Dealing with IoT Defiant Components

Adrilene Fonseca*, Denis Sousa*, Matheus Chagas*, Paulo Henrique M. Maia*,
Lucas Alves*†, Victor Praxedes†, and Ernandes Junior†
*State University of Ceará, Fortaleza, CE, Brazil
{adrilene.fonseca, denis.sousa, matheus.chagas,}@aluno.uece.br, pauloh.maia@uece.br
†Instituto Atlântico, Fortaleza, CE, Brazil
{lucas_alves, victor_praxedes, ernandes_azevedo}@atlantico.com.br

***Abstract*—A defiant IoT component is a smart device of an IoT system-of-systems (SoS) that should adapt its local behaviour to accomplish the new global requirements of the SoS only in exceptional scenarios. There are several self-adaptive approaches for system-of-systems in the literature. However, to handle defiant components, there is currently only one solution, called cautious adaptation, which relies on the use of AOP-based wrappers. That approach may not be appropriate for IoT systems since the components expose their communication API. Therefore, this paper proposes a new cautious adaptation approach for IoT SoS that adds three external components, the Configurator, the Observer and the Effector, that realize the MAPE control loop to apply behaviour adaptation. The solution is evaluated through a proof of concept using two types of IoT device communication mechanisms that demonstrated that our proposal helps the SoS to achieve its global goals.**

***Index Terms*—defiant component, cautious adaptation, internet of things, smart homes**

I. Introduction

A BCC Research study[1] showed that, in the last 20 years, smart home technologies have evolved and improved significantly, allowing the communication among different devices and the interaction of them with users [1]. A smart home is an Internet of Things (IoT) application that provides a set of intelligent services to users by means of smart devices that require the minimum of human intervention [2].

An IoT environment, such as a smart home, is composed by a set of independent devices that have their local requirements and that together cooperate to accomplish a global requirement (e.g, saving energy or assisting elderly people), thus realizing the notion of system-of-systems (SoS) [3]. In fact, IoT is one of the most common example of a SoS [4]. In this new context grounded by the communication of a variety of devices, emergent behaviours may arise due to the fact that the devices may behave differently from their original design. Therefore, the need for device integration in order to achieve a common goal may require some adaptive behaviour to meet new global requirements or to avoid possible conflict of their local requirements [5].

Smart devices should be able to process information, perform self-configuration, self-maintenance, self-repair and make independent decisions [6], thus allowing IoT systems to become self-adaptive systems (SaS), *i.e.*, systems that are capable to adjust their behavior in response to some variation in the environment [7]. Typically, SaS decisions are partly made at run time through a feedback control loop, such as the MAPE-K (Monitoring, Analyzing, Planning and Executing over a Knowledge base), which is by far the most common type of control loop used to devise self-adaptive software systems [8].

Some work have already addressed self-adaptive IoT systems [9]–[14]. However, they either focus on adapting the system infrastructure or the system exchanging messages, not focusing on device behaviour adaptation to meet the SoS global goals. In this direction, Maia *et al.* [15] proposed a cautious adaptation solution to deal with *defiant components*, which are a SoS component that should change its behaviour only in exceptional scenarios in order accomplish the SoS global requirement. That approach consists of creating a wrapper, implemented using Aspect Oriented Programming (AOP), to change the component behavior only during the exceptional scenario, keeping the original behavior in other situations. In that approach, the use of wrappers was appropriate since the application source code is inaccessible. However, in domains where there is another level of source code access, like IoT systems, which usually provide service APIs and whose components communicate in a different way [14], a wrapper may not be the most suitable solution.

Therefore, this paper brings a new cautious adaptation strategy for IoT environments, in which the devices interact via a communication interface, which can be either a middleware or a broker. The proposal relies on adding three components, the *Configurator*, which sets necessary parameters to run the system, the *Observer*, responsible for monitoring and analysing the message exchange among the devices in order to detect the exceptional scenarios that trigger the adaptations, and the *Effector*, which applies the planned adaptation actions to the defiant component via API calls. Together those two last components realize the MAPE phases. Our main contribution is threefold: (i) a new cautious adaptation mechanism for IoT systems; (ii) a process to instantiate the proposed approach and (iii) two open source exemplars of the analysed system, one with a broker communication and the other one with a real IoT middleware, that can be reused and extended by the community.

The remainder of the paper is structured as follows. Section

[1]https://blog.bccresearch.com/the-evolution-of-smart-home-technology

II introduces a motivating example. Section III details the proposed approach, while Section IV explains the implementation of two exemplars of the motivating example using different communication interfaces. Section V describes the evaluation carried out, while Section VI discusses limitations and a possible extension. The main related work are presented in Section VII. Finally, Section VIII draws conclusions and future work.

II. Motivating example

Ross and Rachel are a couple that live in a smart home equipped with several smart devices, like lamps, voice assistants, and TVs, among others. Every night, they put their daughter, Emma, to sleep in her bedroom. While Rachel sings to make Emma sleep, Ross sets the baby monitor to check baby's health and sleep during the night. The data collected is sent to a specific mobile app. When the app detects a abnormality, it sends a notification to the parents' smartphone, but they could not see it, for being distracted by watching a series on the streaming app of the smart TV, for example. In this case, the app can forward the notification to the TV, interrupting the streaming transmission and alert Emma's parents that she needs care.

In the above example, the IoT system-of-systems is composed by the baby monitor device, the smartphone app that receives and shows the baby's health data, and the smart TV. We assume that the user provides permission for the app to send messages to the smart TV that are displayed in its screen. The global requirement of this SoS is to increase the chances of an emergency notification, such as no breathing detection or intense crying detection, to be seen by parents. Hence, the baby can be assisted when (s)he needs the most.

The devices of this IoT SoS interact by exchanging messages via a communication interface, which can be either a middleware or a broker. The former is a software layer that manages the heterogeneous smart devices, their functionalities and their interaction [16], while the latter performs a publish/-subscribe communication, collecting data from a publisher IoT device and sending it to the corresponding subscribers [17].

Messages coming from the baby monitor can be classified as *normal*, meaning that the baby is fine, or *critical*, indicating that something is not right, such as the child is crying or not breathing for more than five seconds. The first ones are not alerted when received by the smartphone and are only visible when the user access the application. However, critical messages generate a smartphone notification (usually accompanied by a sound and/or a vibration alert) that requires a confirmation within a time limit (ten seconds, in this case), otherwise the baby monitor device will keep sending those messages. When the confirmation is not sent, the Baby Monitor app forwards the critical message to the smart TV and waits for the acknowledgment of TV that the message has been displayed. When that happens, it sends the confirmation to the baby monitor device, thus making it stopping sending the critical messages.

The smart TV has a feature that allows messages, which can come from either other devices or the TV itself, to be shown while it is displaying a channel. However, it also has a local requirement of blocking messages to be displayed when it is running third-party applications, such as a movie streaming application. As the TV was not designed to work in conjunction of a baby monitor, it did not provide the feature of unblocking the critical messages that come from the baby monitor app and, for this reason, may not be willing to change its behaviour. Consequently, a conflict emerges in the exceptional scenario in which the smart TV is in the same IoT environment of the baby monitor device and its corresponding mobile app, which tries to forward the notification to the TV, but it is blocked. Therefore, in this example, the smart TV can be seen as a *defiant component* [15]. In this example, the adaptation action could be forcing the TV to stop the application such that the critical messages forwarded by the smartphone can be displayed, therefore increasing the chance of the baby to be assisted.

In a general way, a possible cautions adaptation solution for exceptional scenarios in a IoT SoS consists on a mechanisms that monitors the messages exchanged by the devices, detects when an exceptional scenario occurs and then executes the necessary adaptation actions. Note that this solution is only applicable for the identified exceptional scenario, i.e., it does not interfere in the original behaviour of the smart devices. In the next section we present a way to implement that approach.

III. The proposed IoT cautious adaptation solution

A. Solution Overview

Our solution consists of introducing three new components into the IoT SoS: *Configurator*, *Observer* and *Effector*, as shown by Figure 1. To define the approach, we made the following architectural decisions: *(i) implementing the MAPE-K feedback loop*, since it is the most common approach for promoting adaptation in SaS [8]; *(ii) dividing the MAPE-K activities between the new components*; and *(iii) using the device's API to adapt the defiant component*, as it is a common feature on IoT devices.

Regarding the item (ii), the Observer component is responsible for watching the system scenarios; thus, it runs the monitoring and analysis phases. The Effector is responsible for applying the adaptation actions on the necessary components, hence, it performs the planning and executing activities. Both components run collaboratively and in parallel, therefore, they depend on each other to work effectively. Finally, to a certain extent, we can say that the Configurator plays the role of the knowledge base, since it contains the information about the adaptation strategies set up at SoS initialization, even though it is not updated at runtime.

The *Configurator* is responsible for providing the necessary information to run the system and to load the Observer and the Effector. Technically, it is a Restful web service that uses the HTTP protocol and receives the input data from the user in a JSON format. This data is divided into three parts: *(i)*

communication interface; *(ii) scenarios*; and *(iii) adaptation actions*.

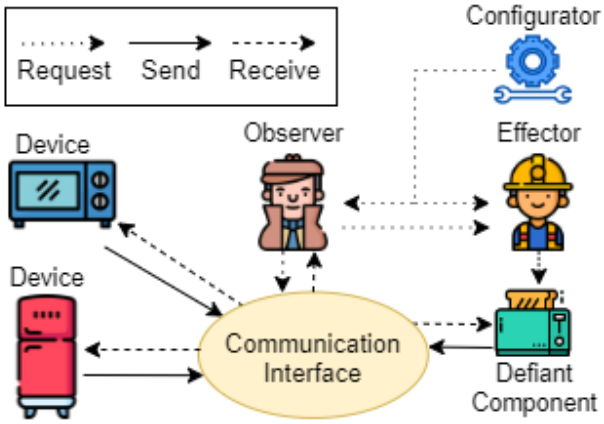

Fig. 1: Solution overview in IoT systems

The communication interface block contains the information about the interface that the devices use to communicate to each other, such as the type of the interface (broker or middleware), the address where it is hosted and authentication data. Depending on the type of interface, other parameters may be added in order to define communication logic, protocol definition, among others.

In the scenarios area, the user inputs the message structure that defines both normal and exceptional scenarios. The structure can be a single or a sequence of messages. In addition, each exceptional scenario of the system must be represented by a name.

Finally, in the adaptation part, the user defines both the adaptation actions and the actions to return the system original behaviour. To implement the adaptation actions, we follow a common practice in IoT systems that is using the devices' API to interaction. Therefore, it is informed the method to the API request, the endpoint (url) and the request body. For each set of actions, they also must be represented by a name that matches to the specified exceptional scenario.

To monitor the system, the Observer gets the messages that are exchanged via the used communication interface. It knows the normal and exceptional scenarios of the SoS by accessing this information that has been loaded from the component *Configurator*. Thus, for each exchanged message, it compares them to the defined scenarios in order to analyse whether an adaptation is needed, *i. e.*, if an exceptional scenario is occurring.

Likewise the Observer, the Effector also accesses the information loaded by the Configurator, but now aiming at obtaining the sequence of actions that were planned to execute the adaptation. Thus, every time an adaptation is required, it executes the actions for the exceptional scenario that is occurring.

In summary, the Observer detects when an adaptation is required and triggers the Effector to do it. Then, when the system returns to the normal scenario, the Effector is triggered again to return the defiant component to its original behaviour.

B. Solution Instantiation

Figure 2 presents the process with the steps necessary to use our solution. It is divided into two phases: modelling solution, which occurs at design time, and application of the cautious adaptation, whose activities happens at run time. Firstly, it is necessary to configure the JSON used by the Configurator. We start by setting the communication interface parameters, followed by configuring both normal and exceptional scenarios, and then end up defining the adaptation actions. After that, the user starts the system execution by sending a request to the Configurator, which loads the Observer and the Effector with the information provided. This finalises the design-time phase.

Considering the runtime phase, the Observer connects to the communication interface and starts monitoring the messages exchanged by the devices. Since the scenarios are defined by messages, the Observer keeps comparing each message received to the defined exceptional scenarios to analyse whether an adaptation is needed, so it will trigger the Effector to adapt the system.

The Effector applies the modelled actions by sending a request to the defiant component's API. While the adaptation is carried out, the Observer keeps monitoring the system expecting that a normal scenario occurs. When this happens, it will again send a request to the Effector, but now to apply the actions that will make the defiant component return on its original behaviour.

IV. Example Simulation

To demonstrate our solution, two IoT environments were created: one using a communication via broker and the other one using a middleware to manage the devices. In both cases, the devices were simulated using Flask[2]. All the implementations are publicly available in our Github[3]. In this section, we firstly explain how the simulations work using the two different types of communication and, subsequently, how the solution was developed using the previous described components.

A. Simulation using broker

To simulate the approach that uses a broker, we used RabbitMQ[4], an open-source broker with resources like low rate of messages loss and performs communication using AMQP (Advanced Message Queuing Protocol), which uses a publish/subscribe architecture to provide reliability [18].

We implemented a web page that shows at real time the messages exchanged by the devices, as depicted by Figure 3. The interface provides buttons to interact with each device, being able to start and stop. Considering the smartphone, there is a possibility to confirm a notification, if any, while the smart TV can be blocked by the user.

Before the communication execution begins, the broker must be configured with exchanges, queues and routes settings

[2]https://flask.palletsprojects.com/en/1.1.x/
[3]https://github.com/BabyMonitorSimulation
[4]https://www.rabbitmq.com/

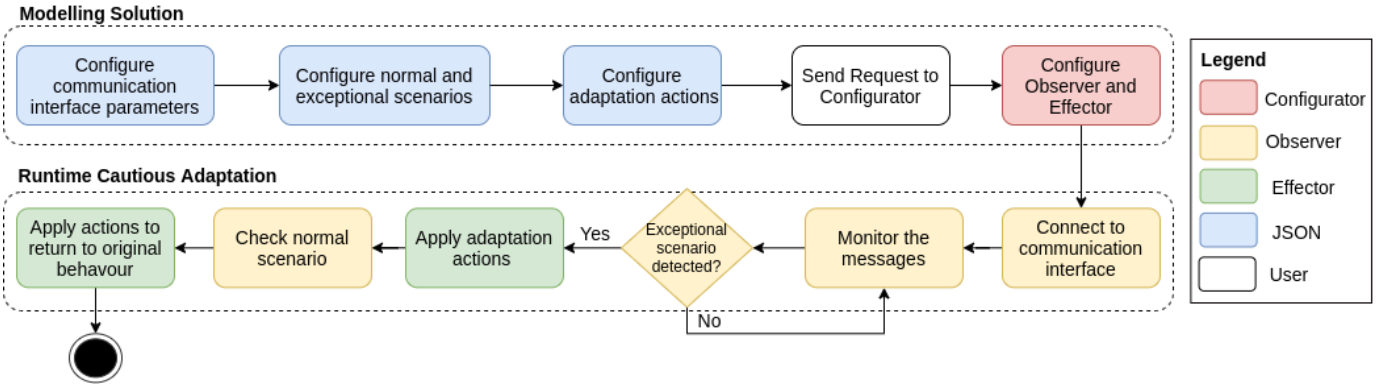

Fig. 2: Solution instantiation process

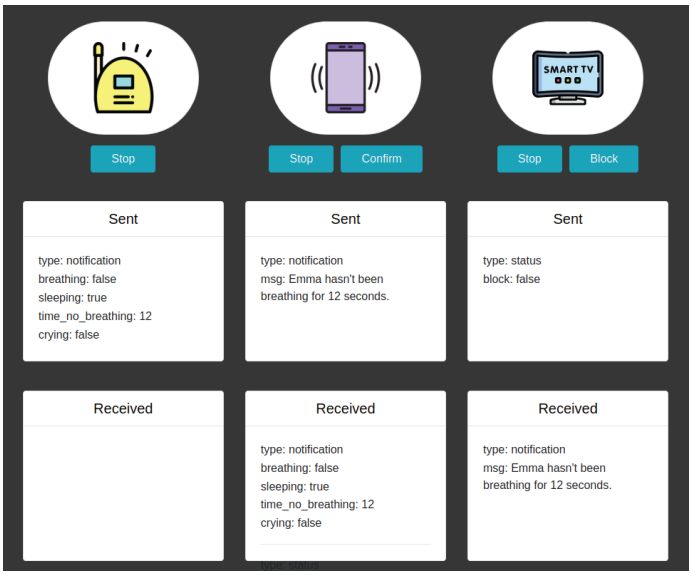

Fig. 3: Interface of Baby Monitor simulation using broker

according to the AMQP protocol. In this example, an exchange was defined to be responsible for routing the messages to the correct queues. In addition, each device has its own queue, which consumes the messages that it receives and that is linked to a route. Thus, a message is sent by a publisher object and delivered to the subscriber's queues.

B. Simulating using a middleware

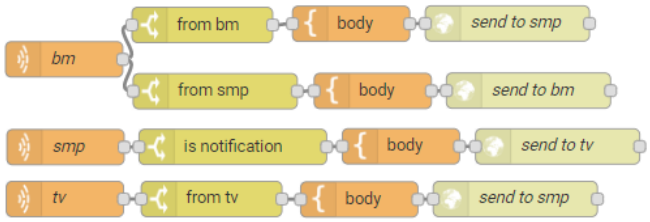

Fig. 4: Execution flow on Dojot platform

For the simulation with a middleware, we used Dojot[5], an open-source platform that helps developing IoT ecosystems. It provides the devices' management, creates data stream and rules for the system, and other features to help developers building IoT systems using best practices. Each device (baby monitor, smartphone and smart TV) was implemented as servers that provide APIs to interact with them.

To make the Dojot working as expected, we must define each device with a virtual representation inside it, so it will be able to receive and manage the data. Also, from the virtual devices, we define a flow that the data will follow. Each message sent from the device to the middleware uses MQTT (Message Queuing Telemetry Transport) protocol. For the messages coming from the middleware to the device, the HTTP is used. Furthermore, to help the Dojot identifying who the message is for, its structure includes labels with information about where it comes from and where it is going to.

To simulate the Baby Monitor, we created the flow presented in Figure 4. Any interaction between the devices is made through the Dojot using the virtual devices. Thus, the middleware reads the messages received by the virtual devices and sends them to the proper simulated device, according to what has been defined in the flow. For instance, when a message is received by the virtual baby monitor, it checks if it is coming from the baby monitor device (baby's data) or from the smartphone (confirmation). In the first case, it sends the data to the smartphone device, while in the other one it sends to the baby monitor device. This way, the data in the Dojot and in the simulated devices remains synced.

C. Solution implementation

Firstly, we developed the Configurator, which provides an API that has an endpoint to receive the configuration that comes as a JSON. When the request is received, it uses methods to separate the information destined to the Observer and for the Effector. Each component also has endpoints to configure them, so the Configurator sends the configuration information to them using these endpoints. Additionally, the

[5]http://www.dojot.com.br/

Effector provides routes to request an adaptation and to return to normal behavior. These routes call the methods that execute the adaptation strategies configured and they are used by the Observer.

```
1  {
2    "interface_type": "broker",
3    "connection_config":
4      { "host": "<broker_host>",
5        "user": "<broker_user>",
6        "password": "<broker_password>",
7        "port": "<broker_port>",
8        "exchanges": ["exg_baby_monitor"] },
9    "normal_scenario": [
10     { "topic": "bm_info",
11       "type": "status" }
12   ],
13   "exceptional_scenario": {
14     "tv_blocked": [
15       { "topic": "st_msg",
16         "type": "notification",
17         "body": "*" },
18       { "topic": "st_info",
19         "type": "status",
20         "body": {"block": true} }
21     ]
22   },
23   "adaptation_actions": {
24     "tv_blocked": [
25       { "method": "POST",
26         "url": "<host>/tv_status",
27         "body": {"lock": false} }
28     ]
29   },
30   "return_to_normal_actions": {
31     "tv_blocked": [
32       { "method": "POST",
33         "url": "<host>/tv_status",
34         "body": {"lock": true} }
35     ]
36   }
37 }
```

Fig. 5: Configuration JSON of broker communication

After that, the Baby Monitor system was modelled. Firstly, we modelled the normal scenario by normal messages coming from the baby monitor, that is, messages of type status. At the exceptional scenario, "tv_blocked" was defined as an order of receiving messages. Initially, it is sent a notification type message to the smart TV and then a message from the TV containing the information that it is blocked is sent. To adapt the "tv_blocked" scenario, a POST request must be sent to TV's API with the body indicating that the lock must be false. To return the system to the previous state, the same request is sent but with the body indicating the lock must be true. Thus, this modelling corresponds to the knowledge that the components have about the system.

Figure 5 depicts an example of a JSON configuration file to the broker communication interface. As our implementation used RabbitMQ, we use the API that it provides to connect to it (lines 1-7). When the Observer establishes a connection, it makes a request looking for all the routes associated to the exchanged defined in the configuration. Then, the Observer creates its own queue and links it to the routes found. As for the scenarios, they were defined indicating the topic where they are published (which corresponds to the routes), the type they have and the body message (lines 8-18). Thus, the Observer compares each message received with them, looking on the route it came from and the type it is. Therefore, the Observer runs as a thread to keep receiving the messages, so it can monitor and analyse them.

By the moment the Observer gets the sequence of messages: (i) messages going to the TV of type notification; and (ii) messages coming from the TV with the *block* key stated as true, it sends a request to the Effector to adapt the component indicating that the scenario "tv_blocked" is happening. Then, the Effector applies the actions defined (lines 29-37), i.e., it sends a request to the TV's API with the *lock* element set to false. As an adaptation was required, when the Observer detects that the normal scenario is running again, it sends a request to the Effector to return the defiant component to original behaviour, that is the "tv_blocked" scenario. Thus, the Effector sends a request to TV's API, setting the lock status to true again, as defined on lines 38-46. In summary, the Effector acts when it is triggered since it is responsible for the adaptation strategies.

Considering the middleware, the configuration is very similar. To connect with Dojot, we used a socket connection, so that the Observer receives in real time each message exchanged inside the middleware. Also, we defined a topic instead of an exchange, to get messages from the Baby Monitor system. To define the scenarios, we defined the destination of the message, instead of the route, as it is present in the body of the message. The adaptation actions and the interaction between the Observer and the Effector work in the same way.

V. Solution Validation

The validation was done through a proof of concept that intended to show whether the approach is viable in an IoT context. We sought to evaluate the following research question:

RQ: Does the approach contribute to increase the chances of a critical message being seen by the parents?

To perform the validation, we used the two simulation tools described in the previous section considering three possible scenarios, based on the exceptional situation described in Section II. Each scenario consisted of a different probability of the defiant component being blocked to receive critical messages and were defined as **favorable**, **medium** and **critical**, in which the TV has 10%, 50%, and 90% of chance of being blocked, respectively.

For each scenario, we performed three tests, called T1, T2 and T3, in which we varied the chances of a confirmation of the critical message being seen by the parents on their smartphone on 90%, 50% and 10%, respectively, which indicates when there is no need to forward the message to the TV. For each combination of scenarios, which encompasses the probability of the messages being sent to the TV and probability of the TV being blocked, 100 executions were

				Simulation Using Broker					Simulation Using Middleware				
				Without the solution		With the solution			Without the solution		With the solution		
	Tests	Block	Confirmation	Success%	Failures%	Success%	Failures%	Obs/Eff%	Success%	Failures%	Success	Failures%	Obs/Eff%
Favorable	T1	10%	90%	100%	0%	100%	0%	0%	96%	4%	100%	0%	0%
	T2	10%	50%	100%	0%	100%	0%	2%	94%	6%	100%	0%	6%
	T3	10%	10%	92%	8%	100%	0%	10%	92%	8%	100%	0%	10%
Medium	T1	50%	90%	94%	6%	100%	0%	2%	98%	2%	100%	0%	10%
	T2	50%	50%	82%	18%	100%	0%	22%	78%	22%	100%	0%	32%
	T3	50%	10%	60%	40%	100%	0%	54%	48%	52%	100%	0%	48%
Critical	T1	90%	90%	94%	6%	100%	0%	4%	92%	8%	100%	0%	6%
	T2	90%	50%	56%	44%	100%	0%	32%	50%	50%	100%	0%	46%
	T3	90%	10%	16%	84%	100%	0%	82%	16%	84%	100%	0%	70%

Fig. 6: Experiments Results

performed, half of them with the proposed solution and the other half without the solution. We consider an execution as *successful* when the critical message sent from the baby monitor device to the smartphone is forwarded to the TV and a confirmation that the message has been displayed is sent back, otherwise the execution is said to be *failed*. The probability values were inserted directly into the source code. Finally, we run the tests with both the broker and the middleware simulation.

The experiment results can be seen in Figure 6. Considering the experiments using broker and without our solution, we can see that, for a favorable scenario, there was failure only in cases where the parents were more inattentive (the notifications were seen only 10% of the times), when 8% of the critical messages were lost. Considering the experiment with the middleware, the failure rate increases as the parents see less the notifications, having also 8% of the critical messages being sent. For the same scenario, but now using our proposed solution, for both types of simulation, 100% of the executions were successful. In addition, we can see that the Observer and Effector components were triggered in only 10% of the times (maximum).

As expected, the more serious the scenarios become, the more important is our solution to achieve the global requirement of the SoS. For example, in the medium and critical scenarios, in the worst situation, there was a failure rate of 40% and 84%, respectively, for the simulation with the broker, and 52% and 84%, respectively, for the middleware counterpart. This represents great risk to the baby's health.

By analysing the experiments with the components intervention, we can see that, regardless of the scenario and the communication interface, there was no block of critical messages by the TV, which means that whenever the baby needed assistance, the parents were able attend to him/her. Furthermore, still considering the worst case of the medium and critical scenarios, the proposed components intervened in 54% and 82% of the times in the simulation with the broker, respectively, and 48% and 70% of the times in simulation with the Dojot middleware. Therefore, it is noteworthy that the proposed approach has a great influence on the success of the IoT SoS global requirement.

By comparing the tests of the simulation with the broker and with the middleware, we can see that the results were very similar, being the differences caused by the randomness of the probability distributions of each scenario. This shows that our solution can work well considering both types of communication interface, wherefore we can answer the posed research question affirmatively, i.e, the approach does contribute to increase the chance of a critical message being seen by the parents.

VI. Discussions

The validation showed that our approach contributed to achieving the SoS global requirement. This section discusses some limitations and new opportunities to apply this work when there are more defiant components.

Limitations and Threats to Validity. Firstly, although the good results, it is not possible to generalise the solution, since the tests were based on a specific example and executed through simulations. Even though we have used a real IoT middleware, the devices were simulated. That factor may affect the results, as it may contain some variables that were not considered. Moreover, the tests were performed using random values that may not reflect what happens in the real life.

It is important to highlight that our approach is not designed to reveal all conflicts in an IoT environment, since the conflicts are particular to the exceptional scenarios caused by the SoS composition. To help that task, there are other approaches, such as a process for modelling adaptation scenarios for SoS [19].

Regarding the validation, the experiments were performed only considering variations of the same scenario of the Baby Monitor system and the smart TV. Therefore, for other scenarios involving other components, new validations are necessary to evaluate the approach.

In the simulations, differences between the execution of the two types of communication were evidenced. For the simulation using broker, due to the execution in a local host, the availability of resources and communication was more stable. Regarding the Dojot middleware, there was a perceptive delay on attending requests because it is hosted on the Cloud. Therefore, the simulation using the broker obtained better

```
1  {
2    "exceptional_scenario": {
3      "tv_blocked": [{...}],
4      "assistant_not_disturb": [
5        { "topic": "assistant",
6          "type": "notification",
7          "body": "*" },
8        { "topic": "assistant",
9          "type": "status",
10         "body": {"msg":"do not disturb"} }
11     ]
12   },
13   "adaptation_actions": {
14     "tv_blocked": [{...}],
15     "assistant_not_disturb": [
16       { "method": "POST",
17         "url": "<host>/assistant_disturb",
18         "body": {"not disturb": false} }
19     ]
20   },
21   "return_to_normal_actions": {
22     "tv_blocked": [{...}],
23     "assistant_not_disturb": [
24       { "method": "POST",
25         "url": "<host>/assistant disturb",
26         "body": {"not disturb": true} }
27     ]
28   }
29 }
```

Fig. 7: Voice Assistant JSON Configurator

performance due to is local host structure. On the other hand, the simulation using Dojot is closer to a real world application, since it is a concrete platform for IoT system development.

New Defiant Component. In the motivating example, we considered only one defiant component and one exceptional scenario. Nonetheless, in a more realistic IoT environment, there can be other devices for which the Baby Monitor app could forward the notification to. For instance, consider that in the smart home of the analysed example there is a voice assistant device that could also receive the critical messages not seen by the parents and then speak them out to alert that the baby needs help. Assume that the voice assistant has a "Do not disturb" mode, which disables notifications. Then, following the same rationale applied to the smart TV in Section II, we can say that the voice assistant is also a defiant component in the exceptional scenario of alerting the parents about the critical condition of the baby.

This new scenario could be configured in the JSON file that is inputted to the component Configurator as shown in Figure 7. It can be seen as an extension to the JSON file shown in Figure 5 to increment the original motivating example with a new defiant component. Note that lines 2, 11 and 17 represent the pieces of code shown in lines 12-18, 20-23, and 25-28, respectively, of Figure 5. Lines 3-9 defines this new exceptional scenario, which occurs when the smartphone app forwards a critical message (type notification) to the voice assistant. Lines 12-15 describe the actions that should be taken to adapt the behaviour of the new defiant component. In this case, turning the "do not disturb" mode to off. Lately, lines 19-21 represent the actions to return the voice assistant to its original "do not disturb" mode.

The Observer detects the exceptional scenario that is happening (whether regarding the smart TV or the voice assistant) by verifying the message structure that has been sent. In case where both devices are available in the smart home, the solution can apply a prioritization strategy to decide which defiant component should be adapted. For instance, a baby-crying message may be less important than a no-breathing one. In the first case, the Observer may decide to adapt only one of the defiant components, while in the second one both components should be adapted to maximize the chance of the baby being assisted. This prioritization is a property that is also addressed in the original cautious adaptation proposal [15]. After the correct exceptional scenario has been identified, the Effector applies the corresponding adaptation actions.

VII. Related Work

Self-adaptive IoT systems have been addressed recently. Focusing on architecture-based adaptation, MARTAS is a model that uses probabilistic models in the feedback loop, where the adaptation decision is based on the quality estimate of each configuration of adaptation available, and provides verification that is suitable for the resource constraints that are present in IoT systems [9]. In [10] the authors propose a control architecture model that uses multiple MAPE-K loops that interact and provide a decentralized adaptation. That architecture model aims at providing autonomic capabilities and decentralized control to smart homes.

The previous approaches propose a way to use feedback loops to perform a better adaptation of the IoT devices. Similarly to our work, those studies focus on the devices configuration as the target of the adaptation. However, they use the architecture of the system to apply the adaptation, while in this work we use the devices' API.

FIoT is a framework that uses artificial intelligence based agents to provide IoT environments that can be managed and recognise smart objects [11]. For each device, an adaptive agent is set and, for the collection of devices, there is an observer agent whose job is to check the environment and if an adaptation is needed. Those agents actions are very similar to the ones proposed in this work, but the main difference is that they set an agent to an individual device, while our agents (Observer and Effector) work over the whole system. Also, their adaptation focuses on the environment management, while in this work we aim at attending the SoS global requirements.

Dealing with queuing messages and real time process, in [12] the problem of queues bottleneck in real-time system process, which can decrease the system performance, is addressed. Similarly to us, the authors also observe the message exchange to perform adaptation. However, the messages are the adaptation target, whereas in this work, they are used to verify the occurrence of exceptional scenarios and, consequently, the need of adaptation.

Motivated by the difficulties faced by IoT engineers on identifying exceptional problems that can appear at runtime,

in [13] three approaches for self-adaptive IoT systems that are linked to infrastructure aspects, like network interference and bandwidth, are presented. Differently, we perform behavioural adaptations that may not related to the system hardware resources.

In [14] it is proposed an approach to adapt the message flow of a SoS through a server. Similarly to it, we also use the data traffic to adapt the systems, but while their adaptation target is the server, in this work we adapt the devices

At last, but not least, none of the aforementioned approaches address the adaptation of defiant components in IoT systems. Therefore, our solution brings a novel contribution for this topic.

VIII. Conclusion

In this paper, we proposed a new cautious adaptation approach of IoT defiant components that consists of adding three new components, called Configurator, Observer and Effector, into the system. The Configurator sets the necessary parameters to run the system, while the Observer is responsible for monitoring the messages exchanged among the devices and triggers the Effector to perform the adaptation actions when an exceptional scenario occurs.

To evaluate our solution, we executed two simulations considering a smart home example, one using a message broker and another one using a real IoT middleware called Dojot. In both cases, the experiments demonstrated that our approach helps the system-of-systems to achieve its global requirement. We also discussed how to deal with a new defiant component introduced in the example.

We are aware that the tests carried out in this work may not fully demonstrate its effectiveness. Thus, we intend to make more exhaustive and efficient tests to validate the proposed solution. Also, as future work, we intend to extend the example with new exceptional scenarios and apply the solution to other IoT systems with different architectures. Finally, we also plan to investigate uncertainties in the context of IoT system-of-systems and propose new adaptation mechanisms.

IX. Data Availability

All data and source code related to the implementation of this work are publicly available at the links mentioned in the footnotes throughout the paper.

X. Acknowledgments

This work is partially supported by CNPq/Brazil under grant Universal 438783/2018-2 and Funcap/Brazil under grant UKA-0160-00005.01.00/19.

References

[1] A. Zaidan, B. Zaidan, Q. M. Yas, O. S. Albahri, A. Albahri, M. Alaa, F. Jumaah, M. Talal, K. L. Tan, W. Shir, and C. K. Lim, "A survey on communication components for iot-based technologies in smart homes," *Telecommunication Systems*, vol. 69, pp. 1–25, 2018.

[2] M. Soliman, T. Abiodun, T. Hamouda, J. Zhou, and C.-H. Lung, "Smart home: Integrating internet of things with web services and cloud computing," in *2013 IEEE 5th international conference on cloud computing technology and science*, vol. 2. IEEE, 2013, pp. 317–320.

[3] M. W. Maier, "Architecting principles for systems-of-systems," *Systems Engineering: The Journal of the International Council on Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.

[4] H. Cadavid, V. Andrikopoulos, and P. Avgeriou, "Architecting systems of systems: A tertiary study," *Information and Software Technology*, vol. 118, p. 106202, 2020.

[5] T. Viana, A. Zisman, and A. K. Bandara, "Identifying conflicting requirements in systems of systems," in *2017 IEEE 25th International Requirements Engineering Conference (RE)*, Sep. 2017, pp. 436–441.

[6] Lu Tan and Neng Wang, "Future internet: The internet of things," in *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, vol. 5, 2010, pp. V5–376–V5–380.

[7] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–26. [Online]. Available: https://doi.org/10.1007/978-3-642-02161-9_1

[8] R. Calinescu, S. Gerasimou, K. Johnson, and C. Paterson, "Using runtime quantitative verification to provide assurance evidence for self-adaptive software," in *Software Engineering for Self-Adaptive Systems III. Assurances*, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds. Cham: Springer International Publishing, 2017, pp. 223–248.

[9] D. Weyns, M. U. Iftikhar, D. Hughes, and N. Matthys, "Applying architecture-based adaptation to automate the management of internet-of-things," in *Software Architecture*, C. E. Cuesta, D. Garlan, and J. Pérez, Eds. Cham: Springer International Publishing, 2018, pp. 49–67.

[10] P. Arcaini, R. Mirandola, E. Riccobene, P. Scandurra, A. Arrigoni, D. Bosc, F. Modica, and R. Pedercini, "Smart home platform supporting decentralized adaptive automation control," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1893–1900.

[11] N. M. do Nascimento and C. J. P. de Lucena, "Fiot: An agent-based framework for self-adaptive and self-organizing applications based on the internet of things," *Information Sciences*, vol. 378, pp. 161 – 176, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025516313664

[12] P. Chindanonda, V. Podolskiy, and M. Gerndt, "Metrics for self-adaptive queuing in middleware for internet of things," in *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2019, pp. 130–133.

[13] D. Weyns, G. S. Ramachandran, and R. K. Singh, "Self-managing internet of things," in *SOFSEM 2018: Theory and Practice of Computer Science*, A. M. Tjoa, L. Bellatreche, S. Biffl, J. van Leeuwen, and J. Wiedermann, Eds. Cham: Springer International Publishing, 2018, pp. 67–84.

[14] R. Bustamante and K. Garcés, "Managing evolution of api-driven iot devices through adaptation chains," in *Proceedings of the XXIII Iberoamerican Conference on Software Engineering, CIbSE 2020, Curitiba, Paraná, Brazil, November 9-13, 2020*. Curran Associates, 2020, pp. 85–95.

[15] P. H. Maia, L. Vieira, M. Chagas, Y. Yu, A. Zisman, and B. Nuseibeh, "Cautious adaptation of defiant components," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 974–985.

[16] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for internet of things: A survey," *IEEE Internet of Things Journal*, vol. 3, no. 1, pp. 70–95, 2016.

[17] S. Sen and A. Balasubramanian, "A highly resilient and scalable broker architecture for iot applications," in *2018 10th International Conference on Communication Systems Networks (COMSNETS)*, 2018, pp. 336–341.

[18] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, 2017, pp. 1–7.

[19] M. Maciel, P. H. Maia, F. C. M. B. Oliveira, and F. Maciel, "Adore: An adaptation-oriented requirement modeling approach for systems of systems," in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, ser. SBES 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 166–171. [Online]. Available: https://doi.org/10.1145/3350768.3353814