An Agent Architecture to Enable Self-healing and Context-aware Web of Things Applications

Rafael Angarita[1,2], Maude Manouvrier[2] and Marta Rukoz[1,2]
[1]*Université Paris Ouest Nanterre La Défense, Paris, France*
[2]*PSL Université Paris-Dauphine, CNRS, LAMSADE UMR 7243, 75775 Paris Cedex 16, France*

Keywords: Internet of Things, Web of Things, Self-healing, Fault-tolerance, Context-aware.

Abstract: The Internet of Things (IoT) paradigm promises to connect billions of objects in an Internet-like structure. Applications composed from connected objects in the IoT are expected to have a huge impact in the transportation and logistics, healthcare, smart environments, and personal and social domains. The world of things is much more complex, dynamic, mobile, and failure prone than the world of computers, with contexts changing rapidly and unpredictably. The growing complexity of IoT applications will be unmanageable, and will hamper the creation of new services and applications, unless the systems will show "self-*" functionality such as self-management, self-healing and self-configuration. The Web of Things (WoT) builds on top of the IoT to create applications composed of smart things relying on standard and well-known Web technologies. In this paper, we present a new agent architecture to enable self-healing and context-aware WoT applications.

1 INTRODUCTION

The Internet of Things (IoT) paradigm has gained ground, both in the industry and in research worlds (Atzori et al., 2010). It was also included by the US National Intelligence Council in the "Disruptive Civil Technologies - Six Technologies With Potential Impacts on US Interests Out to 2025" conference report (National Intelligence Council, 2008). The EU has invested more than 100 million euros in projects related to the IoT, and the government of China released the 12th Five-Year Plan for IoT development (Chen et al., 2014). Failures in IoT applications may lead to loss of production time, equipment damage, environmental catastrophes, or loss of human life (Alho and Mattila, 2015).

The world of things is much more dynamic, mobile, and failure prone than the world of computers, with contexts changing rapidly and unpredictably (Mattern and Floerkemeier, 2010). In the *IoT Strategic Research Roadmap* (Vermesan et al., 2011), Vermesan and his coauthors place autonomous and responsible behavior of resources as one of the fourth macro trends that will shape the future of the IoT. We extract the following:

> " ... the trend is towards the autonomous and responsible behaviour of resources. The ever growing complexity of systems will be unmanageable, and will hamper the creation of new services and applications, unless the systems will show "self-*" functionality such as self-management, self-healing and self-configuration."

In another IoT research directions paper, Stankovic stated that the areas of distributed and adaptive control are not developed well enough to support the open, dynamic environment of the IoT (Stankovic, 2014). Petersen and his coauthors argue that fault-tolerance and survivability play a key role in the designing of IoT applications (Petersen et al., 2015). They propose a mandatory "disaster mode" for IoT devices, allowing applications to continue working only with vital functionalities even in the presence of failures. Athreya and his coauthors suggest that the natural direction for IoT devices is to manage themselves in terms of software, hardware, and resource consumption (Athreya et al., 2013). Cherrier and his coauthors also underline the importance of fault tolerance, recovery, and coherence mechanism in the IoT applications (Cherrier et al., 2014). It is clear that fault-tolerance, resilience, self-healing, and other self-* research are very active

Angarita, R., Manouvrier, M. and Rukoz, M.
An Agent Architecture to Enable Self-healing and Context-aware Web of Things Applications.
DOI: 10.5220/0005932200820087
In *Proceedings of the International Conference on Internet of Things and Big Data (IoTBD 2016)*, pages 82-87
ISBN: 978-989-758-183-0

areas since they face new challenges in the IoT context due to its promise of connecting billions of devices in an Internet-like structure.

The Web of Things (WoT) (Guinard et al., 2011) builds on top of the IoT to create applications composed of smart things using standard and well-known Web technologies. In this paper, we present an agent architecture to enable self-healing and context-aware WoT applications. Our agents are the representation of physical objects, Web services, or humans in the Web, and they may be hosted and run inside physical objects or in a cloud infrastructure. In our architecture, an agent may also be responsible of managing and monitoring WoT applications.

The rest of this paper is organized as follows. We provide an overview of our proposed architecture in Section 2. We present a case study to illustrate our proposal in Section 3. This is followed by related work, and conclusions and future work directions in Sections 4 and 5, respectively.

2 OUR AGENT ARCHITECTURE

In this section, we present the main aspects of our agent architecture for self-healing and context-aware WoT applications. First, we describe the building block of our approach: the WoT Agent. Then, we present the WoT Application Manager, which is in charge of WoT applications. Before going into further detail, we present our definition of a WoT application.

Definition 1. *WoT Application. A WoT application is a composition of things linked by data or control dependency, where things may be physical objects, Web services, or human beings. A WoT application has an associated QoS (e.g., execution time) which is the aggregation of the participating things QoS.*

In our approach, all *things* participating in a WoT application, independently of their nature, are represented virtually by a WoT agent, which we define in the following paragraphs.

WoT Agent. Inspired from our previous work (Angarita et al., 2015), we introduce the WoT application component agent depicted in Fig. 1. It is in charge of executing an operation in a WoT application, communicating with peers, and applying fault-tolerance mechanisms if necessary. It may be implemented using Node.js, and its main components are the following:

- *API:* an interface to communicate with the WoT agent via JSON objects indicating the id of the WoT application, the sender, and data types and their values. A WoT agent may also act in a context-free way.
- *Core:* it contains the basic execution control elements of WoT application component agents; that is, waiting for inputs, invoking its corresponding operation, and sending produced outputs, if any.
- *Autonomic Component:* it represents the control loop of the autonomic computing (Psaier and Dustdar, 2011), and it detects local or global degradations on the application behavior, selects an appropriate action, and applies it.
- *Context Manager and Knowledge Base*: It handles information regarding the WoT applications the WoT agents is participating in. Its main components are:
  - *Knowledge Base:* it contains information described in RDF regarding WoT applications: rules indicating normal behavior and collaborators, requirement verification, self-healing actions, replacement operations, and replacement WoT agents.
  - *Inference Engine:* it infers logical consequences from a set of asserted facts in the knowledge base. Apache Jena may be used to reason over RDF graphs and query them using SPARQL.
- WoT Application Manager: stores and monitors WoT applications.
- Object Interface: it is in charge of the communication with its corresponding object.

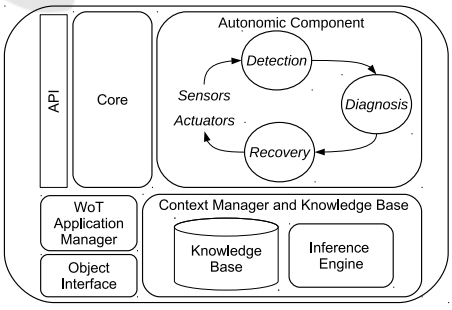

Figure 1: WoT Agent Architecture.

We present in detail the autonomic component of the WoT agent architecture in Figure 2. The *detection* component (Figure 2 (a)) takes into account one external and two internal data sources. The external information regards the expected QoS; for example, the WoT application manager may allow a certain QoS

degradation. The internal information refers to the QoS degradation or failure of its corresponding operation

The *diagnosis* component (Figure 2 (b)) analyzes the current degradation and computes a solution. The three possible diagnosis correspond to the three states of a self-healing system: normal; degraded; and broken. The choice of the recovery mechanism is influenced by available options (e.g., retry or replacement), and constraints imposed by the WoT application manager (e.g., expected QoS).

The *recovery* component (Figure 2 (c)) is in charge of applying the selected fault tolerance mechanism. A WoT agent may retry or replace its own operation, retry the communication with another WoT agent, or replace another WoT agent.

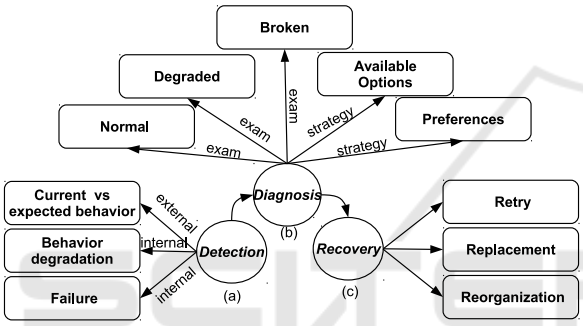

Figure 2: Autonomic Component.

WoT Application Manager. We present an overview of the WoT Application Manager in Figure 3. A WoT agent may manage a set of WoT applications consisting of its participants components and their data relationship, a set *rules* and *requirements* of the WoT application, a *triggering event* indicating when the WoT application starts its execution, a *disaster mode* specifying what to do in case of irreparable failures, its *historical* executions, and a *deployment protocol*.

The deployment protocol analyzes participating components, defines the WoT application execution mode, creates the necessary components, and sends them the required information. A WoT application manager may create agents to control participating components or communicate with existing agents. The deployment protocol contacts participating components and verifies their capacity to manage application context. A WoT application has the following execution modes (or a combination of them):

- Locally-hosted: WoT agents are hosted and run in the component managing the WoT application. This may happen when dealing with agentless components of a WoT application such as resourcesless objects or RESTful services, in the absence of a cloud infrastructure, and when the host component has enough resources to manage the WoT application.
- Cloud-based: WoT agents are hosted and run in a cloud infrastructure. This also may be the case when dealing with agentless components. Also, WoT agents may be already provided by their corresponding objects but hosted in a a cloud infrastructure.
- Distributed: WoT agents are managed and hosted by their respective objects.

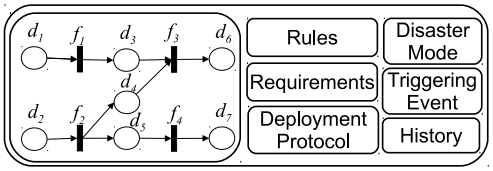

Figure 3: WoT Application Manager.

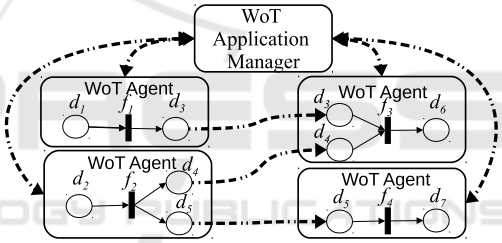

Figure 4: WoT Application Deployment.

Figure 4 shows a deployed WoT application. The WoT Application Manager sends the required information to all participant WoT agents, which may then communicate between them during the WoT application execution.

WoT Agent Failure. The corresponding operation of a WoT agent may fail; in this case, the failure may be fixed by operation retry or replacement, similarly as showed in our previous work (Angarita et al., 2012; Angarita et al., 2015). In the WoT application context, WoT agents may fail themselves, specially if they are hosted in mobile physical objects. In this case, the failed agent must be detected and the WoT application must be reconfigured. We propose two WoT agent failure detection mechanisms:

1. *Predecessor Detection:* when a WoT agent sends a message to another WoT agent, it waits for the status code `OK`. If it receives another status code, or it gives timeout, it may retry or replace its successor WoT agent, or execute the disaster mode.

2. *WoT application timeout:* if a WoT agent crashes during the executing of its operation, the WoT application manager eventually gives timeout, check which WoT agents are not alive, and performs retry, replacement, or executes the disaster mode. For further details about application timeout detection see (Angarita Arocha, 2015).

3 CASE STUDY

Figure 5 shows a fictional e-Health application we adapted from (Angarita Arocha, 2015). This application is built from 9 components, and it is installed in the mobile phone of a patient called Jenny; that is, the phone has a WoT agent which also plays the role of the WoT application manager, represented by $phone_{\diamondsuit}$ and $phone_{\blacklozenge}$. The WoT application triggering event specifies that it runs every 30 minutes. Jenny wears the *SugarImplant* and *VitalSignsImplant* smarts devices that gather information about her health. When the WoT application starts, $phone_{\diamondsuit}$ tells both devices to send their data to *SugarAnalysis* and *VitalSignsAnalysis*, which send their conclusions to *Diagnoser*. *Diagnoser* sends its results to the appropriate components depending on the necessary actions.

Table 1 shows the manager and location of each of the components according to the deployment protocol described in Section 2. *SugarImplant* and *VitalSignsImplant* are powerful objects hosting WoT agents. *SugarAnalysis*, *VitalSignsAnalysis*, and *Diagnoser* are not physical objects; however, they are provided as WoT agents hosted in a cloud infrastructure. *CallEmergency*, *NotifyContact*, *NotifyDoctor*, and *DisplayMessage* are RESTful services which do not have WoT agents. Their corresponding WoT agents are created in Jenny's mobile phone (*Phone*).

Table 1: WoT agents managers and hosts.

Component	Manager	Host
SugarImplant	*Itself*	*Itself*
VitalSignsImplant	*Itself*	*Itself*
SugarAnalysis	*Itself*	*Cloud*
VitalSignsAnalysis	*Itself*	*Cloud*
Diagnoser	*Itself*	*Cloud*
CallEmergency	*Phone*	*Phone*
NotifyContact	*Phone*	*Phone*
NotifyDoctor	*Phone*	*Phone*
DisplayMessage	*Phone*	*Phone*

To understand context-awareness, suppose that the application started its execution, *SugarImplant*, *VitalSignsImplant*, *SugarAnalysis*, and *VitalSignsAnalysis* analysis were successfully invoked. *SugarAnalysis* and *VitalSignsAnalysis* send their messages to *Diagnoser* (Listings 1 and 2). Note that both messages indicate the id of the WoT application they belong to. In this case, the id corresponds to Jenny's application. *Diagnoser* may receive data from other applications or processing context-free requests.

```
{ "id": "e427b92cfb07e68215110b6e8f7b",
  "sender": "SugarAnalysis",
  "data": [ {"name": "sugarResult1", "
      value": "someValue1"},
    {"name": "sugarResult2", "value": "
       someValue2"}]
}\vspace{-1.6mm}
```

Listing 1: SugarAnalysis Message.

```
{ "id": "e427b92cfb07e68215110b6e8f7b",
  "sender": "VitalSignsAnalysis",
  "data": [{"name": "vitalSignsResult1"
      , "value": "vitalSignsValue"}]
}
```

Listing 2: VitalSignsAnalysis Message.

When *Diagnoser* receives Jenny's data from *SugarAnalysis* and *VitalSignsAnalysis*, it performs its corresponding operation which may produce the following output: *NORMAL*, *WARNING*, or *EMERGENCY*. *Diagnoser* also contains the monitoring rules associated to Jenny's application showed in Listing 3. These rules are stored in the knowledge base of *Diagnoser* and verified by its autonomic component (see Figure 1). The rules showed in Listing 3 are all post-condition rules, but other kind may exist such as QoS monitoring rules.

```
id: "e427b92cfb07e68215110b6e8f7b"

if output = NORMAL, send Jenny's
    information to DisplayMessage;

if output = WARNING, send Jenny's
    information to DisplayMessage, and
    NotifyDoctor;

if output = EMERGENCY, send Jenny's
    information to DisplayMessage,
    NotifyDoctor, NotifyContact, and
    CallEmergency;

if FAILURE, execute DISASTER_MODE;
```

Listing 3: Jenny's Rules.

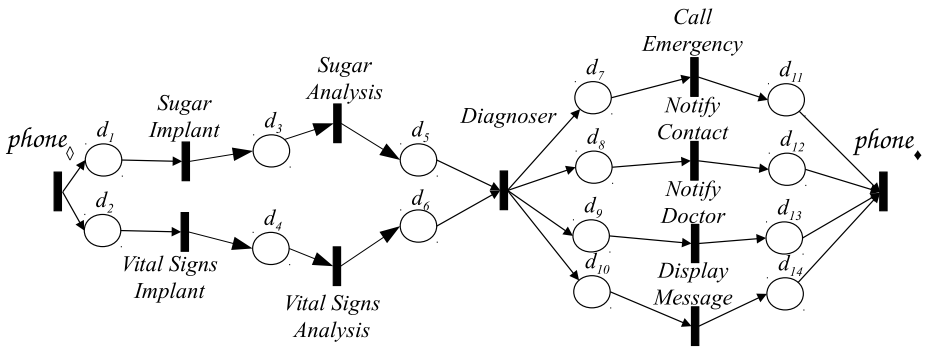

Figure 5: Case Study: the e-Health WoT application.

At the end of the execution, *CallEmergency*, *NotifyContact*, *NotifyDoctor*, and *DisplayMessage* send a message to *phone*♦ indicating they finished successfully.

4 RELATED WORK

In 2001, IBM published the *Autonomic Computing* (IBM, 2001) manifesto expressing their concerns about the inevitable increasing of the size and complexity of computer systems. For them, it was clear that such complexity of heterogeneous and distributed systems will minimize the benefits of future technology; therefore, solving the increasing complexity problem was the "next Grand Challenge". Two years later, we had the *Vision of Autonomic Computing* (Kephart and Chess, 2003) where the authors reaffirmed that the only solution to the software complexity crisis was through computing systems that can manage themselves. They presented the concept of self-management as the building block of autonomic computing, which includes four main aspects: *self-configuration*, *self-optimization*, *self-healing*, and *self-protection*.

Kephart and Chess described the self-healing property of autonomic systems as the system's ability to automatically detect, diagnose, and repair software and hardware problems. In a survey published in 2007, Ghosh and his coauthors presented the now well-known concepts of self-healing states and properties (Ghosh et al., 2007). They explained that the vision of large scale systems was already a reality and that self-healing research was active. Psaiser and Dustar published a survey showing the advancements on self-healing research (Psaier and Dustdar, 2011). The collected self-healing research areas in this survey included embedded systems, operating systems, architecture based, cross/multi-layer-based, multi agent-based, reflective-middleware, legacy application and Aspect Oriented Programming, discovery systems, and Web services and QoS-based.

In (Mrissa et al., 2015), the authors propose a software component called avatar for the WoT. The idea is to give objects a virtual representation in the Web so the may rely on Web languages, protocols, and semantic annotations. These avatars collaborate together in composed applications; however, they exhibit neither fault-tolerance nor self-healing properties. Also, they do not provide mechanisms to handle application context; that is, avatars function in a request-response way, and WoT applications need to be managed by a central coordinator. Moreover, they are not able to act differently depending on the application the are participating in.

5 CONCLUSIONS AND RESEARCH DIRECTIONS

We have presented a proposal to enable self-healing and context-aware WoT applications. The building block of our approach is a software component called WoT agent, which are the representation of physical objects, Web services, or humans in WoT applications. These agents are equipped with a knowledge base to handle application specific information, and with an autonomic component to exhibit self-healing properties. Our next steps are the formal definition of our approach, the implementation of its main functionalities, and its experimental evaluation.

REFERENCES

Alho, P. and Mattila, J. (2015). Service-oriented Approach to Fault Tolerance in CPSs. *J. Syst. Softw.*, 105(C):1–17.

Angarita, R., Cardinale, Y., and Rukoz, M. (2012). *The Semantic Web: ESWC 2012 Satellite Events, Heraklion, Greece, May 27-31, 2012.*, chapter FaCETa: Backward and Forward Recovery for Execution of Transactional Composite WS, pages 343–357. Springer Berlin Heidelberg.

Angarita, R., Rukoz, M., and Cardinale, Y. (2015). Modeling dynamic recovery strategy for composite web services execution. *World Wide Web*, 19(1):89–109.

Angarita Arocha, R. E. (2015). *An approach for Self-healing Transactional Composite Services*. Theses, Université Paris Dauphine - Paris IX.

Athreya, A., DeBruhl, B., and Tague, P. (2013). Designing for self-configuration and self-adaptation in the IoT. In *Collaboratecom, 2013 9th Int. Conference on*, pages 585–592.

Atzori, L., Iera, A., and Morabito, G. (2010). The IoT: A survey. *Computer Networks*, 54(15):2787 – 2805.

Chen, S., Xu, H., Liu, D., Hu, B., and Wang, H. (2014). A Vision of IoT: Applications, Challenges, and Opportunities With China Perspective. *IoT Journal, IEEE*, 1(4):349–359.

Cherrier, S., Ghamri-Doudane, Y., Lohier, S., and Roussel, G. (2014). Fault-recovery and coherence in IoT choreographies. In *IoT (WF-IoT), 2014 IEEE World Forum on*, pages 532–537.

Ghosh, D., Sharman, R., Raghav Rao, H., and Upadhyaya, S. (2007). Self-healing Systems - Survey and Synthesis. *Decis. Support Syst.*, 42(4):2164–2185.

Guinard, D., Trifa, V., Mattern, F., and Wilde, E. (2011). *Architecting the IoT*, chapter From the IoT to the Web of Things: Resource-oriented Architecture and Best Practices, pages 97–129. Springer Berlin Heidelberg.

IBM (2001). Autonomic computing: IBM's Perspective on the state of IT. IBM.

Kephart, J. and Chess, D. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Mattern, F. and Floerkemeier, C. (2010). From the Internet of Computers to the IoT. In *From Active Data Management to Event-Based Systems and More*, volume 6462 of *LNCS*, pages 242–259. Springer Berlin Heidelberg.

Mrissa, M., Medini, L., Jamont, J.-P., Le Sommer, N., and Laplace, J. (2015). An Avatar Architecture for the WoT. *Internet Comp., IEEE*, 19(2):30–38.

National Intelligence Council (2008). Disruptive Civil Technologies - Six Technologies with Potential Impacts on US Interests Out to 2025 - Conference Report CR 2008-07, April 2008.

Petersen, H., Baccelli, E., Wählisch, M., Schmidt, T. C., and Schiller, J. (2015). The Role of the IoT in Network Resilience. *IoT. IoT Infrastructures: First International Summit*, pages 283–296.

Psaier, H. and Dustdar, S. (2011). A Survey on Self-healing Systems: Approaches and Systems. *Computing*, 91(1):43–73.

Stankovic, J. (2014). Research Directions for the IoT. *IoT Journal, IEEE*, 1(1):3–9.

Vermesan, O., Friess, P., Guillemin, P., et al. (2011). IoT strategic research roadmap. *IoT: Global Technological and Societal Trends*, 1:9–52.