A self-managing IoT-Edge-Cloud architecture for improved robustness in environmental monitoring

Andrea Bombarda
University of Bergamo
Bergamo, Italy
andrea.bombarda@unibg.it

Giuseppe Ruscica
University of Bergamo
Bergamo, Italy
giuseppe.ruscica@unibg.it

Patrizia Scandurra
University of Bergamo
Bergamo, Italy
patrizia.scandurra@unibg.it

ABSTRACT

The new distributed paradigm Edge-Cloud Continuum (ECC) is fundamental to provide a continuum of computing from Edge to Cloud and create more dependable Cloud-powered IoT applications. In line with this computing paradigm, this paper presents a self-managing IoT-Edge-Cloud computing architecture for enhancing robustness in environmental monitoring. In addition to the sensor data transmission to a cloud server node for late processing, a semi-decentralized self-adaptation architecture with feedback control components running on edge nodes (gateways) is adopted to monitor the end devices and make decisions in the edge. The proposed control architecture was implemented as part of SEIROB, an IoT-Edge-Cloud exemplar for air quality monitoring, using the Python programming language and the open-source ChirpStack LoRaWAN network server stack. In this paper, we share the architecture design of SEIROB, report our experiments about its effectiveness in achieving some robustness scenarios, and discuss faced challenges and lessons learned in using a ready-to-use IoT infrastructure (like the ChirpStack platform) for ECC.

CCS CONCEPTS

• **Computer systems organization → Dependable and fault-tolerant systems and networks**.

KEYWORDS

IoT-based Environmental Monitoring, Software architecture for IoT-Edge–Cloud computing continuum, self-adaptive IoT

ACM Reference Format:
Andrea Bombarda, Giuseppe Ruscica, and Patrizia Scandurra. 2025. A self-managing IoT-Edge-Cloud architecture for improved robustness in environmental monitoring. In *Proceedings of ACM SAC Conference (SAC'25).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

An Edge-Cloud deployment model, in which data processing and control of operations occur partly as close as possible to where the data is generated (edge/fog computing) and partly on the cloud, is a key solution to cope with several problems of IoT-based applications of *environmental monitoring*. These include excessive manual management and maintenance of devices, network latency, lack of bandwidth, data loss, etc., and are due to sources of uncertainty, which are difficult to predict and solve through the conventional cloud computing model. In particular, it is fundamental to provide a continuum of computing from the edge to the cloud – the *Edge-Cloud Continuum* (ECC) [2, 11]. However, most IoT applications focus only on making data (pre-)elaboration on the edge nodes; less attention has been put till now on deploying also control and decision functions in the edge. In general, ECC infrastructures, frameworks, and engineering methods are still in their infancy and lack standardization and experimentation [8].

In line with the ECC paradigm, this paper reports our experience in engineering SEIROB (SElf-managing Iot-edge-cloud exemplar for ROBust environmental monitoring), a self-adaptive IoT-Edge-Cloud exemplar for enhancing robustness in environmental monitoring. In this respect, developing IoT-based solutions for environmental monitoring poses several challenges. In general, they require hand-tuning and manual maintenance. Moreover, they have to operate in open/semi-open spaces, so it emerges the need for a capillary distributed monitoring and control software architecture to track, and eventually mitigate, device issues and processes with environmental impact. Current ready-to-use management infrastructures for IoT networks available on the market do not provide native support for such features and for ECC, in general.

SEIROB was designed for the IoT domain of environmental monitoring and was first evaluated for air quality monitoring. In architecting and implementing SEIROB, we tried to reflect the ECC deployment model, thanks to the adoption of Arduino-based devices with low-cost sensors (wireless sensor nodes), Raspberry Pi-based micro-computers (edge nodes) with high computing capabilities working also as gateways, and the open-source IoT network/server infrastructure ChirpStack LoRaWAN[1] with lightweight communication protocols – LoRaWAN and Message Queue Telemetry Transport (MQTT). The control software architecture of SEIROB is made of a collection of control components that follow and/or support the activities of the feedback control loop model MAPE-K (Monitor-Analyze-Plan-Execute over a Knowledge base) [7] for self-adaptation. These control loop components are developed as microservices in the Python programming language and are deployed in a semi-decentralized manner over the edge and server/-cloud nodes to discover and automatically report and/or manage robustness-related device issues at runtime (e.g., low device's battery level, silent nodes, network instability, etc.). Such control loop

SAC'25, March 31 –April 4, 2025, Sicily, Italy

ACM ISBN 979-8-4007-0629-5/25/03…$15.00
https://doi.org/xx.xxx/xxx_x

[1] https://www.chirpstack.io/project/

components form the control cycles along the continuum. To evaluate the capability of SEIROB to meet robustness goals along the continuum, we conducted experiments on both simulated and real IoT physical setups based on some robustness scenarios of interest. In this paper, we share the architecture design and validation results of SEIROB, and discuss faced challenges and lessons learned in using a ready-to-use IoT infrastructure (like ChirpStack) for ECC.

This paper is organized as follows. Sect. 2 introduces the problem domain, key challenges, and some robustness goals to achieve at runtime. Sect. 3 presents the SEIROB software architecture. Sect. 4 evaluates by scenarios the proposed architecture and analyzes the practical usage of SEIROB using examples of IoT deployments, while Sect. 5 discusses how we addressed the key challenges and lessons we learned in developing SEIROB. Finally, Sect. 6 describes relevant related work, and Sect. 7 concludes the paper.

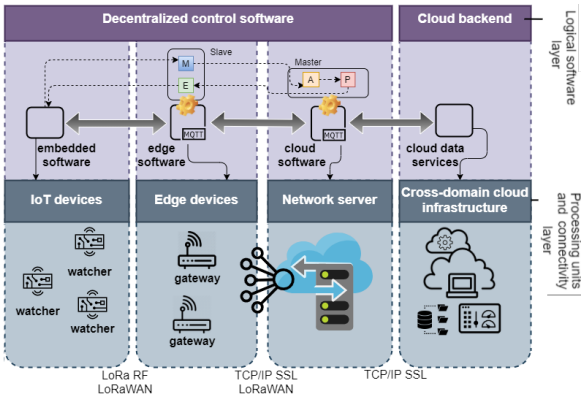

Figure 1: SEIROB's conceptual architecture

2 APPLICATION DOMAIN AND CHALLENGES

Environmental monitoring poses several challenges for IoT systems. The devices are placed in (semi-)open spaces, are often poorly guarded, easily breakable, and can cease to function in an unpredictable way (the battery, for example, has a limited lifespan). There is the need, therefore, to improve the robustness of the IoT deployment itself and reduce the number of manual maintenance interventions and their costs. There is also a need for reducing unnecessary data flows to the cloud and increasing efficiency and reliability at the edge/end nodes for these IoT systems. To this end, monitoring and self-adaptation mechanisms are prominent to target these robustness goals [6, 16]. In particular, the main key challenges we see in developing a robust IoT-Edge-Cloud system for environmental monitoring can be summarized as follows:

Automated maintenance and failure recovery. Due to the decentralized nature of the information flow within and across nodes, and to the nodes' redundancy (e.g., redundancy of LoRa gateways), there are no single points of failure. However, silent nodes/services, lost devices, and hardware/software failures are to be detected and recovered since they cause loss of data and/or not up-to-date information on a possible remote cloud dashboard about the monitored phenomena, and cause the system degradation itself. How to automate these diagnostic and control functions over the intermediary edge-cloud network infrastructure, and with the types of sensor nodes available?

Control loop design complexity. How to design control loop components for monitoring and adaptation over the IoT-Edge-Cloud computing continuum? How to exploit the native support of the underlying infrastructure management platform to implement control and communication functions?

Control loop deployment complexity. How to allocate and start computational activities of the controller components on the three types of nodes with the support of the underlying infrastructure management platform?

As a solution domain, in our vision, the robustness of an IoT-Edge-Cloud application for environmental monitoring shall be increased by self-adaptation over the computing continuum. In the SEIROB exemplar (see Sect. 3 for more details), in particular, we adopted the solution to realize a semi-decentralized control software, i.e., controller components for self-managing functions are deployed over the edge-cloud nodes and cooperate to automate and/or make the diagnosis more robust and mitigate problems and malfunctions of the devices themselves (such as low battery levels, silent devices, unstable network/back-end services with possible data loss, etc.).

3 SEIROB EXEMPLAR FOR ENVIRONMENTAL MONITORING

This section describes the infrastructure and software architecture of the SEIROB exemplar[2]. First, we present our architectural vision for SEIROB (see Fig. 1), and then its concrete implementation.

Sensor nodes (also called *watchers*) in the IoT network relay their sensor data to the gateway(s), assuming they are directly reachable by at least one gateway (single-hop routing). Both device-to-edge/cloud (*uplink payload*) and edge/cloud-to-device (*downlink payload*) communication are supported. IoT watchers periodically send telemetry from the sensors to control services hosted in the edge/cloud computation nodes and, vice-versa, control services may send configuration commands to the watchers. Received uplink data could be further forwarded to cloud back-end applications for analytics, alerting, and data visualization dashboards for human operators. A set of MAPE-K [7] feedback loops run on edge and cloud/server nodes and ensure the achievement of robustness goals along the continuum. These MAPE components may be centralized or decentralized[3] according to the distribution granularity and IoT architectural patterns for self-adaptation [13, 14, 20].

The main robustness scenarios we devised for SEIROB are summarized in Tab. 1. The scenarios are organized by the type of robustness goal that the system adaptations should achieve, and the type of adaptation strategy and distribution (sensor, edge, or cloud nodes) along the continuum.

3.1 SEIROB infrastructure

An IoT-Edge-Cloud installation, according to our exemplar's network topology, may consist of: wireless sensor nodes, which are

[2]Replication package for SEIROB: https://doi.org/10.5281/zenodo.7944213
[3]Fig. 1 shows an illustrative example of a decentralized master/slave configuration.

Table 1: Examples of robustness scenarios for IoT-Edge-Cloud applications

Robustness scenario	Adaptation strategy	Type of distribution	Adaptation Goal
S_1: Watcher's battery rationing	According to certain thresholds (in percentage) of the battery level: • increase or decrease data transmission frequency of the watcher • increase or decrease the transmission power of the watcher	Sensor nodes and cloud nodes	Self-configuring
S_2: High availability of watchers	Upon identification of silent watchers, alert a human operator	Edge nodes and cloud nodes	Self-healing
S_3: High availability of edge nodes	Upon detection of a silent edge node: • alert a human operator • force the attached watchers to connect with a different edge node/gateway	Cloud nodes	Self-healing
S_4: Limit the loss of measured data when the app server is silent	Upon detection that the app server is silent: • edge buffering of latest watchers' data till the app server resumes • alert a human operator	Edge nodes	Self-healing

Arduino MKR-based hardware prototypes (developed in-house); edge nodes, which are Raspberry Pi-based LoRa gateways equipped with the ChirpStack Gateway OS; and server/cloud nodes hosting the Chirpstack network/application server. In the current prototype of SEIROB, geolocated multi-sensor devices (sensor nodes) and gateways (edge nodes) can be deployed and connected using a single-hop routing model. So, we assume the target environment is coverable by the radio range of single sensor nodes connected to gateways[4]. Occasionally, sensor nodes and edge nodes could be manually moved to optimize the environment coverage at scale or to obtain a better area coverage for difficult zones [17].

The Chirpstack LoRaWAN Network/Server is an open-source ready-to-use platform for LoRaWAN and IoT networks. It has a user-friendly web interface for device management and APIs for integration with external services. Device payloads are encoded in the JSON format and sent and/or received through the application protocol MQTT, which is seeing increased attention and industry adoption for its robustness [9]. MQTT follows a publish-subscribe model over TCP; appropriate Chirpstack event types (up, ack, join, echo, and status error events) and command types (down and ping) are represented by MQTT topics.

3.2 SEIROB air quality watcher

The watcher's embedded software runs on an Arduino MKR WAN with LoRa connectivity (i.e., the sensor node). Such a board is equipped with low-cost components and sensors for measuring air pollutants and monitoring the device itself. These include sensors for PM10, temperature, humidity, ozone, benzene, ammonia, aldehydes, GPS location, and battery level. The watcher prototype was developed in-house by adopting code-level idioms/patterns for reconfigurability: dynamically and remotely configurable parameters to change the frequency of readings from sensors[5] and of their sending to the gateway, bit array to activate/deactivate long sensors readings or calibration tasks, etc. The watcher is therefore able to receive and execute specific configuration actions as dictated by the remote controllers hosted in the edge and server nodes.

Fig. 2 shows the life cycle behavior of the watcher using a UML state diagram. The watcher goes through two main phases, *Setup* and *Loop*, that correspond to the two main steps of an Arduino sketch. First, during initialization (state *Setup*), initial default values are assigned to internal variables and all (remotely settable) parameters enabling the proper operation of the sketch. Then, the watcher initializes the LoRa module and, after that, it tries to join the LoRa network by going through an activation process according to the Over-The-Air Activation (OTAA) join procedure. To this purpose, the watcher communicates with the LoRa network server and shares appropriate session keys (*appEUI* and *appKey*, provided by the network server). If the watcher fails to find an available gateway among those pre-integrated with the network server, after 1 minute it tries again the join procedure. As long as it cannot connect to a gateway, its execution does not proceed. Once a connection to a gateway is established, the *Loop* phase is entered. The watcher moves to state *Data Read*, where sensors' data are read after a certain (sending) period elapses. Then, it goes to state *Send uplink message* to send an uplink message to the gateway with the sensors data, and possibly goes to state *Receive downlink message* to receive a downlink message from the gateway with a valid payload containing the new configuration values to actuate in state *Variables configuration*. In case a watcher loses connection with the gateway during the *Loop* phase, it tries to re-join to it (this is not shown in Fig. 2 to keep the diagram simple).

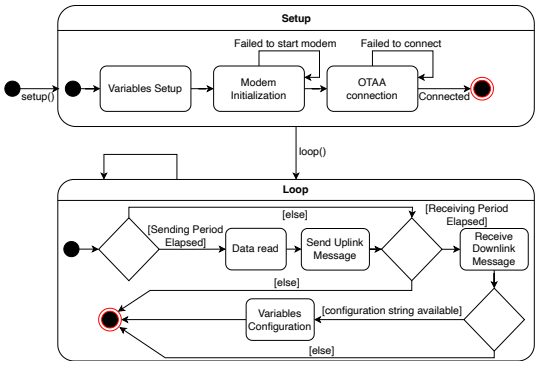

Figure 2: Watcher's life cycle

The watcher's probes and effectors are concretely represented in terms of sensor values transmitted via, respectively, uplink and downlink message payloads (base64 encoded) of the IoT Chirpstack-based network. An example of an uplink message payload is:

```
0.62|19.40|50.00|0.05|302320|12.34|6.29|45.80|9.80|99.80
```

[4]We did not adopt the multi-hop routing model, where out-of-range devices use other devices as relays towards a gateway to connect to the Internet.
[5]It could be necessary to decrease the frequency to rationate the device's battery, but sometimes also to increase it to better observe an environment critical phenomenon.

where the first seven real values are pollutant measures (parsing/unparsing of such string format is position-based and depends on a fixed array of sensor names), followed by the GPS latitude and longitude values, and the final value is the percentage of battery level. Similarly, an example of a downlink message payload is as follows:

```
1|1|1|1|1|1|1|1|20000|30000
```

where the first 8 bit-values (0 or 1) represent the request to activate or de-activate the reading from a sensor (the correspondence is position-based). The last two values (in ms) are the parameters *tts* (time to send) and *ttr* (time to receive) used in the Arduino code as part of conditions regulating the frequency at which it reads and sends telemetry to the gateway; the admitted value range for such parameters is $[10000, 200000]$, otherwise the command is ignored as it would be not compliant with system requirements.

3.3 SEIROB architecture design

We here describe the design of the SEIROB software architecture as a concrete implementation of the conceptual architecture shown in Fig. 1. Specifically, we briefly describe two design options that we shaped in an incremental way to cover all scenarios: the first option is for scenarios S_1, S_2, and S_3, while the second option allows also the realization of S_4. Overall, both the two architecture options reflect two *multi-adaptation* patterns [14] with the MAPE-like adaptation mechanism: Synthesize-Command (SC) and Collect-Organize (CO). SC is a centralized architectural pattern where all the sensor nodes and other components communicate with a central coordinator, which is in charge of making decisions. CO is a semi-decentralized pattern that uses additional controller components to receive the data from the sensor nodes and make decisions.

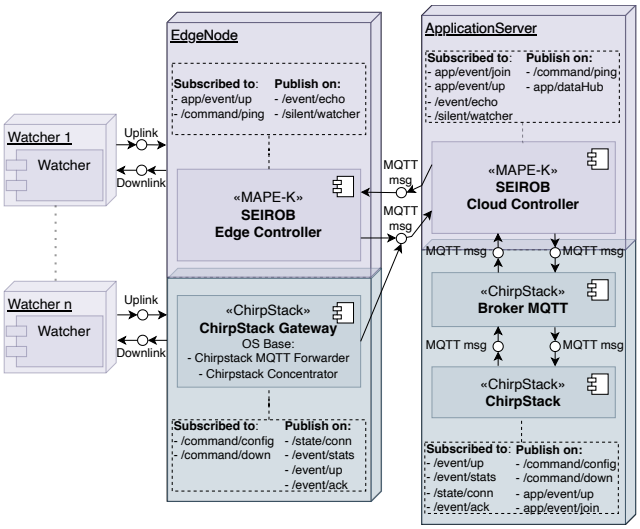

Figure 3: SEIROB architecture: first design option

First design option: semi-decentralized architecture with a central server broker. The UML-like deployment/component diagram for this first design option is shown in Fig. 3. Essentially, a central coordinator *SEIROB Cloud Controller* running on a server node is in charge of making high-level managing decisions for the overall system and of interacting with the human operators. Multiple controller components *SEIROB Edge Controller* are distributed over edge nodes and act as local coordinators for a group of sensor nodes (*Watcher* components).

Control software components running on edge and cloud nodes can publish messages on the (Chirpstack broker's) specific topics and can receive published messages (see the labels *Subscribed to* and *Publish on* shown in the dashed boxes in Fig. 3) that match their topic filter in order to monitor and realize high-level managing decisions. As an example, the excerpt of Python code in Listing 1 shows the function *on_message*, which is a topic filter responsible for enacting some (or parts of) feedback loops upon the occurrence of a proper triggering event (a MQTT topic event). Specifically, it shows the firing of a cascade of MAPE functions (server side) related to scenario S_1 for rationing the battery life of a watcher. The function *analize_planning* shows the logic adopted, as an example of two merged Analyze and Plan components, for generating a new configuration for a watcher and regulating its transmission frequency (parameters *timetosend* and *timetoreceive*) on a threshold basis.

All data from the devices are sent over MQTT as payloads in the Chirpstack-specific JSON data format. The MQTT topics are those supported natively by the Chirpstack infrastructure. The identifiers of the application and of the devices appear in the prefix part of the topics. The main events are topics of the form `/event/[EventType]` where `[EventType]` is a placeholder for an event type ranging in the set: `up` – received uplink frame, `stats` – gateway statistics, `ack` – confirmed downlink acknowledgement, and `join` – device join event. The default topic for scheduling downlink payloads to a device is `/command/down`, while `command/config` is used to request the gateway to re-configure the signal strengths of the LoRa channels. The topic `/state/conn` is for a gateway to publish its connection state.

In addition to these native MQTT items, we added further topics for implementing the robustness scenarios, such as `/silent/watcher` for scenario S_2, `/command/ping` and `/event/echo` for S_3. In scenario S_1, a (downlink) configuration message can be sent by the server controller to a watcher to change the watcher configuration parameters. In scenario S_2, an edge controller may notify the server controller about a silent watcher publishing on the `/silent/watcher` topic. In scenario S_3, the SEIROB server controller periodically checks if the edge controllers are alive by sending a message on the `/command/ping` topic. The edge controllers reply to the message by posting on the `event/echo` topic. If after 3 pings the server controller does not receive an echo from an edge controller, this last is considered silent.

Second design option: semi-decentralized architecture with cross-edge brokers. Other than being incomplete (not all scenarios are supported), the first design option has the following limitations. If the application server goes down, the LoRa gateways stop working as well. To implement all scenarios and avoid this single point of failure, we decided to make redundant the *Chirpstack MQTT broker* at the edge node (see Fig. 4). This edge broker runs on the gateway, on top of all the features that are provided by a full version of

```
def on_message(self, client, userdata, msg): [...]
    if topic_type == "eventup":
        newBattery = self.monitoring(payload_decoded)
        newConfiguration, newBatteryConfig = self.analize_planning(newBattery)
        self.execute(payload_decoded, newConfiguration, newBatteryConfig)
        ...
def analize_planning(self, battery_level):
    watcher_config = DownlinkConfigurationPayload()
    if battery_level > 50:
        watcher_config.timetosend = AppServer.MIN_TIME_TO_SEND
        watcher_config.timetoreceive = AppServer.MIN_TIME_TO_RECEIVE
        batteryConfig = WatcherBatteryConfigEnum.NORMAL.value
    elif battery_level > 30:
        watcher_config.timetosend = round(AppServer.MIN_TIME_TO_SEND * 1.5)
        watcher_config.timetoreceive = round(AppServer.MIN_TIME_TO_RECEIVE * 1.5)
        batteryConfig = WatcherBatteryConfigEnum.SOFT_ENERGY_SAVING.value
    elif battery_level > 15:
        watcher_config.timetosend = round(AppServer.MIN_TIME_TO_SEND * 3)
        watcher_config.timetoreceive = round(AppServer.MIN_TIME_TO_RECEIVE * 3)
        batteryConfig = WatcherBatteryConfigEnum.ENERGY_SAVING_.value
    else: watcher_config.timetosend = round(AppServer.MIN_TIME_TO_SEND * 5)
        watcher_config.timetoreceive = round(AppServer.MIN_TIME_TO_RECEIVE * 5)
        batteryConfig = WatcherBatteryConfigEnum.HARD_ENERGY_SAVING.value
    return watcher_config, batteryConfig
```

Listing 1: MAPE-k loop for S_1 - watcher's battery rationing

the *Chirpstack Gateway OS*, which comprises a full infrastructure stack with *ChirpStack Network Server* and *ChirpStack Application Server*. Essentially, the broker at the edge publishes identical information coming from sensors' nodes. These data are collected and housed by a new sub-component, *Client GW*, of the edge controller (this sub-component is subscribed to the topics on which the edge broker publishes), and then forwarded to the broker at the server node (when it is up) by the *Client_Server* sub-component. Another advantage of this second architecture is that it allows the implementation of the fourth scenario in the case where the application server is silent. This architecture also allows for expanding the set of functionalities at the edge nodes, including data storage and processing.

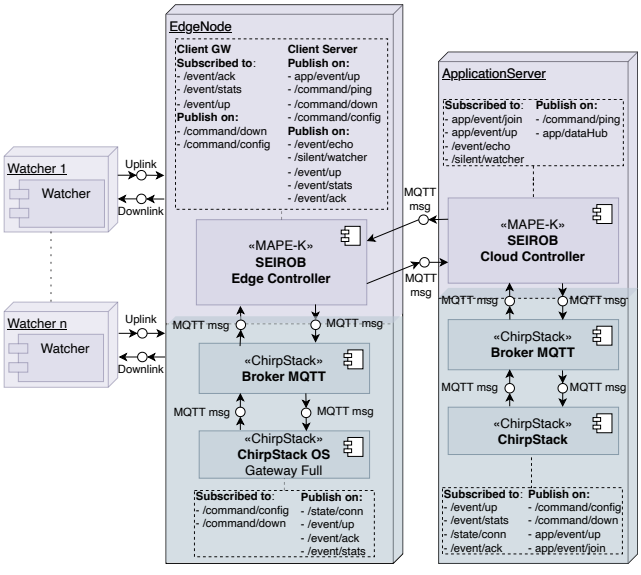

Figure 4: SEIROB architecture: second design option

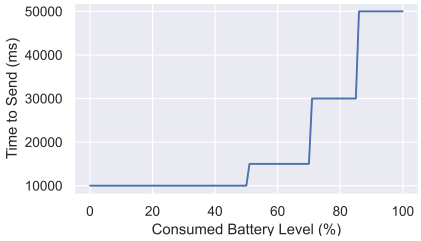

Figure 5: RQ1: Battery rationing

4 EVALUATION

In this section, we describe the evaluation that we have conducted of SEIROB w.r.t. the first three robustness scenarios (see Table 1) during an experimental campaign with a hybrid IoT installation made of both physical and virtual devices. The experimental campaign consisted of several executions of different lengths (from a few minutes to several hours). Thanks to the logs we recorded, our evaluation allows us to reply to the following research questions.

- RQ1 Is SEIROB able to identify the need for adaptation in the case of battery shortage, adapt the frequency of monitoring, and transmit data consequently?
- RQ2 Does SEIROB achieve the self-healing requirement in the case of a silent watcher?
- RQ3 Does SEIROB achieve the self-healing requirement in the case of a non-working edge node?

More specifically, we here reply to the three research questions by analyzing some execution traces reporting the behavior recorded as usual of a hybrid setup of the SEIROB exemplar, implemented by using the second design option presented in Sect. 3.3, for exemplifying scenarios S_1, S_2, and S_3. The considered installation is made of: 4 watchers – 2 physical (`pWatcher_1` and `pWatcher_2`) and 2 virtual (`vWatcher_1` and `vWatcher_2`); 2 gateways – one physical and one virtual (`pGateway_1` and `vGateway_2`); and a cloud node hosting the ChirpStack network/application server and the `AppServer`.

4.1 Results

RQ1: Battery rationing. In this research question, we are interested in evaluating the self-configuring property of SEIROB in scenario S_1. In other words, we here analyze the log of a virtual watcher to identify if it is able to reconfigure itself based on the commands received by the application server and edge node. Fig. 5 reports a plot on the transmission rate (`time to send`) used by `vWatcher_1` depending on the consumed battery level. It can be noticed that up to 50% of consumed battery, the transmission rate remains unchanged, according to the MAPE-k loop reported in Listing 1. Then, when the battery decreases, the `time to send` increases, implying a reduction of the transmission rate. This change in parameter configuration is visible in the watcher's log as reported in Listing 2. The excerpt of log reports the first cycle (cycle 5001) in which the battery level drops below 50%. At line 6, the log reports the downlink message received by the application server in which we can see the new `time` to send and receive. This message is sent by the application server, embedding the MAPE-k control loop, as reported by the log in Listing 3 at line 1.

```
12:03:56 -> ------------------- CYCLE : 5001 -------------------
12:03:58 -> Groove Dust Sensor => Detecting PM10 concentration.. [...]
12:04:02 -> 0.6|20.6|53|-|46752|276|6.5|45.80|9.80|49.99
12:04:04 -> time to send: 10000 - time to receive: 10000 [...]
12:04:05 -> Received downlink message correclty..
12:04:05 -> 1|1|1|0|1|1|1|1|15000|15000
12:04:08 -> Sensors activated/deactivated as wanted..
```

Listing 2: `vWatcher_1` serial console output for RQ1

```
11/03/2023 12:04:05 [INFO]: APPSERVER ENQUEQUE WATCHER vWatcher CONFIGURATION
    -> 1|1|1|0|1|1|1|1|15000|15000
11/03/2023 18:24:07 [INFO]: AppServer received message from topic: application/fda23245-f5e7
    -411f-8d9f-d1cac850c286/device/a8610a3237268901/event/up
```

Listing 3: `AppServer`'s log output for RQ1

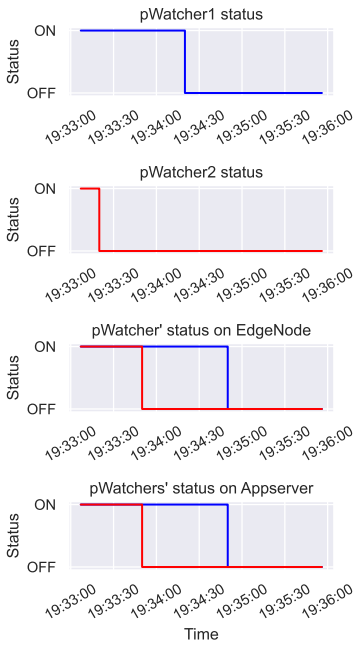

Figure 6: RQ2: Silent Watchers

In conclusion, we can state that the adopted architecture for SEIROB allows the system to automatically reconfigure the watchers as the percentage of available battery decreases.

RQ2: Silent watchers. In this research question, we analyze whether SEIROB achieves the self-healing property when a watcher is silent (scenario S_2). In this case, the proposed architecture should allow edge nodes and the application server to identify that the watcher is silent. This scenario is illustrated, with a focus on a limited time range, in Fig. 6. In the figure, we report the status, respectively, for `pWatcher1` and `pWatcher2`, and their status as seen by the edge node and the application server. The physical watcher `pWatcher1` works till the time instant 19:34:20 and, then it becomes silent. Thanks to the proposed architecture for SEIROB, the edge node and the application server identify the silent watcher within the timeout (30 seconds, in our case). This can be seen in the log outputs reported in Listing 4 and 5. Concerning Listing 4, it can be seen that the edge node receives (line 1) an `/event/up` message from the

```
11/03/2023 19:34:20 [INFO]: Edgenode vGateway1 received message from topic: application/
    fda23245-f5e7-411f-8d9f-d1cac850c286/device/101d244c6bad962d/event/up [...]
11/03/2023 19:34:50 [INFO]: Edgenode vGateway1: WATCHER pWatcher1 IS SILENT. LAST
    POSITION: {'latitude': 45.6472792, 'longitude': 9.5944264}
```

Listing 4: `EdgeNode`'s log output for RQ2

```
11/03/2023 19:34:20 [INFO]: AppServer received message from topic: application/fda23245-f5e7
    -411f-8d9f-d1cac850c286/device/101d244c6bad962d/event/up [...]
11/03/2023 19:34:50 [INFO]: WATCHER pWatcher1 IS SILENT. LAST POSITION: {'latitude':
    45.6472792, 'longitude': 9.5944264}
```

Listing 5: `AppServer`'s log output for RQ2

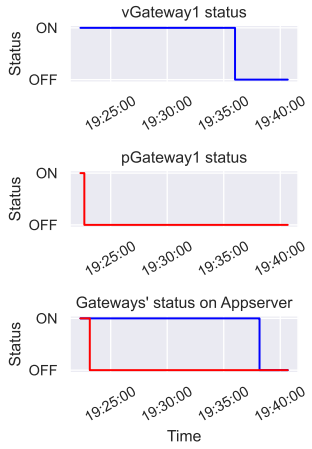

Figure 7: RQ3: Non-working Edge Nodes

device with UID 101d244c6bad962d (`pWatcher1`). Then, at line 2, the watcher is considered silent because no more messages are received from it within the 30 seconds timeout. Similarly, Listing 5 shows the same messages as received by the application server.

All in all, this analysis shows that SEIROB is able to identify silent watchers and quickly report them to the edge nodes and application server.

RQ3: Non-working edge nodes. In this research question, we analyze whether SEIROB achieves the self-healing property when an edge node is not working (scenario S_3). In this case, the proposed architecture should allow the application server to identify which is the non-working edge node. This scenario is reported, with a focus on a limited time range, in Fig. 7. In the figure, we report the status, respectively, for `vGateway1` and `pGateway1`, and their status as seen by the application server. In this figure, we can see that the virtual edge node `vGateway1` works till the time instant 19:36:00 and, then it stops working. However, the identification of a non-working gateway by the application server is not immediate (always less than two minutes in our experiments, as shown in Fig. 7), depending on the set transmission period, as the mechanism embedded in the proposed architecture requires the application server to send 3 `ping` messages without any response before declaring an edge node as non-working. Similarly, `pGateway1` is declared to be non-working by the application server. This can be seen in the

```
1 11/03/2023 19:33:10 [INFO]: AppServer send message to topic eu868/gateway/b827ebfffe09d724/
      command/ping [...]
2 11/03/2023 19:37:40 [INFO]: AppServer send message to topic eu868/gateway/b827ebfffe09d724/
      command/ping [...]
3 11/03/2023 19:38:10 [INFO]: EDGENODE vGateway1 IS NOT WORKING. LAST POSITION: {'
      latitude': 17.382750235675026, 'longitude': 9.790534973144533}
```

Listing 6: `AppServer`'s log output for RQ3

log output reported in Listing 6 where, after a set of ping messages, the edge node is declared to be non-working.

4.2 Validity and limitations

The experimental analysis shows that the proposed architecture allows for identifying non-working edge nodes and for achieving the adaptation goals for the first three scenarios. Unfortunately, we could not make experiments for the fourth scenario due to a limitation of Chirpstack Gateway OS that does not easily allow to deploy functionalities at the edge node. As better explained in Section 5, through a workaround we were able to deploy the controller components on the edge nodes, but they are not allowed to access sensor data for local buffering and processing, which are fundamental functionalities for implementing scenario S_4.

We limited external validity threats by selecting common communication protocols (like MQTT) and open-source technology stacks (like ChirpStack LoRaWAN network stack) adopted by practitioners to develop IoT systems for environmental monitoring.

To mitigate threats to internal validity, we evaluated SEIROB through a diverse set of experiments by varying the type of IoT setups (with simulated devices, with real devices only, hybrid) and the injection of device failures.

We ensured the validity of our conclusions by repeating the experiments multiple times and varying their durations, ranging from a few minutes to hours. However, as future work, we plan to study scalability aspects involving multiple watchers and gateways.

5 DISCUSSION AND LESSONS LEARNED

Here, we discuss how we tried to address the challenges introduced in Sect. 2 in developing SEIROB and we share the lessons we learned.

Regarding the issue of *automating maintenance and failure recovery in IoT-Edge-Cloud applications*, the current state of practice suggests the distribution and seamless integration of feedback control loops to monitor and realize the self-* properties over the continuum. This is especially true for the achievement at runtime of high-level requirements related to robustness scenarios in an IoT-Edge-Cloud application for environmental monitoring. However, the work done in this respect allowed us to identify that a very critical aspect to consider is the feasibility of implementing the required adaptation scenarios with the type of sensors and hardware available and the underlying infrastructure. The proper choice of these technologies is fundamental to accurately observe the main environmental phenomenon but also to make the system itself (with all its parts) and its operating context observable along the continuum. In our case, we had a lot of freedom in making the watcher observable since the Arduino sketch was developed in-house, so we decided the format of the probe/effector messages. However, not all information regarding the device status is easily accessible and trackable; for example, the battery level feature is supported only in the last versions of Arduino MKR boards (such as MKR1310). Other information related to wireless communication are commonly used as signal strength indicators (e.g., SNR – Signal-to-Noise Ratio, RSSI – Received Signal Strength Indicator, etc.), and so usually means for observing and modulating these factors are supported natively by the IoT network/server infrastructure (like Chirpstack) to get a good wireless connection. For other aspects that are at the level of the software running at the edge nodes and are to be monitored without an infrastructure with native support for edge/fog computing, it is necessary to explicitly make them observable and remotely controllable. In our case, for example, for our availability objectives, we had to implement ping-echo tactics at the level of the controllers running on the edge and server nodes.

The *Control loop design complexity* can be mitigated by splitting the control logic into a set of distributed and cooperating feedback loops (and their control functions) that ensure the achievement of the system objectives over the continuum. To this purpose, we learned that defining offline the responsibility and/or a dependency graph among the controller components and their control functions for implementing the decentralized adaptation (robustness) scenarios is helpful. To this end, reasoning on how to better distribute the control/adaptation functions over the continuum and about their effective implementation is necessary. We learned that, during the design and feasibility analysis of the adaptation requirements, it is also crucial to look at the native support of the underlying infrastructure platform for collecting state information about distributed devices and running services, and for actuating proper actions quickly (e.g., for re-starting up, recovering from a failure or intermittent connectivity). On the one hand, the lack of native ECC support by the infrastructure (like Chirpstack) results in significant design complexity and low flexibility in resource monitoring and management at the level of the edge and cloud nodes. Furthermore, the advantages brought by an ECC infrastructure can be lost if unintended, hidden dependencies are introduced to vendor-specific and version-specific external components (e.g., dependence on APIs, data formats, integrations with various cloud platforms, etc.).

Concerning the *Control loop deployment complexity*, we found it difficult to configure the gateway and the services running on it. We had to manually deploy the controller components into the SD card of the gateways, configure them as gateway services so they can be started during the Chirpstack Gateway OS boot and restarted via the open-source utility *Monit* for Unix-like systems. So, automating pipelines for the cross-deployment and re-deployment of the edge controllers dynamically is a necessary feature for ECC infratuctures, but not available in Chirpstack at time we made such experiments. Such ECC deployment pipelines may exploit methods such as Over-The-Air (OTA) to update and change the embedded software of the sensor and edge nodes without reissuing them.

6 RELATED WORK

Several cloud service providers are proposing edge infrastructures and software tools for cloud-native edge computing. Examples of such initiatives are Azure IoT Edge [10], Cisco IOx [3], IBM Edge Application Manager [5], and Google Anthos [4]. Though they use open protocols and standards for inter-operability, such

cloud-native solutions suffer from the well-known lock-in vendor problem. As long as the operating systems of industrial embedded software and cloud platforms are proprietary and variegated, real interoperability between the edge and the cloud is hard to achieve.

Open initiatives for developing IoT applications for edge devices exist. These include, for example, the W3C Web of Things (WoT) [18] in the smart home domain, and ChirpStack and Things Stack [15] for LoRaWAN networks. However, as we have experimented with ChirpStack, ECC deployment pipelines are not yet full supported in such infrastructures.

Decentralized control loop architectures [7, 20] have been adopted in some research prototypes in the form of simulators or tested on a small scale in smart home scenarios [1], open spaces [19], and smart grid networks [12]. Among self-adaptive IoT exemplars, the most influential for our work are DeltaIoT[6] and DingNet[16]. DeltaIoT provides a simulator and a physical deployment of an IoT network for a small in scale environment, while DingNet targets larger IoT deployments. Both are aimed at supporting IoT applications with QoS-based adaptation goals; essentially, a centralized adaptation manager dynamically adapts the IoT mots and network (in a multi-hop configuration) settings to ensure reliable and energy-efficient packet delivery. Our exemplar aims instead at goals of robustness for diagnosis and mitigation of device failures. It focuses on functional requirements that the overall system exhibits in terms of feedback loops distributed along the continuum, and also incorporated into the embedded software of the watcher that, therefore, we had to develop in-house.

7 CONCLUSIONS AND FUTURE WORK

In this experience report, we discussed on how we architected and implemented a more robust IoT-Edge-Cloud exemplar for environmental monitoring. We described how we built such an exploratory exemplar by leveraging on a semi-decentralized self-adaptive architecture along the continuum, with the support of an open-source ChirpStack LoRaWAN network infrastructure. To validate the capability of our exemplar to meet some initial robustness goals, we conducted several experiments with simulated and real deployment topologies. The results of these experiments reveal that the approach satisfies the initial robustness scenarios we have considered. We also shared some of our experiences and gave insights into the most relevant challenges we had to face.

As future work, we plan to perform a more extensive evaluation with larger IoT deployment topologies and make a quality-driven analysis (e.g., performance, latency and transmission periods of both hybrid and physical installations at different architecture scales). We also want to design placement strategies of the sensor nodes to meet the desired coverage goals of a certain environment/phenomenon to be monitored. Moreover, we want to conduct some analysis about the pollutant measurements accuracy. Finally, we would like to include strategies for adapting the controllers' deployment in response to resource overload or failure, and thus ensuring more performing and continuous environmental monitoring. This would be extremely useful for IoT applications that require a response time ideally near to zero and non-negotiable. This, in turn, would imply the need for load balancing and task re-allocation over a set of edge nodes. These features are not well supported yet, but will probably become part of many advanced infrastructures for ECC computing in the near future.

ACKNOWLEDGMENT

We would like to thank Filippo Barbieri and Lorenzo Mazzoleni for the preliminary work done during their master thesis. The work of Andrea Bombarda is supported by PNRR - ANTHEM (PNC0000003) – CUP: B53C22006700001. The work of Patrizia Scandurra was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

REFERENCES

[1] Paolo Arcaini, Raffaela Mirandola, Elvinia Riccobene, and Patrizia Scandurra. 2020. MSL: A pattern language for engineering self-adaptive systems. *J. of Systems and Software* 164 (2020), 110558.
[2] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi. 2019. A Unified Model for the Mobile-Edge-Cloud Continuum. *ACM Trans. Internet Technol.* 19, 2, Article 29 (apr 2019), 21 pages. https://doi.org/10.1145/3226644
[3] Cisco. 2023. Cisco IOx Network Infrastructure Products - Cisco. https://www.cisco.com/c/en/us/products/cloud-systems-management/iox/index.html
[4] Google. [n. d.]. Anthos at the Edge. https://cloud.google.com/solutions/anthos-edge
[5] IBM. [n. d.]. IBM. https://www.ibm.com/cloud/edge-application-manager
[6] Muhammad Usman Iftikhar, Gowri Sankar Ramachandran, Pablo Bollansée, Danny Weyns, and Danny Hughes. 2017. DeltaIoT: A Self-Adaptive Internet of Things Exemplar. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 76–82.
[7] Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (Jan. 2003). https://doi.org/10.1109/MC.2003.1160055
[8] Danylo Khalyeyev, Tomás Bures, and Petr Hnetynka. 2023. Towards a Reference Component Model of Edge-Cloud Continuum. In *20th International Conference on Software Architecture, ICSA 2023 - Companion, L'Aquila, Italy, March 13-17, 2023*. IEEE, 91–95. https://doi.org/10.1109/ICSA-C57050.2023.00030
[9] Elizabeth Liri, Prateek Kumar Singh, Abdulrahman Bin Rabiah, Koushik Kar, Kiran Makhijani, and K. K. Ramakrishnan. 2018. Robustness of IoT Application Protocols to Network Impairments. In *2018 IEEE International Symposium on Local and Metropolitan Area Networks, LANMAN 2018, Washington, DC, USA, June 25-27, 2018*. IEEE, 97–103. https://doi.org/10.1109/LANMAN.2018.8475048
[10] Microsoft. [n. d.]. IoT Edge Microsoft Azure. https://azure.microsoft.com/it-it/services/iot-edge/
[11] Dejan S. Milojicic. 2020. The Edge-to-Cloud Continuum. *Computer* 53, 11 (2020), 16–25. https://doi.org/10.1109/MC.2020.3007297
[12] Mahyar T. Moghaddam, Eric Rutten, Philippe Lalanda, and Guillaume Giraud. 2020. *IAS: An IoT Architectural Self-adaptation Framework*. Springer International Publishing, 333–351. https://doi.org/10.1007/978-3-030-58923-3_22
[13] Henry Muccini and Mahyar Tourchi Moghaddam. 2018. IoT Architectural Styles - A Systematic Mapping Study. In *Software Architecture - 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24-28, 2018, Proceedings (LNCS, Vol. 11048)*, Carlos E. Cuesta, David Garlan, and Jennifer Pérez (Eds.). Springer, 68–85. https://doi.org/10.1007/978-3-030-00761-4_5
[14] Angelika Musil, Juergen Musil, Danny Weyns, Tomas Bures, Henry Muccini, and Mohammad Sharaf. 2017. *Patterns for Self-Adaptation in Cyber-Physical Systems*. Springer International Publishing, Cham, 331–368.
[15] The Things Network. [n. d.]. The Things Network. https://www.thethingsnetwork.org/
[16] Michiel Provoost and Danny Weyns. 2019. DingNet: A Self-Adaptive Internet-of-Things Exemplar. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 195–201.
[17] Azar Taufique, Mona Jaber, Ali Imran, Zaher Dawy, and Elias Yacoub. 2017. Planning wireless cellular networks of future: Outlook, challenges and opportunities. *IEEE Access* 5 (2017), 4821–4845.
[18] W3C. [n. d.]. W3C Web of Things (WoT). https://www.w3.org/TR/wot-thing-description/
[19] Danny Weyns, M Usman Iftikhar, Danny Hughes, and Nelson Matthys. 2018. Applying architecture-based adaptation to automate the management of internet-of-things. In *Software Architecture: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24–28, 2018, Proceedings 12*. Springer, 49–67.
[20] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. 2013. *On Patterns for Decentralized Control in Self-Adaptive Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 76–107.