Evaluation of IoT Self-healing Mechanisms using Fault-Injection in Message Brokers

Miguel Duarte  
up201606298@fe.up.pt  
Faculty of Engineering,  
University of Porto  
Porto, Portugal

João Pedro Dias  
jpmdias@fe.up.pt  
BUILT CoLAB and  
Faculty of Engineering,  
University of Porto  
Porto, Portugal

Hugo Sereno Ferreira  
hugosf@fe.up.pt  
INESC TEC and  
Faculty of Engineering,  
University of Porto  
Porto, Portugal

André Restivo  
arestivo@fe.up.pt  
LIACC and  
Faculty of Engineering,  
University of Porto  
Porto, Portugal

ABSTRACT
The widespread use of Internet-of-Things (IoT) across different application domains leads to an increased concern regarding their dependability, especially as the number of potentially mission-critical systems becomes considerable. Fault-tolerance has been used to reduce the impact of faults in systems, and their adoption in IoT is becoming a necessity. This work focuses on how to exercise fault-tolerance mechanisms by deliberately provoking its malfunction. We start by describing a proof-of-concept fault-injection add-on to a commonly used publish/subscribe broker. We then present several experiments mimicking real-world IoT scenarios, focusing on injecting faults in systems with (and without) active self-healing mechanisms and comparing their behavior to the baseline without faults. We observe evidence that fault-injection can be used to (a) exercise in-place fault-tolerance apparatus, and (b) detect when these mechanisms are not performing nominally, providing insights into enhancing in-place fault-tolerance techniques.

CCS CONCEPTS
• Computer systems organization → Sensors and actuators;  
Reliability; Distributed architectures;  
Software and its engineering → Software testing and debugging.

KEYWORDS
IoT, fault-injection, self-healing, dependability, fault-tolerance, middleware

1 INTRODUCTION
Internet-of-Things (IoT) is being largely adopted, being ubiquitous across application domains, connecting the physical and virtual realms to provide services that improve the quality-of-life and business processes [43]. IoT devices are typically constrained in both computational power and energy, highly distributed (both logically and geographically), and heterogeneous (e.g., different manufacturers and competing standards). The nature of these systems drives the creation of communication protocols that are lightweight and thus compatible with the computational and energy constraints of these systems. Among those protocols, MQTT has been largely adopted as a lightweight TCP-based Machine-to-Machine (M2M) and IoT connectivity protocol [24]. MQTT leverages a publish/subscribe pattern, in which a middleware broker guarantees the delivery of messages from publisher entities (i.e., entities that publish messages in a given topic) to one or more subscriber entities (i.e., entities that are subscribed to a given topic) [24].

IoT dissemination required end-users to program their own systems, thus leading to the proliferation of low-code development platforms such as Node-RED which has a visual programming language to mashup together hardware devices, APIs, and online services [26, 33, 40, 42]. However, most of these low-code development environments lack of verification and validation mechanisms [13], and do not provide fault-tolerance mechanisms or suggest how to improve the dependability of these systems. This is expected, as most IoT systems disregard these concerns or achieve redundancy only by device/service replication (with additional costs and management complexity) [10, 28, 36]. This is a mostly direct result of the complexity of creating and using fault-tolerance mechanisms in IoT systems, as pointed by Javed et al., "building a fault-tolerant system for IoT is a complex task, mainly because of the extremely large variety of edge devices, data computing technologies, networks, and other resources that may be involved in the development process" [28].

The use of self-healing[22] — the ability of a system to automatically detect, diagnose and repair system defects at both hardware and software levels — to attain fault-tolerance on IoT systems has been suggested by several authors [1, 4, 20, 35, 37, 38]. In previous work [14] a set of patterns to achieve fault-tolerance in IoT systems by adding self-healing mechanisms was introduced, along with a reference implementation in Node-RED. This reference implementation, so-called SHEN\(^1\), consists of a set of self-healing add-on nodes to the Node-RED visual programming language that can be used to improve the visual flows with error detection and system health recovery/maintenance mechanisms [15–17].

One way of ensuring that a self-healing/fault-tolerance mechanisms work as intended, is to actually exercise them. In the research field of fault-tolerance, fault-injection has been used as a technique to deliberately cause errors and failures in systems by introducing faults and then observing how it behaves and recovers from them; a now common practice in Chaos Engineering [8].

For the purposes of this work, we assume that the IoT system under study uses MQTT as the communication substrate, thus requiring a message broker to manage all the combinations, and that there is no redundant broker units. Thus, we can instrument an MQTT broker to inject faults in the messages as they are exchanged in the broker. And, as stated in the fault model introduced by Esposito [21], we are able to introduce several types of faults at the network level (more specifically, at the message level) namely:

\(^1\)SHEN is available on GitHub: node-red-contrib-self-healing, https://github.com/jpmdias/node-red-contrib-self-healing.
(1) omission, (2) corruption, (3) reordering, (4) duplication, and (5) delay. With an instrumented broker capable of injecting faults on-demand in a running IoT system with self-healing mechanisms, we can: (a) exercise the in-place fault-tolerance mechanisms, and (b) know when these mechanisms are not working correctly, thus finding improvement targets. We proceed to collect empirical evidence that supports these claims by defining two experimental scenarios and a total of six experiments.

The main contributions of this paper are twofold: (1) an instrumentable MQTT broker that allows fault-injection in IoT systems either using pre-defined operators or user-defined ones, and (2) a validation of the self-healing extensions for Node-RED [15, 16].

The remaining of the paper is structured as follows: Section 2 presents some related work and Section 3 introduces fault-injector proof-of-concept, and Section 4 details the experimental setup. Section 5 presents the carried experiments and the obtained results. Finally, Section 6 presents an overall discussion of the experiments and results, Section 7 discusses some of the threats to the validity of the experiments carried in this work, and Section 8 gives some closing remarks and points to future work.

2 RELATED WORK
Although most literature regarding IoT and fault-injection focuses on hardware faults via physical interaction with devices [25, 29, 46] rather than interfering with the application layer logic (including communication channels and middleware components), the usage of software techniques to exercise in-place fault-tolerance mechanisms can be traced to as early as 1993 [5]. More recently, particularly in cloud computing, Chaos Engineering became an umbrella term to techniques that inject, observe and collect information concerning the impact of faults on a running system [8, 11, 41].

Looker and Xu [30] presented an approach to carry fault-injection in the Open Grid Services Architecture (OGSA) middleware, which ensures the exchange of messages across services (e.g., cloud). Jacques-Silva et al. [27] suggested the use of partial fault-tolerance (PFT) techniques to improve the dependability of stream processing applications, given that faults in computational nodes or stream operators can provoke massive data losses due to the high data exchange rates of these systems. Esposito [21] surveyed research between 1995 and 2013 on fault-injection in the context of communication software; they found most of them focused on faults at the message-level and other networking failures (38.8%), with the remaining on process crashes (12.24%), application-level faults (18.4%), memory corruptions (8.2%), and others (22.36%). They also introduce a fault-injection approach for publish/subscribe systems that encompass most of the faults presented in previous works.

Yoneyama et al. [45] performed model-based fault-injection on MQTT, by mimicking unstable network environments via simulated network errors (e.g., connection lost) and delays. Han et al. [44] introduced TRAK as a message-agnostic testing tool for injecting delays and provoke higher packet loss in Apache Kafka [23], thus asserting its QoS capabilities. Zaiter et al. [47] presented a distributed fault-tolerance approach for e-health systems based on the exchange of messages between LACs (Local Controlling Agents), reducing the reliance on one global agent controller (GAC). Britland et al. [9] explores faults where third-parties inject fabricated data and expect it to modify the system’s behaviour over time; they propose a Domain-Specific Language (DSL) to generate altered data that can then be injected into the system to observe its behaviour.

In summary, previous software-based fault-injection literature mostly explores faults at the communication/protocol level [6], with few tackling domain-specific behaviors (e.g., modifying sensor readings). Most also rely on fault-injection agents as new system’s components, with a single work preferring to modify the middleware [44]). This limits their usage in IoT due to the computational constraints of most entities. Finally, very few works use fault-injection to evaluate the behaviour of in-place fault-tolerance mechanisms [27, 44, 45, 47]. This paper improves on existing work by (1) creating faults by semantically changing messages passed between different parts of the system, (2) providing a DSL comprised of reactive operators\(^2\), (3) modifying a common middleware to target any MQTT-based system, and (4) designed to support in-place evaluation of fault-tolerance mechanisms.

3 INSTRUMENTED MQTT BROKER
To execute the experiments, we modified an commonly used open-source MQTT broker\(^3\) to inject faults into IoT systems. We choose fault-injection at the message broker level due to the fact the broker is a point of convergence of most IoT systems, mostly independent of the message publish/subscribe complexity, thus reducing the impact of the heterogeneity of the IoT system in the fault-injection strategies. The modifications were done to the broker allowed to use it as a proxy to intercept and modify messages before being published to a specific topic. The modified broker is available on GitHub\(^4\); more details on its inner workings can be found in [19].

Each fault-injection rule consists of a topic (where the rule will be applied), and an array of operators each one transforming the incoming message and passing it to the next one (as in a pipes and filters architecture). Each rule can also have a startAfter and stopAfter fields that define the number of messages before the faults start and stop being injected. The implemented operators in this proof-of-concept are the following: (1) map, which takes a function as an argument (e.g., multiply by two) which is then applied to the message, (2) randoDelay that delays the publication of a message, (3) buffer which captures all the messages until a time interval and/or number of messages is reached, publishing them all at once, and (4) randoDrop, which randomly drops some of the messages according to a specified probability. While all operators could be implemented through the map one, others are provided for ease the configuration of fault-injection, being the map operator useful for creating additional faults and transformations on-demand.

These operators were chosen to be implemented since they can be used to mimic some common faults in sensors and IoT systems as a whole [17, 32], faults which can result from the malfunctioning of a given device (e.g., out-of-spec sensor readings can be mimic with a map that multiplies the readings by a random number), network issues (e.g., message loss with randoDrop and lag with randoDelay), or a combination of these factors (and others). These concrete operators were also selected since they can be used

---

\(^2\)Inspired by the ReactiveX operators [31], http://reactivex.io

\(^3\)AEDES MQTT broker, https://github.com/mosca/eades

\(^4\)Instrumentable-Aedes, https://github.com/SIGNEXT/instrumentable-aedes
to validate the existent Node-RED self-healing nodes. Given the aforementioned operators, we can inject all the faults presented in the fault model of Esposito [21].

4 PRELIMINARIES

The possible combinations of the system with and without self-healing or fault-injection result in four variations of the system under test (SUT). We called these BL (baseline), self-healing (SH), fault-injection (FI), and self-healing with fault-injection (FI×SH).

If the fault-injection and self-healing mechanisms are working correctly we expect that: (1) the behavior of SH approximates BL, as no fault-injection is performed in either system and self-healing mechanisms should have a low impact in a nominal system; (2) the behavior of FI is very different from BL, since the base system, without self-healing components, should not be able to recover from injected faults, provided the fault is enough to deviate it from nominal operation; and (3) the behavior of FI×SH is similar to that of FI×SH, showing that the self-healing mechanisms are able to bring a system with injected faults back into nominal behavioral.

These assumptions are expected to hold since the self-healing mechanism for each one of the scenarios was selected in accordance with the type of sensing data, sensing data cadence, sensor maximum, and minimum reading values (i.e., device specifications). Other aspects, including network latency and packet loss, are expected to have no impact since the experiments were run in a single machine. Further, the injected faults are directly related to the type and frequency of sensing data being exchanged; thus, it is expected that such errors would emerge in a real-world system.

The experiments were done on a standard Linux laptop with Node-RED version 1.3.2, and the modified AEDES MQTT broker was run with NodeJS version 14.15.5. A replication package for these experiments is available on Zenodo [18].

5 EXPERIMENTS AND RESULTS

Two test scenarios (S1 and S2) were devised and several experiments were done for each scenario. Each one of these used a separate fault-injection configuration so that different operations were applied to the same dataset. For each experiment, messages were relayed to both the baseline (BL) and the corresponding self-healing (SH) flow simultaneously to ensure that both systems received the same input and that their outputs (i.e., alarm level) could be directly compared. This also ensured that the fault-injection operators, which rely on the MQTT broker’s random number generation, do not produce different results when compared to the baseline.

To remain as close as possible to a real-world system, we used a real-world dataset6 as sensing data. The dataset used contained NOx (GT) readings from a device “located on the field in a significantly polluted area, at road level, within an Italian city” [12]. The sensing data was replayed — i.e., each data point was emitted using a Python script with a reduced time interval between readings (in the dataset one data point was collected per hour) — reducing the time required for each experiment to run, thus allowing to increase the number of experiments and refinements. Since the dataset does not provide the concrete specification of the sensor’s reading range of values, we considered a widely available sensor6 as a reference, which has a hardware range of 0-1 ppm (0-1000 ppb) and a minimum detection limit of 0.005 ppm (5 ppb). Thus, only values ranging from 5 to 1000 ppb should be considered valid readings. We considered each reading should trigger an alarm according to three concern levels7: off (0), warn (1), and danger (2). A threshold of 53 ppb was considered as a warning value, and 212 ppb as a danger value.

Each experiment replayed a total of 120 messages from the used dataset, and faults are injected to messages 10 thru 110 — this allows the system to gain stability before entering a degradation state and can also resume normality after fault-injection stops. In order to collect insights on the behavior of the SUT, all the messages flowing in the different MQTT topics are monitored and logged during the course of the experiment. All the experiments use the same BL, with three NOx sensors used to trigger an alarm according to the defined thresholds. The corresponding BL system, implemented as a Node-RED flow, parses the reading received from the sensors and sends the respective alarm level as output, filtered by a report-by-exception node to ensure that values are only emitted when the current alarm level changes. The Node-RED flows with self-healing capabilities used in the following scenarios use a subset of the SHEN nodes, as presented in previous work [16].

5.1 Sensor Readings Issues (S1)

Here (S1) we performed four experiments, each with different types of fault-injection operators applied to the sensor messages passing through the MQTT broker (i.e., simulating sensor malfunctions).

A Node-RED flow, with self-healing mechanisms, was developed to deal with these issues (SH). This system expands upon BL by introducing self-healing capabilities via SHEN nodes. It filters extraneous messages that are outside the expected operating range, compensates for missing values after a certain timeout, joins messages so that they are considered in groups of 3, considers the majority of values with a minimum consensus of 2 (with a 25% difference margin), and compensates for readings for which there is no majority with a mean of the previous readings, besides the basic functionality implemented by BL. The join and compensate nodes are configured with a timeout of 6 seconds to have a margin of 1 second in relation to the readings’ periodicity (5 seconds).

5.1.1 Experiment S1E1. In this experiment no fault injection was performed, with only the baseline system (BL) and the system with added self-healing mechanisms (SH) being considered. This allows us to compare the behavior of BL with SH in normal operation, and creates a base of comparison for the remaining experiments (i.e., experiments with fault-injection). This also provides us a behavioral profile of BL when compared with SH, giving us insights on how self-healing mechanisms’ operate with no added entropy.

We expect the SUT to remain stable during this experiment, outputting the expected alarm levels for the sensor readings’ thresholds. Despite the expected similarity in behavior, it is expected that SH’s alarm level output will be more stable than that of BL. This is due to the latter not implementing any type of consensus or majority

6https://www.aeroqual.com/product/nitrogen-dioxide-sensor-0-1ppm.
7While the EPA National Ambient Air Quality Standards list 0.05ppm as the avg. 24-hour limit for NO2 in outdoor air[2], for validation purposes we will consider these limits for each data point and not the 24-hour average value.
voting and instead simply using the received values directly as a stream.

Fig. 1 shows the experiment results for BL and SH. Despite the alarm output (represented as shaded areas near the horizontal axis) being very similar for both experiment outputs, stability is higher for SH. This can be observed by the lack of fast alarm state changes for borderline values for SH, which occur several times for BL (e.g., around 35 to 40 seconds into the experiment or around 415 to 420 seconds). The cause of this is likely to be BL’s lack of consensus mechanism, given that this system instead simply considers the most recent reading in order to determine the alarm state. When the three sensors report in quick succession, if the values of their readings are near the alarm level thresholds, fluctuations in the alarm level are expected.

The output similarity is confirmed by the alarm level overlap percentage between these two outputs, which is 97.3%. This is also a good sanity check to confirm that the addition of self-healing capabilities to the base system does not significantly change the alarm output status, which means that comparisons for SH between S1E1, and further experiments, will be meaningful in validating self-healing recovery of injected faults.

Table 1: Count of alarm level state transitions for S1E1-4.

|          | S1E1 | S1E2 | S1E3 | S1E4 |
|----------|------|------|------|------|
| **BL**   | Off 0| 8    | 10   | 5    | 8    | 4    | 8    |
| **SH**   | 4    | 4    | 10   | 5    | 8    | 4    | 8    |
| **S1E2** | Warn 1| 20   | 13   | 68   | 14   | 26   | 13   | 20   | 13   |
| **S1E3** | 11   | 8    | 70   | 8    | 17   | 8    | 11   | 8    |
| **S1E4** | Danger (2) | 39   | 25   | 148  | 27   | 51   | 25   | 39   | 25   |

Table 1 supports the previous claim — that SH provides an improvement in system stability in comparison to BL — due to the lower number of alarm state transitions. Additionally, this experiment presents evidence that both BL and SH correctly implement the expected core functionality (triggering the different alarm levels for different sensor reading thresholds), given that the alarm level at a given point in time corresponds to the sensor readings’ distribution along the thresholds (represented in the mentioned figures by the horizontal lines).

5.1.2 Experiment S1E2. Considering the baseline (BL) and the self-healing (SH) systems (which S1E1 shows to be similar in normal operation), we proceed to inject faults on both, obtaining systems FI (corresponding to the injection of faults in BL) and FI×SH (corresponding to the injection of faults in SH).

The fault being injected corresponds to an erroneous Sensor 3’s reading. As a result, this sensor’s readings are altered to be stuck at the upper operating bound (1000 ppb). This experiment simulates a fault in which a sensor malfunctions by continuously emitting readings in its top operating bound.

We expect that this fault-injection will provoke an inconsistent output in FI, especially if the third sensor is frequently the last to emit its reading, even if only by a slight delay. Due to relying on a majority of at least two values to decide on the alarm level to emit, we expect that the self-healing mechanism will be able to deal with the faults injected in FI×SH.

Fig. 2 shows the experiment results for FI and FI×SH. The faults injected (FI) disrupt the normal function of the system, resulting in constant alternation between alarm states, spending most of the experiment’s time in the highest alarm level. Meanwhile, FI×SH successfully recovers from the injected faults, having a near-perfect performance in comparison to this system’s output for S1E1.

These statements are supported by the overlap in alarm levels between FI×SH and SH, with a near-perfect overlap of 98.1%, and FI and BL, with a much lower overlap percentage of 40.0%. Table 1 also illustrate these conclusions, in which FI is much more unstable in comparison to BL, being the total number of state transitions for this experiment 148, while there were only 39 state transitions for the base experiment. Furthermore, the number of state transitions with self-healing has increased only marginally, going from 25 in the base experiment (SH) to 27 in this experiment (FI×SH).

Therefore, S1E2 demonstrates that the original system cannot handle sensor stuck-at issues since the behavior of FI is considerably affected, which validates that the performed fault-injection was meaningful enough to disturb the system’s regular operation. On the other hand, FI×SH can recover from the injected faults, having a remarkably similar behavior to SH. This happens since the stuck-at fault only affects one sensor, and with the usage of the replication-voter node, this reading will be discarded and the other two sensors’ values will be considered instead, resulting in a system that operates similarly to the SH.
5.1.3 Experiment S1E3. In this experiment, we injected faults in BL and SH to obtain FI and FI×SH by multiplying 40% of the readings done by Sensor 3 by a random factor in the range [0.2, 2.2], simulating spikes in sensor readings. The factor is randomized for each spike occurrence.

We expect that FI may output incorrect alarm values (in comparison to BL), especially when the altered values switch between alarm level thresholds. On the other hand, FI×SH should be able to handle the spikes since that, even if one of the three sensors outputs a value considerably different from the others, it will be discarded and the other two sensors’ values will be considered instead — due to the usage of the replication-voter node.

![Figure 3: NOx concentration and alarm status for S1E3.](image)

Fig. 3 shows the experiment results for FI and FI×SH. FI has had a good performance in the presence of the spikes (both when increasing and decreasing the read value), but there were still several situations in which the sensor reading spike caused the output alarm level to differ from the expected value in BL. FI×SH has held up to the defined expectations, handling almost all the injected faults and operating similarly to SH.

These statements are supported by the overlap in alarm levels between FI×SH and SH, with a near-perfect overlap of 97.4%, and FI and BL, with a lower overlap percentage of 76.3%, showcasing the disruption provoked by the injected spikes.

Table 1 supports the role of the self-healing mechanisms. Despite the difference not being as remarkable as that of the overlap percentages, it is of note to mention that FI×SH has the exact same number of alarm level state transitions of SH while the number of state transitions has increased for FI when compared with BL.

Despite this experiment not causing a variation as significant as that of the behavior of FI in S1E2, we were still able to observe a mismatch between the behavior of this system between the base case and this experiment, even if to a lesser extent. This shows that for the system under study, the spikes faults are less concerning when compared with the stuck-at ones injected in S1E2. Nevertheless, due to the decline in the overlap percentage for FI in comparison with BL, we can conclude that the faults injected were significant enough to affect the system’s correct functioning.

Since FI×SH behaved similarly to SH, we can confirm that for this experiment the presence of self-healing capabilities are beneficial for the system’s correct operation, thus improving its resilience.

5.1.4 Experiment S1E4. In this experiment, we injected faults in Sensor 3 so that it has a 20% chance of losing messages. The system does not receive any of the lost messages, as these are suppressed before leaving the message broker.

We expect that FI may report erroneous alarm values (compared to the base experiment, S1E1), especially when the missing values are in proximity to the alarm thresholds. FI×SH should be able to handle the injected faults by compensating the missing values by replaying the last message in the expected time interval.

![Figure 4: NOx concentration and alarm status for S1E4.](image)

Fig. 4 shows the experiment results for FI and FI×SH. FI is capable of handling the loss of some readings, thus the alarm output is quite similar to BL. FI×SH is also able to handle the loss of readings, similarly having almost the same behavior as SH.

The similarity in outputs when compared with S1E1 is a direct result of the low probability of losing one reading. Also, since the values for different sensors in the original dataset are close to one another, even by increasing the probability of values being suppressed from one sensor would result in FI outputting the expected alarm values for most of the experiment’s duration.

These statements are supported by the overlap in alarm levels between FI×SH and SH, 98.7%, and FI and BL, 99.8%, showing that FI has a similar behavior to BL, and that FI×SH has a similar behavior to SH. The previous observations are also supported by the results in Table 1, which shows the number of state transitions for both systems to be identical to those that occurred in S1E1.

This experiment did not cause a significant enough deviation from the base experiment’s behavior for FI, which is corroborated by the high overlap percentage with BL of 98.4%, as well as the fact that the number and type of alarm state transitions are identical to those of S1E1 (cf. Table 1). Thus, we conclude that the used dataset may not be the best candidate for this type of fault injection. A higher deviation in operation could possibly be observed if the fault-injection was done to more than one sensor at a time and with a higher probability of losing a message.

5.2 Timing Issues (S2)

In this scenario (S2), the Node-RED flow self-healing mechanisms used in S1 was enriched with nodes that detect and mitigate issues with timings (e.g., readings frequency issues) by introducing debounce nodes which can filter out extraneous messages based

---

8This fault may occur when a sensor is disconnected, has an intermittent power supply, or the network is unstable [32].
5.2.1 Experiment S2E1. No faults are injected for this experiment. Similarly to S1E1, the purpose of this experiment is to confirm that the system’s base functionality is correctly implemented for both BL and SH, as well as to provide a base experimental output with which to compare the behavior of the systems in following experiments. As with S1E1, we expected that the systems under observation remain stable during this experiment since there are no injected faults and that SH’s alarm level output will be more stable than that of BL. The results were similar to those of S1E1 with a similarity of 97.4%.

5.2.2 Experiment S2E2. In this experiment, to introduce additional noise into the system, each message for Sensor 3 is repeated after 6 seconds, as depicted in Fig. 5. Since the periodicity of the system’s readings in regular circumstances is of 5 seconds, the repeated message will be outputted in close proximity to the next reading. We expect FI to have an output that is less stable than it was for BL due to the injected faults; this may be problematic for FI since it does not have any concept of message timing. On the other hand, FI×SH should be able to cope with the injected faults since the debounce node will filter out the additional messages that come out of the expected frequency, thus behaving similarly to SH.

Figure 5: Marble diagram of messages for S2E2. The top diagram depicts the regular flow of messages, while the bottom diagram shows the messages after the fault injection.

FI (Fig. 6) performed significantly better than expected, despite the issues with messages near the alarm level thresholds, i.e., repeating the previous message would cause the system to output the previous alarm level once again until it received the following message and went back to the expected state. This is caused by the fact that the periodicity of the sensor readings messages is quite high (i.e., number of sensor readings per unit of time) and, whenever a fault is injected, it is not in effect for a long duration. This can be checked by observing the number of alarm state transitions and their duration — even with more state transitions, the time that the alarm stays at that given alert level is short — resetting to a normal state when a new reading appears.

As expected, FI×SH was able to cope with the injected faults (Fig. 6), having a near-perfect behavior in comparison to SH. These statements are supported by the overlap in alarm levels between FI×SH and SH, with a near-perfect overlap of 95.7%, and FI and BL, with a slightly lower overlap percentage of 83.4%, showcasing the disruption provoked by the injected spikes. Despite the high overlap percentages for FI with BL, Table 2 shows that for S2E2, FI has output nearly two times the amount of alarm level state transitions in comparison to S2E1.

S2E2 shows that despite the introduction of faults in FI the difference shown by the overlap percentage to BL is minimal. Despite this, FI×SH cope better with the injected faults, operating closer to SH. FI also performs worse than FI×SH when taking into account the number of alarm level state transitions (cf. Table 2). And, while it is not possible to conclude if this experiment caused enough deviation from BL to FI by taking only in consideration the overlap percentage, the difference in the count of alarm level state transition for FI in comparison to BL provides evidence that the injected faults have caused issues on the baseline system which the self-healing system is able to sustain.

6 DISCUSSION

The fault-injection experiments S1E1, S1E2, S1E3, S2E1, and S2E2 allowed us to observe that: (1) the self-healing systems (SH) do not deviate too much in behavior from the baseline system (BL); (2) the faults injected are consequential since there is a deviation on the baseline system in comparison to the base experiment when no fault is being injected; and (3) when the faults injected are consequential, the self-healing systems were able to recover from it, conforming with the normal service, and thus confirming that the self-healing mechanisms were being exercised and performing as expected.

More concretely, by analyzing the Table 1 and Table 2, we can see the impact that fault-injection has in a system without any fault-tolerance mechanisms versus a system with self-healing capabilities. The number of times that the alarm changes state between its three alert levels is considerably higher in all experiments, with a

| Alarm Level | Off (0) | Warn (1) | Danger (2) | Total |
|-------------|--------|----------|------------|-------|
| FI          | 8      | 20       | 11         | 39    |
| FI×SH       | 4      | 13       | 8          | 25    |

Table 2: Count of alarm level state transitions for S2E1-2.
clear impact in the experiments S1E2, S1E3, and S2E2, where the number of transitions was more than two times higher than the expected number of transitions. While the overlap percentages of the different experiments do not provide enough evidence to draw conclusions for most experiments, we can see that S1E2 performs considerably better when self-healing mechanisms are present.

Additionally, S1E4 allowed us to verify that it is paramount for the behavior of BL to be noticeably different when faults are being injected (FI) in comparison to the regular operating circumstances. This factor made it so that we considered this experiment inconclusive due to the low entropy caused in the system. Nevertheless, it shows that it is necessary to find this stark difference in expected versus observed output for the baseline system to be sure if the self-healing components are doing any work at all since a naïve system would already be able to recover from most injected faults.

It is also noticeable that for the created scenarios, due to the used dataset, some types of fault-injection did not result in much instability. This is due to the fact that all three sensors output readings with values in close to each other. As such, if one or even two sensors fail, it is likely that a naïve system (e.g., BL) will still perform as expected, outputting the correct alarm levels for most cases even in the presence of faults (FI). This is an indicator that further validation should be performed with other types of datasets and systems, as well as different types of faults.

7 Threats to Validity

The experiments presented in this work were carried out using a real-world dataset replayed (i.e., simulated) using a Python script. While all the messages were replayed, even if they contained erroneous or invalid readings, we cannot recreate any errors that could exist with the data if any pre-processing was done to the dataset before being put public available. This could be mitigated by using additional datasets of different IoT systems and carrying some extra, even if smaller, experiments in a real testbed. Further, while running the experiments in a simulated environment eases reproducibility, it has the downside of not covering sporadic faults that could occur in the physical testbed, e.g., electromagnetic interference’s and network disruptions.

An inadequate selection of the faults injected into the system also poses a threat to the validity of the experiments carried in this work since they have been hand-picked with prior knowledge of the fault-injection and self-healing capabilities. This can result in a bias in the selection, favoring issues that we know about and having more confidence that the proposed solution will handle correctly, instead of the ones that are mostly like to occur. While we attempt to mitigate this by using a real-world dataset, we have the bias of picking the faults injected and the self-healing logic.

Injecting faults at middleware (i.e., message broker) also limits the range of possibilities. While we attempted to replicate some common sensor fault types [32], other faults, including ones at the network level, were not covered by this study. Faults and implementation quirks of the underlying infrastructure — i.e., Python message emitter script, modified MQTT broker, Node-RED, and the self-healing extensions — might also have influenced the outcome of the carried experiments, so they are a confounding variable. We believe this has been mitigated by careful analysis of the expected and actual results, though it is something of concern.

8 Conclusions

Ensuring the dependability of software systems has been the goal of most fault-tolerance research in the past years [7]. In IoT, ensuring security, reliability, and compliance is becoming a paramount concern due to the recent increase in mission-critical contexts. IoT fault-tolerance is considerably challenging due to several aspects, including (1) high heterogeneity of devices, (2) interaction and limitations of systems deployed in a physical environment, (3) field fragmentation, ranging from the high number of communication protocols to the different and competing standards, and (4) intrinsic dependability on hardware that might simply fail [3].

Fault-injection becomes paramount to ensure that fault-tolerance mechanisms perform as they are expected when required. By instrumenting an MQTT broker, we enable the injection of faults at the message-passing level that allows to observe how well components deal with such faults. This provides a means to assert if the self-healing mechanisms configured in the system are sufficient to deal with them. The carried experiments showcase that the self-healing extensions do, indeed, work as expected, with the injected faults causing little to no impact on the delivery of normal service.

As future improvements to the instrumented MQTT broker, we consider the following: (1) simplify the fault-injection configuration by supporting more native language constructs (e.g., arrow functions) and other configuration abstractions (e.g., leverage visual notations), (2) support wildcard topics as per the MQTT specification, and (3) enable switching configuration at run-time instead of having to specify the configuration file when starting the broker. Regarding the experimental stage, it would be interesting to (1) expand the scenarios with more experiments, including more extensive fault-injection pipelines; (2) replicate the experiments using different datasets; (3) extend the usage of self-healing mechanisms, especially the ones implemented as part of SHEN [16]; and (4) explore decentralized IoT systems and orchestrators [34, 39].

Acknowledgments

This work was partially funded by the Portuguese Foundation for Science and Technology (FCT), ref. SFRH/BD/144612/2019.

References

[1] Mehmet S. Aktas and Merve Astekin. 2019. Provenance aware run-time verification of things for self-healing Internet of Things applications. *Concurrency Computation 31, 3*(2019), 1–9.
[2] Ammar A. Al-Sultan, Ghufran F. Jumaah, and Faris H. Al-Ani. 2019. Evaluation of the Dispersion of Nitrogen Dioxide and Carbon Monoxide in the Indoor Café – Case Study. *Journal of Ecological Engineering 20, 4*(2019), 256–261.
[3] M. Aly, F. Khomh, Y. Guérin-Escud, H. Washizaki, and S. Yacout. 2019. Is Fragmentation a Threat to the Success of the Internet of Things? *IEEE Internet of Things Journal 6, 1*(Feb. 2019), 472–487.
[4] Rafael Angarita. 2015. Responsible objects: Towards self-healing internet of things applications. *Proceedings - IEEE International Conference on Autonomic Computing, ICAC 2015*(2015), 307–312.
[5] J. Arlat, A. Costes, Y. Crouzet, J.C. Laprie, and D. Powell. 1993. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Trans. Comput. 42, 8*(1993), 915–923.
[6] Cyrille Valentine Artho, Armin Biebe, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuhashi Yamamoto. 2013. ModBat: A Model-Based API Tester for Event-Driven Systems. In *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel,*
November 5-7, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8244). Valeria Bertacco and Axel Legay (Eds.). Springer, 112–128.

[7] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. 2001. Fundamental Concepts of Dependability. Technical Report SeriesUniversity of Newcastle Upon Tyne Computing Science 1145, 010028 (2001), 7–12.

[8] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Roseenthal. 2016. Chase Engineering. IEEE Software 33, 3 (2016), 35–41.

[9] Mathias Brilane. 2021. A Language for Modelling False Data Injection Attacks in Internet of Things. (2021). 1–8. http://www.flowbird.group

[10] Bin Cheng, Gurkan Solmaz, Flavio Cirillo, Erno Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. 2017. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. IEEE Internet of Things Journal 4662, c (2017).

[11] D. Cotronone, L. De Simone, P. Liguori, and R. Natella. 2020. Fault Injection Analytics: A Novel Approach to Discover Failure Modes in Cloud-Computing Systems. IEEE Transactions on Dependable and Secure Computing. (2020).

[12] S. De Vito, M. Piga, L. Martinotto, and G. Di Francia. 2008. On field monitoring scenario. Sensors and Actuators B: Chemical 129, 2 (2008), 750–757.

[13] João Pedro Dias, Flávio Couto, Ana C.R. Paiva, and Hugo Sereno Ferreira. 2018. A Brief Overview of Existing Tools for Testing the Internet-of-Things. In IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). 104–109. https://doi.org/10.1109/ICSTW.2018.00035

[14] João Pedro Dias, Hugo Sereno Ferreira, and Tiago Boldt Sousa. 2019. Testing and Deployment Patterns for the Internet-of-Things. In Proceedings of the 24th European Conference on Pattern Languages of Programs (Isere, Germany) (EuroPLoP '19). ACM, New York, NY, USA, Article 16, 8 pages.

[15] João Pedro Dias, Bruno Lima, João Paesello Faria, Andre Restivo, and Hugo Sereno Ferreira. 2020. Visual Self-Healing Modelling for Reliable Internet-of-Things Systems. In Proceedings of the 20th International Conference on Computational Science (ICCS) Springer, 27–36.

[16] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. 2021. Empowering Visual Internet-of-Things Mashups with Self-Healing Capabilities. In 2021 IEEE/ACM 3rd International Workshop on Software Engineering Research Practices for the Internet of Things (SERP4IoT).

[17] João Pedro Dias, Tiago Boldt Sousa, André Restivo, and Hugo Sereno Ferreira. 2020. A Pattern-Language for Self-Healing Internet-of-Things Systems. In Proceedings of the European Conference on Pattern Languages of Programs 2020 (Virtual Event, Germany) (EuroPLoP '20). ACM, New York, NY, USA, Article 25, 17 pages.

[18] Miguel Duarte, João Pedro Dias, Hugo Sereno Ferreira, and André Restivo. 2021. Dataset of Fault-Injection experiments in a publisher/subscriber IoT system. https://doi.org/10.5281/zenodo.5148566

[19] Miguel Pereira Duarte. 2021. MQTT Chan Engineering for Self-Healing IoT Systems. Master’s thesis. Faculty of Engineering, University of Porto.

[20] Bahadir Dundar, Merve Astekin, and Mehmet S. Aktaş. 2016. A Big Data Processing Framework for Self-Healing Internet of Things Applications. In 2016 12th International Conference on Semantics, Knowledge and Grids (SKG). 62–68.

[21] Christian Esposito. 2013. Evaluating Fault-Tolerance of Publish-Subscribe Services. Springer, Milan, 115–130.

[22] Alan G. Ganek and Thomas A. Corbi. 2003. The dawning of the autonomic computing era. IBM Systems Journal 42, 1 (2003), 5–18.

[23] Nishant Garg. 2019. Node-RED. 813–818.

[24] Kevin Ni, Nithya Ramanathan, Mohamed Nabil Hajj Chebade, Laura Balzano, Sheela Nair, Sadaf Zahedi, Eddie Kohler, Greg Pottie, Mark Hansen, and Mani Srivastava. 2009. Sensor Network Data Fault Types. ACM Trans. Sen. Netw. 5, 3, Article 25 (June 2009), 29 pages.

[25] OpenJS Foundation. 2019. Node-RED. Flow-based programming for the Internet of Things. https://node-red.org/. Last Access 2019.

[26] Duarte Pinto, João Pedro Dias, and Hugo Sereno Ferreira. 2018. Dynamic allocation of serverless functions in IoT environments. In 2018 IEEE 16th international conference on embedded and ubiquitous computing (EUC). IEEE, 1–7.

[27] Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguair, and Hugo Sereno Ferreira. 2017. Patterns for things that fail. In Proceedings of the 24th Conference on Pattern Languages of Programs. 1–10.

[28] Partha Pratim Ray. 2017. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. IEEE Internet of Things Journal 4662, c (2017).

[29] Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguair, and Hugo Sereno Ferreira. 2018. Patterns for things that fail. In Proceedings of the 24th Conference on Pattern Languages of Programs. 1–10.

[30] Atsushi Kitazawa. 2017. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. IEEE Internet of Things Journal 4662, c (2017).

[31] Andreas Magle. 2016. Reactivejs and rxjava. In Reactive Java Programming. Springer, 1–9.