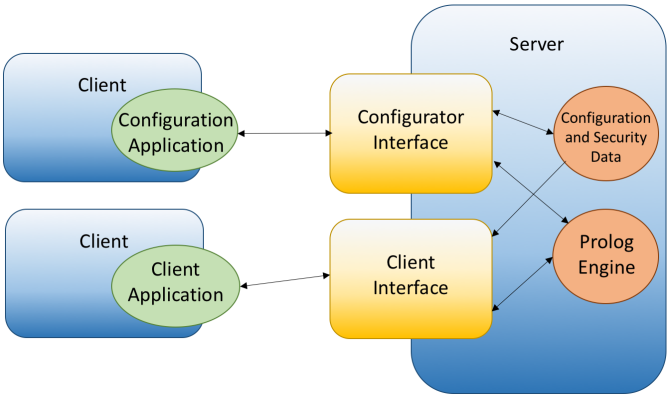

Fig. 3. The LPaaS RESTful Web Service

TABLE II

LPaaS Client Interface. The service returns its currently configured properties when method `getServiceConfiguration(-ConfigList)` is invoked, where `ConfigList` is `[isStateful, isStateless, isDynamic]`. `isStateful` is true if the service is configured to be stateful (thus all the methods of the stateful interface are available), `isStateless` is true if the service is configured to be stateless (thus all the methods of the stateless interface are available) and analogously `isDynamic` is true if the service is configured to be dynamic (thus all the methods of the dynamic interface are available). Observational methods are `getServiceConfiguration`, `getTheory`, `getGoals`, `isGoal`, which return, respectively, configuration parameters (stateful, stateless and static/dynamic), 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`.

	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(+Time))`	`solve(-Solution, within(+Time))`
	`solveN(+Goal, +NSol, -SolutionList, within(+Time))`	`solveN(+NSol, -SolutionList, within(+Time))`
	`solveAll(+Goal, -SolutionList, within(+Time))`	`solveAll(-SolutionList, within(+Time))`
	`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(+Time), ?Timestamp)`	`solve(-Solution, within(+Time), ?Timestamp)`
	`solveN(+Goal, +NSol, -SolutionList, within(+Time), ?Timestamp)`	`solveN(+NSol, -SolutionList, within(+Time), ?Timestamp)`
	`solveAll(+Goal, -SolutionList, within(+Time), ?Timestamp)`	`solveAll(-SolutionList, within(+Time), ?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 server implementation is realised exploiting a plurality of technologies that are commonly found in this field: in particular, the Business Logic is realised on the J2EE framework[3], exploiting EJB[4], while the database interaction is implemented on top of JPA[5].

The Prolog engine is implemented on top of the tuProlog system [17], 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,

[3]http://docs.spring.io/autorepo/docs/spring-framework/1.2.x/reference/

[4]http://www.oracle.com/technetwork/java/javaee/ejb/

[5]http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

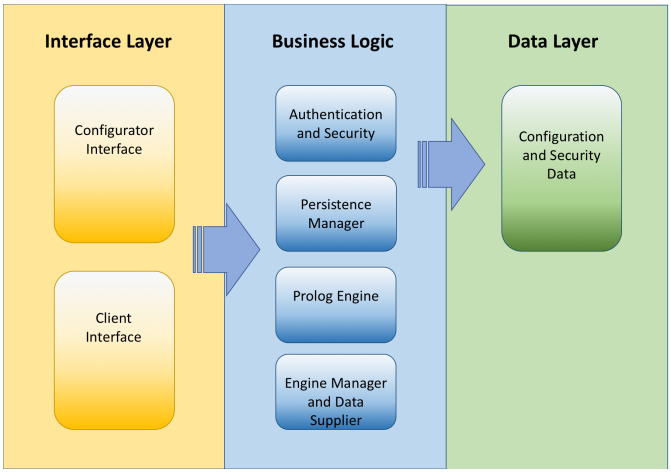

Fig. 4. The LPaaS Web Service Server architecture

thus paving the way towards further forms of interaction and expressiveness. tuProlog also supports JSON serialisation natively, ensuring the interoperability required by a WS. The tuProlog engine, distributed as a Java JAR or Microsoft .NET DLL, is easily deployable and exploitable by applications as a *library service*—that is, from a software engineering standpoint, a suitably encapsulated set of related functionalities.

The service interfaces exploit the EJB architecture, but can also be accessed as RESTful Web Services, realised using JAX-RS Java Standard (Jersey)[6]. Security is based on jose.4.j[7], an open source (Apache 2.0) implementation of JWT and the JOSE specification suite. The application was deployed using the Payara Application Server[8] (a Glassfish open source fork), and its source code is freely available on Bitbucket[9].

A. Case Study

As testbed scenario, let us consider a Smart Bathroom to monitor physiological functions so as 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 tuProlog Android application. The Smart Bathroom system is composed by three different tuProlog-enabled LPaaS services processing data collected by:

- toilet sensors analysing biological products, like temperature, volume or glucose sensors[10] (Toilet Server)
- nano sensors integrated into the toothbrush (Toothbrush Server)
- ultrasonic bathtubs, pressure sensing toilet seats and other devices to monitor people's cardiovascular health (Personal Server).

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. Messages are classified into urgent ones (i.e., presence of Streptococcus infection, positive Diabetes Tests, etc.) and less important ones (i.e., need to drink more water, recharge batteries, and so on). An excerpt of the knowledge base of the services is shown in Fig. 5.

```
%%LPaaS Personal Server KB
disease('possible diabetes') :- measurement('glucose'), not(disease('diabetes')).
disease('diabetes') :- measurement('glucose'), measurement('ketones').
symptoms('high sodium') :- measurement(sodium(X)), X > 200.
symptoms('dehydration') :- measurement(water(X)), X > 1028.
symptoms('whiteBloodCells') :- measurement('whiteBloodCells').
warning('drinkMoreWater') :- symptoms('dehydration').
warning('limitSodiumIntake') :- symptoms('high sodium').

%%LPaaS Toothbrush Server KB
disease(infections(X)) :- symptoms(infection(X)).
symptoms(infection('streptococco infection')) :- measurement(bacteria(X, Y)), X >100, Y='streptococco'.
warning('toothbrushLowBattery') :- measurement(battery(X)), X < 16.
```

Fig. 5. KB extraction of LPaaS Personal Server and LPaaS Toothbrush Server.

The system is built on the following network configuration:

- Toilet Server: on Raspberry Pi 3 (Ubuntu Mate Arm)
- Toothbrush Server: on Lubuntu laptop
- Personal Server: on Windows 10 laptop
- Client: tablet Lenovo A10 (Android 5.0.1)
- Client: desktop application on Windows 10 laptop

Currently, all the data collected by sensors are simulated. Fig. 6 shows some screenshots of the Android application giving evidence of urgent messages (red box) and minor warnings (green box).

Despite its simplicity, the case study shows the potential of LPaaS approach, where local sensors could perform *situated reasoning*, applying their *local knowledge* to aggregate the raw data and produce higher level synthesised information. Such higher-order data would both enable the creation of new computing services that autonomously respond to a user, and support much more accurate predictions based on situatedness.

V. Conclusions & Future Work

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; the architecture is based on the usual SOA infrastructure—namely, RESTful Web Services [16].

Future work will be devoted to more complete tests in pervasive deployment scenarios, mainly in the IoT landscape—i.e., testing directly LPaaS tuProlog over Bluetooth Low Energy connections. Also, space-awareness and situatedness will be investigated, exploring the idea to opportunistically federate LP engines by need as a form of dynamic service composition.

[6]http://jersey.java.net
[7]http://bitbucket.org/b_c/jose4j/
[8]http://www.payara.fish
[9]http://tuprolog.unibo.it
[10]http://www.wired.co.uk/article/yaniv-j-turgeman

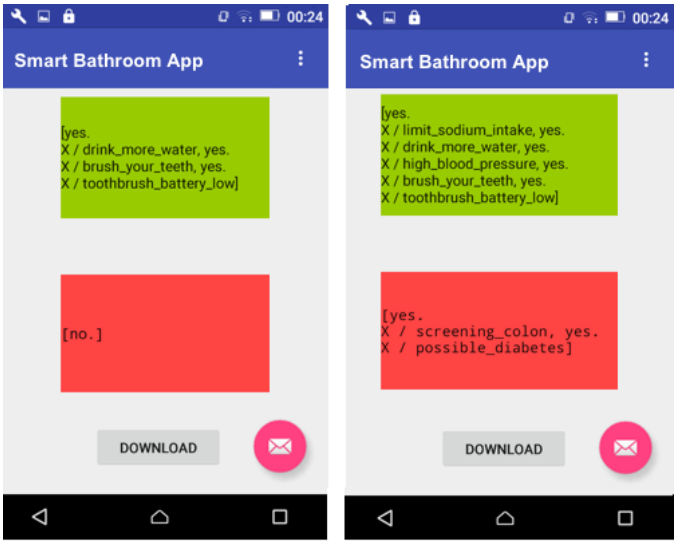

Fig. 6. The LPaaS Android application: non-urgent messages are shown in green, urgent ones in red. The user can download new messages at any time via the download button (that forwards the request to the server). The left screenshot shows three non urgent messages (drink more water, brush your teeth, 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).

A focal point of our forthcoming activity is the design and implementation of a specialised tuProlog-oriented middleware, dealing with heterogeneous platforms, as well as with distribution, life-cycle, interoperation, and coordination of multiple, situated Prolog engines – possibly based on the existing TuCSoN middleware – aimed at exploring the full potential of logic-based technologies in the context of IoT scenarios and applications.

Acknowledgements

Authors would like to thank Dipl. Eng. Andrea Muccioli for his contribution to this project and his work on the prototype of the LPaaS tuProlog.

References

[1] F. Cicirelli, G. Fortino, A. Guerrieri, G. Spezzano, and A. Vinci, "Metamodeling of smart environments: from design to implementation," *Advanced Engineering Informatics*, 2016, in press. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474034616302063

[2] F. Zambonelli, "Engineering self-organizing urban superorganisms," *Engineering Applications of Artificial Intelligence*, vol. 41, no. C, pp. 325–332, May 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197614002449

[3] N. Bicocchi, D. Fontana, M. Mamei, and F. Zambonelli, "Collective awareness and action in urban superorganisms," in *2013 IEEE International Conference on Communications Workshops (ICC)*, Jun. 2013, pp. 194–198. [Online]. Available: http://ieeexplore.ieee.org/document/6649227/

[4] G. Aloi, G. Caliciuri, G. Fortino, R. Gravina, P. Pace, W. Russo, and C. Savaglio, "Enabling iot interoperability through opportunistic smartphone-based mobile gateways," *Journal of Network and Computer Applications*, vol. 81, pp. 74 – 84, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804516302405

[5] A. Omicini and F. Zambonelli, "MAS as complex systems: A view on the role of declarative approaches," in *Declarative Agent Languages and Technologies*, ser. Lecture Notes in Computer Science, J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, Eds. Springer, May 2004, vol. 2990, pp. 1–17, 1st International Workshop (DALT 2003), Melbourne, Australia, 15 Jul. 2003. Revised Selected and Invited Papers. [Online]. Available: http://link.springer.com/10.1007/978-3-540-25932-9_1

[6] B. D. Heumesser, A. Ludwig, and D. Seipel, "Web Services based on Prolog and XML," in *Applications of Declarative Programming and Knowledge Management*, D. Seipel, M. Hanus, U. Geske, and O. Bartenstein, Eds. Springer, 2005, pp. 245–257. [Online]. Available: http://link.springer.com/10.1007/11415763_16

[7] S. Bromuri, M. I. Schumacher, and K. Stathis, "Towards distributed agent environments for pervasive healthcare," in *Multiagent System Technologies*, ser. Lecture Notes in Computer Science, J. Dix and C. Witteveen, Eds. Springer, 2010, vol. 6251, pp. 125–137, 8th German Conference, MATES 2010, Leipzig, Germany, September 27-29, 2010. Proceedings. [Online]. Available: http://link.springer.com/10.1007/978-3-642-16178-0_13

[8] A. C. Kakas, R. A. Kowalski, and F. Toni, "Abductive logic programming," *Journal of Logic and Computation*, vol. 2, no. 6, pp. 719–770, 1992. [Online]. Available: http://logcom.oxfordjournals.org/cgi/doi/10.1093/logcom/2.6.719

[9] G. Fortino, R. Giannantonio, R. Gravina, P. Kuryloski, and R. Jafari, "Enabling effective programming and flexible management of efficient body sensor network applications," *IEEE Transactions on Human-Machine Systems*, vol. 43, no. 1, pp. 115–133, Jan. 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6392962/

[10] G. Fortino, A. Guerrieri, W. Russo, and C. Savaglio, "Integration of agent-based and cloud computing for the smart objects-oriented iot," in *Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, May 2014, pp. 493–498.

[11] G. Fortino, A. Guerrieri, M. Lacopo, M. Lucia, and W. Russo, *An Agent-Based Middleware for Cooperating Smart Objects*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 387–398. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38061-7_36

[12] R. Calegari, E. Denti, A. Dovier, and A. Omicini, "Labelled variables in logic programming: Foundations," in *CILC 2016 – Italian Conference on Computational Logic*, ser. CEUR Workshop Proceedings, C. Fiorentini and A. Momigliano, Eds., vol. 1645. Milano, Italy: CEUR-WS, 20-22 Jun. 2016, pp. 5–20, Proceedings of the 31st Italian Conference on Computational Logic. [Online]. Available: http://ceur-ws.org/Vol-1645/paper_7.pdf

[13] R. Calegari, E. Denti, and A. Omicini, "Labelled variables in logic programming: A first prototype in tuProlog," in *Proceedings of the Doctoral Consortium of the 14th Symposium of the Italian Association for Artificial Intelligence (AI*IA 2015 DC)*, ser. CEUR Workshop Proceedings, E. Bellodi and A. Bonfietti, Eds., vol. 1485, AI*IA. Ferrara, Italy: Sun SITE Central Europe, RWTH Aachen University, 23–24 Sep. 2015, pp. 25–30. [Online]. Available: http://ceur-ws.org/Vol-1485/proceedings.pdf#page=30

[14] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM*, vol. 12, no. 1, pp. 23–41, Jan. 1965. [Online]. Available: http://dl.acm.org/citation.cfm?id=321253

[15] P. Deransart, A. E. Dbali, and L. Cervoni, *Prolog: The Standard. Reference Manual*. Springer, 1996. [Online]. Available: http://link.springer.com/10.1007/978-3-642-61411-8

[16] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, May 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=514185

[17] E. Denti, A. Omicini, and A. Ricci, "tuProlog: A light-weight Prolog for Internet applications and infrastructures," in *Practical Aspects of Declarative Languages*, ser. Lecture Notes in Computer Science, I. Ramakrishnan, Ed. Springer, 2001, vol. 1990, pp. 184–198, 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 Mar. 2001. Proceedings. [Online]. Available: http://link.springer.com/10.1007/3-540-45241-9_13