Context-aware design of reflective middleware in the Internet of Everything

Marina Mongiello, Tommaso Di Noia, Francesco Nocera, Eugenio Di Sciascio, Angelo Parchitelli

Politecnico di Bari, Via Orabona, 4, 70125 Bari, Italy
{firstname.lastname}@poliba.it

Abstract. We daily experience the interaction with physical objects which are becoming smarter and smarter with the ability to communicate with each other as well as with different information systems. While, on the one hand, we are assisting to the rise of a pervasive Internet of Things (IoT) or an Internet of Everything (IoE), on the other hand we face the need of a new generation of objects able to adapt to external inputs coming from the environment they are dipped in.
New modeling techniques, pattern and paradigm for composing and developing software and services able to deal with changing context and requirements are necessary.
Self-adaptive systems are modern applications whose running part should be able to react on its own, by dynamically adapting its behavior, in order to sustain a required set of qualities of service, and dynamic changes in the context or in the user requirements. Here, we propose a solution allowing a IoT Middleware to conform to Reflective programming paradigm thus giving more flexibility and adaptability to the network behavior.

1 Introduction

The Internet of Things (IoT) and its extension Internet of Everything (IoE) is for sure one of the most influential technological shift we are facing in the last years. Thanks to the spreading of sensors and to the diffusion of low-cost miniaturized computational resources, we are able to make objects that produce data and interact with each other thus producing a network of interconnected things, data, processes. IoE makes possible to distribute data sources and data consumers in the real world to produce, collect, exchange and consume information usually with a high rate. Elements of the IoT are quite evolved piece of hardware that may be programmed and covered by a software layer implementing complex functionalities.

One of the challenges in building Internet of Services and Things is the way software will be developed and composed on the top of flexible infrastructures and integration architectures.

Despite the great interest in software composition and verification methods, when developing service- and thing-based software systems, strong challenges still regards the way a smart IoT-based architecture is designed in order to make it robust with reference to the contextual changes it continually undergoes. Therefore, new software composition paradigms and patterns that deal with heterogeneity, dynamicity, adaptation are needed.

We all known that the physical world is not a static environment and it changes accordingly to sometimes unpredictable events. Then, informative objects which are dipped in the physical world should be able to adapt and change their behavior accordingly to the surrounding context. This means that the software they have on board should be able to react in order to change and modify its functions to comply with the new environmental variables. From the point of view of a software architecture, IoT poses many interesting challenges due to its unpredictable yet adaptive requirements. In order to be as effective as possible, in IoT solutions the intelligent objects are usually coordinated by a middleware that acts as a facilitator towards a smoother and homogeneous communication among the various components. A huge work is available in the literature about IoT middleware. Interesting and structured surveys are in [2, 7, 15, 10]. IoT middleware is also facing multiple challenges [6, 1] that are mainly induced by IoT features. In this paper we propose a reflective extension of an IoT middleware. The approach aims to show a possible solution to make a IoT middleware conforming to the pattern of reflective programming, which allows a software system to dynamically change its logic without internal changes to the code.

The remainder of the paper is structured as follows: in the next section we describe the a formal model of the approach we propose. Section 3 describe a practical example of a use case scenario. In Section 4 the formal model is instantiated in the use case scenario. The details of the reflective implementation are provided in Section 4. Conclusion and future work close the paper.

2 Modeling Reflective Middleware

While designing a IoT solution, the use of traditional methods may lead to excessive complexity of the code that may become slow and/or little maintainable. Among the different patterns available to design a software system, *Reflection* allows the developer to produce an extensible architecture from the beginning. A IoT middleware can surely benefit from the use of a reflective approach. As an example, we have the possibility of designing a completely configurable system and adaptable to different operating environments.

The main concept in *Reflection* pattern is the distinction between *base-level* and *meta-level*. A base level includes the core application logic. Its runtime behavior is observed by a meta level that maintains information about selected system properties to make the software self-aware. Changes to information kept in the meta level thus affect subsequent base-level behavior. [4] The adaptation of the meta level is performed indirectly with help of a specific Interface, *M*eta-*O*bject *P*rotocol (*MOP*). This allows users to specify a change, checks its correctness, and automatically integrates the change into the meta level. [5] MOP also makes possible the change of connections between the *base-level* and *meta-objects*. A *meta-object* is an object that manipulates, creates, describes, or implements other objects (including itself). Thus, a proper configuration of the base-level and meta-objects defines the behavior of an application. [11] In addition, using a programming language that supports reflection, it is possible to change the structure of the objects themselves at run-time and then make the software system more flexible.

In this Section, we define a formal model of a reflective middleware for Internet of Things. Let S be the set of sensors $S = \{s_1, s_2, ..., s_n\}$ where each s_i is the stimulus, i.e. the physical, biological, chemical etc, quantity that the sensor detects and $D = \{d_1, d_2, ..., d_n\}$ the Domain of sensor's output for each sensor in S.

Definition 1. *(Condition). Given S and D, a Condition C is a relation R between s and d where $s \in S$ and $d \in D$ or a boolean combination of relations between s and d.*

For example, suppose sensor s_i detects changes in concentration of pm_{10} in the air, a condition may be: IF pm_{10} GREATER THAN 35 or IF HUMIDITY MINOR THAN 15 AND pm_{10} GREATER THAN 35.

Let $AR = \{arg_1, arg_2, ..arg_n\}$ a set of arguments that can applied to given operations:

Definition 2. *(Action). Given a set $A = \{a_1, a_2, ..a_n\}$, an Action a_i is the operation to be performed when a Condition occurs, applied to the argument arg_j, $a_i(arg_j)$.*

For example: if an increase in the fine dust emission is detached, an *action* necessary for managing the problem resolution may be sending an e-mail to an office responsible for safeguarding, the *action* is notification, the *argument* is sending e-mail.

Definition 3. *(Rule). A Rule is defined as a function associating Condition to an Action $R : C- > A(arg)$.*

Let B be the MESSAGE BROKER component that models the Publish/Subscribe mechanism for defining the relationships between physical sensors and actions:

Definition 4. *(Message). A message m_i is defined as $m_i =< s_i, d_i, a_i, arg_i >$ where s_i is the sensor, d_i the sensor's output, a_i the given action and arg_i the argument passed to the action a_i.*

Definition 5. *(Message bus). The Message Bus is the channel where the* MESSAGE BROKER *publishes messages from sensors and receivers subscribe to receive notifications of a relevant message.*

The proposed model maps an Iot middleware on the three levels of a classical reflection pattern according to the following matching: the *Condition* to the *Meta level*, the *Action* to the *Base level* and the *Rule* to the *Meta-Object Protocol*. RULES are modeled and stored in a RULE-BASED SYSTEM. A RULE ENGINE models the reasoning algorithm that automates the rules R. Figure 1 shows a graphical schema of the proposed approach using an abstract architecture. The architecture models the main elements of our approach that are represented as components. The abstract architecture can be instantiated in several contexts and scenarios. Conditions are represented at the condition level where conditions are defined, Actions at several action levels, the rule engine that implements the reasoning algorihtm on the rules. The ADAPTOR component works as a driver and translates the received command in a real action. It is possible to use place more Adaptor, one for each desired application.

Steps for performing the reflective mechanism:

- Step 1: Sender publishes messages on the *Message Bus*.
- Step 2: Receivers subscribe to the *Message Bus*.
- Step 3: The MESSAGE BROKER passes the message m_i through the *Message Bus* to the *Condition level*.
- Step 4: The RULE ENGINE executes the rule.
- step 5: An *Action level* is activated.

The reflection mechanism is activated precisely in the last one step. Especially the Reflection is implemented through rules triggered by the messages.

3 Use Case Scenario

To explain the formal model let us consider its instantiation in a use case scenario, i.e. consider a smart environment, specifically in a Smart city domain. Particularly, in this domain management of energy, environmental protection and mobility are some interesting applications of Internet of Everything. The devices can be spread over the whole area of urban land to monitor, through appropriate sensors, meteorological and environmental variables as depicted in Figure 2.

Typically variables to be monitored are air, temperature, humidity, fine dust emissions.

Other variables to be monitored are related to the use of municipal services such as the position of city buses at their respective sections, the residual capacity of the car parks, the urban lighting control. Suppose now that a network of sensors monitors the city traffic and at the same time the fine dust emissions.

A slowdown occurs in an area of the town, the data is sent from the controller to the middleware that analyzes the data and compares it with a set of pre-established rules. The controller will trigger a series of procedures to solve the slowdown (special signals indicating for alternative routes are lighted on). Suppose now that there is an increase in the fine dust emissions: the constant monitoring unit sends the data to the middleware which will activate the necessary actions for the resolution of the problem. The two main action to solve the critical events maybe: sending an email to the municipal section for safeguard of environment and of the land or to the section for land and infrastructure and mobility.

4 Instantiation of the model

In this section we instantiate the model defined in Section 2 in the use case scenario described in Section 3.

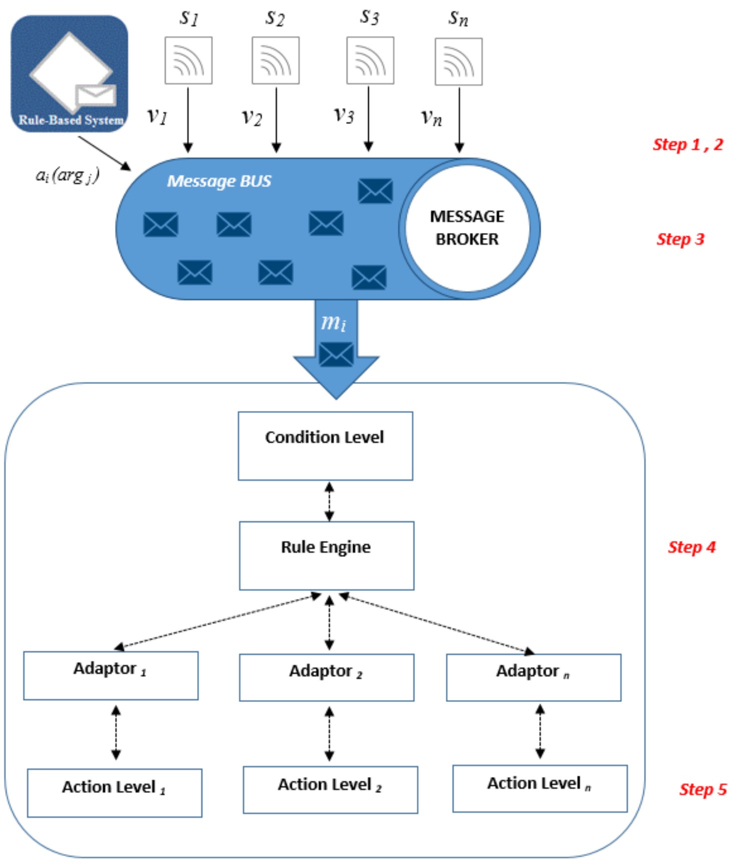

Fig. 1. Graphical schema of proposed model.

The DeviceHive middleware. We choose DeviceHive as IoT middleware for several reasons. Ease of installation, the rich documentation and the high integration with a wide range of programming languages and IoT protocols. By using DeviceHive, the set up of a IoT system is made by the following three steps:

– Create: the user must create an instance of the *Cloud by DeviceHive*. There are instances of *Microsoft Azure*, *Juju*, *Docker* and *Cloud Playground*. In our implementation, we used the shell instance *Cloud Playground*.
– Connect: using the specific IoT Toolkit, a dedicated gateway is installed. This is the link between the devices and the cloud of DeviceHive. The gateway is written in Go, and in our case communicates with a Python script that is responsible for receiving the data from the sensors within the network.
– Visualize: sent data can be displayed via a Web page.

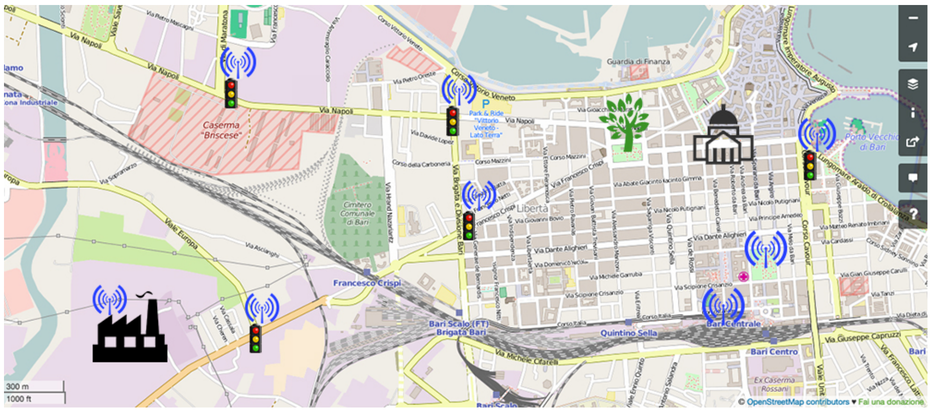

Fig. 2. Smart environment scenario.

Redis as a Message Broker. In our instantiation of the model we adapt Redis as MESSAGE BROKER. Redis[1] is a NoSQL DBMS. It is based on the key-value paradigm but has some characteristics that make it different from other database of its own category: it works in RAM, provides support to the storage of the pair key-value, it offers four data structures: lists, sets, ordered sets and hash. We used Redis as MESSAGE BROKER, which allows to translate a message from the messaging protocol of the sender to the recipient's messaging protocol. Through appropriate Publish/Subscribe directives Redis implements the mechanism of Publish/Subscribe, whereby senders do not send messages directly to recipients; rather the messages are published and sorted into channels, without knowing who is really writing to the channel. Interested parties express their intention to subscribe to notifications of one or more channels of interest. This decoupling between publishers and subscribers allows for greater scalability and a dynamic network topology.
Observer. An OBSERVER component observes rules extracted from the Rule based-systems. In our implementation, the rule based systems is a config file. The OBSERVER (of the data flow) notifies the MESSAGE BROKER about the arrival of new data and the active rule.
Event-driven component. We use Node JS as Event-driven component. It receives data about the sensor and the variable of the sensor through a REST interface.
Message Bus. Instantiation of the *Message Bus* of our model is obtained using the Redis Message channel. To implement the Publish/Subscribe mechanism, publishers publish messages on the Redis channel, recipients subscribe on the channel and read information about the available services.
Condition Level. The meta-level of the reflective mechanism is the *Condition Level* of our model. In the instantiation of the model it contains a simple call to a generic function, with the string identifying the function and the class of the function. In is instantiated in a separate component of our reflective component connected to the RULE

[1] http://redis.io/

ENGINE. According to this mechanism, the logic of the program can be dynamically changed depending on the data got from the sensors and changes are transparent to the internal structure of the software. The *Condition Level* can be extended with the meta-objects to develop considerably more advanced functionality external to the server. In fact, it is sufficient to connect the component to the Redis server for listening to the messages coming from the server. The *Condition Level* has access to basic levels via the object DBCLASS containing instances of basic objects identified by a particular name which is specified in the configuration file of the *Action Level*, that will be eventually analyzed. In the message received by Redis the class name and the name of the function to call are specified in special strings. The data to be written is contained in a separate object, which will be the argument of the function.
Rule Engine. A RULE ENGINE component implements the algorithm for the extraction of the action from the rule, given a condition.
Adaptors and Action level. In order to determine which are the *Action Levels* available at runtime, a system for loading dynamic components within the same *Condition Level* has been implemented. Two files for each *Action Levels* are available: a JSON file containing configuration data and a JS file containing the class itself. Objects instantiated with the data in the JSON file will then be stored in an array, and referenced by the name of the same class, specified within each rule.

5 Reflective implementation of the middleware architecture

In this Section we analyze the flow of control in the implementation of the use case scenario. In our implementation, the system will automatically perform actions according to the values received by the sensors of devices by matching the set of formal rules.

The hardware used to provide a source of data is based on the *Arduino* platform, a small electronics board equipped with a microcontroller, which is used to quickly achieve hardware prototypes. In addition to the *Arduino* board we have used the *RaspberryPi B+* board to send and receive data from/to the local server which then makes a transition of data to *Cloud DeviceHive* servers.

In Figure 3 the architectural scheme of the proposed solution is depicted. Within the component, a configuration file will contain the definition of the rules on the data and the actions to take if the rule is checked. The OBSERVER (of the data flow) notifies the NODE JS that forward to the MESSAGE BROKER about the extracted active rule. The active rule together with the information about sensors and variables, (the predicate of our Condition) is published on the *Message Bus*. On the message channel is published information about the arrival of new data. The MESSAGE BROKER works as a through for data flow and the active rule that are forwarded to the Reflective part of our component.

We can analyze the exchange of information between the various components involved in which the environmental monitoring unit sends a message to the middleware with the following form:

```
{"device":"Arduino_Raspberry_PM10_sensor",
 "value":40,
 "sensor":"pm10_sensor",
 "timestamp":"1452357172"}
```

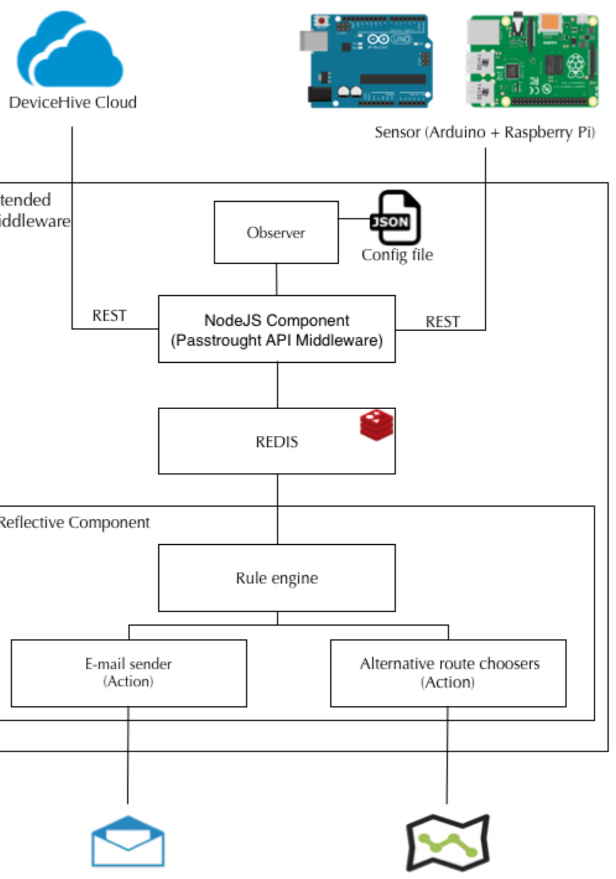

Fig. 3. Architectural schema of proposed scenario.

The message reaches the middleware where it is published on the *Message Bus*; now it has the following form:

```
{"id":"EJH8rmcDl",
 "rule":"notification",
 "rule_where":"send_email",
 "data":{"device":"Arduino_Raspberry_PM10_sensor","value":40,
         "sensor":"pm10_sensor",
          "timestamp":"1452357172"}
}
```

Receivers subscribe to the MESSAGE BROKER and are notified of the message. The MESSAGE BROKER forwards the whole message made up of sensor, variable, (the predicate of the condition) and the rule, (extracted from the config file) to the Rule

Engine that executes it. Reflection is applied thanks to the signing of the receiver component of the Redis channel. This component will read the published messages that contain specific information on the function to be called, the class of which the function is a member and topics. The *Condition Level* and the RULE ENGINE contains enable the action level of the reflective component. The RULE ENGINE contains among other specifications a simple call to the generic function to notify to execute the email application to communicate the proper recipient the exceeding the threshold with respect to the value contained in the catalog of rules.

Implementation of the *rule* is:

```
{
    "id" : "Arduino_Raspberry_PM10_sensor",
    "sensors" : ["pm_10_sensor",
    "temperature_sensor"],
    "rules" : [
        {
            "sensor": ["pm10_sensor"],
            "action": "notification",
            "argument": "send_email",
            "variable_action" : {
                "condition" : "moreThan",
                "value": "35
            }
        }
    ]
}
```

In this case the PM10 value greater than admitted threshold value of 35. The level of detail in the rules of the system is of the single sensor and can group multiple sensors in logic ”AND” between them. The activations are managed via a plugin system. Each plugin is self-consistent and controls an appropriate device in Plug & Play mode.

In Figure 4 we show the class diagram of our architecture. Our component is placed between the devices and the *Cloud DeviceHive*.

The wrapper replaces the middleware for passing the requests delivered from devices that will connect to our component, instead of connecting directly to the middleware. This part of the server, implemented by classes `GenericServer` and `Server`, contains a simple HTTP server. Performed steps are:

1. receives the HTTP request from the gateway and analyzes the content;
2. composes a new HTTP request to the server DeviceHive, creating all the headers required by the function `getDHHeaders`;
3. as soon as the server has received the answer from DeviceHive, forwards it to the gateway of the device.

Note that with this approach it may be also possible to implement a set of classes that replace the function `getDHHeaders`, pushing the `Strategy` design pattern, to process requests sent to the middleware always based on a REST approach.

Data sent by the gateway initially directed to the middleware, i.e. DeviceHive in our framework, are intercepted by the server. This latter performs a series of comparisons to evaluate if the appropriate component of base level will be activated. Comparisons are based on checking the rules in the Configuration File, each one related to one or more sensors present on a given device.

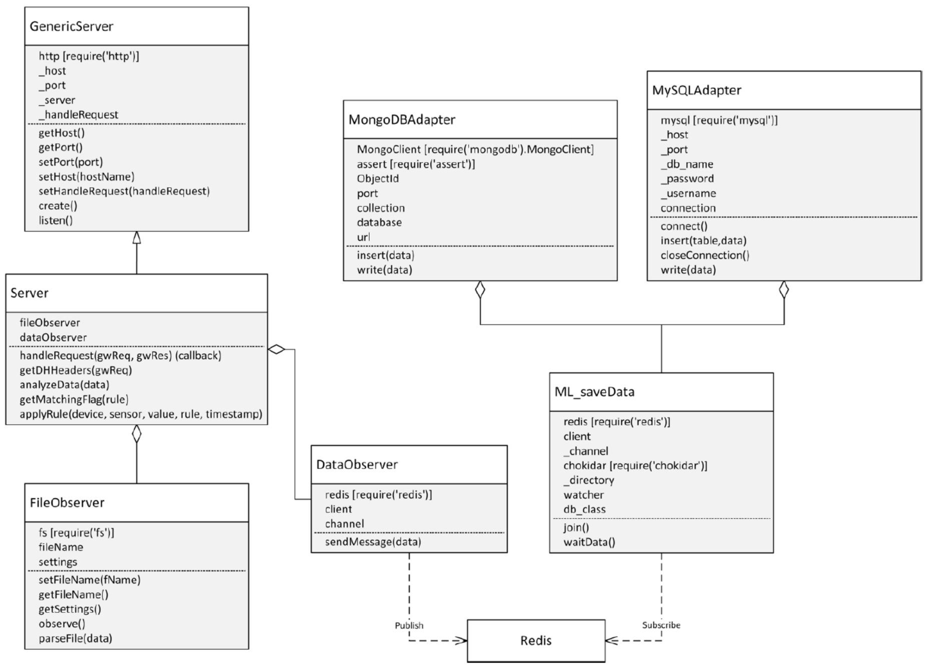

Fig. 4. Class diagram of the proposed architecture.

The class delegated to parsing the configuration file (FileObserver) scans the file and analyzes the rules. In fact, a single rule could be related to several sensors. In this way, given a NOTIFICATION message received by the gateway, it only needs to check if the condition specified in `variable_action` is verified. In case it is verified, the wrapper publishes a message on the Redis Publish/Subscribe server via a DATAOBSERVER class object. FILEOBSERVER parses the configuration file in case of modification while the server is running.

6 Related work

In this section we present a state of the art of existing approaches for reflective middleware. A survey of reflective middleware for Iot is in [13]. The survey addresses a broad range of techniques, methods, models, functionalities, systems, applications, and middleware solutions related to context awareness and IoT. The paper analyzes, compares and consolidates past research work. One of the first approaches is in [3] that presents an architecture for reflective middleware based on a multi-model approach. Through a number of working examples, they demonstrate that the approach can support introspection, and fine- and coarse- grained adaptation of the resource management framework. More recent relevant approaches of reflective middlewares are in SOAR (SOA with Reflection) [8]. [9] presents a chemical reaction-inspired computational model using the concepts of graphs and reflection, which attempts to address the complexities associated with the visualization, modelling, interaction, analysis and abstraction of information in the IoT. [12] presents Internetware which consists of a set of autonomous software entities distributed over the Internet, together with a set of connectors to enable collaborations among these entities in various ways. To support on-demand collaboration, Internetware middleware employs an RSA and reflection mechanisms on its own application server. [14] extends the Multinetwork INformation Architecture (MINA), a reflective (self-observing and adapting via an embodied Observe-Analyze-Adapt loop) middleware with a layered IoT SDN controller.

7 Conclusion and Future Work

In the recent years, there has been a huge effort to provide an immediate access to information about the physical world through Internet technologies. IoT vision aims to integrate the virtual world of information to the real world of things. The role of a IoT middleware is to provide the connectivity between the virtual world and physical world and an interface between heterogeneous physical devices and applications. We present a reflective extension of an IoT middleware which makes possible the design of a software resulting completely configurable and adaptable to different operating environments. The proposed framework enables to automatically perform actions according to the values received by the sensors by triggering a set of rules. Our current implementation relies on the commercial DeviceHive middleware and wraps it with an adaptable layer thus transforming the whole middleware in a reflective architecture. The approach is general enough and can be extended to deal with diverse functionalities other than the ones currently implemented. We are working to extend the proposed framework with the addition of new Plug and Play actions. Also we are working to extend the framework and enable it for the integration of several middlewares. The idea is to build an "adaptive wrapper" of middlewares able to adapt the reflective middleware to the different clouds.

Acknowledgment

The authors acknowledge partial support of PON03PE_00136_1 Digital Services Ecosystem: DSE. Furthermore, the author Francesco Nocera acknowledges support of Exprivia S.p.A Ph.D grant 2016.

References

1. Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. Role of middleware for internet of things: A study. *International Journal of Computer Science & Engineering Survey (IJCSES)*, 2(3):94–105, 2011.
2. Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. A survey of middleware for internet of things. In *Recent Trends in Wireless and Mobile Networks*, pages 288–296. Springer, 2011.

3. Gordon S Blair, Fábio Costa, Geoff Coulson, Fabien Delpiano, Hector Duran, Bruno Dumant, François Horn, Nikos Parlavantzas, and Jean-Bernard Stefani. The design of a resource-aware reflective middleware architecture. In *Meta-Level Architectures and Reflection*, pages 115–134. Springer, 1999.
4. Frank Buschmann, Kevlin Henney, and Douglas C Schmidt. *Pattern-Oriented Software Architecture, Volume 4, A Pattern Language for Distributed Computing*. Wiley, 2007.
5. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
6. Moumena Chaqfeh, Nader Mohamed, et al. Challenges in middleware solutions for the internet of things. In *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, pages 21–26. IEEE, 2012.
7. Ghofrane Fersi. Middleware for internet of things: A study. In *Distributed Computing in Sensor Systems (DCOSS), 2015 International Conference on*, pages 230–235. IEEE, 2015.
8. Gang Huang, Xuanzhe Liu, and Hong Mei. Soar: towards dependable service-oriented architecture via reflective middleware. *International Journal of Simulation and Process Modelling*, 3(1-2):55–65, 2007.
9. Ahsan Ikram, Ashiq Anjum, Richard Hill, Nick Antonopoulos, Lu Liu, and Stelios Sotiriadis. Approaching the internet of things (iot): a modelling, analysis and abstraction framework. *Concurrency and Computation: Practice and Experience*, 2013.
10. Valérie Issarny, Nikolaos Georgantas, Sara Hachem, Apostolos Zarras, Panos Vassiliadist, Marco Autili, Marco Aurelio Gerosa, and Amira Ben Hamida. Service-oriented middleware for the future internet: state of the art and research directions. *Journal of Internet Services and Applications*, 2(1):23–45, 2011.
11. Pattie Maes. Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, volume 22, pages 147–155. ACM, 1987.
12. Hong Mei, Gang Huang, and Tao Xie. Internetware: A software paradigm for internet computing. *Computer*, 12(6):26–31, 2012.
13. Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. *Communications Surveys & Tutorials, IEEE*, 16(1):414–454, 2014.
14. Zhijing Qin, Grit Denker, Carlo Giannelli, Paolo Bellavista, and Nalini Venkatasubramanian. A software defined networking architecture for the internet-of-things. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–9. IEEE, 2014.
15. Mohammad Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhan Clarke. Middleware for internet of things: a survey. *Internet of Things Journal, IEEE*, PP(99), 2015.