proceedings

Proceedings

A Research Agenda for IoT Adaptive Architectures †

Jairo Ariza *, Camilo Mendoza *, Kelly Garcés * and Nicolás Cardozo *

Department of Systems and Computing Engineering, School of Engineering, Universidad de los Andes, Bogotá, Colombia

* Correspondence: ja.arizac1@uniandes.edu.co (J.A.); ca.mendoza968@uniandes.edu.co (C.M.); kj.garces971@uniandes.edu.co (K.G.); n.cardozo@uniandes.edu.co (N.C.)

† Presented at the 12th International Conference on Ubiquitous Computing and Ambient Intelligence (UCAmI 2018), Punta Cana, Dominican Republic, 4–7 December 2018.

Published: 17 October 2018

Abstract: Adaptation is very important in IoT systems, due to their continuously changing environments. Changes may come from different elements of the architecture underlying an IoT system. Existing literature pays special attention to changes in the Service layer using evolution agents or context aware approaches to manage adaptations to said changes. In this paper, we elaborate on eight challenges that developers face when building adaptive IoT systems. Such challenges take into account changes at the Services layer, but also in the Middleware and Physical layers. These challenges serve us as a research agenda to foster IoT technology. As a starting point, we design an architecture to deal with the posited challenges. Various of the architectural components are inspired on a reference architecture, and complemented by new components to manage dynamic adaptations in response to the identified challenges. Preliminary experiments provide an initial insight about the feasibility and/or impact of our adaptive architecture.

Keywords: reference architecture; service-based architecture; dynamic adaptation

1. Introduction

Internet of Things (IoT) environments propose new means for users' interaction by means of software systems augmenting users' physical surrounding environment with a multitude of interconnected sensor and actuator devices [1]. Software systems available in the environment are supported by collected information from sensors, which is then analyzed and exploited to interact with users through the services or actuators.

The IoT naturally fosters collaboration between different systems. In particular, IoT aided critical systems present a continuous communication between deployed sensors and services in an environment (e.g., to alert users about sensed hazardous situations). Under normal conditions, sensors provide continuous values within a given range dictating safe situations. Emergency situations, present fluctuations of sensed variables, as a signal to trigger an alarm about the unexpected hazardous situation.

IoT devices are present in a multitude of application domains for critical systems, as transport systems, smart grids, smart homes, or Industry 4.0. In these domains, software services interact with and relay on the IoT infrastructure to function properly. As other software-based systems, IoT systems are in continuous evolution, be that due to software updates to the services running on top of the infrastructure, or to hardware updates on devices. Two main problems are identified when developing software systems for such evolving IoT environments:

1. interaction between devices of different makes, and
2. coping with the ever-changing surrounding environment [2,3].

In the first case, new devices may be added to, or removed from the environment unannounced. Running services should be able to interact with these devices regardless of the specific make of a device, and therefore, the communication protocols used. New devices may collect new information of interest to running services, or provide more accurate readings for a sensed variable. The system should be able to take such newly collected information seamlessly, so to ensure that services are working in accordance to their surrounding environment. Similarly, in the second case, the environment in which an IoT system is deployed may change for multiple reasons: changes in services' behavior, mobility of services and/or devices, or multiplicity of information sources. The system should be able to continue working in face of all these situations.

The aforementioned problems evidence that changes in the surrounding environment of IoT systems (both internal and external), require the system to adapt to said changes to continue working seamlessly. However, current (reference) IoT architectures [4] are not equipped to deal with these problems (Section 7).

This paper proposes a service-based adaptive architecture for IoT systems (Section 4). The main concern addressed in such architecture is resiliency and means of adaptation to the changing environment. The architecture consists of two innovative components (i.e., a format translator, and an Application Programming Interface (API) conflict solver) that integrate with classical components needed in an IoT architecture (e.g., resolution, fault and state manager components). All these components are distributed among the three layers of an IoT architecture: devices, middleware, and services. The proposed architecture is motivated by a list of challenges, identified from our experience working in both industry and academic projects, and a literature review of current issues in IoT adaptation (Section 7). We do not pretend the list of challenges to be exhaustive, but it is an initial approach to issues when adapting the system, that may be refined in the future. The main purpose of this paper is to provide a roadmap for adaptive IoT systems.

We put the proposed architecture into practice (Section 6) for a particular case-study of a household fire and gas-emission manager (Section 3), demonstrating the feasibility of our approach with preliminary results for the components associated to address Challenge 6. We conclude the paper and outline future directions in Section 8.

2. Challenges in Building IoT Systems

This section puts forward eight challenges identified from our experience in building IoT systems, and the state-of-the-art [2,3]. Rather than focusing on the technicalities of building a system that uses *IoT devices*, (for example, artifact provisioning) the challenges focus on the adaptation aspect of the system. Therefore, the challenges take into account, the actions required to ensure the system's continuous functioning in the face of faults or changes to any of the connected components across the different layers of the IoT architecture. Note that there are existing solutions for some of the proposed challenges. Our proposed architecture is designed to integrate such solutions.

1. **Resiliency to transient IoT devices.** IoT devices may become temporally or permanently unavailable due to a (hardware or software) failure, or users/devices mobility. In this case, services would become unresponsive, as readings from the surrounding environment are not received. The system should dynamically adapt to deal with these situations.
2. **Inclusion of new IoT devices.** As new IoT devices appear in the environment, they should be seamlessly incorporated in the system. If the system cannot recognize new devices, services will be agnostic to new available information. Changes in the topology should be integrated dynamically.
3. **Data format change in a device.** IoT devices may change the format in which measured data is sent as response to devices' updates, or the replacement of an existing device by another one. Modifying service's format would break the service, as the received data cannot be interpreted, or the service would yield wrong results. Both entities should change in unison.
4. **Middleware Unavailability.** The *Middleware* layer may become unavailable due to maintenance or malfunction. If the *Middleware* is not available, communication between devices and services is

lost. In such a case, the communication mechanism between services and devices must adapt to search for new components (re-)enabling the interaction between physical and virtual entities.

5. **Authorization and authentication.** As discussed in Challenge 2, new IoT devices should be incorporated in the system. In addition to functionality, security aspects should be taken into account too. The devices should be authorized and authenticated prior to establishing new connections, to monitor and regulate all the devices in the system.
6. **Service connectivity loss.** There are multiple situations in which connectivity with a service might be lost. For example, due to connectivity problems or services leaving the environment. All of these situations cause data sent by IoT devices not to reach their associated services. The system should adapt to discontinue the service.
7. **Abnormal information from different devices.** Due to the nature of collaborative systems, a variation of the measured variable is expected in case of abnormalities. In such cases, the IoT device must adapt its behavior, to properly analyze the environment.
8. **Services' API change.** Services evolve over time. As developers modify services' APIs, these may become unresponsive as the data sent from the IoT devices may no longer be consumable. In such cases, the Middleware should be able to search for other potential suitable devices to supply services' data.

2.1. Challenges to Evolve IoT Systems

2.1.1. AC.1—Adaptation to Transient *IoT Devices*

The IoT device connected to the *Middleware*, must send an identification mechanism in a regular basis. This way, the Resolution Manager in the *Middleware*, can identify the device and his current state. If there is no direct communication of the IoT device with the *Middleware*, the last one will order other *IoT devices* to communicate with the unavailable device to get its current sensors readings. If it is not possible to establish communication between the two *IoT devices*, the *Middleware* will notify the *Service* about the issue and will start a fixing request for the affected IoT device. If the unavailable IoT device re-start its operation, the fixing order will be canceled. In case the affected *IoT devices* experiences recurrent unavailable statuses, the maintenance order will also be created. During the inactivity period of the affected IoT device, the *Middleware* will block the receiving port of the IoT device, in order to stop receiving bad or corrupted data, with the potential to disrupt the proper function of the *Service*.

2.1.2. AC.2—Adaptation to Inclusion of New *IoT Devices*

As stated in scenario 1, each IoT device has an identification mechanism sent to the Status Manager in the *Middleware*. If the *Middleware* stops receiving the identification mechanism of a specific device and now it's communicating with a new IoT device, the *Middleware* will check the type of the variables supplied by the new IoT device and verify if these data are appropriate for the *Service*. If the new IoT device cannot establish communication with the *Middleware* using the primary established protocol, it must communicate directly with the *Middleware* using another available protocol. In case it is not possible the communication with the *Middleware*, the new device must communicate with another IoT device to guarantee the transmission of the measured variables.

2.1.3. The IoT Device Sends the Measured Data with a Different Format

In this case we propose a couple of options to the required adaptation in this scenario. First, just like in scenario 1 and 2, the *Middleware* will block the receiving port of the affected IoT device, to avoid the improper data format. The second approach, consist in a format translation *Service* that is developed to convert the improper data format into a suitable data for the *Service*. The translation *Service* include a semantic and a syntax approach to solve the issues.

2.1.4. AC.4—The *Middleware* is Temporally or Permanently Unavailable

In this scenario there are two adapting layers due to *Middleware* unavailability. The first one is the physical layer and the second one is the *Service* layer. In both cases the adaptation is similar, there are additional *Middleware* to ensure that the *IoT devices* and the *Services* can connect with each other. When the associated *Middleware* is not available due to a malfunction, the devices and the *Services* will look for a new *Middleware* to connect.

2.1.5. AC.5—*Service* Adaptation in the Event of a Cybernetic Attack

The required adaptation in this case, is also to block the port associated with the attacked IoT device, to prevent a *Service* malfunction due to corrupted data.

2.1.6. AC.6—Adaptation to Loss of Network Connection to the *Services*

We have considered, that the *Middleware* will be a device with better hardware capacities than the *IoT devices*. These characteristics allows the *Middleware* to assume the functions of the *Service*, at least for a limited time, like the present scenario. When there is no connection to the Web *Service*, the IoT *Middleware* will assume control and it will persists the measured variables internally. Also, the *Middleware* can analyze the variables to trigger the fire sound and visual annunciation if necessary. How long this operation can be performed will depend on the memory capacity of the *Middleware*.

2.1.7. AC.7—Adaptation to Abnormal Values of the Measured Variable

To save IoT's battery and to optimize data storage, the sample rate determined to measure the physical variable is low. Also there are another situations, where the system is categorized as a low demand system, i.e., it is not expected that the variable reaches its predetermined threshold in a very long time. But if an abnormal situation, the *Service* will detect a perturbation in the measure and it will send a request to the IoT device to increase its sample rate, allowing the *Service* to perform better analysis of the abnormal situation.

3. Smart-Home Alert System

As case study, to position the adaptation challenges of IoT systems, we present a residential smart-home monitor and alert system. The purpose of this system is to detect abnormal conditions in the environment, and alert the inhabitants about possible dangerous situations. For example, fires or high Carbon monoxide (CO) levels sensed in the house.

In the smart-home systems, the *Physical* layer is composed of two IoT devices. First, we have a fire alarm node composed of a temperature sensor, and a flame sensor. The main purpose of this node is to monitor the room temperature and detect the presence of flames. The second device is a CO node. This node is composed of a CO sensor that monitors the level of CO particles present in the environment, for example, by an inadequate fire combustion in an oven or a stove.

The CO node continuously monitors gas levels sending its reading to a web service (in the *Service* layer). The service analyzes the data to identify abnormal parameters, like CO levels exceeding a threshold, or inconsistent readings over a defined interval of time. Received data is persisted in a database.

If a high level of CO is detected, the corresponding web service alerts inhabitants about the hazardous situation. As part of this process, the service requests an appointment with the gas service provider, offering possible dates for the appointment to inhabitants, before it is sent to the gas provider. In case the CO levels are too high or the presence is detected for a long period of time, it will also emit visual, or sound alarms, with respect to users' location (e.g., pushing a notification to the user's smart T.V., or mobile phone).

The fire alarm node works in a similar manner to the CO node. The temperature and flame sensors monitor their associated physical variables. As in the previous case, a detection of fire sends an alert to the corresponding authorities through a fire alarm web service. In addition to this notification, the fire alarm service also enunciates the event locally with a visual, or sound alarm, depending on the location and activities of inhabitants.

Figure 1 illustrates the physical design of the alert system implemented. The CO node, the fire alarm node and the local visual and sounds alarms, were developed using an Arduino Uno Wifi. These devices communicate to the fire alarm and Natural gas Web services through an IoT gateway, implemented in a Raspberry Pi 3 model B. The IoT devices use the MQTT protocol to connect with the gateway and the Web services use the HTTP protocol. The IoT gateway acts as a proxy and enables the transmission of data between the different protocols used in our case study.

Section 5 describes how the challenges arise in the context of our case study; together with the adaptation required by our adaptive architecture to deal with the changes.

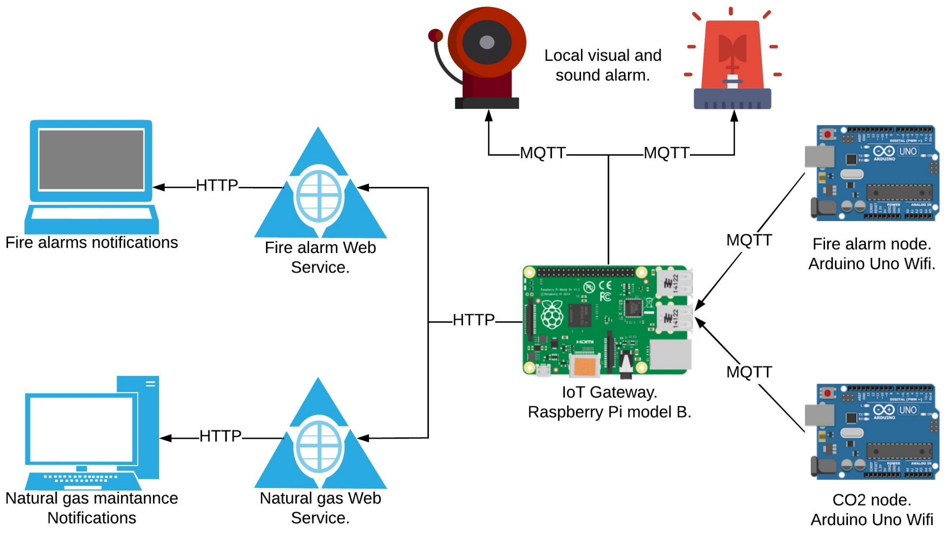
Local visual and sound alarm.
MQTT
MQTT
HTTP
Fire alarms notifications
Fire alarm Web Service.
MQTT
Fire alarm node. Arduino Uno Wifi.
HTTP
IoT Gateway. Raspberry Pi model B.
MQTT
HTTP
Natural gas maintannce Notifications
Natural gas Web Service.
CO2 node. Arduino Uno Wifi

Figure 1. Residential Smart-home and alert system.

4. An Adaptive IoT Architecture

This section presents our proposed architecture, shown in Figure 2, to deal with the challenges of the changing environment put forward in Section 2. Various of the architectural components are inspired on the reference IoT architecture [4], complemented by new components to deal with dynamic adaptations.

4.1. Physical Layer

The first layer in our architecture concerns the *physical layer*. This layer is composed of IoT devices. The main purpose of the devices is to measure physical variables, sending gathered information (i.e., *payload*) to IoT services. In the domain of critical systems, devices should be deployed with redundancy to measure each physical variable, assuring continuity in the payload. In this layer, each device provides a health-check functionality to announce that it is operational. In case, the health-check is not received, one of the redundant devices should be used.

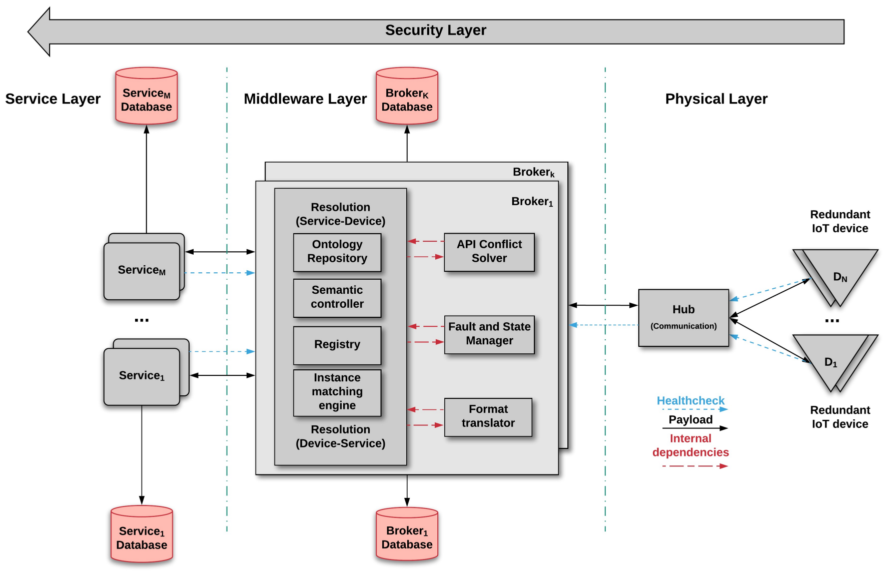

Figure 2. IoT architecture for adaptation.

4.2. Middleware Layer

This layer consists of two main components, a Hub and Broker.

4.2.1. Hub

The Hub provides connectivity from the IoT devices to services using wireless communications. The Hub offers a multi-protocol communication proxy [5], to be compatible with any devices available in the environment.

4.2.2. Broker

The Broker is composed of five main components, designed to address the challenges. Its main responsibility is to link IoT devices and services. Brokers have to ensure that both parts interact successfully even if changes happen in the environment affecting either of them. We consider redundant brokers to provide means of adaptation in case an operating broker becomes unavailable. Brokers are composed of five components: Resolution, Fault and State Manager, Format Translator, API Conflict Solver, and Database.

Resolution: This component discovers both; services that match the information provided by existing IoT devices, and IoT devices providing data required by deployed services (if such a matching exists). The functionality of the resolution component is carried out through the following four sub-components: *Ontologies Repository*, *Semantic Controller*, *Registry* and *Instance Matching Engine*.

The *Ontologies Repository* manages access to two ontologies: the Semantic Sensor Network (SSN) ontology [6], and the Service Integration Ontology (SIO) [7] ontology, both defined by the W3C. We extend these ontologies by adding the *Body Field* concept to the SSN ontology, and the *Property* and *Parameter* concepts to the SIO ontology. Figure 3 illustrates some of the ontology concepts instantiated in our case study. In the SSN ontology, the principal concept is *Sensor*. A *Sensor* is a device attached to a platform. Each *Sensor* has capabilities to measure *Environmental Properties*. Each capability has *Body Fields* representing the type of data that can be measured by a given device. In the SIO ontology, the main concept is *Service*. Each *Service* has a *URL*, *Properties*, *Methods*, and a set of *Parameters*. The *URL* is the address of the server that hosts the *Service*. The *Property* concept describes the environmental variables to which the *Service* is related to. There are four types of *Methods* that can be exposed by a service's API: Create, Read, Update, and Delete. Each *Parameter* has a name and a datatype. The

Ontologies Repository stores the ontologies together with instances describing the particular devices and services of a given domain.

The *Semantic Controller* checks the *Registry* every time the Broker receives a new request from an IoT device or service.

The *Registry* stores information in a tree-like structure where the parent nodes correspond to a part of the system (devices or services) and the children represent the counterparts that match a particular need. If the *Registry* indicates that a part has been previously bound to an available counterpart, then the request is forwarded to the latter. If not, *the Semantic Controller* queries the ontologies looking for the instances that describe the set of potential IoT devices and services. This set consists of services and devices with a number of parameters that corresponds to the number of devices' Body Fields. Each *Broker* shares its registries with other *Brokers* to make information available in case of downtime.

The *Instance Matching Engine* compares the ontology instances of services and devices by using heuristics that can be classified into two categories:

1. Similarity, and
2. Filtering.

Similarity heuristics compute a similarity value by comparing the name, multiplicity or context of a pair of instances. Filtering heuristics select matchings satisfying a given set of conditions. For example, a basic filtering heuristic is to compare whether registered values meet a threshold. In particular, the *Instance Matching Engine* compares environmental *Property* of *Sensors* (from the SSN ontology) with *Property* of *Services* (from the SIO ontology), and *Body Fields* with *Service Parameters*. Figure 3 illustrates the result of instances matching for our case study. In this example, matching is quite trivial since the strings correspond exactly. However, more elaborated heuristics may be required in other cases. The matching challenge has been well studied in the past, therefore, existing solutions will be integrated with our architecture [8].

If a list of matches is found, the service with the highest similarity in the list responds the device's request. The list is stored in the Registry to handle future requests from similar devices. In contrast, if there is no correspondence, the system sends an exception to the device for the developer to deal with.

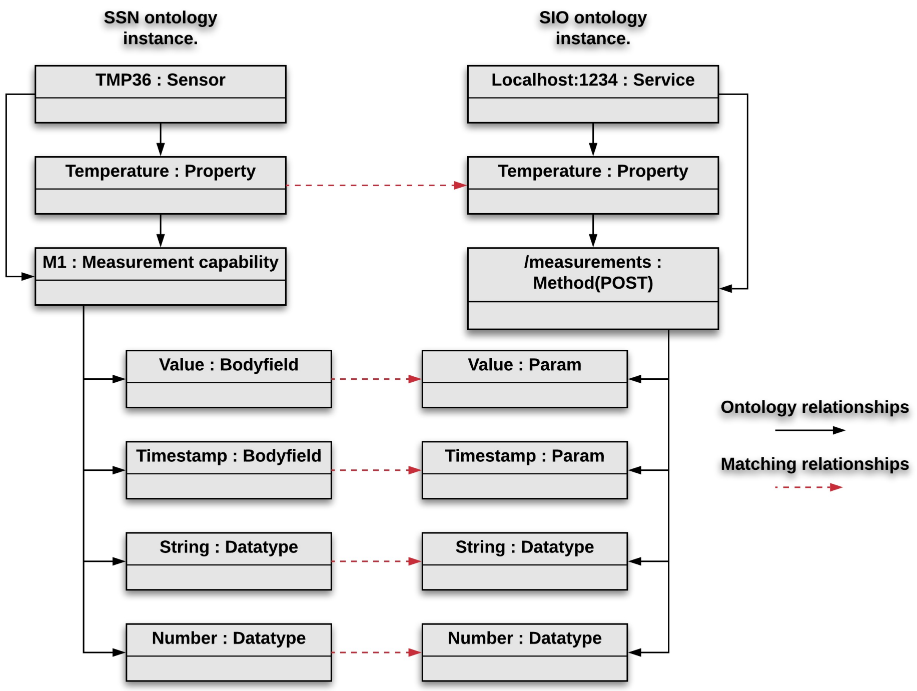

Figure 3. Result of instances matching for our case study.

Fault and State Manager. In our architecture, each IoT device or Service offers a health-check, sent to the state manager, to announce its availability. Whenever the state managers fails to receive a number of health-checks from a device or services, this is marked in the Registry as unavailable.

The Fault Manager component is in charge of handling faults during resolution and system execution. Once the communication between devices and services is established, this component forwards an exception message whenever some (part of the process) fails. There are three kinds of faults that trigger exceptions:

1. Unmatched services,
2. Unavailable services, and
3. Abnormal service responses.

Format translator. The format translator adapts the data provided by IoT devices to services (and vice versa), in case an incompatibility between the formats arises.
API Conflict Solver. This solver is able to adapt requests, sent by the devices, to changes on services' APIs. The adaptation depends on the kind of change the API undergoes, such changes can be classified into three categories [9]:

1. Non breaking changes,
2. breaking and resolvable changes, and
3. breaking and unsolvable changes.

No adaptation is needed for the changes from the first category. *The API Conflict Solver* integrates adaptations at run time, to solve the conflicts for the second category. Finally, the third category requires developer assistance. While the request is manually adapted, the Broker may temporally storage the data coming from the IoT devices in the Database, constrain data submission, or seek for another suitable service, to consume the data, in the Registry. The action taken by the Registry depends on how critical is the data, its frequency of emission, and volume.
Broker Database. Each Broker is equipped with a *Database*[1] to temporarily store data provided by the IoT devices. In particular the database can be used to hold the data while API conflicts are solved. To account for Broker failures, the associated databases for active brokers are synchronized periodically with their redundant nodes. This synchronization assures the activation of new brokers will not disrupt the system. All broker databases have to be optimized for speed and scalability.

4.3. Service Layer

At the top of our proposed architecture, one finds *services*. Services acquire the information supplied by IoT devices, in order to perform different functions, as providing alerts and notifications to a final user, persisting the gathered data in a *Database*, and performing data analysis of gathered information. Adaptation of services' APIs in the *Service* layer will be supported by the aforementioned *API Conflict Solver*.

4.4. Security Layer

The security layer cuts across the Physical, Middleware, and Service layers. Security functionality handles potential threats to the system by denying access to suspicious or not suitable IoT devices. Many of the components in the architecture must deal with security concerns, however, the concrete distribution of responsibilities has to be further analyzed, since there are trade-offs with other architectural quality attributes. This analysis is out of the scope of this paper, and we will tackle it as part of our future work.

[1] The database type depends on the type of information to store. Typically a NoSQL database is used, but this should be transparent to the broker.

5. Addressing the Adaptation Challenges

This section presents how our proposed architecture addresses the different challenges put forward in Section 2. The proposed solutions are explained from the perspective of a particular application domain, the alarm system case study presented in Section 3.

1. **Transient IoT devices:** Whenever a device becomes unavailable (e.g., the fire alarm, or the gas detection nodes), the system must adapt to continue sensing the environment and alert users about the fault. In our example, a maintenance order for the affected IoT device is generated by the Web Service associated to the faulty device. Moreover, the *Broker* will try to find an alternative (set of) device(s) to supply the missing information to the web service. The Broker must update its IoT device registry, mapping new devices to services, and changing the status of the absent device to inactive. If the device restart its operation during this process, the maintenance order will be canceled and the device registry will be updated to active. If the device keeps an intermittent behavior, the service will generate a maintenance request, explaining the intermittent behavior.
2. **Inclusion of new IoT devices:** Let us take an environment that initially contains devices connecting through a wireless (e.g., WiFi) network with the Hub. If new devices using a cellular network appear in the environment, the Hub seamlessly connects with the device (regardless of the difference in protocols), given this is a valid device according to the SSN ontology. New devices are registered in the Broker, to use in cases the currently active devices fail.
3. **The IoT device sends the measured data with a different format:** Measurement provided by IoT devices can use different semantic formats, for example the CO node can use particles per million (ppm), or milligram per liter (mg/L), or milligrams per kilogram (mg/kg). To ensure proper system operation, the *Broker* will request the *Format translator* to transform the CO measurement to the format required by the corresponding *Service*, using the Metadata provide by the IoT device. Syntactic format inconsistencies are resolved in a similar fashion.
4. **The Middleware is temporally or permanently unavailable:** The *Broker* connecting the web service and the fire alarm may become unavailable. In such a case, as with IoT devices, a redundant *Broker* is made available to maintain the communication between IoT devices to access the Services. Upon the *Broker* connection failure, IoT devices and the Services adapt their behavior to search for another *Broker* to establish proper communication. To make this task easier, each *Broker* of the group will share its registry and database snapshot with other Brokers, so the new connection between Services and devices will propagate to other Brokers. Registry and database synchronization is vital in case of a new unavailability or the recovery of the Broker. This behavior should always take place independently, from the specific application domain using the architecture.
5. **Authorization and authentication of new IoT devices:** Third party interference, data manipulation, or devices may cause the device to malfunction, possibly leading to a hazardous situation. The system's security component is responsible for providing authorization and authentication mechanism to new connected IoT devices. This process provides proper regulation of IoT devices, in order to assure suitable devices to the required needs of services.
6. **Service connectivity loss:** The web services analyzing the readings provided by the CO device may become unavailable. Three possible adaptations are presented to deal with this problem. Each of the solutions offers adaptations for one layer in our architecture. First, our proposed architecture considers that each service has at least a redundant copy. If a service is not available due to infrastructure or connectivity failure, the *Broker* must connect the associated IoT devices to a copy of such service, to avoid losing data or service malfunction. In case all copies are unavailable, the *Broker* must search for another suitable service in the *Registry*. The second adaptation option is to temporally store the data in the *Broker* database to be processed once the service is available anew. Finally, the third adaptation option is to enforce IoT devices to stop sending information, until the *Broker* can find a new suitable Service for the provided information.
7. **Adaptation to abnormal values of the measured variable:** Our case study is an example of a low demand system. This means, it is not expected that the fire alarm and CO detection nodes,

announce abnormal events (e.g., fire or CO presence) in a regular time basis, as these scenarios represent emergencies. As a consequence, sensed variables can be stored with a low sample rate. If there is a significant perturbation in the measured variable, the IoT device must increase the sample rate, to get more detail about the abnormal situation.

8. **Service API change:** Services can change their API and the *Broker* must adapt the requests to satisfy the new API. The adaptation required depends on the kind of change. As an example, take the initial API version of the CO web service that triggers maintenance orders if the data received from a single CO device is intermittent. Suppose further that the API evolves to a new version, in which the maintenance order is created whenever multiple IoT devices, located in the same area of the house, present an erratic behavior. In such a case, the API Conflict Solver has to resolve a suitable combination of IoT devices, to comply with the new data requirements of the evolved API.

6. Experiments and Results

This section presents preliminary results providing initial insights to the feasibility and impact of our adaptive IoT architecture. This experimentation focuses on challenge 6.

6.1. Prototype

To evaluate our architecture and in particular the integration of devices, services, a hub, and Fault and State Manager components of the Resolution component in the Broker,[2] we develop part of the fire alarm system. We use temperature sensors connected to an Arduino UNO Wifi [10]. The microcontroller reaches the Broker (which is available over the Internet) through an access point. The broker forwards temperature data to a Spring Boot [11], a web service exposing a create method and a health-check. In the Broker, the Semantic Controller is a Java program and the registry is a ZooKeeper [12] instance. The Ontologies Repository is developed on top of an Apache Jena Fuseki server. Finally, Duke is used as the Instance Matching Engine [13]. The Broker runs on a single machine. Three ZooKeeper servers are deployed on different machines, working as a cluster. The Semantic Controller and Ontologies Repository are also deployed on different machines. All the machines are equipped with 8GB of RAM and 4 cores.

6.2. Evaluation

We evaluate the accuracy of the Instance Matching Engine, the functionality offered by the Fault and State Manager and the Resolution components' performance.

Instances to the ontologies are manually added, executing the Instance Matching Engine over them. We found that matching failed when comparing device/service properties expressed as abbreviations with properties expressed with long words (7 or more letters). From this we learned that our architecture has to be flexible enough to allow developers to plug, compare and tune new matching heuristics.

We also carry out a functional test over the defined components. The results of the tests matched our expectations: when a service becomes unavailable, the *Broker* effectively selects another appropriate service to forward the request to. Error messages are effectively returned, in the exception cases planned for the tests.

We execute load tests to see the behaviour of the Resolution component with respect to performance. We executed several iterations in which the number of requests ranged from 100 to 12,500, with a ramp-up of 1 s in each iteration. The upper bound was chosen empirically, as after 12,500 requests the *Broker* began generating "overloaded server" errors. The X axis in Figure 4 shows the number of requests sent in each iteration, and the Y axis shows the response time for the requests per

[2] The implementation of the remaining components are needed to address the remaining challenges, but fall outside of the scope of this paper.

iteration. As it can be seen in Figure 4, the average response time of the Resolution component has a linear growth rate with respect to the number of requests.

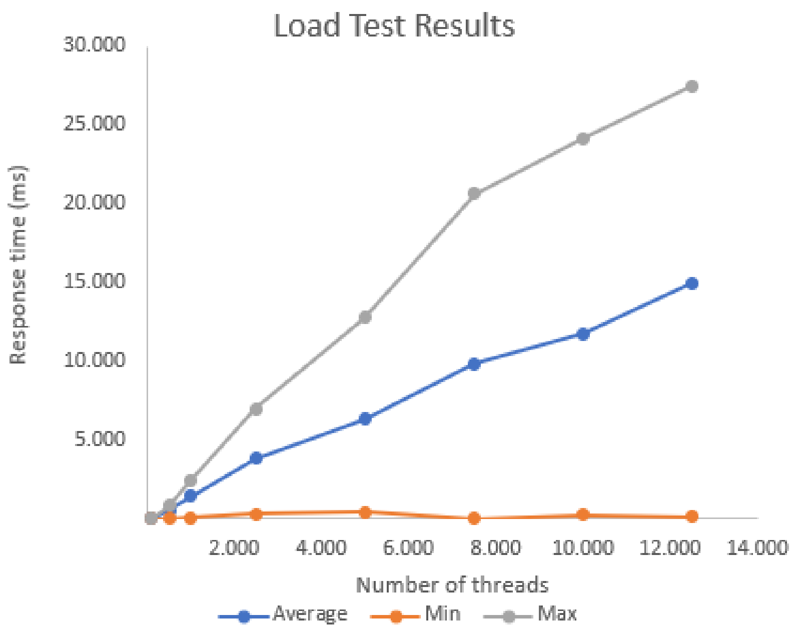

Figure 4. Load test results for Device-Service Resolution.

6.3. Discussion

This experiment tests device-to-service resolution. However, we need to make an additional effort to test service-to-device resolution. Even though the Fault and State Manager components have been designed in a generic way to support both directions of interaction, at the implementation level, we identify the following extensions:

1. an additional registry where parent nodes (in the tree representation) represent services (those initiating communication) and children represent the devices that match such services; and
2. a low-cost mechanism for reaching devices from the Middleware.

Currently, the Resolution component is a bottle-neck since all requests from devices/services pass through the *Broker*. As a consequence, the *Broker* resources rapidly run down. We plan to introduce at this level different tactics to improve performance and scalability. First, a publish/subscribe architectural style where devices/services can consume/publish requests from multiples topics. This introduces concurrency that should favor performance. We need to complement the style with health checking for devices/services to stop consuming/publishing data from a given topic, if the corresponding counterpart is unavailable. Second, only a broker is treating requests at a time, resulting in an architecture that does not scale. To cope with this, we plan to implement horizontal scaling where multiple brokers serve requests in parallel. Third, the State and Fault Manager components may overload rapidly because they perform health checks on multiple devices/services, like a ping. We need to implement a heart-beat where devices/services are those that notify the Manager about their state; this also unloads Broker.

7. Related Work

Evolution and adaptation in IoT systems has raised the attention of many researchers, given the opportunities for collaboration with other such systems. This section puts existing approaches that address adaptation of IoT systems in perspective of our proposed architecture.

Evolution Agents (EVAs) [14] consist of a notification management architecture exploiting a semantic service registry based on ontologies. Services are described and registered in the registry using semantic web ontologies. The registry's main function is to provide clients and EVAs with suitable information about a required service. This information is used by the evolution algorithm to compare different versions of a service, in order to adapt it (automatically or manually). EVAs detect and coordinate the evolution of the services available in the registry. Whenever a service changes, the EVA associated with the service reports the change to all dependent services; triggering their EVAs to evolve accordingly. [14] describe a scenario where several IoT devices from different suppliers provide different services. Each service in the environment is equipped with an EVA. The EVA analyzes the changes and if possible, it will automatically update the service with a suitable version, provided by its own repository or by another EVA. In case the version evolution cannot take place, the EVA notifies the developer (to perform the adaptation manually).

Context-aware adaptation is also explored for the IoT domain. [15] propose a web framework to adapt IoT systems in response to context changes (e.g., sensed information, networks, or power supplies). Developers can define resource-independent applications through a mT-Res. The mT-Res architecture implements modular self-contained code, that can be moved from one resource to a similar one, when a change is found in the context [15]. The two main components in this architecture are the Resource Administrator, and the Applications Manager. The Resource Administrator component monitors the context from sensors, and detects variations in the environment. Upon variations, the component informs the Applications Manager to deploy the functional blocks of code in another resource with the same capabilities of the affected one, effectively adapting the resource to the environment.

Ref. [16] provide a context-aware approach based on ontologies to describe resources provided by Physical devices or even virtual components. With the description provided by the resources' ontology, the *Middleware* handles *IoT devices*' data distribution, and connection security.

Our adaptive architecture evolves services' APIs by adapting the communication data payload between the *Physical*, *Middleware*, and *Services* layers to the latest API version. In contrast, EVAs handle evolution by keeping former and new versions of services' APIs, mapping devices' requirements to the appropriate version of the API. mT-Res composes devices' behavior, by moving functional blocks between them, in response to physical changes in the system's surrounding environment.

EVAs manage virtual entities and physical devices uniformly as services, both represented in a single ontology. Similarly, [16] use a single ontology to represent both devices and virtual entities. Our proposed architecture uses two ontologies; an ontology to represent Devices, and another to represent Services. We keep two ontologies to clearly separate the physical domain from the virtual domain, to facilitate ontology modification by stakeholders. In both cases matching is necessary, so having two ontologies introduces no overhead to the solution.

8. Conclusions and Future Work

This paper enumerates the different adaptation challenges found in IoT applications, according to our experience. The adaptation challenges can take place at different layers *(Physical, Middleware, or Services)* of the IoT reference architecture using specific adaptive components. To cope with these challenges we propose an adaptive architecture. This architecture, and our contribution, constitutes a roadmap for the next five years in IoT adaptation development. The architecture is validated with an initial experiment focusing on a particular challenge. This experiment evidences the relevance of the proposed components, as they are aligned with the challenges. Nonetheless, we noticed the initial implementation needs to be improved, to satisfy the performance and scalability requirements of IoT systems. The remaining challenges will be studied and analyzed as part of our future work.

While in this paper we only consider a subset of the challenges, this sets a first milestone for the proposed architecture. In the future we will extend and improve our current implementation to address the other challenges. A key factor in future development is the integration of the different components.

References

1. Mattern, F.; Floerkemeier, C. From the Internet of computers to the Internet of Things. *Informatik-Spektrum* **2010**, *33*, 107–121.
2. van Kranenburg, R.; Bassi, A. IoT Challenges. *Commun. Mob. Comput.* **2012**, *1*, 9. doi:10.1186/2192-1121-1-9.
3. Gazis, V.; Goertz, M.; Huber, M.; Leonardi, A.; Mathioudakis, K.; Wiesmaier, A.; Zeiger, F. Short Paper: IoT: Challenges, projects, architectures. In Proceedings of the 2015 18th International Conference on Intelligence in Next Generation Networks, Paris, France, 17–19 February 2015, pp. 145–147. doi:10.1109/ICIN.2015.7073822.
4. Bauer, M.; Boussard, M.; Bui, N.; De Loof, J.; Magerkurth, C.; Meissner, S.; Nettsträter, A.; Stefa, J.; Thoma, M.; Walewski, J.W., IoT Reference Architecture. In *Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 163–211. doi:10.1007/978-3-642-40403-0_8.
5. Desai, P.; Sheth, A.; Anantharam, P. Semantic Gateway as a Service Architecture for IoT Interoperability. In Proceedings of the 2015 IEEE International Conference on Mobile Services, New York, NY, USA, 27 June–2 July 2015, doi:10.1109/mobserv.2015.51.
6. Compton, M.; Barnaghi, P.; Bermudez, L.; García-Castro, R.; Corcho, O.; Cox, S.; Graybeal, J.; Hauswirth, M.; Henson, C.; Herzog, A.; Huang, V.; Janowicz, K.; Kelsey, W.D.; Le Phuoc, D.; Lefort, L.; Leggieri, M.; Neuhaus, H.; Nikolov, A.; Page, K.; Passant, A.; Sheth, A.; Taylor, K. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semant. Sci. Serv. Agents World Wide Web* **2012**, *17*, 25–32.
7. Ryu, M.; Kim, J.; Yun, J. Integrated semantics service platform for the Internet of Things: A case study of a smart office. *Sensors* **2015**, *15*, 2137–2160.
8. Euzenat, J.; Shvaiko, P. *Ontology Matching*; Springer-Verlag: Berlin/Heidelberg, Germany, 2007.
9. IBM Corporation. Evolving Java-based APIs, 2017.
10. Arduino. Arduino uno WIFI. 2018 Available online: https://store.arduino.cc/usa/arduino-uno-wifi (accessed on 10 October 2018).
11. Pivotal Software. Spring Boot. 2018. Available online: https://projects.spring.io/spring-boot/ (accessed on 10 October 2018).
12. The Apache Software Foundation. Apache ZooKeeper. 2017. Available online: https://zookeeper.apache.org/ (accessed on 10 October 2018).
13. Garshol, L.M. Duke. 2017. Available online: https://github.com/larsga/Duke (accessed on 10 October 2018).
14. Tran, H.T.; Baraki, H.; Kuppili, R.; Taherkordi, A.; Geihs, K. A Notification Management Architecture for Service Co-evolution in the Internet of Things. In Proceedings of the 2016 IEEE 10th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA), Raleigh, NC, USA, 3 October 2016; pp. 9–15. doi:10.1109/MESOCA.2016.8.
15. Gaur, S.; Rangarajan, R.; Tovar, E. Extending T-Res with Mobility for Context-Aware IoT. In Proceedings of the 2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI), Berlin, Germany, 4–8 April 2016; pp. 293–296. doi:10.1109/IoTDI.2015.16.
16. Pötter, H.B.; Sztajnberg, A. Adapting Heterogeneous Devices into an IoT Context-aware Infrastructure. In Proceedings of the 2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) Austin, TX, USA 16–17 May 2016; pp. 64–74. doi:10.1145/2897053.2897072.