Managing Evolution of API-driven IoT Devices through Adaptation Chains

Rafael Bustamante, Kelly Garcés

Department of Systems and Computing Engineering, Universidad de los Andes (CO)
{r.bustamante32,kj.garces971}@uniandes.edu.co

Abstract. Technological advancements, updates to legal regulations, and emerging industry standards have immersed the IT industry in a never-ending cycle were RESTFul APIs are required to keep up with the latest trends. IoT developers often hard-code the API invocation in the clients (i.e., IoT devices); this makes the continuous upgrade/delivery of new API versions laborious or even impossible. In this article, we introduce an API Adaptation approach that addresses the problem by transforming messages exchanged between devices and the API server. In our approach, compatibility is achieved by adjusting the information after it leaves the source and before it reaches the target destination. The evaluation over a large/real API offers initial insights into the feasibility of our approach.

Keywords: API; Changes; Software Defined Networking; Middleware; Internet of Things.

1 Introduction

With the latest technological advancements and the ever increasing number of devices connected to the Internet, humanity has been able to gather data and integrate electronic devices with the physical world like never before. All of this has enabled a technological trend known as the *Internet of Things* (IoT). The Internet of Things is based on machine to machine communication, cloud computing and networks of sensors [2]. This integration of systems has the possibility of making every device smart by combining all gathered data with the power of cloud computing and machine learning.

An IoT architecture consists of three tiers[1]: IoT devices, connectivity, and services. The connectivity tier typically has gateway devices responsible for interfacing directly with the services and providing an API (Application Program Interface) (e.g., RESTFul) to IoT devices. As developers modify services' APIs, these may become unresponsive as the data sent from the IoT devices may no longer be consumable. This challenge can be expressed as the need of maintaining compatibility between constantly developed and improved APIs and their continuously increasing number of clients.

Keeping new and legacy API Clients compatible with the latest changes of an API is a complex process. The evolution and adaptation of APIs for Web-based

software have been studied in recent years [15] and some solutions have been proposed, e.g., Software Defined Networks (SDN). However, when it comes to evolution of APIs for IoT systems, such solutions may not be fully applicable because client devices may: (1) have very few resources which constrain their ability to properly adapt to changes in the API or (2) either be hard-coded or hard to access, which makes continuous upgrade/delivery laborious or even impossible.

These factors represent a challenge that deserves attention and research. To reach a more appropriate solution, adhesion to modern trends in software engineering such as continuous integration and DevOps is required. In this paper, we propose an approach in early stages of research that suggests a revisited perspective of SDN applicable to IoT. This work is aligned with the H2020 2018-2020 work package; specifically with the activities ICT-15-2019-2020 [5] about engineering methods for cyber-physical systems of systems (CPSoS) and ICT-16-2018 about software engineering methods and tools applicable across domains such as IoT and Cloud[6]. This work contributes initial insights to the feasibility of software engineering practices that result in improvements in maintenance for API-driven IoT devices.

Our approach (see Section 4) focuses on addressing compatibility between an API and its clients[1] by adapting messages as they leave the source to make them understandable by the target destination by applying successive transformations to the exchanged data. The approach can handle input and output interface changes of individual APIs, that correspond to syntactic changes, thus changes in the semantics of the APIs are out of the scope of this paper. We adapt existing strategies in the IoT research landscape. The rest of this document is organized as follows: Section 2 presents related work. Section 3 describes an API evolution scenario to provide more context for the work presented here. Section 5 summarizes the results obtained after testing the proposed strategy over a real API in production environment. Section 6 contains thoughts and further considerations for the adaptation approach outlined in this document.

2 Related Work

There are several papers in the areas of API adaptation (e.g., [16, 13, 12, 10, 18]), business process adaptation (e.g., [17]), web service composition adaptation for changing workflow/business processes (e.g., [9, 4]). In this paper, we took inspiration from the algorithms found in those works and adapt them for APIs in the IoT domain. Whereas most of the approaches found in the literature tackle the challenge of adapting Web/mobile/third party systems to evolving software artifacts, we found only two approaches addressing the API compatibility issues for IoT clients. The next paragraphs briefly describe these two approaches and their limitations, and a comparison ends the section.

[1] API clients and IoT devices will be used as interchangeable names in the remaining sections

Software Defined Networking (SDN) [15]: This approach proposes a segmentation of concerns between *how the API is navigated* and *what data is passed between calls to the API*. In the same manner, the API is configured by setting available navigation paths in a programmatic sense instead of the traditional resource path in a REST API [14]. Under this approach, a software Petri Net is configured with the API Paths (represented as nodes) and the client is redirected across them as required. To do this, the client incorporates two software components: a Client Oracle that knows how the API is traversed and a Client Agent that knows what data is to be passed to the current node indicated by the navigation. In this way the evolution of the API is managed by sharing navigation and API data changes with the client, which in turn updates the definitions of its Oracle and Agent. Although this approach offers flexibility and guarantees that clients are updated with the changes deployed in the API, it represents a major shift away from the RESTful API standard. Also, this approach means that existing clients should be modified to include the Oracle and Agent, which is difficult or even impossible in constrained IoT devices.

Multi-Versioning [8]: This approach focuses on keeping multiple versions of the same API published and available at the same time. This way, compatibility between all clients with the API is guaranteed as each client will be able to consume their corresponding version. However, this approach does represent major challenges in terms of proper resource allocation as it requires additional servers and platform resources for each deployed API version. Additionally, this approach also implies that corrections and upgrades to any API likely need to be propagated to other versions, meaning that additional maintenance costs would be incurred.

The improvements on maintainability offered by SDN-based approaches (including our own) cause a degradation on performance. In multi-versioning approaches is the other way around. In certain contexts (e.g., safety-critical systems) this degradation cannot be tolerated. Therefore, both approaches are valid and can be seen as complementary to satisfy the requirements from a broad variety of IoT devices. All the other approaches focus on helping the developers change their source code to make applications compatible with a new version of an API, which in the context of IoT where the code that makes use of the API is on a device, changing that source code is something difficult or in some cases impossible to achieve.

3 Case Study

The case study consists of a residential smart-home system that monitors and alerts house residents about fire situations. The system works by continuously reading data from a fire detection device, detecting anomalous measurements, and warning stakeholders of possibly dangerous situations.

The fire alarm device is composed of a temperature sensor and a flame sensor. The main purpose is to monitor the room temperature and detect the presence of flames. The temperature and flame sensors monitor their associated physical

variables. A detection of fire sends an alert to the corresponding authorities through a fire alarm service. In addition to this notification, the fire alarm service also announces the event locally with a visual or sound alarm depending on the location and activities of the residents. The target API is assumed to be Open API 3.0.0 compliant.

The case study API is composed of a single fire report resource published on an HTTP server. This resource represents the interface with the fire department service and exposes two operations: a GET and a POST request. The GET operation returns an array of JSON objects representing a fire report. The POST operation is used to deliver a JSON object that represents a fire sensor data report and receives, as response, a JSON object that describes the fire department that will attend the emergency and their estimated time of arrival to the house. The object diagram in Fig. 1 depicts two overlapped versions of the fire alarm service's API. This case study is introduced to add additional context to the proposal discussed in the following section.

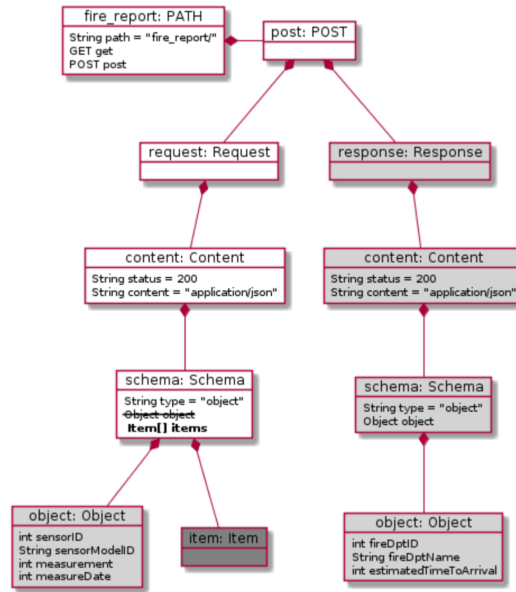

Fig. 1. Object diagram representing an excerpt of the fire alarm service API

4 The API Adapter

Keeping in mind the reasons for continuous/mandatory API evolution and the hurdles that hinder compatibility between APIs and clients, the API Adaptation approach was designed with the following considerations as guidelines: (1)

Since the technological gap between legacy and modern API clients will broaden throughout the upcoming years, the solution should cover the widest range of devices consuming the target RESTful API. (2) Despite the latest advances, IoT devices still have a low amount of resources when compared to those of API servers, thus being constrained devices. The solution should compromise as little resources as possible from API clients. Hence, the solution is designed as an independent component with no ties to any of the IoT devices. (3) The solution should keep the API version management as simple as possible without additional complexity. Therefore, only the newest version of the target API should be required.

The solution incorporates a middleware, which will be referred to as *API Adapter*, focused on keeping all client interactions compatible with the most recent version of the published API, as well as all responses delivered by the API compatible with the clients' expectations. The overall strategy by which compatibility between different versions of the API is obtained is divided into two phases: The *Conflict Identification Phase* and the *Message Adaptation Phase*.

The *Conflict Identification Phase* takes place whenever a new API version is deployed. This phase identifies the differences between the source version of the API and the new target version; it creates the message adaptation transformations required to make both versions (source and target) compatible. This part of the strategy is summarized in the following steps: (1) Load source and target API definitions. (2) Compare the source API version against the target and distill all differences as atomic differentials. (3) Process each API differential and map it against its corresponding category within a previously established taxonomy. The category indicates which service performs the adaptation. (4) Format this information as a request to an Adaptation Service.

The second phase is the *Message Adaptation Phase* and it occurs whenever a message is dispatched from an API client to consume a resource in the API server. This side of the approach is briefly described in the list up next: (1) Receive a request message from any API client. (2) Find the client's expected API version. (3) Apply successive transformations to the messages to make the request from the client compatible with the newest API version. (4) Transform the response from the API server back into the client's expected API version via successive calls to Adaptation Services. The architecture of the *API Adapter* is depicted in Fig. 2. A description of its components follows.

4.1 API Version Manager

This component ensures compatibility of any supplied API with the OpenApi 3.0.0 specification [11]. Once a new API version has been validated as compliant with the standard, it is submitted to the *Differential Comparator* to evaluate the changes in relation with the former API version.

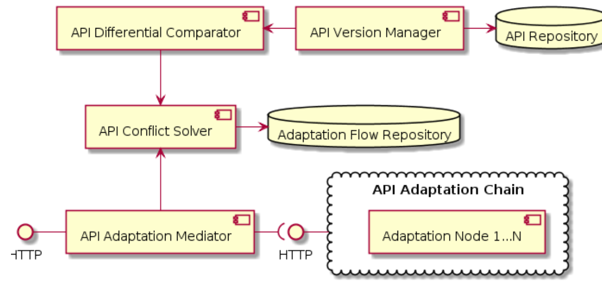

Fig. 2. Component diagram of the API Adapter

4.2 API Differential Comparator

This component compares two different version specifications of the same API. Any differences found are expressed as a set of small differentials, describing whether the changes are modifications to existing elements, additions of new elements, or deletions of existing elements.

The following list describes the steps and considerations followed by the *Differential Comparator* to evaluate two versions of the API. (1) Each API version is loaded into its own *Abstract Syntax Tree* (AST) structure [3].(2) The two ASTs are compared against each other in search of mismatching branches. (3) Each mismatching branch is expressed as an *API Differential* and is described with the following fields: *Element Path:* This is the section of the AST branch where both versions of the API are similar. *API Element:* This is the last node in the Element Path, which is the one undergoing the change. *Differential Type:* This field defines whether the new version of the API is either a *modification* of existing data, an *addition* of new data, or a *deletion* of data. *Differential Phase:* This field defines whether the new version of the API is modifying the API's server *request* or its *response*. *Source Element Value:* This is a representation of the data stored in the nodes directly below the *Element Path* in the tree of the source API. (f)*New Element Value:* This is a representation of the data stored in the nodes directly below the *Element Path* in the tree of the target API.

To further illustrate how differentials are processed, suppose that a change in the new version of the case study API is introduced. In this example, the schema for the POST request is changed from a JSON object to an array of items with no specific structure. In Fig. 1 the two versions are overlapped; the common elements between APIs are colored in white, elements from the source API are colored light gray and the elements from the target API are colored dark gray. White colored objects represent the **Element Path**. As seen in the figure, the change takes place under the *request* node, which is the **API Differential Phase**. This means that the adaptation should take place when the client's request is received in the system. The lowest node in the **Element Path** represents the **API Element** of the path up to which the two APIs are similar. The light gray colored element represents the **Source Element Value** in

the source API. The dark gray colored element represents the **New Element Value** in the target API. The striked line over the property *object* in the schema represents the **Differential Type**, which is a modification.

4.3 API Conflict Solver

This component is responsible for processing API Differentials delivered by the *Differential Comparator* and mapping them into properly structured adaptation requests to services in the *Adaptation Chain* (see Section 4.5).

In order to match API Differentials against Adaptation Services, the *Conflict Solver* relies on a *Taxonomy of Changes* [19]. Table 1 summarizes the taxonomy. Each change has a type, an element undergoing the change and an Adaptation Service. The latter transforms the changed element into compatible data.

Table 1. Differential Taxonomy

Name	**Adaptation Service**
Modify Property Type	Caster
Modify Property Name	Property. Mapper
Add Property	Property Mapper
Remove Property	Property Mapper
Modify Schema Type	Schema Parser
Modify Content Type	Content Mapper
Modify Status Code	Status Mapper
Add Status Code	Status Mapper
Remove Status Code	Status Mapper

The process by which an API Differential is mapped into an Adaptation Service request is summarized in the following steps. (1) Match the incoming *API Element Type* against the taxonomy Element. (2) Match the incoming *API Differential Type* against the taxonomy Differential Type. (3) Match the incoming *API Differential Phase* against the taxonomy Differential Phase. (4) With all the elements matches, find the proper *Adaptation Service* and register the proper call to it as a link of the adaptation chain.

4.4 API Adaptation Mediator

This component is the interface between the API and all its clients. It is responsible for receiving all incoming API client requests, orchestrating calls to Adaptation Services and exchanging the transformed messages between API clients and API servers.

4.5 API Adaptation Chain

This component is responsible for supplying the Adaptation Services for the *API Adaptation Mediator* with the purpose of applying successive transformations to messages exchanged between the API Server and its clients.

Each *Adaptation Service* is focused on providing a specialized transformation which is determined by how an *API Differential* is matched against a taxonomy item by the *Conflict Solver*. Since each API Differential is conceived as the smallest, most representative change applied to an API Element from an API Version to the next, the Adaptation Services are designed to be consumed following a Chain of Responsibility pattern [7] similar to the behavior described in Fig. 3, where the *Adaptation Mediator* calls a Caster Adaptation service. This adaptation should take place before calling the POST method since the incoming JSON object has to be transformed into an array of items. The latter may require casting of data types.

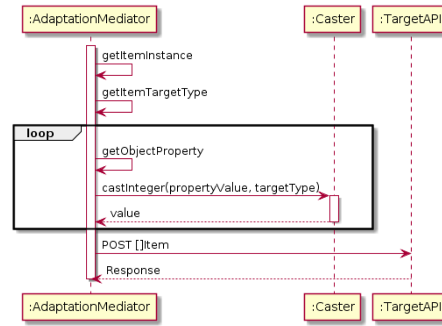

Fig. 3. Sequence diagram representing an adaptation flow for the case study

5 Preliminary Results

The purpose was to test the accuracy of the API Adapter at identifying changes between API's versions. First, we built a prototype of the API Adapter. We provided the API Adapter with two versions of a large API, where the last version included at least three occurrences of each type of change reported in our Change Taxonomy. Then, we manually compared the output from the API Adapter against a table of expected results.

Some characteristics of the API under evaluation are: 23 Resources, 1-4 HTTP Verbs per Resource, 2-4 Status Codes per Response, 1 Application Content, 0-2 Nested Objects, 1-14 Properties per Object. Further elements that describe the complexity of the API follow: Each of the 23 resources exposes on average 2.5 http verbs (GET, POST, PUT, DELETE). Each operation available in a resource has on average 2.5 status codes (200, 400, 401, 403, 500). Only the *application/json* content is considered throughout the API. A resource's schema

may have a nested object definition which, in turn, may wrap up to one object as a property. Resource schema objects have on average 6 properties; meaning that some may have a minimum of 1 property and a maximum of 14.

The API adapter is 100% accurate when it comes to determining whether a new element was added or an existing element was removed. Whilst the API comparison is always able to identify element modifications in leafs of an API AST representation, it is only 66.66% accurate when identifying modifications in non-leaf nodes. The reason for this lack of precision is explained by an unexpected behavior in the differential comparator. In order to speed up the process, the current implementation navigates in depth the branches of the AST until a change is detected. As a consequence, operations like modifying the property *name* of elements are not always identified as such; being incorrectly marked as an addition of a new element or a removal of an existing one. To work around this issue, before marking a source element as changed, the algorithm should compare the tree nodes hanging under such an element with those of the target element. If the tree nodes are equal, we are facing an element whose internal properties have been altered (e.g., property renaming). Otherwise, we deal with an element whose pendent elements have underwent a set of modifications. Hence, we can decrease the risk of wrongly identifying additions and deletions when, actually, a modification takes place.

6 Conclusions and Future Work

Throughout the course of this research we identified that our API Adaptation approach relies heavily on determining the spots where changes took place in order to properly apply all required adaptations. Future efforts will be focused on adding efficient recursive calls to the existing algorithm to improve its accuracy.

The use of an API Change Taxonomy proved to be a valuable guideline to set out the scope of the adaptation strategy. Further work around this should extend the Change Taxonomy to include all elements of an Open API specification that may undergo changes.

Since the API Adapter is an additional layer between the IoT devices and the API server, an additional processing overhead appears; this increases based on the subsequent adaptations applied to the message. Parallelism and caching strategies can be evaluated to reduce this overhead. In addition, one could integrate SDN-based approaches and multi-versioning in the same solution along with a smart mechanism to choose the most appropriate option to precisely meet the needs of a given IoT device.

Furthermore, as future work, there needs to be an experimentation that determines the cost of maintaining this adaptation service for a prolonged period of time. We must also compare the approach to find advantages and disadvantages with other approaches, like pushing firmware updates to the IoT devices.

Finally, during the evaluation, we observed that a large amount of adaptations are registered in the system every time the process is executed, some of these adaptations even being redundant. Because of this, future work should

include an additional process that compares the target API against all previous registered versions to reduce the amount of adaptations each previous API version would need to match the target.

References

1. Bassi, A., Bauer, M., Fiedler, M., Kramp, T., van Kranenburg, R., Lange, S., Meissner, S.: Enabling Things to Talk: Designing IoT Solutions with the IoT Architectural Reference Model. Springer (2016)
2. Burrus, D.: The internet of things is far bigger than anyone realizes (11 2014), https://www.wired.com/insights/2014/11/the-internet-of-things-bigger/
3. Carnie, A.: Syntax: A Generative Introduction. Wiley-Blackwell (2013)
4. Chafle, G., Dasgupta, K., Kumar, A., Mittal, S., Srivastava, B.: Adaptation in web service composition and execution. In: 2006 IEEE International Conference on Web Services (ICWS'06). pp. 549–557. IEEE (2006)
5. Commission, E.: ICT-15-2019-2020. http://ec.europa.eu/research/participants/portal/desktop/en/opportunities/h2020/topics/ict-15-2019-2020.html, accessed June 11, 2018
6. Commission, E.: ICT-16-2018. http://ec.europa.eu/research/participants/portal/desktop/en/opportunities/h2020/topics/ict-16-2018.html, accessed June 11, 2018
7. Gamma, E.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley (1994)
8. H. T. Tran, H.B., Kuppili, R.: A notification management architecture for service co-evolution in the internet of things. IEEE 10th International Symposium on the Maintenance and Evolution of Service Oriented and Cloud-Based Environments pp. 9–15 (2016)
9. He, Q., Yan, J., Jin, H., Yang, Y.: Adaptation of web service composition based on workflow patterns. In: International Conference on Service-Oriented Computing. pp. 22–37. Springer (2008)
10. Henkel, J., Diwan, A.: Catchup! capturing and replaying refactorings to support api evolution. Proceedings of the 27th International Conference on Software Engineering **27**, 274 – 283 (2005)
11. Initiative, O.A.: Open api (6 2001), https://www.openapis.org/
12. Kim, M., Notkin, D.: Discovering and representing systematic code changes. In: 2009 IEEE 31st International Conference on Software Engineering. pp. 309–319. IEEE (2009)
13. Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: 29th International Conference on Software Engineering (ICSE'07). pp. 333–343. IEEE (2007)
14. L. Li, W. Chou, W.Z.: Design patterns and extensibility of rest api for networking applications. IEEE Transactions On Network And Service Management **13**(1), 154–167 (5 2016)
15. L. Li L., W.C.: Compatibility checking of rest api based on coloured petri net. Lecture Notes in Business Information Processing (246), 25–43 (3 2016)
16. Nguyen, H.A., Nguyen, T.T., Wilson Jr, G., Nguyen, A.T., Kim, M., Nguyen, T.N.: A graph-based approach to api usage adaptation. ACM Sigplan Notices **45**(10), 302–321 (2010)

17. Oberhauser, R.: A hypermedia-driven approach for adapting processes via adaptation processes. In: 2015 8th International Conference on Advanced Software Engineering & Its Applications (ASEA). pp. 73–80. IEEE (2015)
18. Perkins, J.: Automatically generating refactorings to support api evolution. ACM SIGSOFT Software Engineering Notes **31**, 111 – 114 (2006)
19. Sanctis, M.D., Geihs, K.: Distributed service co-evolution based on domain objects. Lecture Notes in Computer Science **9586**, 48–63 (2016)