Semantic-based Context Modeling for Quality of Service Support in IoT Platforms

E. Mingozzi, G. Tanganelli, C. Vallati
Dip. Ingegneria dell'Informazione, University of Pisa
L.go L. Lazzarino 1, I-56122
Pisa, Italy
{e.mingozzi, g.tanganelli, c.vallati}@iet.unipi.it

B. Martínez, I. Mendia, M. González-Rodríguez
ICT-ESI Division, Tecnalia Research & Innovation
Parque Tecnológico de Bizkaia, edif. 700, E-48160
Derio, Spain
{belen.martinez, izaskun.mendia, marta.gonzalez}@tecnalia.com

The final publication is available at IEEE via http://dx.doi.org/10.1109/WoWMoM.2016.7523563

***Abstract*—The Internet of Things (IoT) envisions billions of devices seamlessly connected to information systems, thus providing a sensing platform for applications. The availability of such a huge number of smart things will entail a multiplicity of devices collecting overlapping data and/or providing similar functionalities. In this scenario, efficient discovery and appropriate selection of things through proper context acquisition and management will represent a critical requirement and a challenge for future IoT platforms. In this work we present a practical approach to model and manage context, and how this information can be exploited to implement QoS-aware thing service selection. In particular, it is shown how context can be used to infer knowledge on the equivalence of *thing services* through semantic reasoning, and how such information can be exploited to allocate thing services to applications while meeting QoS requirements even in case of failures. The proposed approach is demonstrated through a simple yet illustrative experiment in a smart home scenario.**

Keywords—IoT; Semantic modeling; Context awareness; QoS.

I. INTRODUCTION

The Internet of Things (IoT) is rapidly evolving from an abstract buzzword to the development of concrete systems that seamlessly integrate *smart things* into information systems. There are however still many open issues that must be addressed to uncover the real impact of IoT systems into our day life.

Context-aware data processing is one of the emerging challenges for IoT platforms and an important research topic [1]. The large availability of things collecting overlapping data or providing similar functionalities, in particular, will require proper context collection and management to detect redundant information and overlapping services. There is today still a lack of common semantic methods to describe elements in the IoT at the highest semantic level [2]. This leads to situations in which people from different cultures or social environments describe the same entities in different ways, which hinders system understanding and interoperability.

Quality of Service (QoS) support has been also identified as a key non-functional requirement for a broad class of IoT applications. However, the heterogeneity of applications and devices raises new issues that must be addressed to design a proper QoS framework for the IoT scenario. On one hand, in fact, we have applications that require services enriched with QoS requirements. On the other hand, however, requested services are provided by smart things with usually limited computational and communication capabilities. For this reason, QoS solutions adopted in other scenarios, e.g., traditional Service Oriented Architectures, cannot be applied directly.

In this work we propose a solution for standardizing the description of things and associated services in IoT platforms, which aims at overcoming semantic interoperability limitations by using *natural language processing*. This would allow applications to perform context-aware IoT service discovery by specifying requirements based on natural language and not on an agreed unified vocabulary. We then show how semantic-based context processing can be exploited by the platform to provide QoS support in an efficient manner, i.e., by exploiting semantic reasoning based on the proposed ontology in order to infer equivalence among services provided by heterogeneous things, and taking advantage of such equivalence to allocate thing services to applications while meeting QoS requirements even in case of failures.

The rest of the paper is structured as follows. In Section II we present the proposed architecture from a functional model perspective. In Section III a detailed description of the approach used to manage context and to exploit semantic reasoning is provided. Section IV illustrates how context is exploited to enforce QoS requirements of applications. In Section V validation results are presented to demonstrate the validity of the proposed approach. Finally, the state of the art is presented in Section VI while conclusions are drawn in Section VII.

II. REFERENCE ARCHITECTURE

We refer to a simple layered architecture as depicted in Fig. 1, which is based on the reference architecture for IoT platforms designed by the BETaaS project [3]. The core of the platform is the *Things-as-a-Service* (*TaaS*) *Layer*, which exposes a uniform content-centric service-oriented interface, named *Thing Service*, that allows applications to access services provided by physical things regardless of their location. To seamlessly integrate smart devices with heterogeneous technologies into the same platform, the TaaS layer leverages the functionalities implemented by the *Adaptation Layer* – one instance per different technology – which abstracts the details

978-1-5090-2185-7/16/$31.00

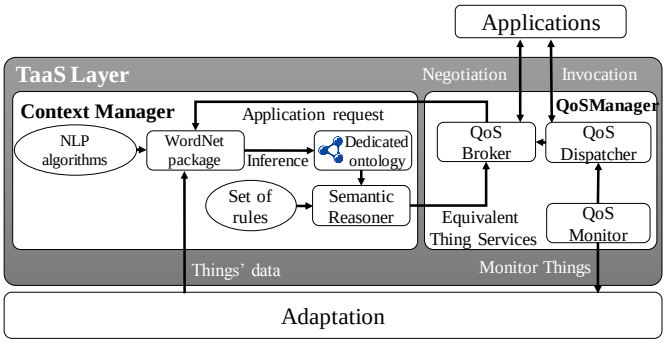

Fig. 1. Reference architecture.

of the respective technology and provides a basic uniform interface to the TaaS layer[1].

For the purpose of implementing the solution described in this work, we consider two main functional components in the TaaS Layer: the *Context Manager* and the *QoS Manager*, respectively. We assume that applications interact with the platform through a two-step procedure: *negotiation*, performed at the beginning to announce the required thing services and corresponding QoS requirements, and *invocation*, during which the requested thing services are invoked. In both steps, applications interact with the QoS Manager only. Smart things, instead, are managed by the Context Manager, which provides to the QoS Manager the input needed to select the correct thing services for each application's request among the set of available thing services.

More in detail, the Context Manager integrates heterogeneous context representations and exploits context data to detect, by means of semantic reasoning, equivalent things, i.e., things that can provide overlapping information or equivalent functionalities. Such functionalities are exposed to applications in order to support automated selection of thing services. Contextual information related to things is described in Natural Language and analyzed through WordNet[2], an open source lexical database that relates English words semantically. Thanks to the lexical classification of words made by WordNet (e.g., synonymy), the platform is able to infer relationships between things. The QoS Manager, instead, enables the advanced management of resources by exploiting context-awareness support. In particular, it allows applications to specify QoS requirements for requested services, which are then enforced through proper assignment of requests to things, exploiting the availability of equivalent thing services inferred from context information.

In the following we provide a detailed description of the solution proposed to model context information, and how this can be exploited to provide QoS support.

III. Context Modeling and Thing Equivalence

A. Modeling data and context

One of the biggest challenges of IoT is being able to integrate a vast amount of heterogeneous entities. To address this heterogeneity, it is necessary to use accurate modeling techniques, such as ontologies, to provide a formal representation of both entities and the data they handle. Ontologies and their ability to represent information provide sufficient level of semantic abstraction to address any IoT domain.

We exploit ontologies to model IoT environments, in particular we have built a dedicated ontology to unify the information coming from heterogeneous resources and applications, and to infer knowledge from this raw data. This ontology is stored in the Context Manager, whose structure is illustrated on the left side in Fig. 1. For ontology development, we have used Apache Jena[3].

Information can have different meanings depending on the context in which it is used. Besides, context is dynamic by nature. In the platform, context refers to all attributes that characterize entities (things) in the IoT and that could be relevant for applications. Thus, the context of the things is composed by elements such as the type of the thing (e.g., temperature sensor) or its location. Location is described by multiple data: geographical coordinates (for outdoor scenarios only), city name (automatically calculated from GPS coordinates by a reverse geocoding online web service offered by GeoNames[4]), a keyword (e.g., street) and a descriptive text (e.g., Via Roma). It is worth to note that our ontology is capable of modeling both the things and the context in which things are immersed. Other contextual information like the battery level of the thing, though semantically modelled by the Context Manager, are stored to be exploited by external modules such as the QoS Manager presented in the following section. Contextual information that is not automatically captured can be inserted manually at time of system configuration using a GUI.

Our ontology is a network of ontologies that we have created reusing several existing ontologies from the IoT domain. The reason for using a network of ontologies is based on the advantage of this solution against the use of unrelated ontologies, as knowledge extraction (inference) is accelerated in a remarkable way when ontologies are related between them. Furthermore, the use of already existing ontologies is always more convenient than to design a new one from scratch, as we can leverage existing knowledge in the area, and maintenance and updating is in the hands of third parties. The most relevant ontology in this network is the SSN[5] ontology, which we consider the most comprehensive and complete currently for modelling sensors and actuators. The scope of this ontology has been expanded by networking it with other domain-specific ontologies. Being a *context-aware platform*, these domain-specific ontologies contribute to model the contextual information associated to each of the connected resources. Specifi-

[1] It is worth to note that the BETaaS architecture also includes a Service Layer for the purpose of service composition; however, such functionality is not relevant for the purpose of this work, and therefore it has been omitted.
[2] http://wordnet.princeton.edu
[3] https://jena.apache.org/
[4] http://www.geonames.org/export/web-services.html#findNearbyPlaceName
[5] http://www.w3.org/2005/Incubator/ssn/XGR-ssn-20110628

TABLE 1. BETAAS THING CONTEXT.

Name	Description (ontology)
Output	e.g. true (if sensor), false (if actuator). **SSN ontology**
Digital	e.g. true (if digital device), false (if analogic device). **SSN ontology**
MaxResponseTime	Thing max response time (ms). **OWL-Time ontology**
BatteryLevel	Thing battery level (%). **SSN ontology**
Measurement	Thing measurement. **SSN ontology**
Protocol	Communication protocol (e.g. zigbee). **FIPA Device ontology**
DeviceId	Gateway Unique ID. **SSN ontology**
Type	Thing Type expressed by means of a keyword (e.g. temperature, humidity). **Phenonet ontology**
Unit	Units of the measurement (e.g. celsius, cars/hour). **MUO ontology**
Environment	Type of scenario (e.g. public, private). **BETaaS ontology**
Latitude, Longitude, Altitude	Outdoor location for *Public* environment. **GeoSPARQL ontology**
Floor	Indoor location for *Private* environment. **SSN ontology**
Location Keyword	Thing Location expressed by means of a keyword (e.g. street, home). **WordNet synsets stored in the SSN ontology**
City name	**GeoNames ontology**

cally, we have used the following ontologies: *i*) *OWL-Time*[6] to model temporal concepts; *ii*) *CF*[7] for the representation of climatic and forecast data; *iii*) *Phenonet*[8] for the representation of the type of sensors and actuators connected to the platform; *iv*) *MUO*[9] to model the units of the measurements taken; *v*) *FIPA*[10] to model the capabilities of the entities; *vi) GeoSPARQL*[11] to model geographical coordinates; and *vii*) *GeoNames*[12] to model the name of the city where an entity has been installed. The network of selected ontologies offers a broad modeling that exceeds by far the actual needs of our platform. In order to adjust this network of ontologies to real needs, some parts of this network have been pruned. Besides, some concepts are not covered by the ontologies selected. Thus, we have added the capabilities needed to model concepts such as Thing Services and Actuators. A partial list of the parameters modelled by the ontology is outlined in Table 1.

For all things that are connected to the platform, one unique *thing service* is created by the Context Manager. We have determined a naming convention to represent these thing services, based in part on the contextual information associated with each of the things connected to the platform, such as the location and the type of the thing, following the nomenclature *setLocationType*/*getLocationType* (e.g. getKitchenTemperature, setViaRomaStreetLighting).

As we mentioned, we propose a solution for standardizing the description of things in IoT environments that uses Natural Language Processing and WordNet. WordNet groups English words into sets of cognitive synonyms called *synonym sets* or *synsets*, and interlinks these synsets by means of conceptual-semantic and lexical relations, forming a network. When a new thing is attached to the platform, a GUI is shown to the installer of the thing asking for all contextual information that cannot be directly provided by this thing (e.g., location and thing type). The GUI asks the user for a name when defining sensor types, and for a verb when defining actuator types. Whenever possible, the GUI looks for the words provided in WordNet, gets the synset ID of the word and stores it in the ontology. In the case of words with multiple meanings, the GUI retrieves all the possible meanings of the word and asks the installer to disambiguate it.

Besides, we use WordNet to *infer knowledge* from raw data. On one hand, the Context Manager uses WordNet to infer *synonymy* relationships between words, so all words with the same synset ID are considered as synonyms. For example, if an application needs to measure location, things classified as localization sensors are valid, as both words share the same synset ID. On the other hand, the Context Manager is also able to infer *families of types of things*. We have defined a procedure that infers that two terms belong to the same family when the first four digits of their WordNet identifiers coincide, e.g., if an application needs to measure humidity, things classified as moisture sensors are valid.

Finally, the Context Manager uses WordNet to infer *relationships between locations of things*, taking advantage on the semantic relationships described by WordNet for all words: hypernymy/hyponymy (the semantic relation of belonging to a higher/lower class), and holonymy/meronymy (the semantic relation that holds between a whole and its parts/a part and the whole). Again, inference can be applied at the time of the execution of applications. In this case, we have defined a procedure that uses the meronyms (or the hypornyms if no meronym is found) of the provided word, in order to determine if it is represented by any narrower concept present in the ontology: e.g., if an application asks for the temperature at home, a temperature sensor installed in the kitchen is valid (*kitchen* is meronym of *home*). Inference can also be applied when registering things in the ontology. In this case we have defined a procedure that uses the holonym (or the hypernym if no holonym is found) of the provided location, in order to determine if the word is represented by any broader concept present in the ontology: e.g., a new thing located in a square would be added as a son of the word city (*city* is holonym of *square*). All synsets inserted in the ontology are stored following these relationships through SKOS[13] (Simple Knowledge Organization System),

[6] http://www.w3.org/2006/time
[7] http://www.w3.org/2005/Incubator/ssn/ssnx/cf/cf-property
[8] http://www.w3.org/2005/Incubator/ssn/ssnx/meteo/phenonet
[9] http://purl.oclc.org/NET/muo/muo
[10] http://www.fipa.org/specs/fipa00091/PC00091A.html
[11] http://www.opengeospatial.org/standards/geosparql
[12] http://www.geonames.org/ontology/ontology_v3.1.rdf
[13] http://www.w3.org/2004/02/skos/intro

$$hasSameType(?x1, ?x2) \quad hasSameLocation(?x1, ?x2) \rightarrow isEquivalent(?x1, ?x2)$$

$$isCombinable(?x1, ?x2) \quad hasNumericMeasurement(?x1, ?x2) \rightarrow hasAverageOperator(?x1, ?x2)$$

$$isCombinable(?x1, ?x2) \quad hasBooleanMeasurement(?x1, ?x2) \rightarrow hasOROperator(?x1, ?x2)$$

Fig. 2. Semantic Rules.

which is a knowledge organization system that offers a common data model to organize classifications in a hierarchical way.

Commenting on the implementation, we would like to point out that WordNet related inference is embedded in the code, and no reasoner is needed in this case. Besides, as WordNet does not offer an online web service, we have downloaded and installed its latest available dictionary within the Context Manager. We have used the MIT Java WordNet Interface[14] library for interfacing with that dictionary.

B. *Inferring Thing Service equivalence*

Apart from the inference mechanisms based on WordNet, we also use semantic reasoning techniques to infer knowledge. The aim is to deduce information which is not explicitly reflected in the ontologies and that cannot be inferred from WordNet synsets. To achieve this goal, the Context Manager uses a semantic reasoner (more precisely, the RDFS rule reasoner of the Jena Framework) and a set of rules. The use of semantic reasoners is widely accepted in IoT environments [4]. Given a set of rules, a semantic reasoner detects which of those rules should apply in a particular time and what is the result of their application. We have analyzed the IoT scenario to detect different rules to be applied. From the analysis of these scenarios, the existence of different types of situations can be deduced: (i) situations where there is more than one suitable thing service to be used (e.g., two presence sensors installed in the bedroom); and (ii) situations where there is a need to combine different services to create a correct response (e.g., an application to detect presence in the home has to combine information from different presence sensors installed in different rooms of the house). From the analysis of these scenarios, we have defined three rules to detect:

1. *equivalent thing services*, which are those associated to things of the same type in the same location. As a result of the execution of this rule, the Context Manager is able to generate lists of equivalent thing services for a particular application; and
2. the *operator* to be applied when things of the same type in different locations have to be combined.

Fig. 2 summarizes the rules we have defined. Though implemented in RDF, here they are expressed in SWRL for the sake of a better understanding.

Based on the requirements of each application, the Context Manager determines the list of thing services involved, the operator to be applied to combine those thing services (if needed) and the list of equivalent thing services found (if any). The decision on which equivalent thing service to use for a given set is taken by the QoS Manager according to the QoS requirements of the application and the status of connected things, as explained in the next Section.

[14] http://projects.csail.mit.edu/jwi/api/

IV. QoS-aware Thing Service allocation

As previously mentioned, we assume that the platform allows the negotiation of the QoS for a required thing service. To this aim, a standard interface for QoS negotiation is exposed to applications by the QoS Manager, which can negotiate the values of the QoS parameters associated to each class of service. For instance, an application that requires real-time temperature measurements can negotiate the *maximum response time* in order to receive updates from a sensor that replies within the specified maximum delay. The use of a standard interface entails the definition of a QoS model to classify the different possible QoS requirement configurations, which however is outside the scope of this work (for a concrete example, please refer to the QoS model defined for the BETaaS platform [5]).

In case the QoS requirements cannot be satisfied by any of the devices or the status of the platform does not allow the application to be accepted, the negotiation is unsuccessful. On the other hand, once the negotiation phase is performed successfully, a Service Level Agreement (SLA) is established and applications can invoke the thing service with the negotiated QoS level. For a detailed description of the interface exposed and the negotiation procedure adopted by the platform, we refer the interested reader to [5]. The main functional components of the *QoS Manager* are illustrated in Fig. 1 (right side), and correspond to the two-phase procedure, namely, *negotiation* and *invocation*, implemented by the platform.

The *negotiation* phase is handled by a sub-component called *QoS Broker*, which manages QoS *negotiation* by performing *admission control* and, most importantly, *managing resource reservation* by exploiting equivalent thing services. The QoS Manager receives from the application a description of the required thing service and the QoS requirements. The description is forwarded to the Context Manager that infers the lists of equivalent things and, if required, the operator to be applied. Based on the currently granted allocations, the capabilities of the available things and the application QoS requirements, the *QoS Broker* evaluates if the application request can be fulfilled or not, and a corresponding SLA can be created accordingly. To this aim, the *QoS Broker* implements a specific reservation algorithm that selects among multiple feasible allocations the one optimizing a carefully chosen objective function. For instance, an algorithm that minimizes the energy consumption of battery-powered devices, thus maximizing the lifetime of the system, has been described and evaluated in [6].

The *invocation* phase, instead, is managed by the *QoS Dispatcher*, which is responsible of allocating each single invocation to the corresponding resources. When a thing service is invoked, the *QoS Dispatcher* selects the actual thing that will provide the service to the application based on the reservations provided by the *QoS Broker* in the previous phase. This two-step procedure allows the *QoS Dispatcher* to react to changes, modifying the allocation through the exploitation of equivalent

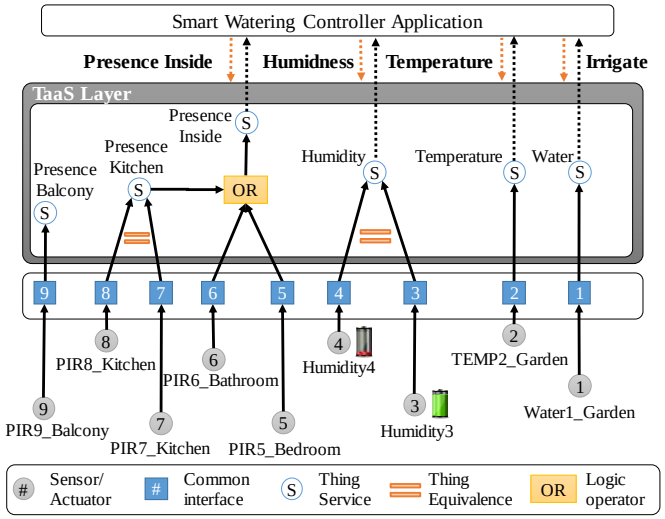

Fig. 3. Validation scenario.

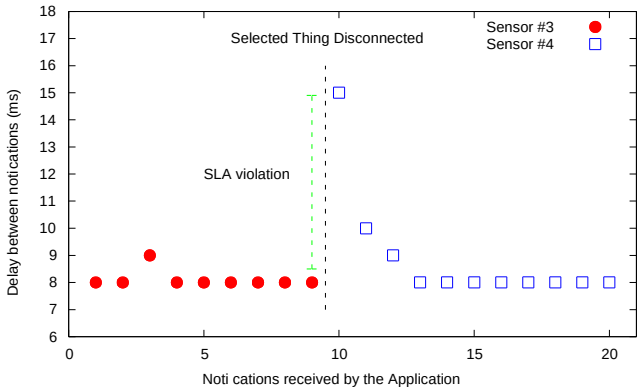

Fig. 4. Notification delay over time.

things, thus ensuring service continuity and resiliency to failure or temporary outage of smart things.

To detect changes in the environment, the QoS Manager includes a sub-component, the *QoS Monitoring*, which is responsible for checking the status of smart things. In fact, the capability of handling the dynamicity that characterizes IoT devices is another requirement of paramount importance, considering that smart things can be battery-powered devices connected through unreliable wireless means, hence can be often unavailable or unreachable. Service continuity and failure management will be demonstrated by a simple experiment in the next section.

V. Validation

In this section, the proposed approach is validated through an experiment that reproduces a smart home scenario. The goal of this validation is twofold: to demonstrate that the context management solution successfully infers a relationship of equivalence among things, and to show how this knowledge is exploited by the QoS Manager to efficiently allocate resources, providing service continuity in case of thing unavailability.

The scenario of the experiment is illustrated in Fig. 3. The platform is connected to the sensors of a smart home system, which includes one *valve* actuator to trigger the *watering* of the garden, one *temperature* sensor placed *outside* in the *garden*, two battery-powered *humidity* sensors placed in the *garden*, and five *presence* sensors, four placed *inside* (two in the *kitchen*, one in the *bathroom* and one in the *bedroom*) and one placed *outside* in the *balcony*. The overall experiment has been carried out on an Intel® Quad-Core i7 @ 3.40GHz. Sensors are emulated by means of CoAP servers, each one running in a separate process, where each sensor provides a context description using natural language. The proposed platform has been implemented in OSGi and runs inside the BETaaS platform [3]. A *smart watering* application that controls the irrigation of the garden is implemented to validate the capabilities of the platform. In particular, the application logic triggers the garden watering system through an actuator if (*i*) the temperature is above a threshold, (*ii*) the humidness is above a threshold, and (*iii*) there is no presence inside.

The experiment goes through the following steps: (*i*) sensors are connected to the platform, the context is retrieved and parsed by the platform, (*ii*) the smart watering control application connects to the platform and requests the required thing services through natural language, (*iii*) a malfunction is simulated disconnecting one of the humidity sensors. When CoAP sensors are connected (step *i*), context information is directly provided by each device using the CoRE link format. Context information is translated into the common format adopted internally by the platform as in Table 1. When the application performs negotiation (step *ii*), four thing services are requested: (*i*) *irrigate*, (*ii*) *presence inside*, (*iii*) *humidness*, and (*iv*) *temperature*. For each thing service, the context manager determines the list of equivalent things that can provide the service. In case the thing service is the result of a composition of different things, the context manager also derives the operator in addition to the equivalence relationships.

In our experiment, the platform successfully identified as equivalent the two presence sensors installed in the kitchen (sensors #8 and #7), and the two humidity sensors installed in the garden (sensors #4 and #3). In particular, the platform disambiguated all synonyms terms found: the nouns ‘humidity’ and ‘humidness’, used within the context description of the sensors and within the description of the thing services required by the application, respectively; and the verbs ‘water’ and ‘irrigate’, being the first the contextual description of the actuator and the second the description of an application requirement. In addition, the platform correctly derived the ‘presence inside’ thing service as the composition by the ‘or’ operator of all presence sensors but the one installed in the balcony, as it is installed outside. As the application starts and thing services are invoked, the QoS Manager runs the allocation algorithm on each list of equivalent thing services to select the actual thing providing the service. In our experiment, in particular, the humidity sensor with the largest battery residual capacity (sensor #3) is selected, as it is the most convenient allocation to maximize the battery lifetime. At the time of disconnection of sensor #3 (step *iii*), the QoS manager automatically executes the allocation algorithm to find another suitable allocation that guarantees the continuity of the service provided to the applica-

tion. In this case, right after the disconnection, sensor #4 is selected.

In order to highlight this behavior and demonstrate service continuity, the notification delay for the humidity thing service, measured as the time between two consecutives notification messages delivered to the application, is shown in Fig. 4. Different colors of the marks identify different service providers, respectively blue for sensor #3 and red for sensor #4. As can be seen, the allocation algorithm successfully reacts selecting sensor #4 as the thing service provider after the disconnection of sensor #3. This event, in particular, causes a temporary peak in the notification delay, which results in a temporary SLA violation. This behavior can be explained considering the overhead introduced by the operations performed by the platform to react to the failure.

VI. Related Work

We are aware of other solutions that address semantic interoperability and context awareness, like OpenIoT[15], VITAL[16], Fiesta[17] or City-Pulse[18] (all developed in EC projects), and even of initiatives emerging in industry like Hyper/CAT[19] or IoT Database[20], but unlike the one presented in this work, all of them require an agreement on a unified vocabulary or taxonomy to describe IoT data. However, as it is explained in the previous sections, we rely on WordNet to propose a solution to describe IoT data using Natural Language. There are larger lexical repositories that overlap with WordNet as ConceptNet[21] or OpenCyc[22], but we consider them less accurate, because, unlike WordNet, they have not been created by linguists. Other common natural language solutions to add metadata to unstructured text are text corpora and Open Calais[23]. We discarded using a text corpus, since they lack information about lexical relationships between words. Equally, we discarded using Open Calais because it is more oriented to social media analysis. Besides, both solutions require manually tagged training data, while WordNet is able to obtain training data automatically [7]. To the best of our knowledge, the architecture proposed in this work is the first to exploit context-awareness functionalities through Natural Language, to provide QoS support to application, and to efficiently manage resources exploiting equivalence among things.

Several proposals aimed at introducing QoS support have been proposed for Service Oriented Architectures (SOA) where QoS is a crucial requirement. However, approaches defined in the context of SOA systems lack of support for constrained devices that introduce new non-trivial issues and would require significant modifications. To the best of our knowledge, a first attempt to address such issues has been done in [8], where the use of web services for sensors integration is proposed. However, this approach aims at implementing a service-oriented middleware directly on the nodes, which is not always feasible due to their constrained environment. To overcome these limitations, in [9] an adaptable middleware is proposed; middleware functionalities can be configured to reduce their complexity in case of constrained devices such as sensors. The proposed solution exposes a SOA interface to applications in which a flexible QoS support is provided by means of Service Level Agreements (SLAs) between the applications and the middleware. This solution, however, is specifically tailored to WSNs.

[15] http://www.openiot.eu
[16] http://vital-iot.eu/
[17] http://fiesta-iot.eu/
[18] http://www.ict-citypulse.eu/
[19] http://www.hypercat.io/
[20] https://iotdb.org/
[21] http://conceptnet5.media.mit.edu/
[22] http://sw.opencyc.org/
[23] http://new.opencalais.com/

VII. Conclusions

In this work we have presented a novel uniform platform designed to include context-awareness functionalities and showed how such functions can be exploited to automate search and selection of things through natural language. In addition, an advanced resource management technique that exploit context information to infer equivalence among things is presented as mechanism to efficiently enforce application QoS requirements. The proposed approach is demonstrated experimentally showing how the proposed approach allows detecting equivalent things and ensure service continuity.

Acknowledgment

This work has been carried out within the activities of the project “Building the Environment for the Things as a Service (BETaaS),” which is co-funded by the European Commission under the Seventh Framework Programme (grant no. 317674).

References

[1] O. Vermesan et al., (2011). Internet of things strategic research roadmap. Internet of Things-Global Technological and Societal Trends, 9-52.

[2] P. Guillemin, F. Berens, O. Vermesan, P. Friess, M. Carugi, G. Percivall, “Internet of Things: position paper on standardization for IoT technologies”, January 2015, online at http://www.internet-of-things-research.eu/pdf/IERC_Position_Paper_IoT_Standardization_Final.pdf.

[3] C. Vallati et al., “BETaaS: A Platform for Development and Execution of Machine-to-Machine Applications in the Internet of Things,” Wireless Personal Communications 87(3): 1071-1091 (2016).

[4] M. Serrano, J. Parreira, P. Cousin, P. Barnaghi, “IoT Semantics for standardisers”, presented at IERC AC4, October 2012, online at http://www.probe-it.eu/wp-content/uploads/2012/11/OpenIoT_WPA_PST_ETSI-IERCAC4Standardisers_Short_20121023_Updated-shrunk.pdf

[5] E. Mingozzi, G. Tanganelli, C. Vallati, “A framework for QoS negotiation in things-as-a-service oriented architectures,” 2014 Int. Conf. on Wireless Communications, Vehicular Technology, Information Theory and Aerospace & Electronic Systems (VITAE), May 2014.

[6] G. Tanganelli, C. Vallati, E. Mingozzi, “Energy-Efficient QoS-aware Service Allocation for the Cloud of Things,” 2014 Int. Conf. on Cloud Computing Technology and Science (CloudCom), Dec. 2014.

[7] C. Leacock, G. A. Miller, M. Chodorow. Using corpus statistics and WordNet relations for sense identification. Comput. Linguist. 24, 1 (March 1998), 147-165.

[8] F. C. Delicato, P. F. Pires, L. Pinnez, L. Fernando and L. F. R. da Costa, "A flexible web service based architecture for wireless sensor networks," Int. Conf. Distributed Computing Systems Wksps, 2003, pp. 730-735.

[9] G. F. Anastasi, E. Bini, A. Romano, G. Lipari “A service-oriented architecture for QoS configuration and management of Wireless Sensor Networks,” IEEE Conf. Emerging Technologies and Factory Automation (ETFA), Sept. 2010.