A Pattern-Language for Self-Healing Internet-of-Things Systems

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

Tiago Boldt Sousa
tbs@fe.up.pt
University of Porto, Faculty of Engineering
Porto, Portugal

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

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

ABSTRACT

Internet-of-Things systems are assemblies of highly-distributed and heterogeneous parts that, in orchestration, work to provide valuable services to end-users in many scenarios. These systems depend on the correct operation of sensors, actuators, and third-party services, and the failure of a single one can hinder the proper functioning of the whole system, making error detection and recovery of paramount importance, but often overlooked. By drawing inspiration from other research areas, such as cloud, embedded, and mission-critical systems, we present a set of patterns for self-healing IoT systems. We discuss how their implementation can improve system reliability by providing error detection, error recovery, and health mechanisms maintenance.

CCS CONCEPTS

• **Software and its engineering** → **Design patterns**; • **Hardware** → Communication hardware, interfaces, and storage.

KEYWORDS

internet-of-things, self-healing, fault-tolerance, patterns

ACM Reference Format:
João Pedro Dias, Tiago Boldt Sousa, André Restivo, and Hugo Sereno Ferreira. 2020. A Pattern-Language for Self-Healing Internet-of-Things Systems. In *European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20), July 1–4, 2020, Virtual Event, Germany.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3424771.3424804

1 INTRODUCTION

One definition of Internet-of-Things (IoT) is that it is a network of uniquely identifiable devices — known as *things* — that can be programmed, and that can sense (*i.e.*, sensors) and change (*i.e.*, actuate) their environment [47]. In contrast to traditional distributed systems mainly composed of servers, computers, and mobile devices, IoT systems are characterized by its unprecedented scale, distribution (both logical and geographical), embeddedness, heterogeneity, and invisibility of devices, which is forcing a paradigm shift on how we build such systems [64].

One of the aspects that are overlooked by both the scientific community and industry is the case for system resilience and, consequently, fault-tolerance [72]. One can argue that one of the main reasons for such is that IoT systems are developed in short development cycles (mostly due *time-to-market* pressures) even if they are larger and more complex systems than traditional ones (*e.g.*, while it is easy to patch some fault in a cloud-deployed system, it is much hard to patch a fault in the firmware of a remotely located device) [66]. Further, while some device's proneness to failure can be reduced by improving their hardware parts or by introducing device redundancy, it comes with additional financial or computational costs [50, 72].

In this paper, the terms fault and failure are used with their fault-tolerance literature connotation. A *fault* (which can be active or dormant) is the adjudged or hypothesized cause of an *error*, being an *error* the manifestation of a fault during runtime that can lead to a *failure* [9]. Failure is a deviation from delivering correct service [9]. Depending on the severity, a failure can be considered partial. We try to avoid the use of *error* for simplicity purposes since it is the term that most differs from fault-tolerance literature to software literature [74].

Some authors argue that the current disregard for resilience and fault-tolerance is mostly due to IoT device's failures typically being *hard faults*, which are consistent, thus easily detectable, reproducible (easy to *debug* and correct), and easy to fix by end-users by replacing the faulty unit. However, the *bigger* problem arises when such a device behaves arbitrarily (intermittent faults), and there is a lack of fail-over options [64, 66, 70]. Consider that if a thermostat stops working (*i.e.*, hard fault), an AC unit can fallback to a predefined working temperature or shut down entirely. However, if the thermostat malfunctions in such a way that it reports high-temperature readings, it can make the AC force the ambient temperature fall below unsafe levels for a newborn. Similarly, if a refrigerator's temperature sensor reports erroneous readings, it may cause food to degrade faster, possibly leading to food poisoning for the entire household [1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroPLoP '20, July 1–4, 2020, Virtual Event, Germany
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7769-0/20/07…$15.00
https://doi.org/10.1145/3424771.3424804

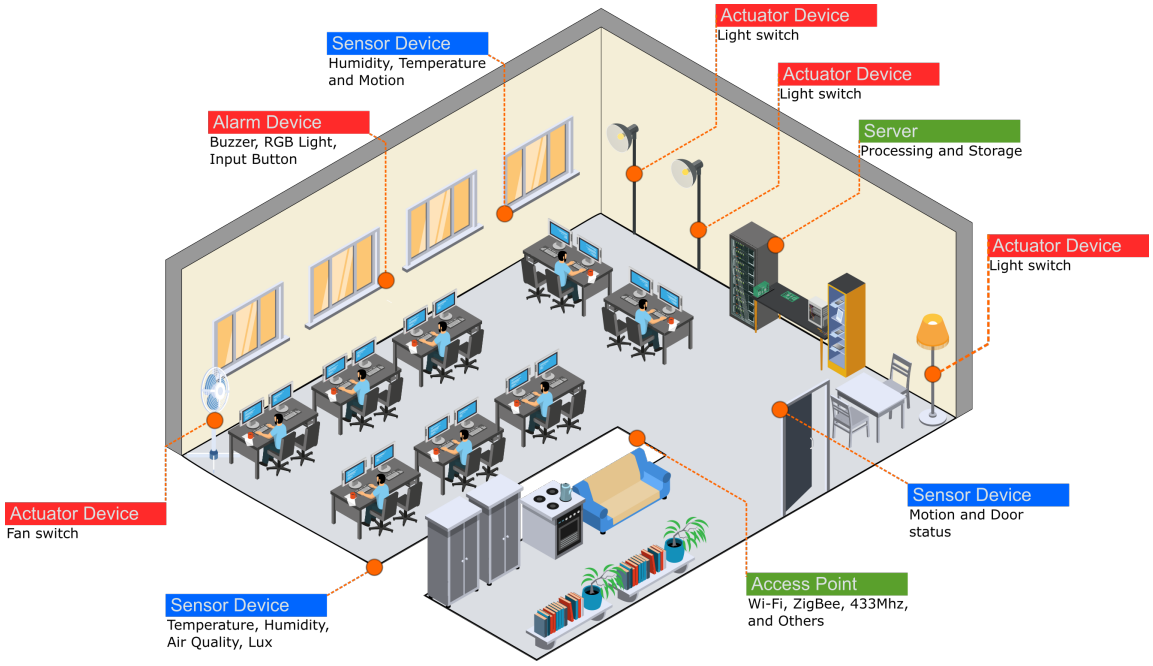


Figure 1: *SmartLab* motivational scenario with interconnected sensors and actuators.

Aware of the impact that failures can have on peoples' comfort, security, and safety (shifting IoT from a life-enhancement technology to a burden, or even provoke nefarious effects — sometimes even life-threatening), several authors have proposed strategies, approaches, and architectures towards improving the reliability of these systems. Most of these solutions strategies to detect problems during operation and fallback maneuvers that counter or compensate those problems, recovering the system (at least to an acceptable minimum) automatically [6, 8, 36].

Research on systems' reliability, especially the one regarding hardware-only systems, has been using the so-called *fault-tolerance* mechanisms to improve system dependability. More recently, several authors have been proposing *self-healing* as an approach to attain system reliability. The difference between the two has been the target of debate [10, 42]. For the scope of this paper, we regard self-healing as the subset of fault-tolerance with a focus on providing a system with the inherent capability of actively attempting to recover *itself* from an *abnormal* to *nominal* state, through strategies that involve software and hardware (and in extreme cases, even human) cooperation [29].

The main contribution of this paper is a pattern language with best practices for increasing the resilience of IoT systems through *self-healing*. Some patterns in the language have been discussed in other domains, but never regarding IoT. Specializing them to IoT enabled us to identify novel forces and solution strategies that did not manifest in other contexts. We consider that this language can be helpful for both system integrators that can add the pattern language patterns as new building blocks when configuring new systems and for IoT developers in general that can integrate the patterns on the systems within themselves as a way of increasing their reliability.

This paper is structured as follows: Section 2 presents a motivational scenario that introduces the reader to a typical IoT ecosystem, and Section 3 presents the fundamental concepts along with a revision of relevant literature. Section 4 introduces the pattern language, and the following sections, Sections 5 and 6, describe a total of 27 patterns that can be used to enable self-healing strategies in IoT system. Some final remarks and future work directions are given in Section 7.

2 MOTIVATIONAL SCENARIO

To better represent and contextualize the reader to IoT systems and their typical functioning in real-world setups, we introduce the *SmartLab*, a system deployed in a laboratory similar to some *smart home* (or *smart office-space*) setups.

An example isometric representation of the laboratory, including sensors, actuators, network infrastructure and other objects is given in Fig. 1. The sensor devices are capable of sensing the physical world, providing data to decision systems (*e.g.*, rule-based systems) that can trigger actions accordingly (actuator devices). Actuators can also be controlled by available applications (*e.g.*, web app) and sensing data observed using available dashboards.

As a motivational example, we can consider the following set of sensing-acting trigger rules:

- When somebody enters the laboratory (door status magnetic sensor), and it is dark (lux sensor), turn on the lights (light switch actuator);
- If there is poor air quality (air quality sensor), turn on the fans (fan switch actuator). Depending on the hazardousness of the values read, turn on the alarm (buzzer or RGB light actuators).

- Depending on the laboratory temperature — *e.g.*, if greater than 35 °C — (temperature sensor), turn on the fans (fan switch actuator) until the temperature drops — *e.g.*, bellow 30 °C.

This small set of rules shows that, depending on the ecosystem and application domain, we can automatize several processes, ranging from safety-concerning ones (*e.g.*, air quality) to comfort and quality-of-life ones (*e.g.*, control ambient temperature and humidity), being the only limitation the diversity of sensors and actuators that one has available.

In scenarios such as this one, dependability becomes a significant concern. The reliability of the components and the system can have direct implications on the system's availability and the conform and well-being of its users. There exists a considerable number of reports which sustain such observation. The roadmap proposed by Ratasich *et al.* [55] groups these and other issues in three categories, namely: security (*e.g.*, eavesdropping, jamming, and denial of service), dependability (*e.g.*, data corruption, protocol violations, and misusage) and long-term concerns (*e.g.*, aging and environmental effects and end-of-life — unsupported — hardware).

Moreover, although many authors focus on such concerns, the deployment and use of their proposed solutions and approaches in scenarios such as the presented one (*smart home*) are most scarce. Using mission-critical-grade components that are more heavily tested and reliable is expensiveness in terms of cost and developing and maintenance complexity (*e.g.*, redundant hardware), which increases development time and testing needs (which can be an issue in time-to-market practitioners' concerns). Nonetheless, dependability concerns must no be disregarded, and *best practices* should be adopted to safeguard the safety and well-being of *smart*-space users.

3 PRELIMINARIES

During system development, mainly verification and validation, IoT systems typically go through iterative fault-removal stages, commonly known as testing. Tests can be executed against simulated environments using SIMULATED-BASED TESTING [24], or in controlled environments using TESTBEDS[1] [24].

It is impossible to guarantee that testing will uncover all possible issues with the system, especially as systems grow in complexity [49]. Developers and other individuals with similar technical responsibilities need to consider that their system might degrade in production, reliability, availability, or even safety [9]. Such faults can result in both *hard* failures (*e.g.*, crash and hang-on) or *soft* failures (generally performance-related), thus the need of fault-tolerance [70].

Exception handling is one of the most common *single-version*[2] fault-tolerance mechanisms, where *exceptions are signaled by the implemented error detection mechanisms as a request for initiation of an appropriate recovery* [69]. However, this kind of mechanism is limited by knowing the possible faults that can occur and is most specific for each one.

[1] A TESTBED consists on a setup *with real devices and end-users interactions that provides an ecosystem to test new IoT systems* [24].
[2] Use of a *single version of a piece of software to detect and recover from faults* [69].

Several authors have contributed to the body-of-knowledge about fault-tolerance, some in the form of patterns. In *Patterns for Fault-Tolerant Software*, Robert Hanmer [31] presents 63 patterns (several based on the work of other authors) for fault-tolerant software systems. It categorizes them as architectural, detection, error recovery, error mitigation, and fault treatment patterns. More recently, Hanmer revisited his work, adapting it to the context of cloud software [33].

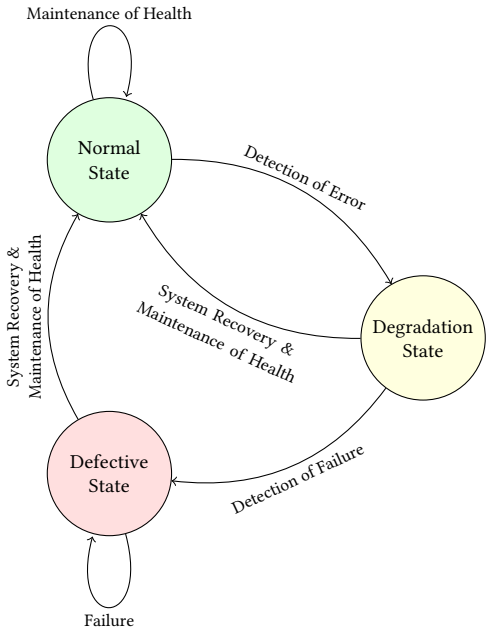


Figure 2: General loop architecture of a self-healing system as defined by Khalil *et al.* [36].

Although IoT systems highly depend on software (both locally and in the cloud), they aggregate hardware concerns that are not contemplated by such literature, mostly disregarding the so-call hardware/software co-design activity, which is fundamental to optimize power/energy, performance, reliability, and overcome privacy, and security issues [71]. The systematic mapping of fault-tolerance solutions supports this statement for IoT by Moghaddam *et al.* , which points to the same observation; however, they mention that some effort is being carried out to bring cloud fault-tolerance techniques to lower tier of the IoT systems, namely fog and edge [48].

While several fault-tolerance solutions are based on creating high-reliable components or allowing components to fail in the most unnoticeable way (*e.g.*, fail-fast, fail-silent), some authors have proposed *self-repairing* systems [16, 37]. These approaches mostly rely on *standby sparing* to recover the system from abnormal states, but they can be considered one of the first strategies of automatic system healing [37].

More recently IBM introduced the concept of Autonomic Computing, with four properties [29]:

- *self-configuring*: the ability to readjust itself *on-the-fly*;
- *self-healing*: the ability to discover, diagnose, and react to, or recover from, failures;

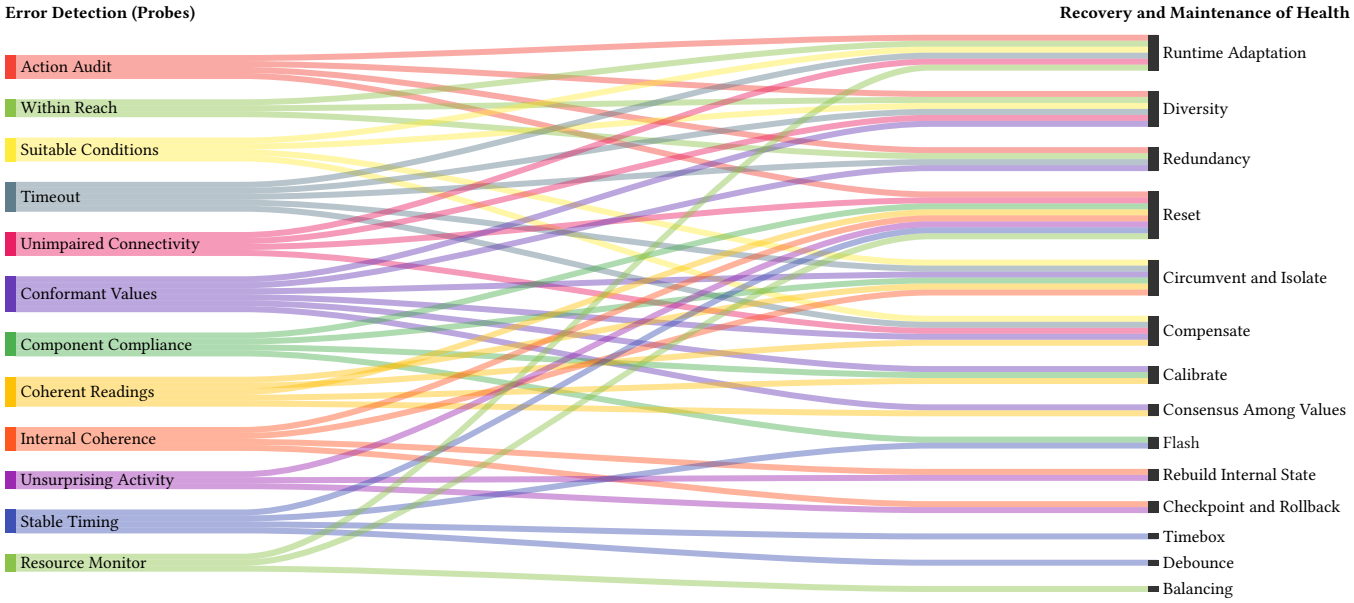


Figure 3: The self-healing pattern language for IoT systems. Each *error detection* pattern on the left can identify issues that can be solved by one or more *recovery & maintenance of health* patterns on the right. Although these mostly direct relations between the two pattern categories, the *error detection* patterns can be used together to enhance diagnosis and *recovery & maintenance of health* patterns can also be used together to recover the system to an healthy state (Fig. 13).

- *self-optimization*: maximizing resource utilization to improve the quality of the service;
- *self-protection*: anticipating, detecting, identifying, and protecting itself from attacks.

IoT systems have been primarily identified as a core example of a system that must contemplate autonomic components [6, 7, 68]. These components — that can range from single devices (*e.g.*, smart locks) to whole systems (*e.g.*, smart homes) — should be capable of self-management, reducing the need for frequent human operation [38]. This becomes even more important in critical systems and when devices are deployed in remote (*e.g.*, wildfire control) or other hard to access areas (*e.g.*, in the user's home).

Some IoT systems are *close-loop* systems. These act based on sensors measurements in order to maintain a predictable output (feedback-loop). Examples are Cyber-Physical Systems (CPS) and some Industrial IoT systems [15]. Other systems are *open-loop*. These take input under consideration but do not react only based on those inputs (no feedback-loop) [12]. As a result, making IoT *open-loop* (there is no verification that an actuator performed the required operation) systems resilient is harder than *closed-loop* ones, due to the lack of feedback.

Nonetheless, any kind of IoT systems should be capable of reconfiguring themselves to recover from failures. A self-healing enabled system should be able to *detect disruptions, diagnose the failure root cause and to derive a remedy, and recovering with a sound strategy* in a timely fashion (Fig. 2) [53].

The existing approaches for fault-tolerance (and *self-healing*) typically follow a *reactive* methodology where errors are detected and then recovered from (*e.g.*, complex event processing, system watchdogs and supervisors) (*cf.* DEVICE ERROR DATA SUPERVISOR [54]), or a *proactive* (also known as *preventive*) methodology where errors are *predicted* and avoided before faults being triggered using machine learning and other predictive mechanisms (*cf.* PREDICTIVE DEVICE MONITOR [54]). A combination of both can also be used [53].

4 SELF-HEALING PATTERN LANGUAGE

The patterns described in Section 5 — *error detection* patterns — and 6 — *recovery & maintenance of health* patterns — are inter-winded and can be mapped into a pattern language. The patterns in this language are presented in Fig. 3. The relationships specified in Fig. 3 are *first-degree* relations which point to possible recovery & maintenance of health patterns that address issues identified by certain error detection patterns. Relationships between the patterns in the same category also exist, and are illustrated to some extent in the following sections.

The patterns are presented as *patlets* of *problem-solution* pairs instead of the more structured traditional fashion of patterns. Thus, these become more general *guidelines* of design, proving a small insight on how to improve an IoT system reliability through self-healing, but are not prescriptive implementation solutions, mostly due to the wide range of application domains, competing standards, and abstraction levels (ranging from hardware concerns to software issues). Giving concrete implementations of each pattern would force us to drill down to the specifies of the technologies used (and, even, specific proprietary protocols and solutions), operational context, and users' capability to interact with the system.

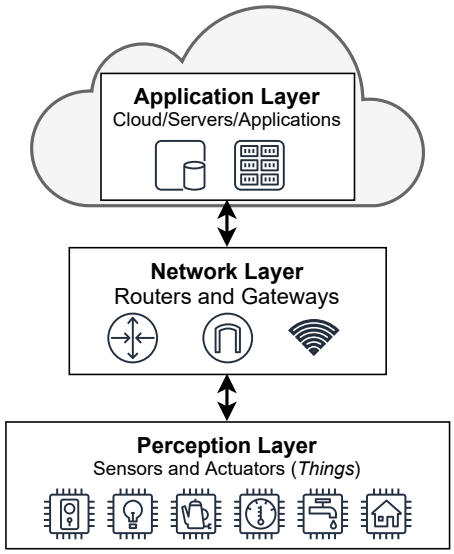


Figure 4: A three-layer view of Internet-of-Things. The perception layer is made up of constrained devices with sensing and actuating capabilities. The network layer interconnects the different parts of the system using a plethora of protocols and radios. The application layer provides storage, processing and services to the end-users [34].

We hypothesize that several pattern sequences can exist, combining the patterns that are part of the language here presented. On the one hand, several *error detection* patterns are used in sequence to diagnosing a problem (or improve the knowledge about a problem). On the other hand, several *recovery & maintenance of health* patterns can be used in sequence to recover the system to a healthy state. Further, several *recovery & maintenance of health* patterns can be used to maintain the system stable while others (*e.g.*, concurrently) try to recover certain parts of the malfunctioning system.

We consider that most of these patterns can be used at different layers as per the specification given in Fig. 4. As an example, missing sensor data can be compensated at the perception layer, where the device has some mechanism to fill in the missing values (*e.g.*, average, maximum or minimum in a timebox) or at the application layer, where, with more computing capability (*e.g.*, cloud), data mining strategies can be used to guess the missing values.

Additionally, we suggest that these patterns can both be used at an integration level, where the user is building the system by "connecting" *boxes* together and some of these are (or have) self-heal features (*e.g.*, visual programming) [25], or at a device/system design and developing level where the logic of the devices, gateways, and servers is coded with these patterns in mind. We believe that, in this way, we can provide insights that apply even when taking into consideration the current IoT fragmentation, which can be useful for practitioners, researchers, and individuals alike.

5 ERROR DETECTION (PROBES)

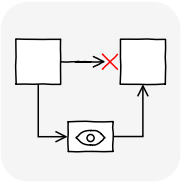
Enabling IoT systems with self-healing capabilities requires systems capable of detecting errors and failures. In this work, we consider error detection the process of detecting issues (both errors and failures) on the running system that can make it enter in a *degradation state* or *defective state*, as per the loop presented in Fig. 2 [36].

An error can be in one of two states: *detected* if an error message/signal indicates its presence, and *latent* if undetected. Error detection can be threefold: (1) identifying that there is an error somewhere in the system (FAULT CORRELATION [33], FAULT OBSERVER [33]), (2) identifying an error root cause (ROOT CAUSE ANALYSIS [33]), and (3) trigger reactive measures to recover and maintain the system health.

The patterns described in the following paragraphs are not recurrent solutions to problems *per se*, but recurrent solutions on *how* to detect different types of issues (faults, errors, and failures) in a system, mostly focusing on one or more parts of the three-part process of *error detection*. The application of these patterns in a system enables it to provide information, mostly at runtime, to *recovery & maintenance of health* mechanisms (which will be further described in Section 6).

Most of the *probe* patterns here presented can be enhanced their diagnostic *precision* by using other *probes*, potentially improving the diagnostic of a specific malfunction or unexpected behavior. Nonetheless, each *probe* has a well-defined target error; thus, they are sufficient to detect the error they target (but each can have limitations in finding the root cause). These relationships are presented in the form of diagrams that appear together with the patterns *patlets*. These patterns are contextualized within the scope of IoT systems like the one presented in Section 2, among others such as *smart farming* and *smart cities* ones. The following paragraphs present the patterns related to the *error detection* category (left side of Fig. 3).

5.1 Action Audit



In an empty home, a smoke detector has been activated; two alarms were triggered in response: one turning on a loud siren, and the other messaging the homeowner. There is a real possibility of a fire raging on, and time is of the essence. The system automatically attempted to perform these actions, but sometimes things do not go as planned. Maybe the message never reached the owner; maybe the siren was broken; maybe both of these things happened. Some actions are critical, and when they fail, counter-measures must be taken. Comfort, and even well-being, can depend on the proper functioning of components. How to guarantee that required actions are triggered when needed?

Therefore, implement a mechanism that validates each action. The siren makes a sound, so it should be audible by a noise sensor. The message can request human acknowledgment via a reply. If these action checks fail, one can resort to alternate pathways to mitigate potential issues, like triggering a light strobe or directly calling 911 (*cf.* RUNTIME ADAPTION, CIRCUMVENT AND ISOLATE), or try resetting the device (*cf.* RESET).

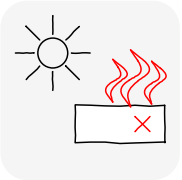
Action Audit

- Component Compliance
- Conformant Values

Figure 5: Diagnostic enhancement tree for the ACTION AUDIT pattern.

Rationale. *Smart home* applications and their underlying platforms take a *fire-and-forget* approach when issuing commands to actuators. The mistake lies in the assumption that everything will work between the triggering of a command and the intended purpose,*e.g.*, , the message was not corrupted, the communication layer is working, and the hardware is not faulty (Fig. 5). From a control-theory perspective, IoT systems could learn to benefit from a closed-loop approach in which, based on observational feedback, commands are not simply assumed to have worked until the desired effective state is achieved [51, 66]. However, performing such checks might need additional infrastructure that allows one to probe the effect. Depending on the nature of the action, this might be accomplished through different hardware (the noise sensor in our example) by performing a reverse computation to check if the output matches the input constraints. [69]. **Also see:** ACKNOWLEDGMENT [31, 60], TEST ALERTS [51], TEST ACTIONS [51], INDIRECT RESPONSE CHECK [56], CHECK PHYSICAL RESPONSE [56]

5.2 Suitable Conditions



It is a particularly hot day outside. So hot that one can fry an egg on top of any of their devices. The outdoor temperature sensors point to an average temperature of 41 °C, but, mysteriously, despite its AC unit being blasting cold air for the last two hours, the indoor garage sensor still points to an abnormally high temperature of 38 °C. Maybe the window is broken, but the shattering glass sensor did not trigger. On further inspection, the specification of this particular temperature sensor state that the *Recommended Operating Conditions* over operating free-air temperature range goes from −10 °C to 50 °C [3]. How can the system or its parts became aware that they can operate correctly?

Therefore, monitor if surrounding conditions are suitable for device operation, usually known as *operation thresholds* or *recommended operating conditions*. If values start getting unacceptable close limits, there is a likelihood that the device could stop working, or (even worse), malfunction[4]. Take preemptive actions to mitigate potential failures (*cf.* RUNTIME ADAPTION, CIRCUMVENT AND ISOLATE).

Rationale. Most commercial Integrated Circuits (*i.e.*, *chips*) have recommended operating conditions that do not perform well below the water freezing point but go as far as 70 °C. The reason most IoT systems disregard this problem is that devices are assumed to be placed indoors. Nevertheless, this assumption can still be broken in a cascading scenario such as the one presented above; 70 °C is not unheard of if the device is behind a glass window with sun shining on it. Checking for environmental constraints is a relatively straightforward process, and most IoT devices must state their operating conditions in the manuals for FCC or CE approval [20]. The capability of flagging a device in an *unreliable state* allows mitigation strategies to be preemptively triggered. However, this depends on having external data sources (independent, and redundant, sensors or third-party services) that provide the data that allows one to detect if some device is operating in ideal conditions or not[5]. **Also see:** Dependability requirements [13], Environment-aware communication protocols [13].

[3] *Recommended Operating Conditions* are detailed descriptions of the conditions in which the device performs as expected is typically documented in the device's user manual or datasheet.

[4] A door that does not automatically open is a problem. A door that automatically opens when it should not is a *bigger* problem.

5.3 Reasonable Values

Things tend to act as expected. Even unexpected events usually present a pattern. An outside luminosity sensor is expected to follow a curve according to the sun's position; unless there are clouds (or a solar eclipse). Temperatures do not drop from 20 °C to 0 °C in the blink of an eye, and then immediately recover back; there is expected inertia to it. Humidity exhibits reasonable gradients; unless someone is taking a shower in the bathroom. All these situations present *reasonable* patterns of readings one is expecting from sensors. If the readings do not fit these patterns, then they might be untrustable.

Therefore, consider the reasonableness of the readings before blindly accepting them as valid values. Use different checks (or a combination of them) depending on the particular sensor to detect unreasonable situations. There will always be a degree of confidence in this assessment, that can vary from *suspicious activity* (spikes in luminosity), to outright *impossibility* (readings outside working intervals). Once they are detected, different strategies can be employed to deal with erroneous values adequately (*cf.* RUNTIME ADAPTION, CIRCUMVENT AND ISOLATE, CONSENSUS AMONG VALUES, COMPENSATE, CALIBRATE, RESET).

Reasonable Values

- Component Compliance
- Suitable Conditions

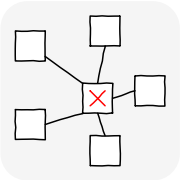
Figure 6: Diagnostic enhancement tree for the REASONABLE VALUES pattern.

Rationale. Something that deviates from what is standard, normal, or expected, is usually called an *anomaly*. There is a whole subfield of computer science and mathematics called *anomaly detection* that might employ sophisticated algorithms to search for situations that deviate from normality, and the data itself defines what

[5] Some approximation can always be made if the read values by the device itself, *e.g.*, humidity, are too close to its the operating limits

establishes as normal [19]. Notwithstanding, there are other scenarios were normality is well-known: reading intervals and physics are two extremely informative sources. In these scenarios, specific strategies might be employed to assess the *degree of confidence* in the values, up to the point of *unreasonableness*. Once we identify these situations, mitigation techniques might be able to extract a *workable* value by filtering out the noise; otherwise, isolation techniques could flag the devices as *unreliable* (Fig. 6). One should note that operating outside the recommended conditions might lead to unreasonable readings, but the reverse implication is not true. **Alias**: *Plausible Values*. **Also see:** Test Periodic Readings, Test Triggered Readings [51], Mean and Variance, Correlation, Gradient and Distance from other readings [50], REALISTIC THRESHOLD [31], Complex-Event Processing (CEP) [52].

5.4 Unimpaired Connectivity



An edge device is attempting to send its reading back to the server (*i.e.*, the message recipient), but the server is not answering back. Depending on the reading's importance, the value might be discarded or saved to be resent later. However, memory is not infinite, and the urgency of the message might require immediate action. If the failure remains, the device might be forced to decide a secondary course of action (*i.e.*, Diversity). How to ensure that the different entities in a system are alive and can communicate with other parts?

Therefore, start by checking if the infrastructure supports the intended connectivity by attempting communication to a secondary target, but through the same connectivity tissue. This can be done by any system part (*e.g.*, a coordinator trying to access a device or a device trying to reach a third-party service). If the message recipient is on the cloud, ping a different known server on the internet can ping another edge device if it is local. If it is concluded that the communication infrastructure is defective, try resetting it or using another one (*cf.* RUNTIME ADAPTION, RESET); if you can communicate with other devices through that same medium, look for the problem elsewhere (*cf.* WITHIN REACH).

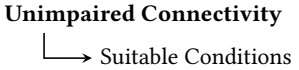
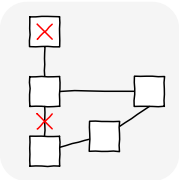


Figure 7: Diagnostic enhancement tree for the UNIMPAIRED CONNECTIVITY pattern.

Rationale. A simple diagnostic (ping to a secondary service) might discard a connectivity problem. Knowing the difference between the two conditions (the message recipient being down, and the connectivity tissue malfunctioning) might allow the system to take different actions; if the connectivity is down, one might consider using an alternate radio to find the recipient (*e.g.*, GSM, Lora or ZigBee). A typical example would be a fog device that triggers a rule that would make an alarm go off via a WiFi connection. Further checks (*e.g.*, ACTION AUDIT) reveal that the order was not fulfilled. Most devices in the local network fail the heartbeat checks, and attempts to connect to other cloud-based services and edge devices are failing. Switching to a secondary radio protocol (*e.g.*, 433MHz) might allow the intended goal to be carried (Fig. 7). **Also see**: Dead Spots (*RF holes*) [4], Locate disconnected client [4], Performance Isolation [4]

5.5 Within Reach



Edge sensors that report frequent reading, such as temperature ones, are usually working continuously, and the system can readily observe that they exist because of their constant message throughput. Edge actuators, like alarms, might only actuate in rare circumstances. The rough confidence that a device will not fail to operate when needed (in this case, disregarding failures of the device mechanical parts) is directly proportional to how well (and often) previous communications were successful. How can one know that some system part is available and responsive when required as it is designed to idle most of the time?

Therefore, if two devices are going to trade messages infrequently, establish a way to increase the confidence in their communication, forcing them to communicate event if to demonstrate that they are operational (*cf.* RUNTIME ADAPTION).

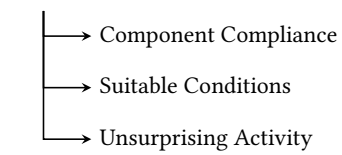
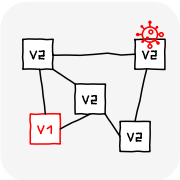


Figure 8: Diagnostic enhancement tree for the WITHIN REACH pattern.

Rationale. There are (broadly speaking) two types of connectivity checks: (1) a deliberate, scheduled broadcast of connectivity is called an HEARTBEAT [26, 31], and it usually occurs from devices (edge) to servers/nodes (cloud/fog); (2) a point check of connectivity is called a PING, and it usually occurs in the opposite direction of a heartbeat, *i.e.*, from the servers/nodes (cloud/fog) to the devices (edge) (Fig. 8). The first is mostly used to preemptively capture potential connectivity failures before action is needed (*e.g.*, a failed heartbeat from a siren might imply the system entering a warning state). The second is mostly used as a diagnostic mechanism to find out if the device is out-of-reach or in an abnormal state. Several mechanisms can be used to meet these *alive* checks, such as the periodic broadcast of status messages or push/pull of telemetry data between system parts. However, one must consider that in low-power solutions (*e.g.*, battery-powered devices), forcing the devices to make themselves *alive* when it is not needed can have a drastic impact on their battery-life (*i.e.*, devices that support *deep-sleep* will drain more energy due to the more frequent power-cycles). **Also see:** ACKNOWLEDGMENT [31], ARE YOU ALIVE [60], I AM ALIVE [60], NACK [22], ACK [22].

5.6 Component Compliance

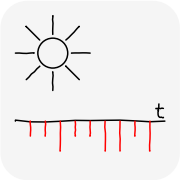


Software is not set in stone, which is the most significant advantage of programmable things and their ultimate curse. We expect things to behave and perform the same unless physical tearing and breakdown occur. Nevertheless, the software can also tear and breakdown, both through usage and time, as well as through malicious intentions or users' configurations. Devices also gain new capabilities, have their configurations changed (*e.g.*, RESET), have their vulnerabilities patched, and their bugs fixed via software updates. Moreover, these can happen more frequently, over-the-air (OTA), and unassisted, with recent advancements. How can we be sure that the devices are executing the software they are expected to be running while complying with the current system configurations (interoperability)?

Therefore, check if a particular device is running what it should in the way it should, by frequently observing their software versions, configurations, firmware hashes, and any other checksum mechanisms. This check can be carried out by different system parts (*e.g.*, a gateway that periodically checks the edge devices versions for updates and checksums for corruptions) or by devices' self-checks (that can detect *modifications* at runtime).

Rationale. Several reasons can render devices running unexpected software, namely tampered devices (detected via integrity checks), newer versions (detected via update checks), known vulnerabilities (detected via audit checks) or misconfigurations (detected using different types of configuration checks [41, 63]). Typical recovery actions include factory resets, reboots (*cf.* RESET), firmware re-installs, re-configurations, updates and downgrades (*cf.* FLASH). In some situations, where recovery is not possible, contingency actions must be done (*cf.* CIRCUMVENT AND ISOLATE). Ensuring the correct use of this pattern depends on having entities (*e.g.*, servers) that provide the latest stable software packages along with verification checksums[6]. These also should have security standards that enhance the confidence of the checks. **Also see:** Firmware Integrity Assurance [5], Update [5], OTA upgrade [30] and OTA downgrade [30], MIDDLEMAN UPDATE [24], PROTOCOL VERSION HANDSHAKE [26].

5.7 Coherent Readings



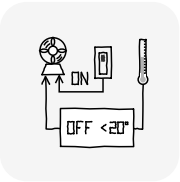
Sometimes, a fact comes to your attention that, although entirely plausible, you know it is probably wrong. Not because of its absurdity *per se*, but because you have other evidence contradicting it. Sensors are the same; sometimes, readings might be perfectly fine on their own, but when confronted with values coming from different sensors, they are not. For example, it is expected that multiple temperature sensors inside the same environment to report slightly different values. Still, when one of them communicates a wildly different one, something must be wrong. How can one ensure that the readings are faithful and that inaccurate/incorrect readings are flagged as such?

[6]Nonetheless, these checks can have some intermediaries, *i.e.*, MIDDLEMAN UPDATE [24]

Therefore, compare values from different sources, and check if they are in accordance so that erroneous readings can be detected (*cf.* REDUNDANCY, DIVERSITY, CONSENSUS AMONG VALUES). If one sensor is consistently reporting widely different values, maybe you should try resetting or calibrate it (*cf.* RESET, CALIBRATE), or maybe just stop trusting it altogether (*cf.* CIRCUMVENT AND ISOLATE).

Rationale. By crosschecking values coming from different sources, we can detect problems that might not be apparent in any other way. Multiple sources must report approximated values if they are deployed in similar conditions. Even if the sources are entirely different (*e.g.*, humidity and rain detection), inconsistencies can still be inferred (detecting rain, while the humidity sensor reports a dry environment would be a strange occurrence). Nonetheless, a sensor that provides "unexpected" values consistently can point to an abnormal situation, *e.g.*, if a fire starts in a home division, only the sensor deployed there will be triggered. **Alias**: *Consistent Readings*. **Also see:** N-Version Programming [21, 69], FAIL-STOP PROCESSOR [60], SICO FIRST AND ALWAYS [3].

5.8 Internal Coherence



Actions are being performed in your system; *e.g.*, turning switches on, regulating the AC and the windows blinds, activating the irrigation system… All these actions lead to changes in the system, whose state is usually mirrored internally (*i.e.*, instead of continually asking if a switch is *on* or *off*, one usually stores the latest known state). Sometimes, though, devices act on their own (*e.g.*, due to a reset or human intervention) and change the state without (or failing to) informing the rest of the system. As an example, consider a light power switch that can be controlled by (1) manually toggling the switch, (2) a mobile application, and (3) a configured light-sensing trigger action. Depending on message delays, packet losses, system reboots or, even misconfigurations, the system can enter in an inconsistent state, where it no longer knows the state of the lights and can lead the user to make incorrect decisions. The problem increases when there are more configurations beyond a simple on/off state, such as using the same example, light colors/temperature. How can we make sure the internal representation of the system reflects its actual state?

Therefore, perform regular checks of the system's internal representation when possible, making sure that it correctly mirrors the actual devices' state. This is specially important after a RESET or a FLASH and might require the device to REBUILD INTERNAL STATE.

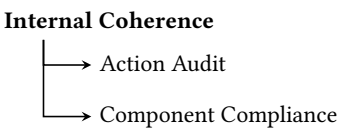


Figure 9: Diagnostic enhancement tree for the INTERNAL COHERENCE pattern.

Rationale. The maintenance of an internal representation of the system exists for several purposes, the most common one being *performance* or to avoid constantly querying a device about its status. However, this assumption that changes in the system are always successfully reported (Fig. 9). Any small overlook in reporting can easily create inconsistencies between the physical setup and its *internal representation*, which might eventually cascade in the decision process leading to a degraded/defective state. **Alias**: *Internal Consistency*. **Also see:** DEVICE REGISTRY [54], SYSTEM MONITOR [62], Resource Discovery for Fault Detection [73].

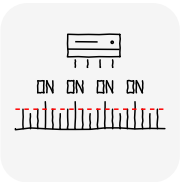
5.9 Stable Timing

Your devices are continuously talking with each other, sending messages to inform about a particular value, or asking another device to execute a specific action. However, timing is everything. Two messages in the wrong order are enough to leave the system in a defective state; delayed readings can be the difference between taking the appropriate action in time to prevent damage or the information no longer being relevant. How can we detect data that is not arriving on time, on an irregular basis, or in the wrong order?

Therefore, have mechanisms that can detect if devices are sending messages at the expected intervals, and taking the expected time to respond to the messages they are receiving. If they are not, you can try to reset them (*cf.* RESET), use other sensors (*cf.* RUNTIME ADAPTION, CIRCUMVENT AND ISOLATE), or mitigate the problem (*cf.* DEBOUNCE).

Rationale. Timing can be critical in IoT systems, and system degradation might cause devices to start taking more time to act upon the messages they are receiving. These delays can cause mischief. Sometimes the time between a reading and a device carrying the appropriate action can be the difference between preventing a fire or irreparable damage, sometimes two steps in the reverse order might be the difference between a vital switch staying on or off. **Also see:** Data-Driven Synchronization [11], Bubble Razor [28], DFix [44]

5.10 Unsurprising Activity



Home is where you know how to set the thermostat just right, and you expect that perfect temperature to be maintained indefinitely. That should be easy enough, turn the A/C off if it is too hot, and the heater if it is too cold. At first glance, these two simple rules might seem obvious enough, but they hide a serious problem. As the temperature fluctuates around that desired temperature, a futile and power-hungry battle between the heater and A/C rages on, with each one taking turns trying its best to push the problem into the other's hands. How can this kind of suspicious activity be detected?

Therefore, check if any device is sending a suspicious number of messages, as that might indicate severe hardware or logic problem.

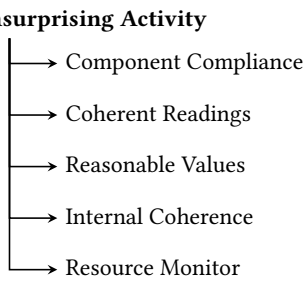
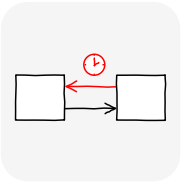


Figure 10: Diagnostic enhancement tree for the UNSURPRISING ACTIVITY pattern.

Rationale. The inappropriate usage of a device might degrade its lifetime or result in an undesired usage from it. Such usage can result from a damaged or malicious device or entity that would continuously ask the device to perform the same operation (Fig. 10). By monitoring the messages being sent to a device, and establishing a reasonable usage restriction to it, it would be possible to identify misuse patterns, informing the *recovery & maintenance of health* mechanisms in place (*e.g.*, Circumvent and Isolateand Reset). **Also see:** BLACKLIST [58], WHITELIST [58]

5.11 Timeout



You want to trigger an action and be sure that it executes within a given time frame. If it does not, an error must have occurred. For example, you want to alert a homeowner that they did not enable his home alarm, despite being out of the house. If the alarm is not enabled within 20 minutes, you want to call him to ensure that they are aware that the alarm is disabled. How can one be sure that a particular action is executed?

Therefore, keep a timer running since the first action and observe if a reaction happened, if the timer runs out without the reaction, an error has occurred. The root cause can range from a device issue, *i.e.*, ACTION AUDIT to a network disruption *i.e.*, UNIMPAIRED CONNECTIVITY, WITHIN REACH.

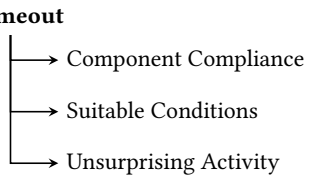
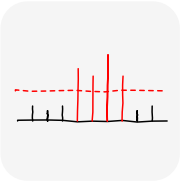


Figure 11: Diagnostic enhancement tree for the TIMEOUT pattern.

Rationale. When a device requests an action from another, the time it takes for the action to be completed might be critical (Fig. 11). If the action fails to complete within the acceptable time frame, the triggering device should be able to trigger an alternate action. For that, it should run an internal timer, during which it observes if

the action has concluded. When the timer runs out, if the desired reaction is not observed, an alternate action is triggered, to which a Timeout probe can be used as well, working as a *maintenance of health* mechanism. **Also see:** LIMIT RETRIES [26, 31]

5.12 Conformant Values



Devices can sometimes have faults in their hardware (and, sometimes, in their software) that can lead them to behave out of their specification. Per example, sensor devices that use percentage values in their reading can not output negative value nor values above 100%. How can we assure that sensors are operating accordingly to their manufacturer specification?

Therefore, check if the device readings are in conformance with the device reading thresholds, which are stated in the device specification.

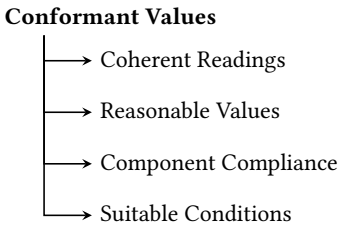
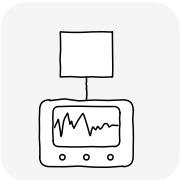


Figure 12: Diagnostic enhancement tree for the CONFORMANT VALUES pattern.

Rationale. Most common hardware present in IoT devices is low-cost and has, typically, a high-proneness to failure. One of the possible outcomes of failure can be, for example, a sensor producing values that are not part of their specification (Fig. 12). In such scenarios, the system should be able to distinguish between valid values or out-of-spec ones, allowing *recovery & maintenance of health* measures to be put in place (*e.g.*, COMPENSATE and CIRCUMVENT AND ISOLATE) **Also see:** Threshold Check [69], Reasonableness Check [69].

5.13 Resource Monitor



Entities in IoT system, ranging from low-power devices to powerful servers, have constraints such as processing power, storage capacity, bandwidth, among others[7]. Spikes in system usage can lead to malfunctions and severally impact the Quality-of-Service (QoS). The need appears to oversee the resources that the system is consuming both in real-time as well as historically.

Therefore, monitor the system resources at all times, ranging from battery levels to network operation and resource usage, providing insights on the system's bottlenecks and related issues.

[7]Even for cloud computing, limitations exist associated with the cost and availability of resources

Rationale. System need for resources varies during its operation. As an example, a *smart home* system can sit mostly idle during the time that the house is empty; however, as inhabitants arrive home and interact with the system, the resource usage can spike. While most cloud-based systems can scale resources on-demand, the devices spread around the house (*i.e.* fog and edge devices), which are typically very limited (*e.g.* processing power) can easily provoke issues in the system (such as QoS degradation). Thus, resource monitor can both provide insights on current issues on the system as well as on potential issues, allowing them to perform actions accordingly (*e.g.*, BALANCING). **Also see:** EXTERNAL MONITORING [65], RESOURCE MONITORING [46].

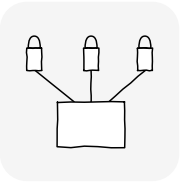
6 RECOVERY & MAINTENANCE OF HEALTH

If errors are detected in a system, enabling the system to *self-heal* requires *recovery & maintenance of health* mechanisms. These mechanisms act per the *probes*, reacting to the information reported by them (*i.e.*, if a probe detects an error recovery or maintenance of health mechanism should be triggered, avoiding system degradation or, even, recovering from a defective state).

The following paragraphs delve into the patterns that correspond to the common *recovery & maintenance of health* mechanisms. These mostly rely on the information provided by the error detection mechanisms described in the previous section; thus, *probes* and *recovery & maintenance of health* mechanisms work in tandem to enable a system to self-heal. Most of these patterns can work together in order to restore the system to a healthy state.

As previously, withing the scope of IoT systems, *smart spaces* such as the one presented in Section 2. The following paragraphs describe the pattern of the *recovery & maintenance of health* (right side of the Fig. 3).

6.1 Redundancy



Things are prone to fail, both in hardware (*e.g.*, power-spikes) and software (*e.g.*, corrupted software update). Even when the different system entities report what seems to be reasonable values (*cf.* Reasonable Values), there is no assurance that it is the real value, since there is no comparison point. In cities, if the air quality control was made by only one sensor, there was no way to distinguish a strange reading (*e.g.*, due to a broken sensor or by a spike provoked by a heavy-duty vehicle passing by) if there is no other record to compare with. How can we ensure that the system provides *correct* service at all times?

Therefore, use redundant mechanisms to achieve the same goal, allowing one to both make decisions on which report to believe in, or to trigger the same action using another way.

Rationale. In more sensitive scenarios, there is a need to deploy redundant units (*i.e.*, redundancy in space) that can report the same measurements or trigger the same actions to make the system survive to partial failures into account the extra costs. Such redundancy can be achieved by deploying similar or equal sensing or acting devices, communication channels (*e.g.*, different radios), processing units (*e.g.*, microservices), third-party services providers (*e.g.*,

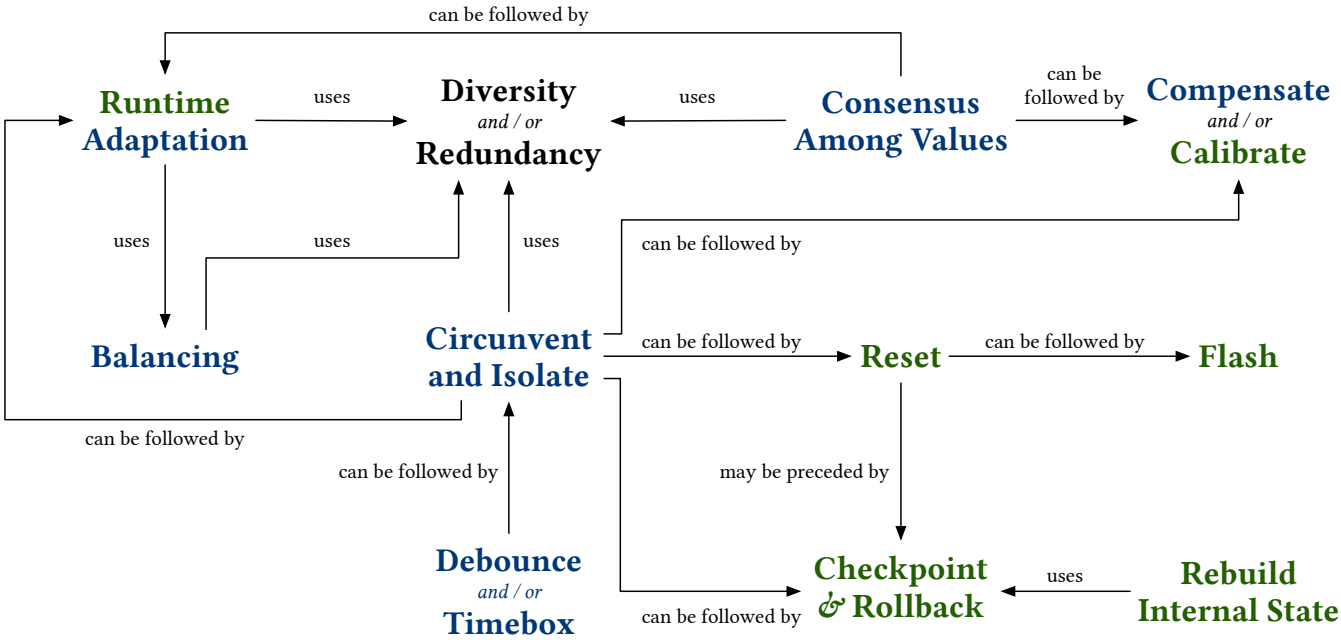
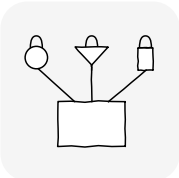


Figure 13: The *recovery & maintenance of health* patterns and their sequences of actions (service restorations) towards normal state. Some of the patterns provide the foundations for others (no color), others work as maintenance of health (blue-colored) and, finally, the remaining work as system recovery mechanisms (green-colored).

using both Amazon Alexa and Microsoft Cortana), and even different power-sources to support the running system (*e.g.*, solar and batteries). In sensing scenarios (*e.g.*, environmental) it can be possible to use *redundancy in time* which consists on taking several measurements in a time-window and only report the most *correct* (*e.g.*, common) reading (*i.e.*, dropping outliers), *viz.*, REASONABLE VALUES. **Also see:** FAILOVER [31], REMOTE STORAGE [31], PASSIVE REPLICATION [60], SEMI-PASSIVE REPLICATION [60], SEMI-ACTIVE REPLICATION [60], ACTIVE REPLICATION [60], Different types of wide-area networks [66], 1+1 REDUNDANCY [26]

6.2 Diversity

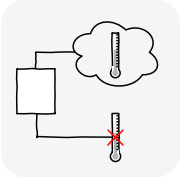


Having multiple components, such as light bulbs, in the same space for the same purpose have the nice side-effect of acting as redundant components when one of them fails. However, this is tightly coupled with the cost of the device; more expensive objects, such as AC units, are usually deployed in a minimum-required number in the same environment. How can we improve the recovering and maintenance of health capabilities in a system where some components are not redundantly deployed?

Therefore, use different entities to achieve a common goal and reduce the impact of faulty parts. There are sometimes *alternative* ways of achieving a common goal without adding new entities to the system. During the daytime, a way to compensate a broken light can be to open the windows' blinds and let in sunlight. Lowering the temperature can also be achieved by opening a window instead of turning on the AC.

Rationale. Redundancy *per se, e.g.*, triple modular redundancy, is rare in IoT systems due to the associated costs for mostly non-critical tasks. Nevertheless, alternative (and possibly already existent) mechanisms can be used to accomplish tasks that are not part of their primary functions, reducing the systems' *points of failure*. Diversity does not need to be applied only to devices; mechanisms such as communication channels (*e.g.*, WiFi, Bluetooth, 433Mhz) can also be the target of diversity, mitigating the effects of Unimpaired Connectivity. Nonetheless, adding diversity to the system will increase its *complexity*, thus possibly impacting the system's overall cost, maintainability, and understandability. Some authors have been proposing the idea of *automatic workarounds* to leverage the already existent diversity (and redundancy) on the system to recover from failures [17]. **Also see:** CIRCUIT BREAKER [39], Design Diversity [9], Automatic Workarounds [17], Protocol Switching [57].

6.3 Runtime Adaption



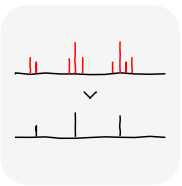
Devices typically have different operating modes which are enabled in different setups or operation conditions. Consider the example of a lightbulb that connected by ZigBee to the main hub; if there is no hub present, the bulb can be controlled by a dedicated remote control. Another example is the case of WiFi-connected

smart plugs; the first time that they are turned on, an Access Point (AP) is created (or a Bluetooth connection) that allows the user to enter the home WiFi credentials, and then the *plug* connects itself to the home network. However, after things are configured once, the devices typically do not employ fallback measures. In the *smart plug* case, if there are disruptions on the WiFi service, the user will not be able to access their smart home system, even if changing the *smart plug* to AP-mode would temporarily fix the issue. Since IoT systems typically rely on low-cost components that might be deployed in harsh environments, which increase the likelihood of *soft-errors* (*e.g.*, intermittent erroneous sensing data or issues on communication) [18], fallback strategies can be crucial in preserving system operation. How can the system deal with different system degradation (partial failures) while still providing services to the user?

Therefore, enable the system to adapt during runtime, allowing the system to use different infrastructure seamlessly (physical or virtual, *i.e.*, Avatar or Digital Twin [58]) during operation. When the usual infrastructure recovers to a healthy state, the system must change back to the usual channels [53]

Rationale. There are several devices/services that already have the built-in capabilities to provide services even when facing some degree of service degradation. However, typically, these capabilities are not taken advantage in a resilience perspective. The *soft-errors* can allow the system to continue operating but can require a certain amount of adaptation to the runtime conditions. If the device is low on battery, its transmission power (*e.g.*, WiFi) can be impacted. However, changing to a more low-powered communication protocol can allow the system to continue operating for more time without human intervention. Further, if a device cannot connect to the communication infrastructure, it can create its own AP for giving users the ability to control or get data from the device. Similarly, if the system depends on a device to report *e.g.*, environment temperature, if there is a failure in receiving data from that device, the system can fall back automatically to a different source (*e.g.*, third-party weather service). Several solutions employ the concept of Avatar or Digital Twin, virtual representations of devices that, beyond making the bridge between the virtual and real-worlds, are capable of some adaptation such as using other physical units in case of detecting errors or simulate the behavior of the real counterpart if none is available (*cf.* COMPENSATE). It must be considered that these adaptations can compromise the system's capabilities (*e.g.*, data-rate), which may not be viable in certain scenarios. **Alias**: *Dynamic Binding*, *Reconfiguration*. **Also see:** CIRCUIT BREAKER [39], REINTEGRATION [31], DEVICE SHADOW [59].

6.4 Debounce

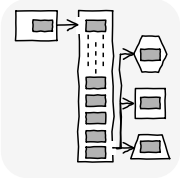


A new temperature sensor was added to the *smart home* to improve the control of room temperature (with more precision than an already existent sensor). However, the cooling system began to show unpredictable behaviors from time to time. The sensor might have been broken; yet, replacing it with a new sensor unit did not solve the issue. Further investigation, mostly by reading the cooling system user manual, showed that the system was trying to collect sensing data from the sensing device at a rate above its sensing period, which lead the cooling system to misbehave (since it had no data to decide upon). How can we meet the device operational constraints without compromise the system operation?

Therefore, filter or aggregate events to meet operational timing constraints, ensuring that the target device receives (or is able to collect) data at the expected frequency. Optionally aggregate them with an average, maximum, minimum, or other strategies to provide the device's relevant data.

Rationale. Both sensors and actuators have several operating constraints, such as power supply, accuracy, and operation range. Amongst those constraints, there is some that limit how many times a device can trigger a certain action, namely, how often readings can be *collected* from a sensor, and how many times or with which frequency an actuator can be triggered, namely: (1) sensing periods and (2) mechanical/electrical life and operation/release time. If a system does not respect these parameters, in the case of sensors, it can result in *undefined behavior* (*e.g.*, ranging from failures to collect values to collecting random data). In the case of actuators, this can reduce the life-spawn of devices, or even, having hazard repercussions. However, in most cases, humans will not notice if the system *delays* the triggering of a device (or delay the report of a value) a second or so (*e.g.*, any delay will possibly lead users to keep pressing the `ON` button until the lights turn on, *cf.* TIMEBOX) [66]. Thus, the system must implement mechanisms that debounce events to meet the system's operational constraints. While these issues are often dealt with at system development and testing phases, end-users can be impacted by the nonexistence of such mechanisms when upgrading their systems. **Also see:** QUEUE FOR RESOURCES [31], REQUEST DELAY [53], PROTECTIVE AUTOMATIC CONTROLS [31], SHED LOAD [31], SLOW IT DOWN [31].

6.5 Balancing



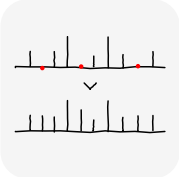
A street access control sub-system (part of a *smart city* system) has variable load. While the system is mostly idle during the night and weekends, during the work-days, the system is at its usage peak (*e.g.*, due to commuting). During this time, other hardware in the *smart city* might be sitting mostly idle. How can we ensure that the system is responsive at all times?

Therefore, distribute software and load between available resources to meet service demands. Do so by abstracting the underlying hardware and distributing the computational units automatically between the available hardware.

Rationale. The processing demands of a system can change rapidly due to peaks in usage (*e.g.*, access control or increase in the home inhabitants' activity) or the appearance of heavier computational tasks (*e.g.*, surveillance video processing). However, even during peaks, there can exist parts of the system that are idle (or, even, available redundant parts, *cf.* REDUNDANCY) that can be leveraged to meet the system usage demands during peaks (returning then to normal operation when they are not further needed, *cf.* RUNTIME

ADAPTION). **Also see:** SHARE THE LOAD [31], PROTECTIVE AUTOMATIC CONTROLS [31], ORCHESTRATION MANAGER [14].

6.6 Compensate

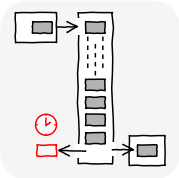


A dehumidifier was brought to eliminate musty odors and prevent the growth of mildew in the house. However, to avoid having it turned on at all times, an additional humidity sensor was brought that provides information that allows controlling when and for how long the dehumidifier is active. However, in several situations, the dehumidifier was not turning on due to issues with its sensing counterpart (due to, as an example, UNIMPAIRED CONNECTIVITY or STABLE TIMING). How to ensure that devices perform as expected even when there have some operational problems in their sensing counterpart?

Therefore, have mechanisms that can compensate missing or erroneous information, at least during an established period of time.

Rationale. While the system operation is best when all of its parts are working correctly, the malfunction of one part (*cf.* REASONABLE VALUES, UNIMPAIRED CONNECTIVITY and SUITABLE CONDITIONS) should not jeopardize the entire system. In these scenarios, mechanisms should be put in place to compensate for the missing information, allowing the system to keep operating. In the case that there is another source of data that can be used (*cf.* RUNTIME ADAPTION and REDUNDANCY), they should be adopted. However, when none of these alternatives exists (or are available), strategies such as using an average of the last measurements can be considered and used [9]. However, the confidence in this calculated values lowers as the time passes by, thus additionally, considerations about a *graceful degradation* should be taken (*e.g.*, by setting and using a default — reference — value) [27]. **Also see:** Kalman Filter [40], Interpolation and Correlation [61, 73], Linear and Non-Linear models [73]

6.7 Timebox

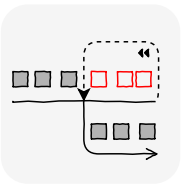


Having components that are both manually controlled *on-site* and remotely (*e.g.*, using a mobile application) leads the user to expect the same kind of behavior in both. However, while operating devices remotely, several constraints can slow the *feedback-loop*, such as network lag. This can lead to the user to keep sending the same request (*e.g.*, turn on the sprinklers) repeatedly until the application reflects the request (*e.g.*, showing a message informing that the sprinklers are on). However, this behavior can provoke malfunctions (even failures) in the system. How to prevent repetitive and similar actions (user or system-induced) in a short period from damaging the system?

Therefore, only process an order in a specific period that respects the system operational constraints, filtering (*e.g.*, dropping) the remaining requests within the timebox.

Rationale. When a state change request is made, sending the same request repeatedly to the make the action happen *faster* does not have real effects (since the system will always take some time to change to the requested state). However, if this behavior is not controlled, it can compromise the system; thus, similar or equal requests made within a pre-defined timebox should be discarded. Even if the system is required to respond to all the requests (or if the requests are different), it can only go as quickly as the system operational thresholds (*cf.* DEBOUNCE). **Also see:** LIMIT RETRIES [31], LIMIT NUMBER OF RETRIES [56]

6.8 Checkpoint

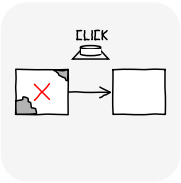


A *smart home* system can keep information about several aspects, such as lights state (on/off), presence (detection of motion in the last minutes) and last device activation's (*e.g.*, vacuum cleaner robot). However, in the case of a general system restart (by some error or on purpose), the system will act as new, changing all devices to default values, thus changing the lights accordingly, resetting the last time presence was detected, and starting to clean the house again (*e.g.*, activating the vacuum cleaner robot) even if it was cleaned just before the system restart. How to avoid the fallback to default values in such cases?

Therefore, preserve the current system state, avoiding the repetition of actions or changing devices states to default values after a disruptive recovery action.

Rationale. The correct functioning of an IoT system depends on preserving parts (or all) of the system current state (*i.e.*, checkpoint), enabling the system to restore to the last known state if a system error or restart/reset happens (*cf.* RESET). This allows the system to recovery seamlessly (*i.e.*, rollback), without repeating tasks and/or bothering the user to restore to the most current configurations. **Also see:** CHECKPOINT [31], WHAT TO SAVE [31], ROLLBACK [31], ROLL-FORWARD [31], CHECKPOINT [26], SNAPSHOT [26]

6.9 Reset



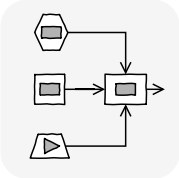
As the system operates, several faults can happen without triggering an *error* (*i.e.*, latent faults), going unnoticed by the users and, even, by the system itself. However, these faults can build up, leading to system errors and, possibly, system failures. How can we reduce the probability of errors and failures being triggered as time passes? Further, several issues can appear during service delivery that can compromise the correct operation of devices (*e.g.*, wrong user inputs, electromagnetic radiation, or power spikes) that can lead the device to present *undefined behaviour*. How can we restore the device to a healthy state?

Therefore, perform system resets, periodically (*e.g.*, during idle periods) or when some error is detected, working both as maintenance of health mechanism and, possibly, as a fault removal procedure.

Rationale. Reboot and reset strategies have been used for a long as a way to improve systems reliability both in terms of software (*e.g.*, application restart [66]) and hardware (*e.g.*, hardware watchdog timers) concerns [1, 2, 23]. The continuous system operation can lead to the creation of several latent faults that can be triggered,

leading to system errors and failures. Even if one could argue that with more resilient (*e.g.*, rigorously tested and verified) software and hardware, the probability of a system entering in an error or failure state is reduced, IoT systems are known to be built with low-cost components with high failure rates (*e.g.*, communication issues, sensing imprecision's) [18, 35]. These resets can be both *soft-resets* and *hard-resets*, depending on what they preserve in terms of the device's internal state. For example, *soft-resets* provably will only work for non-permanent faults (*e.g.*, if a fault is preserved in a checkpoint — *cf.* CHECKPOINT —, a hard-reset should be performed). However, depending on the device or system, rebooting/reset the system periodically can introduce inconsistencies in operation (*e.g.*, system state synchronization) than can negatively impact the system. Further, since reboot/reset can restart all its processes, any dormant storage corruption can provoke malfunctions. **Also see:** ROUTINE MAINTENANCE [31], ROUTINE EXERCISES [31],DATA RESET [31], ROLL-FORWARD [31]

6.10 Consensus Among Values

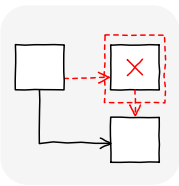


Triggering an alarm can have serious consequences, so, commonly, alarm systems rely on several and, sometimes, different sensors to keep the environment under check. However, a miss-configured or malfunction sensor can report some nefarious condition (*e.g.*, CO2 levels) without being correct. How can the system deal with such a report without erroneously triggering the alarm system?

Therefore, evaluate information from several sources before taking a decision, increasing the level of confidence on the action to take, minimizing the likelihood of mistakes.

Rationale. When several sources report data upon which the system acts, all the information should be considered. In most cases, malfunctions can be easily identified and discarded by comparing the collected information and only considering the information with which the majority agrees with (*i.e.*, voting). More advanced mechanisms can be used that also take into account the trustability of the reported values (*cf.* COMPENSATE) [66]. Nonetheless, there can be situations where the majority of the sensors are reporting erroneous values. In such situations, depending on factors such as the importance of such readings, more sensors can be added and/or the misreports should be considered and, at least, communicated to the system owner/administrator. **Also see:** VOTING [26, 31], Anomaly Detection [67]

6.11 Circumvent and Isolate

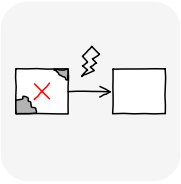


As the system operates along time parts of it can reach end-of-life (*e.g.*, an A/C which no longer can achieve the target temperature), having high-failure rates which can impact the overall system operation and, possibly, threaten the user comfort and well-being. How can the system deal with failing parts without compromising its correct operation?

Therefore, circumvent and isolate failing parts, by disabling faulty components and reconfiguring the system to ignore those, avoiding system-wide disruptions.

Rationale. System parts can have errors and failures, ranging from hardware/software issues to provoked malfunctions (both intentional and unintentional) that can impair the system's operation as a whole. When such problems are detected, and their origin is identified, mechanisms should be triggered that circumvents and isolate the failing part. Suppose that the A/C unit is defective. In that case, the system must not rely on it anymore (until there is a repair intervention) and use alternative ways to achieve the same goal, *cf.* RUNTIME ADAPTION (*e.g.*, during summer, to lower the temperature, opening the home windows can be an acceptable solution). **Also see:** ROLL-FORWARD [31, 60], ERROR CONTAINMENT BARRIER [31], RIDING OVER TRANSIENTS [3, 31], LEAKY BUCKET COUNTER [3, 31], QUARANTINE [31], INPUT GUARD [32], OUTPUT GUARD [32], LOOSE AFFILIATIONS [33], CIRCUIT BREAKER [39], OUTPUT INTERLOCKING [56]

6.12 Flash

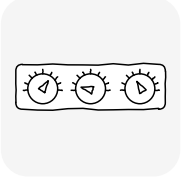


Things can be deployed in remote areas with only periodic maintenance cycles (*e.g.*, every three months). During this time, the devices are exposed not only to environmental conditions, but they also can be stolen or modified by an adversary (*i.e.*, user with malicious intent). In such cases, if the device remains part of the system, it cannot be trusted (since software modifications could have been performed), *cf.* CIRCUMVENT AND ISOLATE. A RESET is not enough if the running software modifications where permanent. How to regain control of the device?

Therefore, flash the device with a trusted software version , remotely (if possible), or by collecting and redeploying the equipment. This can also be known as factory reset or wipe *and* reinstall.

Rationale. Most of the low-cost devices that are part of IoT systems are prone to physical attacks due to the limitations of encryption (mostly resulting from the limited computational power). When a device shows suspicious activity (*cf.* UNSURPRISING ACTIVITY), mechanisms should be triggered to reclaim control of the device, remotely if possible to flash the device with a trusted software version *over-the-air* or physically by collecting and redeploying the device. When recovery is not possible, there should exist a *kill switch* that erases/destroys the device, limiting what the attacker can do to the system. **Also see:** REMOTE LOCK AND WIPE [58], BLACKLIST [58], WHITELIST [58], BUMPLESS UPDATE [26], UPDATEABLE SOFTWARE [26]

6.13 Calibrate



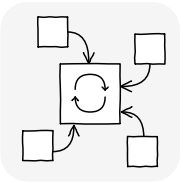
Devices sensors and actuators can deviate from their expected behavior due to decalibrated elements (both in software and hardware). Typically, decalibration errors are consistent, showing up every time a new measurement/action is taken. Regarding sensors, even if they are designed to have high-accuracy, the storage, transport, setup of the devices, along with being subject to heat, cold, shock, humidity, and other nefarious conditions, can lead to

sensor[8] decalibration. We can consider, as an example, that most of the sensors require ADC calibration for accurate readings, and power monitoring chips must be calibrated before being used for measuring the energy consumption [43]. Regarding actuators, as an example, a potentiometer can decalibrate with its usage and lead to unexpected outcomes. How can we ensure the accuracy of the data collected by the sensing devices and the actions carried out by actuating devices?

Therefore, (re)calibrate the device to meet the expected behaviour, by remote or *in-situ* hardware tuning, potentially supported by cooperating sensors.

Rationale. Several sensing devices have calibration requirements that must be realized to make the devices properly function. Further, as time goes by, some of these devices can suffer some decalibration due to several causes, such as usage. As an example, a motion sensor must be calibrated in terms of trigger-periodicity and detection range. If one or both of these configurations are erroneously done or suffer from some decalibration (*e.g.*, malicious modifications), it will cause the device to misfunction, UNSURPRISING ACTIVITY, thus a recalibration is needed. Additionally, we can consider that several communication infrastructure and protocols can require calibration to work properly in the environments they are deployed to (*e.g.*, find the right WiFi channel that is less used can improve communication reliability). **Also see:** ProCal [43], SensorTalk [45]

6.14 Rebuild Internal State



System internal state (partially or as a whole) can suffer from inconsistencies due to the concurrent nature of the IoT systems (*e.g.*, due to multiple ways of interact with the system parts). Refer to INTERNAL COHERENCE for an example on the incoherence that can be introduced by the concurrent inputs of a light system. How can we restore to a stable and coherent system state that reflects the real-world system state?

Therefore, rebuild the internal state of the system to comply with the current system state. INTERNAL COHERENCE probe can trigger this. The system can be restored by querying the existing devices about their state or recover from external state storage.

Rationale. IoT systems are concurrent, with several *inputs* that can come from a wide range of origins (*e.g.*, mobile applications, external APIs or physical device triggers — buttons). However, as the system operates, events (*e.g.*, power-surge) can lead the system to an inconsistent state. In these cases, a CHECKPOINT is not enough since the state of different system parts can be different from the existent checkpoint. Further, there is no need for a RESET. Thus, the system part can rebuild its internal state by observing the current environment or system state. **Also see:** SAFE STATE [26], BOOTSTRAPPER [26], START-UP NEGOTIATION [26]

[8]It is important to note that in a sensing device, the sensor itself is only one component in the measurement system.

7 CONCLUSION

As connected objects (both sensors and actuators) widespread across application scenarios (from domestic applications to industry), these Internet-of-Things systems' dependability has the utmost importance. While several authors address this issue, most of them focus on creating an external system (*e.g.*, auditors and watchers) and rely on those to maintain the system in a healthy state. Thus, there is a lack of a collection of *building blocks* that can be adopted by system developers and integrators to increase their systems' reliability without the need to resort to external solutions.

In this paper, we present a set of 27 patterns and a pattern language that can be used to improve the reliability of IoT systems by enabling them to self-heal. These patterns are organized in two main categories, namely: *error detection* (probes) and *recovery & maintenance of health*; and are mostly based on previous work on related fields such as *cloud computing*, *space systems engineering* and *critical and industrial systems* and are revised under the constraints of the IoT paradigm. Although some of these patterns are not novel *per se*, their contextualization on IoT systems is (at least for the most of them) and imposes new considerations, both from a *fault-tolerance* perspective as well as from a *self-healing* perspective, being our focus the second one. Moreover, these patterns appear in the literature as a disperse, non-systematic way and mostly disregard relationships between them (no pattern language or any other relation is defined).

During this research, we also have potentially identified other patterns that we consider future work. Among those, the ones focused on assessing the resilience of a system. As an example, DRILLS can be used to deliberately provoke failures in a system to check if the maintenance of health & recovery mechanisms is working as supposed (*cf. failure injection* and *chaos engineering*). We also envision future work to evolve further the design and implementation of one or more proofs-of-concept that leverages the pattern language here described [25].

ACKNOWLEDGMENTS

The authors would like to thank Sumit Kalra for helping to review preliminary versions of this work and to EuroPLoP Writer's Workshop members for their insightful feedback. This work was partially funded by the Portuguese Foundation for Science and Technology (FCT), under the research grant SFRH/BD/144612/2019.

REFERENCES

[1] Fardin Abad. 2019. *Safety and Security of Cyber-physical Systems.* Ph.D. Dissertation. University of Illinois. https://doi.org/10.4018/978-1-7998-1482-5.ch016
[2] Fardin Abdi, Rohan Tabish, Matthias Rungger, Majid Zamani, and Marco Caccamo. 2017. Application and system-level software fault tolerance through full system restarts. In *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS).* IEEE, 197–206.
[3] Michael Adams, James Coplien, Robert Gamoke, Robert Hanmer, Fred Keeve, and Keith Nicodemus. 1998. Fault-tolerant telecommunication system patterns. *The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge University Press, New York* (1998), 189–202.
[4] Atul Adya, Paramvir Bahl, Ranveer Chandra, and Lili Qiu. 2004. Architecture and Techniques for Diagnosing Faults in IEEE 802.11 Infrastructure Networks. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking* (Philadelphia, PA, USA) *(MobiCom '04).* Association for Computing Machinery, New York, NY, USA, 30–44.
[5] Alauddin Al-Omary, Ali Othman, Haider M AlSabbagh, and Hussain Al-Rizzo. 2018. Survey of hardware-based security support for IoT/CPS systems. *KnE Engineering* (2018), 52–70.

[6] Rafael Angarita. 2015. Responsible objects: Towards self-healing internet of things applications. *Proceedings - IEEE International Conference on Autonomic Computing, ICAC 2015* (2015), 307–312. https://doi.org/10.1109/ICAC.2015.60

[7] Qazi Mamoon Ashraf and Mohamed Hadi Habaebi. 2015. Introducing autonomy in internet of things. In *14th International Conference on Applied Computer and Applied Computational Science (ACACOS'15)*. ACN.

[8] Algirdas Avizienis. 1997. Toward systematic design of fault-tolerant systems. *Computer* 30, 4 (1997), 51–58. https://doi.org/10.1109/2.585154

[9] Algirdas Avižienis, Jean Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33. https://doi.org/10.1109/TDSC.2004.2

[10] Colin Bell, Richard McWilliam, Alan Purvis, and Ashutosh Tiwari. 2013. Concepts of self-repairing systems. *Measurement and Control* 46, 6 (2013), 176–179.

[11] Terrell R. Bennett, Nicholas Gans, and Roozbeh Jafari. 2017. Data-Driven Synchronization for Internet-of-Things Systems. *ACM Trans. Embed. Comput. Syst.* 16, 3, Article 69 (April 2017), 24 pages. https://doi.org/10.1145/2983627

[12] Gedare Bloom, Bassma Alsulami, Ebelechukwu Nwafor, and Ivan Cibrario Bertolotti. 2018. Design patterns for the industrial Internet of Things. *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS* 2018-June (2018), 1–10. https://doi.org/10.1109/WFCS.2018.8402353

[13] Carlo Alberto Boano, Kay Uwe Römer, Roderick Bloem, Klaus Witrisal, Marcel Carsten Baunach, and Martin Horn. 2016. Dependability for the Internet of Things: From dependable networking in harsh environments to a holistic view on dependability. *e&i - Elektrotechnik und Informationstechnik* 133, 7 (11 11 2016), 304–309. https://doi.org/10.1007/s00502-016-0436-4

[14] Tiago Boldt Sousa. 2020. *Engineering Software for the Cloud: A Pattern Language*. Ph.D. Dissertation. Faculty of Engineering, University of Porto.

[15] Borja Bordel, Ramón Alcarria, Tomás Robles, and Diego Martín. 2017. Cyber–physical systems: Extending pervasive sensing from control theory to the Internet of Things. *Pervasive and Mobile Computing* 40 (2017), 156–184. https://doi.org/10.1016/j.pmcj.2017.06.011

[16] W. G. Bouricius, W. C. Carter, and P. R. Schneider. 1969. Reliability Modeling Techniques for Self-Repairing Computer Systems. In *Proceedings of the 1969 24th National Conference (ACM '69)*. Association for Computing Machinery, New York, NY, USA, 295–309. https://doi.org/10.1145/800195.805940

[17] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. 2008. Self-Healing by Means of Automatic Workarounds. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems* (Leipzig, Germany) *(SEAMS '08)*. Association for Computing Machinery, New York, NY, USA, 17–24.

[18] Tusher Chakraborty, Akshay Uttama Nambi, Ranveer Chandra, Rahul Sharma, Manohar Swaminathan, Zerina Kapetanovic, and Jonathan Appavoo. 2018. Fallcurve: A novel primitive for IoT Fault detection and isolation. *SenSys 2018 - Proceedings of the 16th Conference on Embedded Networked Sensor Systems* (2018), 95–107. https://doi.org/10.1145/3274783.3274853

[19] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41, 3 (2009), 1–58.

[20] Chao Chen and Sumi Helal. 2011. A Device-Centric Approach to a Safer Internet of Things. In *Proceedings of the 2011 International Workshop on Networking and Object Memories for the Internet of Things* (Beijing, China) *(NoME-IoT '11)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2029932.2029934

[21] Liming Chen and Algirdas Avizienis. 1978. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, Vol. 1. 3–9.

[22] Alberto Compagno, Mauro Conti, Cesar Ghali, and Gene Tsudik. 2015. To NACK or Not to NACK? Negative Acknowledgments in Information-Centric Networking. *2015 24th International Conference on Computer Communication and Networks (ICCCN)* (Aug 2015). https://doi.org/10.1109/icccn.2015.7288477

[23] João Carlos Cunha, Antńio Correia, Jorge Henriques, Mário Zenha Rela, and João Gabriel Silva. 2002. Reset-driven fault tolerance. In *European Dependable Computing Conference*. Springer, 102–120.

[24] 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*. 1–8.

[25] João Pedro Dias, Bruno Lima, João Pascoal Faria, André Restivo, and Hugo Sereno Ferreira. 2020. Visual Self-healing Modelling for Reliable Internet-of-Things Systems. In *Computational Science – ICCS 2020*, Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira (Eds.). Springer International Publishing, Cham, 357–370.

[26] Veli-Pekka Eloranta, Johannes Koskinen, Marko Leppänen, and Ville Reijonen. 2014. *Designing distributed control systems: A pattern language approach*. John Wiley & Sons.

[27] Hugo Sereno Ferreira, Tiago Boldt Sousa, and Angelo Martins. 2012. Scalable Integration of Multiple Health Sensor Data for Observing Medical Patterns. In *Cooperative Design, Visualization, and Engineering*, Yuhua Luo (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 78–84.

[28] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. Harris, D. Blaauw, and D. Sylvester. 2012. Bubble Razor: An architecture-independent approach to timing-error detection and correction. In *2012 IEEE International Solid-State Circuits Conference*. 488–490.

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

[30] P. Ganguly. 2016. Selecting the right IoT cloud platform. In *2016 International Conference on Internet of Things and Applications (IOTA)*. 316–320. https://doi.org/10.1109/IOTA.2016.7562744

[31] Robert Hanmer. 2007. *Patterns for Fault Tolerant Software*. Wiley Publishing.

[32] Robert S Hanmer. 2006. Error containment. In *Proceedings of the 2006 conference on Pattern languages of programs - PLoP '06*. ACM Press, New York, New York, USA, 1. https://doi.org/10.1145/1415472.1415493

[33] Robert S. Hanmer. 2014. Patterns for Fault Tolerant Cloud Software. In *Proceedings of the 21st Conference on Pattern Languages of Programs* (Monticello, Illinois) *(PLoP '14)*. The Hillside Group, USA, Article Article 19, 11 pages.

[34] S. A. Hinai and A. V. Singh. 2017. Internet of things: Architecture, security challenges and solutions. In *2017 International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions) (ICTUS)*. 1–4.

[35] Byungseok Kang and Hyunseung Choo. 2018. An experimental study of a reliable IoT gateway. *ICT Express* 4, 3 (2018), 130–133.

[36] Kasem Khalil, Omar Eldash, Ashok Kumar, and Magdy Bayoumi. 2019. Self-healing hardware systems: A review. *Microelectronics Journal* 93, August (2019), 104620. https://doi.org/10.1016/j.mejo.2019.104620

[37] Phil Koopman. 2003. Self-Healing vs. Fault Tolerance. Carnegie Mellon University.

[38] Hermann Kopetz. 2011. *Real-Time Systems*. Springer US, Boston, MA. 307–323 pages. https://doi.org/10.1007/978-1-4419-8237-7 arXiv:96332259

[39] Roland Kuhn and Jamie Allen. 2016. *Reactive Design Patterns (Beta)*. 360 pages.

[40] Xiaozheng Lai, Ting Yang, Zetao Wang, and Peng Chen. 2019. IoT Implementation of kalman filter to improve accuracy of air quality monitoring and prediction. *Applied Sciences* 9, 9 (2019), 1831.

[41] Franck Le, Sihyung Lee, Tina Wong, Hyong S Kim, and Darrell Newcomb. 2006. Minerals: using data mining to detect router misconfigurations. In *Proceedings of the 2006 SIGCOMM workshop on Mining network data*. 293–298.

[42] Rogrio Lemos. 2003. ICSE 2003 WADS Panel: Fault Tolerance and Self-Healing. (08 2003).

[43] Chia-Chi Li and Behnam Dezfouli. 2018. ProCal: A Low-Cost and Programmable Calibration Tool for IoT Devices. In *Internet of Things – ICIOT 2018*, Dimitrios Georgakopoulos and Liang-Jie Zhang (Eds.). Springer International Publishing, Cham, 88–105.

[44] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S. Gunawi, and Shan Lu. 2019. DFix: Automatically Fixing Timing Bugs in Distributed Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 994–1009.

[45] Yi-Bing Lin, Yun-Wei Lin, Jiun-Yi Lin, and Hui-Nien Hung. 2019. SensorTalk: An IoT device failure detection and calibration mechanism for smart farming. *Sensors* 19, 21 (2019), 4788.

[46] Toni Marinucci. 2005. *Characterization and Development of Distributed, Adaptive Real-Time Systems*. Ph.D. Dissertation. Ohio University.

[47] Roberto Minerva, Abyi Biru, and Domenico Rotondi. 2015. Towards a definition of the Internet of Things (IoT). *IEEE Internet Initiative* 1 (2015), 1–86.

[48] Mahyar Tourchi Moghaddam and Henry Muccini. 2019. Fault-Tolerant IoT. 67–84. https://doi.org/10.1007/978-3-030-30856-8_5

[49] Roberto Natella, Domenico Cotroneo, Joao A Duraes, and Henrique S Madeira. 2012. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering* 39, 1 (2012), 80–96.

[50] Kevin Ni, Nithya Ramanathan, Mohamed Nabil Hajj Chehade, Laura Balzano, Sheela Nair, Sadaf Zahedi, Eddie Kohler, Greg Pottie, Mark Hansen, and Mani Srivastava. 2009. Sensor network data fault types. *ACM Transactions on Sensor Networks* 5, 3 (2009), 1–29. https://doi.org/10.1145/1525856.1525863

[51] Pedro Martins Pontes, Bruno Lima, and João Pascoal Faria. 2018. Test patterns for IoT. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 63–66.

[52] Alexander Power and Gerald Kotonya. 2019. Providing Fault Tolerance via Complex Event Processing and Machine Learning for IoT Systems. 1–7. https://doi.org/10.1145/3365871.3365872

[53] Harald Psaier and Schahram Dustdar. 2011. A survey on self-healing systems: Approaches and systems. *Computing (Vienna/New York)* 91, 1 (2011), 43–73.

[54] Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. 2017. Patterns for Things That Fail. In *Proceedings of the 24th Conference on Pattern Languages of Programs* (Vancouver, British Columbia, Canada) *(PLoP '17)*. The Hillside Group, USA, Article Article 7, 10 pages.

[55] D. Ratasich, F. Khalid, F. Geissler, R. Grosu, M. Shafique, and E. Bartocci. 2019. A Roadmap Toward the Resilient Internet of Things for Cyber-Physical Systems. *IEEE Access* 7 (2019), 13260–13283.

[56] Jari Rauhamäki and Seppo Kuikka. 2015. Patterns for Control System Safety. In *Proceedings of the 18th European Conference on Pattern Languages of Program* (Irsee, Germany) *(EuroPLoP '13)*. Association for Computing Machinery, New York, NY, USA, Article 23, 11 pages. https://doi.org/10.1145/2739011.2739034

[57] K. Ravindran and M. Rabby. 2012. Protocol-level reconfigurations for infusion of resilience in distributed network services. In *2012 IEEE Network Operations and Management Symposium*. 1207–1213.

[58] Lukas Reinfurt, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Andreas Riegg. 2016. Internet of Things Patterns. In *Proceedings of the 21st European Conference on Pattern Languages of Programs* (Kaufbeuren, Germany) *(EuroPlop '16)*. Association for Computing Machinery, New York, NY, USA, Article Article 5, 21 pages.

[59] Lukas Reinfurt, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Andreas Riegg. 2016. Internet of things patterns. *Proceedings of the 21st European Conference on Pattern Languages of Programs - EuroPlop '16* c (2016), 1–21. https://doi.org/10.1145/3011784.3011789

[60] Titos Saridakis. 2002. A System of Patterns for Fault Tolerance. In *Proceedings of 2002 EuroPLoP Conference*.

[61] Fabio Sartori, Riccardo Melen, and Fabio Giudici. 2019. IoT Data Validation Using Spatial and Temporal Correlations. In *Research Conference on Metadata and Semantics Research*. Springer, 77–89.

[62] James Scott and Boeing Corporation. 2009. Realizing and Refining Architectural Tactics : Availability. *Solutions* August (2009). https://doi.org/CMU/SEI-2009-TR-006

[63] Sandra Scott-Hayward, Sriram Natarajan, and Sakir Sezer. 2015. A survey of security in software defined networks. *IEEE Communications Surveys & Tutorials* 18, 1 (2015), 623–654.

[64] Sean Smith. 2017. *The Internet of Risky Things*. O'Reilly Media.

[65] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2018. Engineering Software for the Cloud: External Monitoring and Failure Injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. 1–8.

[66] Doug Terry. 2016. Toward a New Approach to IoT Fault Tolerance. *Computer* 49, 8 (2016), 80–83. https://doi.org/10.1109/MC.2016.238

[67] Arijit Ukil, Soma Bandyoapdhyay, Chetanya Puri, and Arpan Pal. 2016. IoT healthcare analytics: The importance of anomaly detection. In *2016 IEEE 30th international conference on advanced information networking and applications (AINA)*. IEEE, 994–997.

[68] Ovidiu Vermesan, Peter Friess, Patrick Guillemin, Sergio Gusmeroli, Harald Sundmaeker, Alessandro Bassi, Ignacio Soler Jubert, Margaretha Mazura, Mark Harrison, Markus Eisenhauer, et al. 2011. Internet of things strategic research roadmap. *Internet of things-global technological and societal trends* 1, 2011 (2011), 9–52.

[69] Torres Wilfredo. 2000. *Software Fault Tolerance: A Tutorial*. Technical Report.

[70] Paul Wood, Heng Zhang, Muhammad-Bilal Siddiqui, and Saurabh Bagchi. 2017. Dependability in Edge Computing. (2017). arXiv:1710.11222

[71] Liwei Yang, Yao Chen, Wei Zuo, Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. 2015. System-level design solutions: Enabling the IoT explosion. In *2015 IEEE 11th International Conference on ASIC (ASICON)*. IEEE, 1–4.

[72] Sen Zhou. 2015. *Supporting Fault Tolerance in the Internet of Things*. Ph.D. Dissertation. University of California.

[73] S. Zhou, K. Lin, J. Na, C. Chuang, and C. Shih. 2015. Supporting Service Adaptation in Fault Tolerant Internet of Things. In *2015 IEEE 8th International Conference on Service-Oriented Computing and Applications (SOCA)*. 65–72. https://doi.org/10.1109/SOCA.2015.38

[74] David Zubrow. 2010. *IEEE Standard Classification for Software Anomalies*. Technical Report. IEEE. 1–23 pages.