IoTRepair: Flexible Fault Handling in Diverse IoT Deployments

MICHAEL NORRIS, Department of Computer Science and Engineering, Penn State University, USA
Z. BERKAY CELIK, Department of Computer Science Purdue University, USA
PRASANNA VENKATESH, SHULIN ZHAO, PATRICK MCDANIEL, ANAND SIVASUBRAMANIAM, and GANG TAN, Department of Computer Science and Engineering, Penn State University, USA

IoT devices can be used to complete a wide array of physical tasks, but due to factors such as low computational resources and distributed physical deployment, they are susceptible to a wide array of faulty behaviors. Many devices deployed in homes, vehicles, industrial sites, and hospitals carry a great risk of damage to property, harm to a person, or breach of security if they behave faultily. We propose a general fault handling system named IoTRepair, which shows promising results for effectiveness with limited latency and power overhead in an IoT environment. IoTRepair dynamically organizes and customizes fault-handling techniques to address the unique problems associated with heterogeneous IoT deployments. We evaluate IoTRepair by creating a physical implementation mirroring a typical home environment to motivate the effectiveness of this system. Our evaluation showed that each of our fault-handling functions could be completed within 100 milliseconds after fault identification, which is a fraction of the time that state-of-the-art fault-identification methods take (measured in minutes). The power overhead is equally small, with the computation and device action consuming less than 30 milliwatts. This evaluation shows that IoTRepair not only can be deployed in a physical system, but offers significant benefits at a low overhead.

CCS Concepts: • **Computer systems organization** → **Reliability**; **Maintainability and maintenance**; *Sensors and actuators;*

Additional Key Words and Phrases: Fault handling, embedded system

ACM Reference format:
Michael Norris, Z. Berkay Celik, Prasanna Venkatesh, Shulin Zhao, Patrick McDaniel, Anand Sivasubramaniam, and Gang Tan. 2022. IoTRepair: Flexible Fault Handling in Diverse IoT Deployments. *ACM Trans. Internet Things* 3, 3, Article 22 (July 2022), 33 pages.
https://doi.org/10.1145/3532194

This work was supported by NSF grants CNS-1900873, CNS-1763681, and CNS-1629915.
Authors' addresses: M. Norris, P. Venkatesh, S. Zhao, P. McDaniel, A. Sivasubramaniam, and G. Tan, Department of Computer Science and Engineering, Penn State University, W209 Westgate Building2, University Park, Pennsylvania, PA, 16801; emails: man5336@psu.edu, {prasanna.rengasamy, zhaoshulin.cn}@gmail.com, mcdaniel@cse.psu.edu, {axs53, gtan}@psu.edu; Z. B. Celik, Department of Computer Science Purdue University, 305 N University St, West Lafayette, Indianapolis, IN, 47907; email: celik.berkay@gmail.com.
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.
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2577-6207/2022/07-ART22 $15.00
https://doi.org/10.1145/3532194

1 INTRODUCTION

In recent years, **Internet of Things (IoT)** devices have grown exponentially in complexity, diversity, and quantity across a wide variety of deployment environments. As the sophistication and abundance of IoT devices continue to increase, the effect they have on everyday life increases in turn. This empowers IoT to increase convenience and efficiency in personal and commercial environments, but also opens the door for potential harm to persons or property should correct operation be disrupted by attackers or system flaws. For this reason, it is increasingly important to improve the dependability of IoT systems to ensure the stability of as many deployments as possible.

A core concern for increasing reliability in IoT is handling faults swiftly and automatically before significant damage is caused to an IoT environment. This is a significant challenge, as IoT devices are prone to faults due to their distributed deployment in physical spaces and limited resources. Possible faults can be caused by factors such as power outages [18] and network disruption [32]; the frequency of faults is also increased due to IoT device design constraints such as low computation capacity [27] and small batteries [13]. Additionally, devices face complex issues, such as disruptive environmental conditions such as weather and collisions [36], user error during deployment in an environment [18], and flaws in the device hardware and software [9]. A study shows that devices in smart homes can experience faults more than four hours a day due to power loss, network disruption, and hardware failure [13]. More recent work shows that a temperature sensor experiences more than 15% faulty temperature readings [32].

The unique requirements for fault tolerance in IoT have been discussed in earlier work [38], and prior efforts have proposed solutions for fault identification [6, 10, 23, 31] and addressing faults [3, 7, 17, 19, 30, 35, 39]. However, previous solutions target a narrow scope (target only certain devices, faults, or environments), and often only notify users about the fault—assuming the user is familiar enough with the deployed devices, installed applications, and fault types to act appropriately. Additionally, the solutions that do perform automatic handling rely on replication, which is not always effective [39] and might be prohibitively costly [38]. Since IoT systems are primarily autonomous, users have less interaction and oversight, which could lengthen the time to resolve the fault manually. The user must recognize the alert and respond to it, but there may not be a user on site. Therefore, an automatic response to faults is desirable to reduce dependency on swift user interventions and increase the reliability of IoT services.

In this article, we develop a flexible multi-layer fault-handling system called IoTRepair specifically designed for IoT. At the lower layer, IoTRepair provides a fault-handling library with a set of functions for handling device faults and a configuration file. To handle diverse faults in IoT, users can write their own configuration files to customize the system and combine the fault-handling functions through the library's API. At the higher layer, IoTRepair includes an automated fault handler on top of the lower layer to handle common situations of IoT faults. The automated handler can be installed onto the edge device of a deployed IoT system to provide autonomous fault handling. A configuration file is generated at installation time through querying the edge device for a list of devices and applications, and also modified at runtime by the fault handler for runtime adaptation; e.g., the fault handler discovers redundant devices at runtime and saves that information in the configuration file.

To evaluate IoTRepair, we implemented 11 distinct IoT apps to manage 17 IoT devices in an Arduino setup designed to mimic the behavior of a sample smart home. We conducted two sets of experiments. In the first, we measured the latency of different fault-handling methods. In the second, we measured the power overhead of IoTRepair's automated fault handler on devices. We injected a comprehensive set of faults into the devices to measure errors that faults would cause,

and how well the fault handler mitigated those errors. In this article, we make the following contributions:

- We study the nature of faults in IoT systems. We show what makes fault behavior in IoT systems different from other environments and why addressing IoT faults is uniquely difficult and requires a generalized solution.
- We design and implement a fault-handling library that implements a common set of functions for handling device faults, such as device restarts, retries, and checkpointing. We provide a flexible API for developers to utilize our fault-handling library in their application code. On top of the fault-handling library, we develop an automated fault handler for IoT. The automated fault handler invokes fault-handling functions in customized orders and configures the functions automatically based on the IoT environment.
- In designing IoTRepair, we propose a set of novel techniques, including a history-based checkpoint/rollback mechanism and a technique for inferring redundant devices according to runtime information.
- We evaluate IoTRepair on a physical setup that mimics a smart home, including 17 devices and 11 IoT apps. We show what harm faults can cause and how IoTRepair mitigates the damage.

A conference version of this article was published at the 5th ACM/IEEE Conference on Internet of Things Design and Implementation (IoTDI 2020) [25]. This article differs from the conference version in the following ways: First, we perform an evaluation on a physical IoT setup detailed in Section 6 to conclusively show IoTRepair's effectiveness; our previous article performed evaluation only in a simulator. Second, we add a power overhead analysis to show that device power consumption, and therefore lifespan, is not significantly impacted by IoTRepair. Finally, we expand the discussion of how IoTRepair is designed and implemented in Sections 3, 5, and 9. Particularly, we describe how IoTRepair can be implemented in any IoT deployment, give detailed descriptions of our fault-handling algorithms, and present detailed examples, both of how faults manifest and cascade through a system and how IoTRepair would mitigate their impact.

2 MOTIVATION AND CHALLENGES

We begin by introducing fault types common in IoT platforms, with an example app to illustrate these fault types. We then motivate the need for a flexible IoT fault-handling system by studying how faults are reported and handled in commercial IoT platforms and existing research literature on fault identification and handling. We end the section with challenges in IoT fault handling.

2.1 Faults in IoT Systems

IoT systems integrate physical processes with digital connectivity. These systems perform simple tasks such as motion-activated light switches, as well as complex tasks such as controlling traffic lights in a smart city. Regardless of purpose and complexity, IoT systems often use an edge device as a centralized gateway that connects devices in physical environments and use a cloud back-end to synchronize device states and provide interfaces to control and monitor devices.

Faults in IoT systems can occur due to flaws in components of devices, the cloud, and communication between them. While faults may happen in any of these components, device failures are more common due to factors such as minimal computational resources, energy constraints, architectural problems, improperly configured systems, and disruptive environmental conditions [17, 18, 24, 27, 31]. For instance, devices in smart homes could experience faults on an average of two hours a day due to power loss, network disruption, and hardware failure [13].

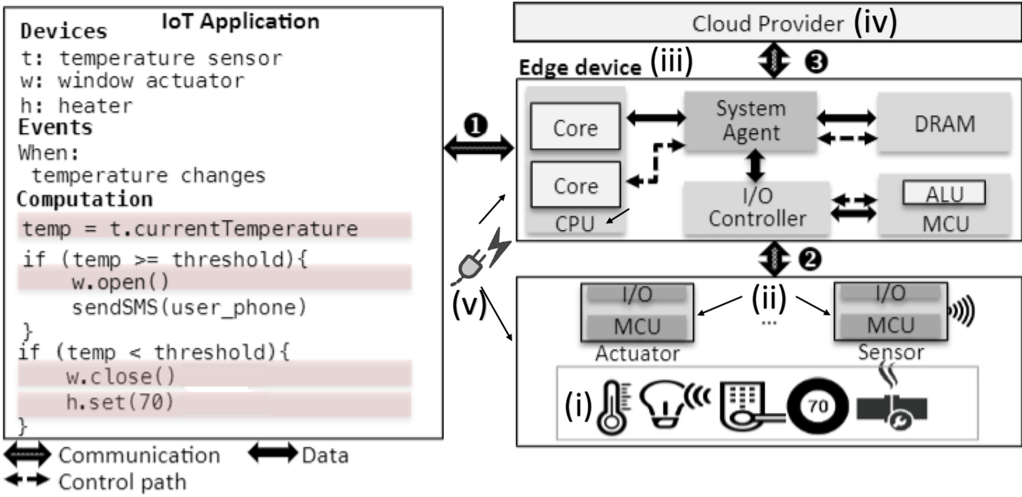

Fig. 1. IoT system architecture and an example IoT app for illustrating fault types.

Table 1. Categorization of IoT Device Fault Types and Causes

Fault Type		Description
Fail stop	Power	A device loses power from battery or outlet and ceases to function
	Communication	A device disconnects from the network or otherwise cannot send or receive packets and ceases to respond requests
	Critical Error	Hardware or software error causes device to cease to function
Non fail-stop	Outlier	A device reports incorrect state for a single poll
	Stuck-at	A device fails to change state when expected to
	High-Variance	A device oscillates between states faster than the environment dictates
	Spike	Numeric device state increases/decreases faster than environment dictates

In a similar vein, experiments in exposed environments recorded half of the devices reporting incorrect states due to severe weather conditions [36].

An IoT system often consists of five components (Figure 1, right): (i) a set of sensors and actuators such as the temperature sensor, light sensor, door lock, thermostat, and leak sensor shown in the figure; (ii) an auxiliary **Micro-Controller Unit (MCU)** to read the raw sensor values; (iii) an edge device with low-power CPU cores and controllers for communicating with sensors, actuators, and the cloud; (iv) a cloud provider for performing advanced computation and interacting with the user; and (v) either a battery or a power supply unit for each sensor/actuator and the edge device to power each of these components. Components (ii) to (v) are often housed in an edge device such as a hub. A fault in any of these components can lead to errors in the system. We will use a `temp-control` app that subscribes to a thermostat, a heater, and window actuators to illustrate the different fault types and their consequences (see Figure 1). The app opens the windows and notifies the user through SMS when the temperature exceeds a user-defined threshold and closes the windows and sets the heater to a user-defined temperature value when the temperature is below the threshold.

We divide faults into two categories, extending the terminology in previous fault tolerance 1 work [6] (see Table 1). Visualizations of the various fault types and the effects they can have on devices are shown in Figure 2. *Fail-stop* faults happen when a device stops functioning and is unresponsive to external requests; for example, when a device loses power, network communication fails; as another example, a software or hardware error halts device operation. A fail-stop fault in the thermostat would cause our example app to halt entirely, and such a fault in the window or the heater would remove the functionality of those actuators.

Non-fail-stop faults relate to a response by the device that diverges from the desired device state. These faults can manifest in a variety of ways with different effects, as shown in Reference [24]: an *outlier* fault causes a device to report an incorrect state for a very brief period, usually the duration of just 1 poll. For example, the temperature reported by a temperature sensor may fall outside of a reasonable range for a single device poll. An unhandled outlier fault in the thermostat could cause the app to send an unintended notification that incorrectly indicates the temperature is above the threshold. A *stuck-at* fault happens when a device cannot change state, maintaining the same state despite changes in environment or actuation commands. An example is if the app opens the window, but the temperature sensor fails to decrease the temperature as it should, which causes a safety issue because an open window enables a burglar to break in the house. Conversely, in commercial buildings with automated doors, a door that is stuck-at closed could cut off access to areas of the building or even trap individuals in certain areas. A stuck-at fault could also cause energy data analytics to be incorrect and home temperature to surge if the heater gets stuck with the always-on state. A *high variance* fault appears when a device's state fluctuates back and forth more rapidly than the environment dictates. For example, the temperature value fluctuates between high and low more than the environment temperature. If this variance crosses the user-defined temperature threshold, then this could cause several issues by repeatedly opening then closing the window and turning the heater on and off. A *spike* fault happens when there is a rapid increase or decrease in the device state; e.g., the temperature decreases faster than the actual temperature in the environment. This could cause safety issues by incorrectly closing the window and turning on the heater, overheating the house—wasting energy and causing discomfort, or possibly even damage if there are temperature-sensitive objects in the environment or the temperature was already very hot. In an industrial setting, a downward spike fault in a temperature sensor for a furnace could cause it to heat to a degree far greater than intended, damaging what is inside and possibly the furnace itself. This could also cause faults in other devices as a now improperly created product is processed as though it were correctly created.

The existing fault tolerance methods discussed in Section 1 and later in this section cannot address this wide array of fault types. Those methods either address only a few specific faults, rely on extensive replication, or cannot handle the effects of cascading behavior.

To evaluate over every possible permutation of these faults would create an impossibly large test size. Therefore, our evaluation in Section 6 targeted only the variables that were consistent across most faults, meaning fault duration and false state, leaving specific elements such as partial fail-stop or the spike change rate as constants. We argue this is acceptable, as these are unlikely to affect the latency of IoTRepair, and have a minor effect on the power consumption.

2.2 Fault Causes

We also divide faults based on what has caused the device to begin faulty behavior. These classifications are important, because they can influence how effective a fault-handling function is, as well as how long it takes to complete. Our fault causes and how they affect system behavior differently are defined below.

A *power failure* occurs when a device no longer has sufficient power to operate, either due to an expended battery or loss of power supply. It completely stops the operation of the device; further, it cannot be repaired by any remote means; so any fault handling other than replication or rollback wastes time and energy.

A *communication disruption* occurs when the hub is either unable to communicate with the device or the communication is unreliable. In the case where communication is impossible, this cause is no different than a power failure, but if communication is unreliable, then normal operation can

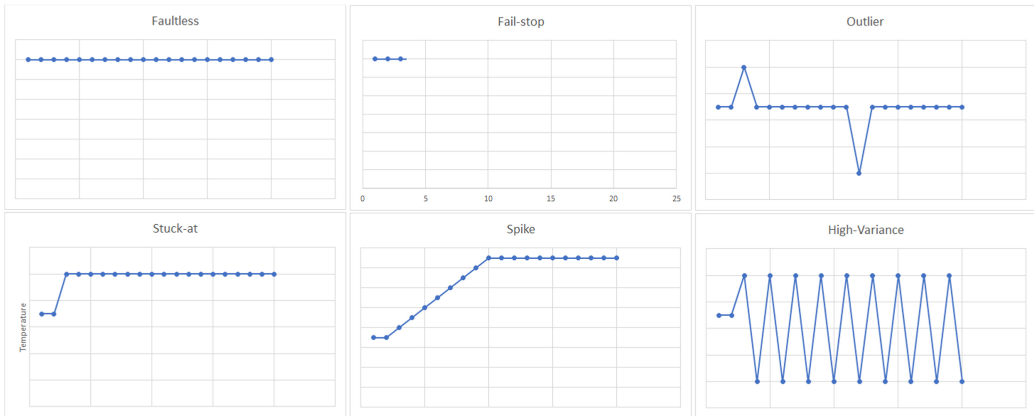

Fig. 2. Visualizing how each fault type would affect the behavior of a device's state.

be approximated by retrying polls and actuations, and it may be possible to repair, as the device can still receive function commands.

A *software error* occurs when a bug in the device's software causes one of the fault types to occur. This can only occur in devices with sufficient software to host bugs, such as a Thermostat, and can most often be repaired by re-initializing the device's software or by power-recycling the device.

A *hardware error* occurs when a defect in the device's hardware causes one of the fault types to occur. This can occur in any device and may be fixable by power-cycling the device one or more times; otherwise, it is no different from a power failure regarding remote fault handling.

As can be seen in the definitions, the cause of a fault mostly affects what form of fault handling may be able to correct the device behavior, if any. Other causes, such as environmental conditions or adversarial attacks, do not add anything significant to this list of causes from IoTRepair's perspective.

2.3 Motivation for Automated Fault Handling

As discussed in the introduction, as IoT systems are primarily autonomous, users generally have little interaction and oversight. Therefore, it is highly desirable to have a largely automated IoT fault identification and handling system.

Fault tolerance in IoT platforms. To understand how IoT platforms identify and handle faults, we have studied eight major IoT platforms to characterize their fault identification and handling methods. The results for each platform are shown in Table 2, with undetected meaning that the fault is not recognized by the platform, silent meaning that the fault is recognized but the system does not inform apps of the fault, general error meaning that an error is thrown but the error does not contain the fault type, and detailed error meaning an error is thrown and it specifies the fault type. For instance, SmartThings provides a developer API to obtain the current device state and can notify an app when the device state is changed [34]; an app can check for the state changes before continuing execution. This requires additional developer effort, and if a developer neglects this check, an app would operate ignorant of whether a device failure exists. Additionally, if a failed device is queried, then SmartThings returns the last known device state or some default value, which may potentially return stale sensor states when a device is unreachable. This enables an app to continue functioning, yet the stale states may inadvertently actuate incorrect device states.

Table 2. IoT Platforms' Response to Different Device Faults

	Fail-Stop Faults			Non-Fail-Stop Faults			
IoT Platform	**Comm.**	**Power**	**Crit Error**	**Outlier**	**Stuck-at**	**High Var.**	**Spike**
SmartThings [29]	⊘	⊘	⊘	⊗	⊗	⊗	⊗
OpenHab [26]	⊘	⊘	⊘	⊗	⊗	⊗	⊗
Vera [40]	⊘	⊘	⊘	⊗	⊗	⊗	⊗
Homekit [2]	⊖	⊖	⊖	⊗	⊗	⊗	⊗
Wink [42]	⊖	⊖	⊖	⊗	⊗	⊗	⊗
AndroidThings [11]	⊕	⊕	⊕	⊗	⊗	⊗	⊗
IoTivity [14]	⊖	⊖	⊖	⊗	⊗	⊗	⊗
KaaIoT [37]	⊖	⊖	⊖	⊗	⊗	⊗	⊗

[⊗]: Undetected, [⊘]: Silent, [⊖]: Generic Error, and [⊕]: Detailed Error.
Undetected means that the fault is not recognized by the platform, *silent* means no message sent to apps, *generic error* means messages do not contain fault information, and *detailed* error means messages contain fault information.

The results in the table show that only Android Things gives enough information to handle fail-stop faults effectively, and no platforms give the means for applications to handle non-fail-stop faults. In contrast, IoTRepair is able to handle these types of faults automatically and further provides a toolset to allow developers on any platform to perform their own handling.

Research Literature on IoT Fault Identification and Handling. Fault identification techniques aim to determine the presence of a fault and determine the faulty device and type. We characterize the fault identification techniques for IoT systems into three groups. First, network traffic-based techniques analyze sensor data packets to detect faults [28, 44]. Network-level techniques are effective at identifying fail-stop failures but ignore non-fail-stop faults. Second, redundant sensor-based techniques use data from multiple homogeneous sensors and exploit the fact that spatial sensing of close sensors should yield similar sensor states [5, 10, 31]. However, these systems are costly due to the requirement of multiple sensors and, more importantly, ineffective when considering that simultaneous sensor faults are common in IoT. In the last group, data obtained from different types of sensors are used for fault identification. These techniques apply to environments such as smart homes where various sensors states are available and often correlated [17, 19, 23]. While these techniques differ in scope, precision, and methodology, they do not identify fault types.

Very little research has been performed in IoT fault handling. The explored techniques often aim at specific environments and faults. A fault-tolerant technique [39] for redundancy-free UAV sensors uses imperfect replication to allow near-correct device operation in the presence of a sensor failure. While this work poses a good argument for the value of imperfect replication, the functions are specific to UAV sensors and contribute little to repairing faults. Rivulet [3] aims at removing an edge device as a single point of failure to mitigate connection faults by distributing communication to devices. The core drawback of this technique is that it does not handle the far more common sensor and actuator faults. Transactuations is a fault-handling technique introduced in Reference [30] to address some flaws that transactions have in IoT environments. This enhanced technique better prevents physical device states from losing synchronization with software variables. While this is useful for preserving dependencies, it does not repair faults and cannot correct errors that occur before the fault is detected. To the best of our knowledge, no current techniques develop a fault handler for diverse IoT device failures and bundle fault identification and fault handling to mitigate faults. CEFIoT [15] and its extension IoTEF [16] use a fault tolerance model designed for distributed systems with the goal of ensuring data processing

and delivery. While it is very effective at increasing the resilience of systems with advanced distributed computation, it assumes the collected data is correct at the source, which may not be true in the case of a faulty device, nor the cascading effects that can cause. The system also does not attempt to repair faulty devices actions or consider the changing state of an IoT environment.

Challenges in IoT Fault Handling. Compared to the fault handling in other computing platforms such as distributed systems and cloud services, fault handling in IoT systems raises a set of unique challenges. We present these challenges below and discuss briefly how we address each.

First, there is no single fault handler technique that can address all possible device faults, due to the vast array of heterogeneous IoT devices and the diverse set of environments in which they are deployed. Implementing an optimal handling technique for a specific fault is highly contextual—one cannot define the impact or correctness of fault handling without understanding the environment. Due to the heterogeneity of IoT devices, various devices often require different fault-handling techniques executed to fit power, environmental, and computation constraints. IoTRepair addresses this challenge by introducing a set of functions that may be invoked in a flexible order with custom parameters through device-driven defined schemes. These schemes can be chosen to meet the system and environmental requirements such as safety, security, and availability. We present the fault-handling functions and schemes in Section 4.

Second, optimal handling of a specific fault is highly contextual due to varying user and developer demands and environmental factors. For instance, some users may prefer energy conservation over real-time fault handling, and a developer may desire their apps suspended when a fault occurs. IoTRepair enables users and developers to define their requirements in a configuration file, which is updated by runtime environmental input.

Third, due to dynamic application code across deployments, faulty behavior can cascade throughout the environment unpredictably. This cascading behavior causes a faulty device in an app to incorrectly trigger an event in another app. To illustrate, we consider a `secure-home` app co-located with the `temp-control` app. The `secure-home` app controls a presence sensor, a door lock, and a window actuator. The app keeps home secure—door-locked and window-closed—when the user is not at home and notifies the user if the house becomes insecure. An unhandled spike fault in the thermostat controlled by the `temp-control` app causes an increase in temperature when the user is not at home and causes the window to open. The `secure-home` app then sends an SMS alert to the user saying that the house is unsafe—making the user become unnecessarily anxious. This behavior is correct from the `secure-home` app, as the window should not be open; however, the window is opened because of a fault that triggers the event handler of the `temp-control` app. This example shows that faults of a faulty device can propagate beyond the device through influencing the environment and interactions between apps. This requires fault tolerance that can either prevent or remedy these behaviors.

Finally, a fault-handling mechanism at installation and runtime must require minimum user interaction and domain expertise. For instance, a solution that only notifies users about the fault may lead to safety issues, as the users might not be available, and the fault may require a real-time response. IoTRepair provides automated fault-handling functionality with user configurations.

3 APPROACH OVERVIEW

The variety of faults in IoT systems, combined with the diversity of IoT deployments, requires a flexible fault-handling approach. It is important that users can customize a fault-handling system to address a variety of faulty devices, fault types, deployment environments, and their preferences. For this reason, we design IoTRepair to be flexible, with major components presented in Figure 3.

One design decision of IoTRepair is that it focuses entirely on the fault-handling aspect and assumes there is already a fault-identification module that can accurately identify the faults in a

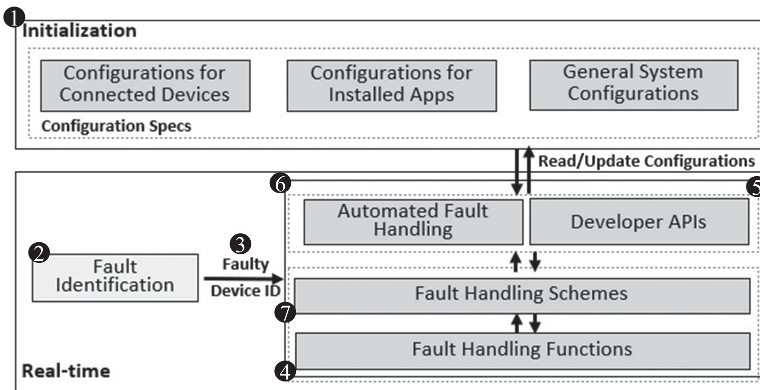

Fig. 3. Overview of IoTRepair architecture.

timely manner. At runtime, the fault identification module sends a signal with the faulty device ID (and optionally the fault type) to IoTRepair's fault handler when it detects a fault. The signal could be in the form of a network packet or an API call; in our implementation the identification module simply imports the IoTRepair library and performs a function call. Each message should only identify one fault, as IoTRepair assumes faults are independent. Multiple faults are considered in design and implementation, however, the number of faults only affects the performance of environmental functions, as increasing faults reduce the ability to accurately gauge the correct system state. We also consider that the identification module may have varying amounts of delay in detecting faults, but our evaluation in Section 6 shows that the delay does not have an impact in IoTRepair's effectiveness beyond the latency of the delay itself.

With that design choice, IoTRepair first provides a menu of fault-handling functions as a library so developers can invoke these functions to mitigate or repair the faults. For instance, it provides a *retry* function, which a developer can invoke to repeat some functionality of a device if a previous use of that functionality returned a fault. This is useful for resolving transient faults.

Second, IoTRepair provides an automated fault handler, which takes a configuration file (discussed soon) and performs fault handling without user involvement. IoTRepair introduces a notion of *fault-handling schemes* to organize and execute fault-handling functions autonomously. A scheme is a list of fault-handling functions to be invoked in a specific order. Each device is assigned a scheme in the configuration file that determines which fault-handling functions will be used to handle faults in that device and what order they will be called in.

Finally, the system includes a configuration file, which is created during an initialization phase and updated by the automated handler at runtime. During the initialization phase, our system requests information from the loaded identification module and obtains a list of installed applications and connected devices, their types, and capabilities; e.g., a motion sensor has active and inactive states and a hardware restart functionality. In detail, there are three types of information in the configuration file: *(i)* general parameters, defining the upper bound on how long fault identification takes defined by the identification module, as well as specifications for checkpoint cleanup and replicated device detection, specified by the user; *(ii)* device-based parameters, defining the list of devices running in the system and a set of parameters (e.g., which fault-handling scheme) for each device; a list of default configurations for various device types is also defined; each device has its default set based on the device's type, or to a conservative default; these settings can be modified by the user, by the automated handler at runtime, or by developers via making API calls in their applications; *(iii)* application parameters, listing what apps are installed on the edge device

and whether the application suppression is enabled for each; by default, application suppression is disabled. These components allow IoTRepair to be flexible in addressing a variety of faults and environments.

The above discussion implies the two modes through which IoTRepair can be used.

Developer API. In this mode, a developer can customize and use IoTRepair through the fault-handling library. The library provides an API for its functions (discussed later). An application developer can utilize the API in two ways. The simplest is to use a set of auxiliary functions to modify the configuration file to customize the automated handler. The second is for the application to call fault-handling functions in the library directly. For our current implementation in Python, installing the library is as simple as importing our Python file. For an application that is not in Python, it can interact with our library through Python's foreign function interface. For faults that cause application code to return errors, a developer can simply use a try/catch block around application code that interacts with devices. For any caught error, the application can call handler functions in conjunction with their own code whenever a fault occurs in a device. This allows them to handle faults flexibly, although it requires more effort on the developer part, as they have to provide all arguments for each call. For faults that do not cause application code to return errors, a developer would need to incorporate a fault identification module and call specific IoTRepair functions on fault identification; for these faults, it is far easier to simply use the configuration file for automated fault handling.

Automated Fault Handling. For Automated Fault Handling, IoTRepair assumes there is a centralized control point where IoTRepair can be installed, which can be an edge device located in the deployment or a cloud service. IoTRepair also assumes this is the only entity other than the devices themselves that can modify device states. While these assumptions eliminate environments with distributed control, centralized control is common in many IoT environments, whether they are smart homes, industrial control systems, hospitals, or otherwise. Once a faulty device is identified, its automated handler performs device suppression, which blocks polls to the faulty device and events generated by faulty devices. This prevents faulty devices from triggering application code and causing incorrect actuations. The handler then uses the configuration file to see which scheme should be used for handling the fault for the faulty device. The handler calls fault-handling functions in the order specified in the scheme, and each function uses the information in the configuration file while attempting to handle the fault. If the fault is repaired at any stage of the scheme, then fault handling immediately ends, and the device is added back for normal execution. If the scheme ends without the device being repaired, then the user is notified that the device must be manually repaired, and the handling ends. Using the automated fault handler in our current setup requires importing our Python file, creating a configuration file, inserting function calls for routinely taking checkpoints, and inserting a function call into the fault identification module for when a fault has been identified. In our implementation, we inserted checkpoint calls at the end of a cycle where device states were polled and any relevant application code was run, which we called a poll-cycle, but this may be different for different environments, such as SmartThings where activity is event-driven not poll-driven.

4 FAULT-HANDLING FUNCTIONS

We introduce a set of *functions* to handle faults effectively. We present these functions in three groups, presented in Table 3. We then discuss four built-in fault-handling schemes that can be integrated into diverse IoT environments.

Design Requirements and Assumptions. We present the requirements for IoTRepair to operate effectively. First, some fault-handling functions, such as rollback and checkpointing, require

Table 3. Fault-handling Functions Prototypes

Device-based Functions
`bool activate_redundant_device (String device_ID)`
`int retry (String device_ID, FP verifyFunc†, String[][] expectedValues†, bool isFailstop†)`
`bool device_software_restart (String device_ID))`
`bool device_hardware_restart (String device_ID)`
`void notifyUser (String device_ID)`
Environment-based Functions
`bool checkpoint (String[] device_values)`
`int rollback (String device_ID)`
`int transaction (String[][] actuations)`
Auxiliary Functions
`bool AddDevice (String[] device_ID)`
`bool RemoveDevice (String[] device_ID)`
`bool UpdateDeviceConfig (String[] device_ID, ConfigOptions‡)`
`bool UpdateAppConfig (ConfigOptions‡)`

([†] Marks optional arguments. [‡] ConfigOptions represents a series of arguments for all configuration parameters for the device/application.)

access to the device states; thus, we assume that device states are available through polling. Second, IoTRepair relies on the edge device (e.g., the hub) being able to query or record the ID and model of all connected devices. Third, we assume that there is a fault-identification module that provides the faulty device ID for each detected fault. Any system that can provide the faulty device ID can be integrated into IoTRepair. Finally, we assume that our implementation is injected into the edge device; it can access the aforementioned capabilities and its library functions are exposed to applications.

4.1 Device-based Functions

Device-based functions are an implementation of *isolated* fault handling, which considers only the state of the faulty device, but not the overall state of the system. The effectiveness of such a function depends only on the specifications of the faulty device and the nature of the fault. These functions do not directly affect and are not affected by any other devices in the system.

Activate Redundant Devices. Functional replication is a well-studied technique in many domains, with applications in cloud environments, distributed systems, and even some IoT implementations. We argue that IoT devices can provide replication in two distinct ways: (a) a duplicate of the primary device; (b) a device that is not a complete duplicate but provides similar capabilities; for example, a camera that is located in the same room as a fault motion sensor can detect motion to replicate the functionality of the motion sensor. IoTRepair currently can only automatically implement replication of type (a), but type (b) is a quite reasonable future addition, as the feasibility was demonstrated in earlier work [39].

The user can use the configuration file to specify any known replicating devices of type (a). IoTRepair's API function `ActivateRedundantDevice` allows the system to continue to run unaffected by a detected fault, so long as the configuration file specifies the replicating device ID if one exists. If there is, then it redirects polls and actuation commands from the faulty device to the replicating device. Using this method, IoTRepair does not rely on native replication support from

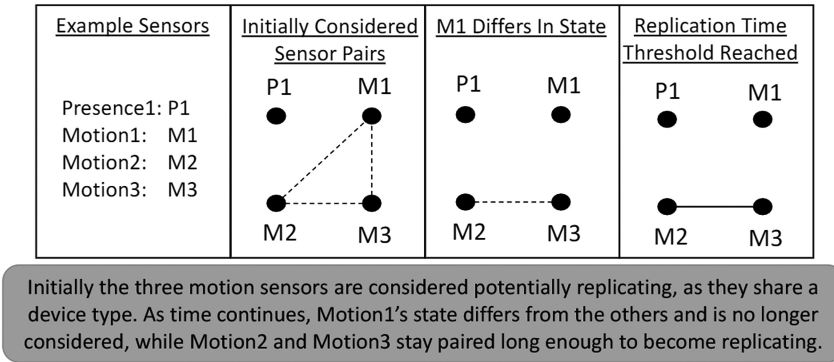

Fig. 4. Example of automatic replication detection.

the platform. This also places all replication logic in the cloud, which keeps latency and overhead to a minimum.

To ease the burden of writing replicating devices in the configuration file, IoTRepair's runtime includes a method to automatically detect likely candidates for replication, similar to the finger-printing method in Reference [12]. It examines the list of connected devices through platform capabilities for devices with matching type and automatically generates relevant configuration data. Finding a replicate device in an IoT environment must consider the fact that similar devices are not necessarily co-located. To address this, IoTRepair observes the sensor states during polling for a configured time and checks whether devices report the same states consistently. If the two devices' states and transition timings remain within acceptable bounds, then these devices are identified to be replicating. Figure 4 illustrates how two co-located motion sensors would be detected as replicating, as their states match consistently. In contrast, an unrelated presence sensor and distant motion sensor are not treated as duplicates.

Retry Device State. `Retry` is effective at handling short, transient faults in devices, or faults that only incur a *chance* of failure. The first purpose of Retry is stalling device execution long enough to let the fault resolve itself and preventing excessive handling from being performed. This can resolve very short faults, such as outliers, that only take place over a few milliseconds. The second is that, if the device is an actuator or the proper state of a device is known, then through repeated polling and/or actuating the fault may be resolved. `Retry` takes parameters of the device ID and three optional arguments: a function pointer for a validation function, a list of expected device states, and whether the fault is fail-stop or not. When used by the automated handler, these may be supplied by the fault identification module. Additionally, each device is configured with a set number of retries. If a function pointer to a validation function is given, then the validation function will be called to check if the fault is resolved (until the retry limit is reached). If the expected device states are passed, then the retry will poll and/or actuate the device continually until an expected state is returned or retry limit is reached. If an expected value is returned, then it will mark the fault as resolved and end fault handling. If the fault is identified as fail-stop, then any successfully returned state will be treated as an expected value.

Software and Hardware Restart. Restarting can potentially correct both software faults, where the onboard software on the device encounters a bug, or hardware faults, where the hardware enters a bad state, so long as it can be remedied by a power-cycle. `SoftwareRestart` and `HardwareRestart` take a device ID and send a signal to the faulty device to initiate a software and hardware restart, respectively. The configuration file will also be read for the device to

determine the max number of attempts to restart. If no acknowledgment signal is received, then the command will be repeated, and if it is not acknowledged a configured number of times in a row, then the restart will be aborted and return with an error code. If the restart is acknowledged, if the fault was not fail-stop, then the function will disable the app and device suppression for the passed device and then wait for a length of time equal to the upper bound on fault detection for the loaded fault identification module. If no fault is detected during this time, then the restart resolves the fault and fault handling ends. If a fault is detected or the fault was fail-stop, then the restart function ends unsuccessfully.

The implementation of restart functions is device-dependent. Many IoT devices such as Honeywell Z-Wave Thermostat [33] and open-source electronics platforms such as Arduino implement the device restart functionality.

User Notification. `Notify` takes a device ID and notifies users of the fault and the result of fault handling. The implementation of the notify call is platform-dependent; we inform users through alert messages on the console, yet text messages or push notifications are possible through IoT platform APIs. Event triggers for user notification can be configured for each device to notify the user when a fault cannot be repaired by the handler, when a fault has been repaired by the handler, when a fault occurs, or any combination of these.

4.2 Environment-based Functions

Environment-based functions are an implementation of *linked fault handling*, which mitigates cascading behavior. It responds to a fault by considering the states of all devices in an IoT environment; e.g., linked fault handling should close the window that was incorrectly opened by an app due to the faulty thermostat. As environment-based functions can affect all device states in an environment, they are capable of mitigating cascading failures across the system.

Checkpoint. `Checkpoint` stores all device states in the system at the time it is called. In general, a checkpoint is a sequence of pairs of devices and device states. For example, in our Arduino deployment each device either has a digital or analog value for its current state. So, a system with a presence sensor, motion sensor, and light may have a Checkpoint that looks like ["Presence": 1, "Motion": 0, "Light": 1], which means Presence is detected, Motion is detected, and Light is on. We divide device states into *sensor* states and *actuator* states. Sensor states are read-only states that collectively represent the state of the environment and drive behavior. Actuator states can be read and modified through actuation, and they are the states that can be rolled back.

We assume that there is a causal relation from the sensor to actuator states; in other words, an app follows the well-known sensor-computation-actuator structure: It takes the sensor states, performs some computation, and then performs actuation to modify the actuator states. Based on this, IoTRepair includes a novel, history-based checkpoint/rollback mechanism (1) that during checkpointing records the history of device states, and (2) that during rolling back restores the most likely actuator states by looking up the history according to the current sensor states. We next discuss this mechanism in detail.

First, the automated fault handler invokes the checkpoint function after every actuation that does not trigger cascading actuations. This means a checkpoint is taken after actuations where no subscribing application initiates an actuation based on the new state. The handler determines these states by checking all apps logic subscribed to the actuated device to ensure no additional actuation is performed as a result.

Second, we must ensure no faults were present at the time a checkpoint was taken. We must wait until a period equal to the upper bound on the fault-detection time has passed without a fault being identified to verify this. If the fault-identification module does not have a known upper bound, then we wait for a user-defined length of time, with a default value of 30 minutes.

Timestamp	System Device States		System Checkpoints			
	Motion	Light	Timestamp	Frequency	Motion	Light
t_0	On	On	N/A	N/A	N/A	N/A
t_1	Off	Off	t_1	1	Off	Off
t_2	On	On	t_2	1	On	On
			t_1	1	Off	Off
t_3	Off	Off	t_2	1	On	On
			t_3	2	Off	Off
t_4	Off	On	t_2	1	On	Off
			t_4	1	Off	On

Fig. 5. Illustration of checkpointing.

When enough time has passed, the new checkpoint is then stored in a history log, which holds checkpoints and their frequencies. If no previous checkpoint has matching sensor states, then a new checkpoint is appended to the log. This Checkpoint will be assigned a frequency for tracking how often it has occurred in the system, with an initial value of 1. This value acts as a tie-breaker when choosing between possible target Checkpoints. If the sensor states match an existing checkpoint's sensor states, then the frequency is incremented if actuator states also match, or the actuator states are overwritten, and the frequency is reset to be one if the actuator states differ. Only storing the most recent actuator states for a set of sensor states keeps the log from exploding in size. For the same reason, checkpoints are removed from history if they remain unused for an extended time, as determined by the configuration. If computation and storage space is relatively high, such as on a cloud server, then we could store all possible actuator configurations for each sensor set. In this case, actuator states could also be used when deciding between Checkpoints, where the Checkpoint that would require less actuations to match is selected. This would allow for causal relationships between actuators to be included in rollback.

Figure 5 provides an example of how checkpoints are taken over time in a system as a result of changes in actuator states. The example uses a simple system with two devices: a motion sensor and a light actuator. The figure shows checkpoints being taken during a series of actuations, which occur at timestamps t_1 through t_4, with initial time t_0 before actuations. On the left side of the figure, the state of the system after the actuation completes is shown; on the right is the list of checkpoints at each time, starting from an empty set. The first two checkpoints at times t_1 and t_2 are new states, since there are no existing checkpoints that match the sensor states. For these new checkpoints, the time and current device states are recorded, and the frequency is set to 1. The checkpoint taken at t_3 has the same device states as an existing checkpoint; so it updates the checkpoint timestamp and the frequency. At time t_4, a checkpoint that matches the sensor states is taken, but the actuator states do not match those of an existing checkpoint; so the matching checkpoint's timestamp is updated, actuator states are overwritten, and frequency is reset.

Rollback. `Rollback` performs a series of actuations to match the system state to the checkpoint that best reflects the current system sensor values. Algorithm 1 details the steps for computing a *rollback* that finds all potential matches, determines the best match, and performs actuation according to the best match.

We have designed three techniques for choosing the best checkpoint to target for rollback: most-recent, fail-norm, and fail-safe. The configuration for the faulty device determines which technique will be used. Lines 1 and 2 of the algorithm involve retrieving which of these to use based on the faulty device.

ALGORITHM 1: Rollback Algorithm

Input : The *ID* of the device that triggered this rollback
Output : Success or Failure

1. Read configuration file to get *rollbackType*
2. Set *targetCriteria* be timestamp if *rollbackType* = most-recent, be current sensor states if *rollbackType* = fail-norm, be fail-safe states if *rollbackType* = fail-safe
3. Search through *Checkpoints* to find checkpoint that fits *targetCriteria* and store it in *bestMatch*
4. **if** *bestMatch* equals ∅ **then**
5.   return Failure
6. **if** any *device* in *systemState* is an actuator, its state does not match *bestMatch*, and is faulty **then**
7.   return Failure
8. Actuate each actuator in *systemState* to match *bestMatch*
9. Change each faulty sensor state in *systemState* to match *bestMatch*
10. return Success

Most Recent targets the last checkpoint that occurred in the system. This is the most basic method and assumes that faults can be detected very quickly relative to the speed that the environment typically changes. This assumption means that most if not all of the recent changes in the environment should be due to the faulty behavior; so they can safely be rolled back to allow the system to continue from a recent safe point. A good example would be in an environment where detecting faults has a high priority, like a hospital. It also works well with complex devices that are more likely to be able to self-diagnose and report faults. Most recent is also good in environments where it is assumed many devices are likely to have faults at once. This is because other rollback methods decrease in accuracy as more faults occur and decrease what can be known about the environment state. An example environment where this would be the case is one that spreads across many networks, where one node going down may result in many devices suffering a network failure.

Fail-norm finds which checkpoint best matches the current known environment. First, only checkpoints where all sensor states match the current sensor states in the system are considered. As the states of faulty sensors cannot be trusted, their sensor states are not considered when determining if a checkpoint matches the current sensor states. Because of this, it is possible that multiple checkpoints match the current known state of non-faulty sensors. If there is more than one match, then the frequency of checkpoints is used as a tie-breaker. As previously mentioned, this method assumes a causal relationship from sensors to actuators, and also assumes a large majority of devices are non-faulty at rollback time. Fail-norm is best suited to devices whose states have a high degree of correlation with other devices. For example, a motion sensor's state may be highly correlated with the state of a presence sensor, which allows the rollback to use the presence sensor to determine what the faulty motion sensor's state should be. It is also better in environments without single points of failure, such as a factory with a backup generator for power failure.

Fail-safe uses the fail-safe configurations for all of the actuators in the system to reduce the list of checkpoints to only those where actuators match their fail-safe states; e.g., the fail-safe configuration for an alarm might be to have it on. Once the list has been reduced, a checkpoint will be selected in the same manner as fail-norm. This is the only method that will proceed with the rollback even if no valid checkpoint is found or one of the devices that must be changed is faulty. Even in these cases, the bare minimum is to put all devices with fail-safe states into those states. Similar to fail-norm, fail-safe assumes devices are correlated for selecting checkpoints, but more importantly assumes that there are devices that have known safe states that are critical for

Current System State:

Motion	Presence	Door Lock
Off	Faulty (stuck at home)	Unlocked

History of Checkpoints:

Frequency	Motion	Presence	Door Lock
30	On	Home	Unlocked
2	On	Away	Locked
3	Off	Home	Unlocked
50	Off	Away	Locked

Bottom two checkpoints match the current motion sensor state, and the second checkpoint has a higher frequency, so rollback will lock the door.

Fig. 6. Example of fail-norm rollback.

ALGORITHM 2: Transaction Algorithm

```
Input  : An ActuationList of
         (ActuatorID, ActuationValue) tuples
Output : Success or Failure return code
1 undoLogs ← ∅
2 foreach tuple in ActuationList do
3     originalValue ← getDeviceState(tuple[0])
4     if actuate(tuple[0],tuple[1]) returns Failure
      then
5         foreach actuation in undoLog do
6             actuate(actuation[0], actuation[1])
7         return Failure
8     undoLog.append(tuple[0], originalValue)
9 return Success
```

them to enter if their proper state becomes ambiguous. The best example for this is when the faulty device drives the behavior of many important devices. In a commercial building with automatic doors, a faulty fire alarm may mean that all interior doors must open to avoid possibly preventing individuals from escaping a possible fire. A similar case would be a smart home that closes and locks all entrances if the presence sensor becomes faulty to secure the home while the location of the resident is unknown.

Line 3 of the algorithm is where a rollback technique is used to iterate through the list of previously taken checkpoints to find the best match for the rollback type. Most Recent will choose the checkpoint with the latest timestamp. Fail-norm finds the checkpoint that matches all known current sensor states best, using how frequently it has occurred as a tie-breaker. Fail-safe is the same is fail-norm, but will choose only checkpoints where all devices with fail-safe states in the configuration file match those states. If there are no matches, then the rollback fails and terminates.

Rollbacks can be dangerous, as a partial rollback can result in an otherwise impossible transition and enter an invalid system state. For this reason, our rollback fails if any actuator that needs to be changed is currently faulty (lines 6–7). We note that this captures the heterogeneous nature of IoT devices compared to similar functions in distributed systems [21] and cyber-physical systems [20]. We further note that as IoTRepair assumes that the identification module does not provide exactly when the fault occurred in the device, all actuators that do not match the chosen checkpoint must be actuated. If the fault time were provided, then it would be possible to rollback only the effects of the fault; we discuss a possible extension in Section 7. We also implement a data rollback as part of the system rollback. Specifically, as long as the rollback does not fail, any faulty sensors are set to their values at the time of the checkpoint until the device is repaired or another rollback occurs (Line 8).

Figure 6 gives an example of how fail-norm rollback would operate in a system with two sensors, motion and presence, and a lock actuator for a door. It shows that IoTRepair's rollback can mitigate dangerous faults that would otherwise persist until the user can repair the faulty device, as long as another sensor's state is correlated with the faulty sensor's state with high frequency. In the example of Figure 6, a presence sensor that is stuck at *home* could cause the door to remain unlocked indefinitely. Fortunately, the motion sensor's state is largely correlated with the presence sensor's state, because it is likely to detect user motion when the user is at home. As a result, IoTRepair's rollback can then use the motion sensor's state to correct the door to be locked and secure the home, even when the presence sensor is faulty.

Transaction. A transaction performs a series of actuations and ensures that the actuations are performed atomically as a group, either all successfully completing or none completing. Currently, the `transaction` function is not used by IotRepair's automated handler, yet app developers can invoke it to cause a series of actuations to execute atomically when a partially executed series may leave the system in an unsafe state. For instance, a developer can issue a call "`transaction([[Window, Closed], [Heater, On]])`" to close the Windows and turn on the heater as a transaction. Transactions are implemented through a standard undo-log-based algorithm. It performs each of these actuations in the order given in the passed array while recording the original states in an undo log. If any actuation fails, then the transaction is aborted, all devices are reset to their previous states, and an error is returned.

4.3 Auxiliary Functions

We describe functions necessary for the operation of the fault handler. These functions are not directly used during fault handling but can be invoked by the automated handler or application developers to modify handler configuration.

Device Suppression. It is not a library function, but rather a capability injected into the code of an edge device (e.g., a hub). This is because it is required to interrupt polls and event hooks at the edge device. Through this capability, the edge device can terminate polling the device's state and block actuation commands sent to the device. This prevents events from a faulty device triggering incorrect actuations to other devices through app code. Device suppression is effective in correcting non-fail-stop faults. For example, if the smoke detector experiences a high variance fault, then it may rapidly switch between smoke-detected and smoke-not-detected states. This would cause the alarm to turn on and off until the fault is resolved. With device suppression, the smoke detector can be suppressed once it is identified as faulty.

Application Suppression. Similar to device suppression, application suppression is also a capability injected into the edge device code. Application suppression can be enabled or disabled for each app based on the configuration file. When a device is identified as faulty, application suppression checks the list of apps subscribed to the device's events. For every app that application suppression is enabled, all events that would be sent to the app and all commands generated by the app are suppressed. This halts the execution of the app, which is desirable when an app has functionality that may put the system into an unsafe state if the state of one of its dependent devices is unknown. For example, suppose there is an app that opens the window when the presence sensor indicates the user is home and the house is too hot; when the presence sensor is faulty, the window may open if the app is not suppressed.

Update Device/App Configuration. These functions allow IoTRepair to update the configuration file at runtime by passing a device or application name and a list of fields to update. If the passed arguments are valid for the given fields, then the values in the configuration file are updated. App developers can also call this function with the same parameters to customize the configuration file for their requirements.

4.4 Built-in Fault-handling Schemes

We introduce a set of built-in schemes to automate the functions introduced above (see Table 4). The schemes control the execution order of the functions to address environmental requirements such as safety and security. We choose to use the device type as a primary determinant of which scheme to use, since IoT device parameters introduce several limitations, e.g., duration of restarts and the presence of replicated devices, which impacts the optimal function ordering. Our four schemes are not designed to address all possible deployments, and additional schemes can easily

Table 4. The Execution Order of Functions in Schemes: (1) Replicate, (2) Retry, (3) Software Restart, (4) Hardware Restart, (5) Rollback, (6) Notify User

Scheme	Function Ordering					
Conservative	1	2	3	4	5	6
Permanent-Expected	1	3	4	5	6	∅
Long-Restart	1	2	5	3	4	6
Time-Sensitive	1	5	2	3	4	6

be created for different deployment requirements and when new handling functions are developed. IoTRepair updates selected schemes based on device behavior during fault handling. We describe the purpose of each scheme. The *Conservative* scheme fits in environments where there are no strict time requirements, and energy conservation is the primary goal. For this reason, the functions are ordered by increasing cost of power to the device. The *Permanent-expected* scheme aims at devices that are unlikely to experience transient faults. It is similar to Conservative, but it skips over Retry, as Retry can only resolve transient faults. For instance, this would be suitable for a temperature sensor deployed in a stable home environment, since it is capable of restarting and does not control time-sensitive operations. The *Long-restart* scheme is used in devices that have excessively long software and hardware restart times, such as security cameras and a smart refrigerator. For this reason, it deploys rollbacks before attempting any restarts to attempt to first correct the system state. The *Time-sensitive* scheme aims to return the system to the desired state as soon as possible, for instance, when an IoT system that is responsible for security is unresponsive. It is similar to Long Restart, but skips Retry, as Retry delays more effective functions. This scheme fits industrial or vehicle environments where system integrity is the top priority.

5 IMPLEMENTATION AND EVALUATION SETUP

For a prototype implementation, we deploy a physical testbed with a set of Arduino IoT devices and applications to mimic a smart home environment. In this section, we introduce our implementation setup, discuss the physical devices and applications in the setup, and then present fault injection techniques we use to evaluate the effectiveness and overhead of IoTRepair.

Implementation Setup. To mimic a real deployment, our setup contains (1) an AVR Arduino Mega 2560 Rev 3 Board [4], which acts as a hub in a smart home environment; (2) a set of devices (sensors and actuators); and (3) a desktop Windows machine, which takes the place of what would be the cloud; the windows machine uses an 8-core 2.2 GHz Intel processor and 12 GB of RAM. An overview of the implementation is shown in Figure 7.

IoT application code and IoTRepair code are installed to the desktop machine. It takes csv configuration files as input and output measurements, discussed in Section 6, into CSV files. During the evaluation, the desktop machine also runs a Python script to drive the evaluation. Polling and actuation functions, as well as the fault-handling functions (Section 4), are uploaded to the Arduino board, which then interacts directly with a variety of devices. The desktop machine and the Arduino board communicate over a serial connection, and Arduino communicates with devices over jumper wires. For evaluation, the Arduino board is coded to perform several commands: (1) Inject Fault, (2) Remove Fault, (3) Poll Devices, (4) Actuate Devices, (5) Check Devices for Faults, (6) Software Restart, (7) Hardware Restart. The timing and power usage of each action is recorded during evaluation.

This setup improves upon our prior work where the evaluation was entirely simulated based on estimated behavior from an observed Arduino setup [25]. In that evaluation, we used the sensor

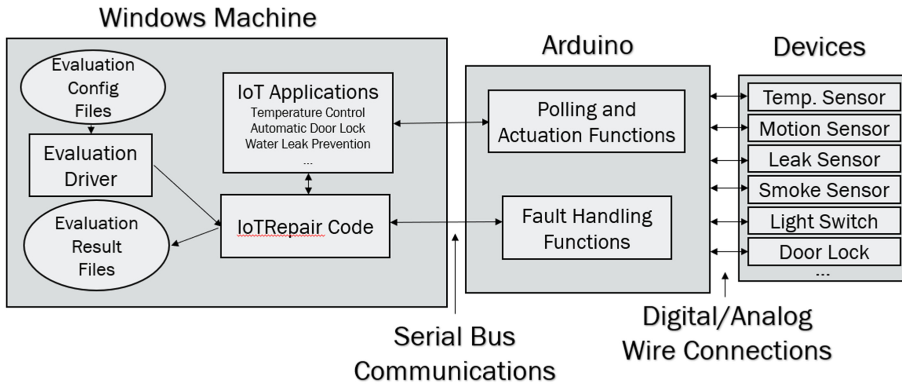

Fig. 7. A graphical representation of our physical implementation, with a desktop Windows machine taking the place of what would be the cloud, an Arduino board acting as the Hub, and a set of devices.

Table 5. Details of Sensors/actuators Used in Our IoT Testbed

ID†	Device‡	Power (mW)	Read time (ms)	Output
S.1	Motion Sensor	66	0.1	double, 8B
S.2	Contact Sensor	0.1	0.1	int, 4B
S.3	Temperature, Pressure, Altitude Sensor	19.5	37.5	double, 8B
S.4	Acceleration sensor	1.3	0.5	int*3,12B
S.5	Flame detector	30	0.96	double, 8B
S.6	Water detector	80	0.1	double, 8B
A.1	LED	0.01	0.1	-
A.2	Buzzer	100	0.1	-

Power is passive power consumed.
[†] S is for Sensor and A is for Actuator. [‡] We use Arduino-compatible devices to simulate the devices in IoT apps. For instance, a flame detector is used for smoke-alarm, and switches are used for the lights (see Section 5).

data observed in a sample Arduino setup to generate traces for 17 simulated devices; those traces represented activity in a smart home, which then drove a trace-driven simulation. That allowed for estimation of the behavior of faults and fault handling in IoTRepair but lacked a guarantee of accuracy in a physical setup.

IoT Devices and Applications. We deployed six different types of sensors and two types of actuators, for a total of eight types of devices. Table 5 presents the types of devices by ID, device description, and device characteristics, which include the power consumption, the amount of time for reading device states, and the type of sensor output values; since actuators do not have output, we use "-" for their output column. Some devices are used as substitutes for devices that would be found in a home environment. For example, the Acceleration Sensor gives X, Y, and Z coordinates of its current position, allowing acceleration to be measured by comparing the difference between two polls. Since a single poll simply gives the current position, it can approximate a presence sensor, which tracks whether the device is present at a location by taking each poll as an isolated current position. Our implemented home environment uses 7 sensors and 10 actuators and includes a set of apps described in Table 6. These apps are selected to cover a wide spectrum of home environment functionality, including green living, convenience, home automation, security and safety, and personal care.

Table 6. The IoT Apps We Developed to Evaluate IoTRepair and Their Descriptions

ID	App Name	Description
App1	Motion-Activated-Lights	When motion detected, turn on the lights. Turn off the lights when motion not active.
App2	Smoke-Alarm	When smoke is detected, sound the alarm and unlock the doors. When no smoke is detected, turn off the alarm.
App3	Temperature-Control	Keep temperature between 70–80 degrees (°F) by turning the heater and the air conditioner on and off.
App4	Water-Leak-Detector	When leak is detected, sound the alarm and close the water valve. When no leak is detected, turn off the alarm and open the water valve.
App5	Welcome-Home	When the user arrives home, unlock the doors and turn on the coffee machine.
App6	Secure-Patio	When user is not present and contact is detected, send text message to user.
App7	Energy-Saver	If the window is open and either the heater or the air conditioner is on, close the window.
App8	Secure-Home	When user is not present at home, lock the doors and close the windows.
App9	Intruder-Detector	When user is not present at home and motion is detected, send text message to user.
App10	Alarm-Safety	When the alarm is activated, turn on the lights.
App11	Morning-Air	Open the windows and close the windows at specific times.

Fault Injection. In our evaluation, we use fault injection to evaluate the effectiveness of fault handling. There is an explosion of possible fault scenarios that can occur in an IoT environment due to the many factors that can influence how a fault appears in the system. Our fault injection aims to take into account of these factors, including fault type, device type, the state of other devices when the fault occurs, and the events that occur during the fault's presence. First, we inject one fault per run. Second, we inject a full spread of every fault type across each device. Finally, we ensure the devices have all entered a stable state prior to beginning each evaluation, meaning that they have finished any startup and are reporting stable state information.

After we have specified the fault injection parameters, we construct an input configuration file that specifies what faults are injected. This allows our system to accept custom sets of faults. Faults are characterized through designated device, fault type, cause, delay, and the false value of every injected fault:

- Designated device can be any valid device ID.
- Fault type is one of Transient-Fail-stop, Transient-Outlier, Transient-Stuck-at, Transient-High-Variance, Transient-Spike, Permanent-Fail-Stop, Permanent-Stuck-At, Permanent-High-Variance, or Permanent-Spike.
- Fault cause is one of Power Failure, Software Failure, Hardware Failure, or Communication Failure. During the evaluation, to inject a fault caused by power failure, the energy wire for a device is turned off; to inject a fault caused by software failure, the variables the device uses to calculate state are corrupted to cause desired behavior; for simulating hardware failure and communication failure faults, their behaviors are coded in the Arduino Board. Hardware failure tags the device as failed and redirects any commands to a function that drives behavior based on the fault type. For example, if the fault type is spike, then it will increase/decrease the state value of each call up to a threshold. Communication Disruption causes any command to the device to fail with a certain probability, defined as 30%, 50%, or 80% in the evaluations.
- Delay determines the number of seconds until IoTRepair becomes aware of the fault; it models the amount of time that a fault-detection module takes to detect the fault.
- Finally, the false value is the state the device falsely reports.

For example, an injected fault entry, `(1,Transient-Fail-Stop,Hardware, F)`, injects a fault into the motion sensor (with device ID 1), with type Transient-Fail-Stop; it is caused by a fault in the hardware, and it causes the device to be nonresponsive (F value for failure). For most fields, our injected faults cover every possible value of those fields, but for delay, false values, and duration, we use a sample set in injected faults. For delay, the values 0 and 100 seconds are used. With false values, for digital devices, we use a complete set of 0 and 1 values, which, depending on device, can represent on/off, detected/not detected, open/closed, or other binary values. For analog devices, we use the extreme ends of the ranges of values; for example, a temperature sensor's value

range is 0 to 38 degrees Celsius. Duration determines how long the fault remains in the system without considering handling. The fault is removed after the duration expires, with duration values being 50, 100, and 150 seconds; we also support permanent faults by making the duration equal to execution time.

Fault Identification. As discussed earlier, IoTRepair is parameterized by a fault-identification module. There is a wide array of identification algorithms available, discussed in Section 2.3. We do not implement our own fault identification system, as there already exists a large body of work on the subject, and it is a different problem than handling faults. We also chose not to implement an existing fault identification system, as the time cost outweighed the minor benefit of using an independent tool to identify our manually injected faults. In our evaluation, we assume an identification module whose amount of identification time is modeled by the delay in fault injection discussed earlier. Further, we assume the module always identifies the correct fault type (after the delay), since IoTRepair focuses on fault repair instead of fault detection.

6 EVALUATION

In our evaluation, we aim to answer the following questions: (1) how effectively can IoTRepair handle a variety of faults?, and (2) what is the runtime efficiency of IoTRepair, in terms of both runtime overhead and power consumption? To this end, we evaluate effectiveness by checking how well IoTRepair removes faults and returns the system to correct operation. We also evaluate IoTRepair's efficiency by measuring the latency between fault detection and the completion of a fault-handling function in IoTRepair. In addition, we measure the power consumption of IoTRepair's fault handling.

We discuss a couple of important notes before we proceed. First, we assume that the motion sensor and smart lights have replicated devices. These devices were chosen to demonstrate device replication, because smart homes are likely to deploy multiple of such devices. Second, in our evaluation, we found that the fail-norm checkpoint is consistently the best match for all devices, and so we only use fail-norm rollbacks in our evaluation.

6.1 IoTRepair Effectiveness

For evaluating the effectiveness of IoTRepair, we constructed a set of faulty scenarios to inject into our setup and test how effective IoTRepair is at handling each scenario. These scenarios and the results are displayed in Table 7. Section 1 presents scenarios where a single fault is injected: For instance, in scenario 1, a failure in a presence sensor causes the door lock to be unlocked incorrectly. Section 2 presents scenarios where multiple faults are injected and Section 3 presents scenarios where a single fault is injected but it causes cascading behavior. For each scenario, we list what faults are injected, what the effect of the fault in the system is, which IoTRepair functions should be used to handle the fault, what the defined desired behavior should be once IoTRepair is finished, and whether IoTrepair was effective at achieving the desired behavior. These scenarios are complete over fault types and applications they affect, though not every possible combination. The scenarios were selected to cover the breadth of fault effects IoTRepair could have to deal with in our setup.

For executions, the state of non-faulty sensors is hard-coded to follow the behavior necessary for the listed fault effects. Faults are injected immediately, but fault detection takes 10 seconds to allow the effects to complete before IoTRepair activates. We manually determine the desired behavior for each scenario and check the states of all devices compared to our desired behavior to evaluate if IoTRepair effectively handled the fault. Desired behavior is generally a return to the state pre-fault, but for some instances where faults are not removed or caused the environment

Table 7. Effectiveness Evaluation Scenarios and the Results of IoTRepair on Different Fault Injections

Group†	ID	Device: Fault	Fault Effect	IoTRepair Function	Desired Behavior	Effective Check
1	1	Presence Sensor: Power Failure	Presence Sensor is unresponsive, so state is unknown	Fail-safe Rollback	Door Lock should be locked, as whether user is present or not is unknown	Door is locked, but Alarm is turned on incorrectly
	2	Door Lock: Communication Disruption	Door Lock does not respond to command to lock	Retry	Door Lock should be locked	Success
	3	Door Lock: Power Failure	Door Lock becomes unresponsive while unlocked	Fail-safe Rollback	Alarm should be turned on, as door cannot be locked	Alarm is turned on, but Water-valve and Window are closed incorrectly
	4	Leak Detector: Hardware Fault	Leak Detector has High Variance between detected and not detected, causing the water valve and alarm to rapidly switch states, causing a large leak	Hardware Restart	Fault is removed and alarm is turned on to alert user of the leaked water	Success
	5	Light: Power Failure	Light is stuck at off when it should be on	Replicate	Replicating Light should be turned on	Success
	6	Motion Sensor: Power Failure	Motion Sensor stops responding, so light is not turned off when there is no motion	Replicate	Replicating Motion Sensor is used and light is turned off	Success
	7	Presence Sensor: Communication Disruption	Poll command to presence sensor fails, so door is not unlocked and coffee not made when user arrives home	Retry	Door Lock should unlock, water valve should open, and coffee maker should be on	Success
	8	Contact Sensor: Hardware Fault	Contact sensor high variance causes user to be spammed with texts that someone is at the door	Hardware Restart	Fault removed and user stops being spammed	Success
	9	Motion Sensor: Communication Disruption	Poll command fails for Motion Sensor when there is motion while user is not home, so they are not alerted	Retry	Poll is completed and user is sent an alert	Success
	10	Window: Hardware Failure	Window is stuck at open and so is not closed when the air-conditioner	Hardware Restart	Fault is removed and window is closed	Success
2	11	Motion & Temperature: Power Failure	Motion and Temperature Sensors are unresponsive, so state is unknown, causing heater to be off when it should not be and light to stay off when there is motion	Replicate and Fail-Norm Rollback	Heater should turn on and light turned on	Success, contingent upon time information indicating heater should be on
	12	Heater & Water-valve: Hardware Failure	Heater is stuck at off and Water-valve is stuck closed	Hardware Restart	Both faults are removed, heater is turned on, and Water-valve is opened	Success
	13	Presence & Smoke & Temperature: Power Failure	Three devices become unresponsive, so door is unlocked, alarm is off, and air-conditioner is on incorrectly	Fail-Safe Rollback	Door Lock should lock, alarm should be on, air-conditioner should be off, since these are safe-states	Success, though would not be if the correct states were reversed, instead would need Fail-Norm
	14	Presence & Light: Hardware Failure	Presence sensor is stuck at home, so door is unlocked and light is stuck on incorrectly	Hardware Restart	Faults are removed, door lock is locked, light is turned off	Success
	15	All sensors: Hardware Fault	A single Outlier event from each device causes all devices to be in incorrect states	Fail-Norm Rollback	Return all actuators to state pre-outlier fault	Success, though most incorrect values are gone by the time IoTRepair commences Rollback from App Code
3	16	Temperature Sensor: Software Failure	Temp. Sensor spikes downward to 10 degrees, causing the heater to turn on, which causes the window to close	Software Restart and Fail-Norm Rollback	Fault is removed, heater is turned off, and window is opened	Success, when Rollback is run second
	17	Smoke Sensor: Hardware Failure	Smoke Sensor becomes stuck-at smoke detected, causing alarm to turn on, which causes light to turn on	Hardware Restart and Fail-Norm Rollback	Fault is removed and alarm is turned off and light is turned off	Success
	18	Smoke Sensor: Software Fault	Smoke sensor stuck at detected, so alarm is turned on incorrectly and light is turned on as well	Software Restart and Fail-Norm Rollback	Fault is removed, alarm and light are turned off	Success
	19	Alarm: Power Failure	Power failure causes alarm not to turn on and so light does not turn on when it should	Fail-Norm Rollback	Light is turned on	Success
	20	Temperature Sensor: Hardware Fault	One poll Outlier causes heater to turn on and window to close	Hardware Restart and Fail-Norm Rollback	Fault is removed, heater is turned off, and window is opened	Success, only if Rollback is run second

[†] **1** is for single faults, **2** is for multiple faults, and **3** is for demonstrating cascading behavior.

state to change, the desired behavior is defined to follow the intent of the applications as best as possible.

As shown in the table, most faults are resolved successfully, but there are a few instances where IoTRepair's success is conditional on some element. For instance, in scenarios 1, 3, and 13, the desired behavior occurs, but, since Fail-safe rollback is used, devices unrelated to the fault are also rolled back. An optimal fault handler would only rollback the devices associated with the given fault; so in scenario 1 the alarm would not be turned on. This behavior is acceptable, because Fail-safe by definition places these devices in a safe state that will cause little to no harm. Additionally, for devices that do not have a Fail-Safe state, Fail-Safe rollback has the same requirements as Fail-Norm.

For Fail-Norm rollback, we also rely on consistent correlation between sensor states. In scenario 11 the correctness of the rollback has requirements, namely, that the non-faulty information indicates the correct state of the faulty devices. In this case, recent timing information in checkpoints indicates that at this time of day the heater is likely to be on. This is acceptable behavior, as it fits into the intended effect of Fail-Norm rollback.

Cascading Behavior. Our applications are susceptible to cascading behavior—which are cases where a fault in one device causes incorrect states in a second device through inter-application interactions. Cascading behavior can be unpredictable, because the application that places a device in a faulty state might not directly utilize the value of the faulty device, and the application interaction may not even be intended. Row **16** of Table 7 shows an example of this. In this scenario, a temperature sensor becoming stuck at a low temperature causes the following sequence of events and actuations in App3, App7, and App11:

$$\textbf{heater-off}\xrightarrow{\textbf{temp<60}}\textbf{App3: heater-on}\xrightarrow{\textbf{heater-on}}\textbf{App7:close-window}$$

In this example, App7 causes the window to close as a cascading result of App3's action responding to a fault in the temperature sensor. The complicated inter-application relationship of cascading behavior lends additional value to environmental functions such as rollback. Even if another function corrects the fault, the cascading behavior may not be automatically removed by application code. In this example, while the heater turning on causes the window to close, there is no code to open the window when the heater is turned off; so the window would incorrectly remain closed even after the temperature sensor fault is removed. In this case, a correct rollback can put the system back into the correct state.

For Table 7's cascading behavior scenarios, we consider fault-handling schemes and whether the order of fault-handling functions in a scheme affects the behavior of IoTRepair. The only function whose position in a scheme matters is rollback. This is because not knowing the state of the device is only relevant for rollback, as it may affect which checkpoint is selected. Therefore, if a previous fault-handling function corrects a faulty device, then rollback is more likely to select the correct checkpoint to remove any cascading behavior the fault had. In scenarios 16 and 20, rollback must be run second, because in these instances Fail-Norm chooses the incorrect Checkpoint as the true sensor state is unlikely given other state information. In other scenarios rollback selects the correct target whether the fault is present or not, as the true device state of the faulty device is likely given the other state information. This is acceptable as it mostly relies on the accuracy of rollback, which, as shown below in our discussion of cascading behavior, is very high. Other than the interaction between rollback and other fault-handling functions, the only other interaction is that software restart should never follow hardware restart, since hardware restart corrects anything software restart would.

To further understand the effectiveness of rollback, we discuss the pros and cons of the three types of checkpoint selection that allow rollback to handle different types of cascading behaviors based on desired behavior: most recent, fail-norm, fail-safe.

Most Recent is the best at resolving cascading behavior that is detected quickly and is reversible. Reversible in this case means the state changes caused by the cascade can be undone. For example, a window that was opened can be closed, but it is not possible to reverse making coffee. A good example of when Most Recent rollback would correct a cascade in our implementation is the sample given in the text above, where a fault in the Temperature Sensor closes the window; a Most Recent rollback would turn off the heater and open the window, correcting the issue.

Fail-Norm is the best rollback for when there are only a few faults in the system and there are no possibly highly dangerous states based on the faulty devices. For example, in Table 7 scenario **16**, we see that turning on the heater can cause the window to close incorrectly. Here, fail-norm may find a checkpoint where non-faulty sensors indicate the heater should be off and the window should be open and correctly actuate the devices. However, if the temperature sensor also caused a dangerous action, such as unlocking the door, then it may be better to rely on fail-safe.

Fail-Safe is the best rollback when neither of the conditions for Most Recent or Fail-Norm is true. As this rollback attempts to move all devices to their "secure" state, it should avoid any unsafe states, to the maximum degree that can be achieved given the fault set. For example, if we were to consider the Smoke Sensor cascade in Table 7 scenario **18**, then the state of the alarm could be potentially dangerous if there is a fire; so Fail-Safe would turn on the alarm to establish the safest state until the fault is properly resolved.

For Most Recent, the effectiveness and accuracy are determined by how quick the fault is detected; this is independent of IoTRepair, which relies on a separate fault-identification module. Fail-Safe can be statically evaluated, given a set of faulty sensors the target rollback is known; so the only determinant is if it is possible. For example, we know fail-safe will always attempt to close the window; but if the window becomes stuck-at open, then there is nothing fail-safe can do to correct the state.

The only rollback that must be evaluated empirically during out evaluation is fail-norm, as its effectiveness and accuracy depends on how well the checkpoints IoTRepair collects represent system states. To this end, we evaluate Fail-Norm by running the program with 0 faults for 1,000 polls to get a baseline and gather checkpoints. We then run again with rollback enabled and attempt a rollback at each poll timestamp. For each rollback, we analyze if the correct checkpoint is chosen across every possible device being faulty, and if it is, then we mark that as accurate and if not, then it is marked as incorrect. We indicate a correct checkpoint by the checkpoint selecting the actuator target states that match the baseline at this timestamp. The primary cause of an incorrect checkpoint is when a sensor is faulty; it cannot be considered when selecting the best fit checkpoint; thus, two or more checkpoints may be "valid." In this case, fail-norm selects the one that occurred the most during checkpointing. Within this experiment, only one rollback was incorrect by these standards, giving a 99.9% accuracy, at least in a system that stays stable for long periods of time and only experiences a single fault.

Overall, this evaluation shows that IoTRepair can effectively handle a wide range of faults, most of the time returning the system to normal operation or at the very least making the system behavior closer to intended behavior.

6.2 Function Latency

In this experiment, we evaluate the latency of functions in IoTRepair's fault-handling library. Across all possible fault injection parameters there is a huge variety of scenarios and because of this large state space it would be time-consuming to experimentally cover the entire state space

to measure the latency of each fault-handling function for each possible scenario. Therefore, for our evaluation, we decided to analyze a select subset of those to cover typical kinds of devices and typical faults that can happen to those devices. Among the 17 devices in our IoT testbed, we selected two sensors and two actuators. The two sensors include a simple digital sensor (a motion sensor) and an analog sensor (a temperature sensor). The two actuators include one with high frequency of actuation (a light actuator) and one with low frequency of actuation (a heater). These four devices are representative of the devices in our testbed. For example, a motion sensor and a contact senor are nearly identical in their characteristics. We also performed experimental analysis on devices not selected for evaluation and confirmed that there is little variance between the data for the representative devices and the data for the rest.

For each selected device, we picked the most typical kind of fault to inject for that device; we will explain this selection and its rationale soon. To get timing information, we record the total time between when a fault is injected and when a fault is either resolved by a fault-handling function or the function determines it cannot resolve the fault. The latency is calculated from an average of three executions for each scenario and each possible set of parameters. We note some parameter combinations are invalid; specifically, since power and communication disruption can only cause fail-stop faults, there is no need to run executions with a fault delay greater than or equal to fault duration, and only communication disruption has a communication probability parameter. Furthermore, when getting the timing information of a fault-handling function, we remove the communication time spent between the core machine, representing the cloud, and the Arduino board, representing the Hub. In our experiments, our communication time was on average 1.23 seconds, which is similar to cloud-based IoT environments [22], and significantly longer than localized implementations, such as mobile devices [8]. This communication time is removed because it is independent of the execution time of our IoTRepair functions, which is the main objective for this evaluation.

The evaluation results are presented in Figure 8 to Figure 11. For these figures, we label fault-handling functions by Success/Fail labels. A label specifies whether a fault-handling function completes its goal or not. For different functions, goal completion can mean different things; for restart, the goal is to remove the fault, while rollback and retry simply aim to mitigate the faulty behavior. Furthermore, even though a fault may have multiple parameters, through our experimental data, we found most parameters did not cause variation on the latency of fault-handling functions. Therefore, each figure will use only the most dominant parameter (in terms of its effect on fault-handling latency) as the x-axis.

Figure 8 presents the results of a Communication Disruption fault injected into a Motion sensor. We selected the Communication Disruption for two reasons. First, the motion sensor (located on a porch or at the front door) is the most likely to be far from the Hub, and therefore more likely to have periods of unreliable connection to the Hub. Second, since most motion sensors are battery-powered and simple, they are relatively unlikely to suffer other types of faults. The x-axis in this graph is the Communication Disruption Probability, which represents how likely the fault is to cause any given communication to fail. Each failed message causes additional time where the action must be repeated until completion or reaching the limit of attempts. For this reason, of all of our parameters this has the most impact for most functions, with the exception of rollback. Because rollback does not consider faulty devices when selecting the best checkpoint, the communication failure probability of a device has no impact on latency. Additionally, this Communication Disruption Probability parameter causes the highest standard deviation among sampled results, because each execution can vary wildly in latency based on how many times the action fails. But from this graph, we can see that even with an extremely high (80%) chance of failed communication,

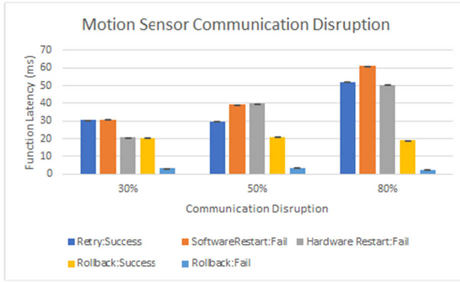

Fig. 8. The average time for each function to execute on a Communication Disruption fault injected into a Motion Sensor with increasing disruption probabilities.

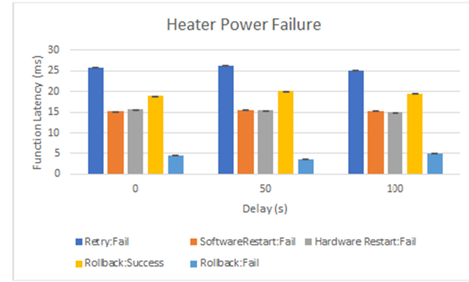

Fig. 9. The average time for each function to execute on a Power Failure fault injected into a Heater across increasing delays before the fault is detected.

the latency only doubles compared to the case of having 30% chance to fail and remains under 60 milliseconds.

Figure 9 shows a power failure injected into a heater actuator. We selected power failure because it is the most likely fault for a heater to experience; anytime a home loses power or the power flickers, the heater can lose power—or if its connection to an outlet is disrupted in any way. The x-axis shows the amount of fault delay (i.e., the amount of time the fault-detection module detects the fault); even though the fault delay causes the largest change among applicable parameters, it still causes only a fraction of a percentage change in latency. With 25 ms as the highest latency, it is clear that latency is low for power failures in this evaluation; however, it is worth noting that only rollback can succeed in case of a power failure.

Figure 10 shows a hardware fault injected into a light actuator. This fault was selected because among all our devices lights have the shortest lifespan, generally burning out or otherwise failing far before other devices would suffer major hardware faults. The x-axis is selected to be duration, as it has the highest impact; though similar to the previous figure, the change is small in magnitude. The only exception is for failed rollbacks. This is because rollback attempts to actuate light to a certain state, and because it is faulty this cannot succeed; however, for each run the selected checkpoint and system state cause high variance in how many actuations are attempted. This is more impactful than successful rollbacks, since the devices that did succeed need to be reverted to the pre-rollback state. Despite this variance, the latency still remains below a fast 40 ms, on average.

Finally, Figure 11 presents a software fault injected into the temperature sensor. Of our deployed devices, only Temperature Sensor and Smoke Sensor are complex enough to suffer proper software faults and allow a software restart, and the temperature sensor is the more complex of two. Fault duration is the parameter with the most impact; so it is selected as the x-axis. But just like the previous two graphs, the impact on latency is still small. The average latency for all functions remains small, even with the relative complexity of temperature sensors.

It can be seen across these graphs that only the communication disruption probability makes any real impact on latency. For delay and duration, changes across the axis are small and do not exhibit any trend. For fault duration, this is because the typical duration is much longer than any element of IoTRepair latency. For fault delay, when the time spent on fault identification is removed, most IoTRepair functions are not sensitive to the state change over that time. The only function that can be is rollback, and in our experiments, the delay did not drastically change how many devices needed actuated. Also, besides the latency of these functions, transactions incur an average of 44.82-millisecond latency per-transaction. This is because the transaction must complete and verify each actuation before moving on to the next and record the previous states of devices in case

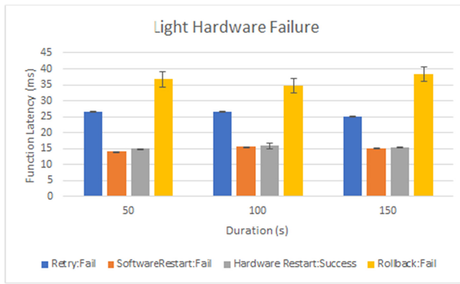

Fig. 10. The average time for each function to execute on a Hardware Fault injected into a Light across increasing fault duration.

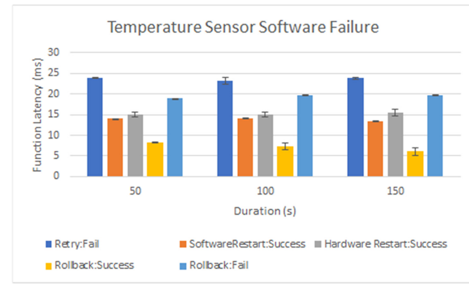

Fig. 11. The average time for each function to execute on a Software Fault injected into a Temperature Sensor across increasing fault duration.

a transaction rollback becomes necessary. We separate transaction and measure it independently, as unlike the other functions it operates whether a fault is present or not. This means it imposes a constant latency overhead on the system that must be accepted by the user, and that it would be pointless comparing it to functions that only have to be called once per fault. Finally, we also tracked the latency of checkpointing and replication. We chose not to include checkpointing in the graph in part because it occurs constantly whether a fault is present or not, and because both checkpoints and replication take fractions of milliseconds to finish. This is due to the lack of reliance on the Arduino Hub, with all calculation taking place on the far-faster machine representing the cloud implementation. Additionally, only motion and light devices have replicating devices in our setup.

Based on our evaluation, we argue that the amount of latency in IoTRepair is acceptable compared to the current state of IoT. First, the current state of the art of IoT fault identification takes far longer to detect most faults than our system takes to handle them. This is shown in more detail in our Related Work section; one such example is DICE [6], one of the fastest identification methods, and it still takes at least 10 minutes to identify faults. Compared to this, the latency of IoTRepair is insignificant. This cannot be said for fail-stop faults, though, which some systems may detect near-instantly, or at least within a matter of seconds, depending on the polling/event generation setup. Even in these cases, we argue our latency is acceptable, as the average fault duration is also quite high relative to the latency. As seen in Reference [13], faults can last hours, in which case, while handling them as fast as possible is still desirable, IoTRepair is fast enough to handle the vast majority of the harm the fault will cause. In the case of outlier faults, such as in Reference [36], the fault will be gone before IoTRepair can handle it, but IoTRepair can still rollback and likely fix any changes that the brief fault caused.

Comparing IoTRepair's latency to other systems' faces several challenges. First, most existing systems incur some constant latency cost to normal system operation rather than a time-to-fix latency like IoTRepair. Additionally, many papers do not report latency in a way conducive to comparisons. Many papers either omit latency information, evaluate some other metric such as throughput, or do not provide a baseline for comparison. Despite this, with some justifications, we can compare to several systems that show IoTRepair has acceptable latency. Transactuations [30] provide the execution time of 10 SmartThings apps with and without their system. They observe an average of 50% overhead when Transactuations are implemented. IoTRepair's transactions incur slightly less overhead, with an overhead of 37.5% when compared to normal actuations. The comparison is not perfect, since Transactuations focus on preserving the synchronization of soft and hard states, while our transaction avoids unsafe state transitions; however, the implementation

from a user's standpoint is similar. Another system [1] implements several blockchain consensus algorithms to increase fault tolerance when some devices may provide incorrect information. It measures the latency of sending packages of 360 bytes, though it does not provide a ground-truth for when there is no blockchain implementation. Depending on the algorithm, number of nodes, and propagation model, the latency ranges from around 3 milliseconds to almost 500 milliseconds. So, depending on the circumstances, the latency may be better or worse than IoTRepair functions, but even in the best case where the overhead is only a few milliseconds this method imposes a constant cost. This cost is incurred far more frequently than our Transaction or Checkpoint and will slow the system far more than IoTRepair. Another method for implementing fault tolerance is to design an IoT architecture from the ground up to be Fault Tolerance, such as IoTEF [16]. As IoTEF is an independent architecture, it has no baseline to compare to. However, it does provide the latency of processing images. When sending images from the camera to the edge device it takes roughly 10–60 ms, depending on the image size. The median is similar to each of our fault-handling functions, showing that IoTRepair's fault-handling functions have comparable latency to normal operating processes in IoTEF. Finally, Wang et al. [41] propose a system that adaptively determines whether edge devices need to replicate data in case of data loss. In their evaluation, the system imposed only an overhead of a couple milliseconds at worst from a retransmission-only baseline. This is less overhead than IoTRepair functions besides Replicate and Checkpoint, but once again it occurs far more frequently and targets only preventing data-loss due to a fault. Overall, IoTRepair imposes very little latency overhead during normal runtime when compared to other fault tolerance solutions, and the time to handle faults is relatively low.

6.3 Power Consumption

In this evaluation, we measure the power overhead caused by IoTRepair and show that it is low cost in terms of power consumption compared to normal system operations. For this, we first tracked how much power was consumed across each execution and compared the power overhead of our fault-handling functions. We performed this using a Monsoon Power Monitor to measure the power consumed by the Arduino board during execution. We show these results with the same scenarios used in Section 6.2. The power consumption is shown as the average power consumption used during IoTRepair function calls.

Figure 12 presents the average power consumption of a fault-handling function for the scenario when a communication disruption fault is injected into a motion sensor. Most functions consume less than 200 milliwatts, which is an insignificant amount when compared to the fact that the average power consumed per poll cycle is 4.28 watts. A poll cycle represents the time from when a poll begins to when the next poll begins. The time it takes a cycle to complete affects how much power is consumed, since idle power consumption is relatively high for the system. Time is primarily consumed by communication time in the system, as discussed in Section 6.2. A non-faulty poll cycle with no actuations takes an average of 2.56 seconds, while a cycle with actuations can add 2.31 seconds or more, depending on how many devices, and are actuated and if an actuation causes more actuations to occur due to application code. During a cycle, power is consumed for receiving events, running applications to determine what actuations are needed, and performing actuations. We use the power consumption of a poll cycle as our base power consumption, as it represents the power consumption of the normal behavior of the system. Figure 12 also has the highest standard deviation of the four figures we present, because communication disruption alters with some probability how many times the functions need to attempt to communicate with the device. rollback has the highest standard deviation and cost across most graphs, because it must operate on various numbers of devices based on the chosen checkpoint. Overall, even rollback costs a tiny fraction of the power consumed during normal system operations.

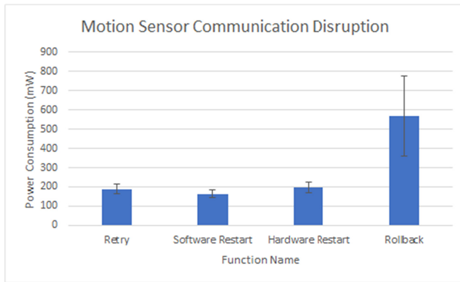

Fig. 12. The average power consumption of each function on a Comm. Disruption fault injected into a Motion Sensor.

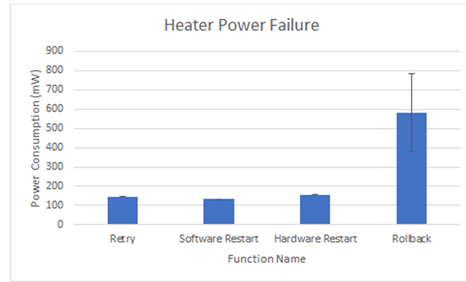

Fig. 13. The average power consumption of each function on a Power Failure fault injected into a Heater.

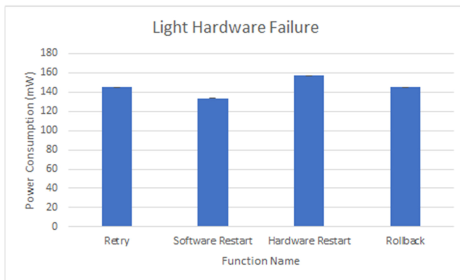

Fig. 14. The average power consumption of each function on a Hardware Failure fault injected into a Light.

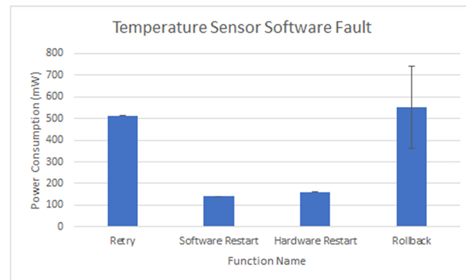

Fig. 15. The average power consumption of each function on a Software Failure fault injected into a Temperature Sensor.

Figure 13 shows the average power consumption of a fault-handling function when a power failure is injected into a heater. The amount of power consumption is also low. For the first three functions, the cost is less than the numbers for the communication-disruption scenario, as the functions cannot succeed and therefore do not consume extra time and power. The only exception is rollback, which can succeed so long as the heater does not need to be turned on for a given checkpoint. All functions still remain relatively small on power consumption, compared with this scenario's average poll cycle costing 3.80 watts.

Figure 14 presents the average power consumption of a fault-handling function when a hardware failure is injected into a light. Among the presented scenarios, rollback has a lower cost compared to other fault-handling functions. This is because in this scenario the light must change state for a successful rollback; so the rollback simply fails to execute once this is determined. Aside from that, this graph is similar to the others with all functions consuming a small amount of power compared to the poll cycle cost of 3.68 watts.

Figure 15 presents the average power consumption of a fault-handling function when a software fault is injected into a temperature sensor. The only noteworthy difference from previous graphs is the high cost of Retry. This is a side-effect of polling the Temperature Sensor being much more expensive than polling other much simpler devices. Since Retry uses polling when attempting to resolve the fault, it also increases the cost of Retry. The higher cost of Retry would hold true for all complex devices, such as Presence, Leak, and Contact sensors that report analog rather than digital values. Despite this increase, it is still less expensive than rollback and relatively small compared to the poll cycle cost of 3.82 watts.

Table 8. The Power Consumed by the Devices and Board Computation for Each Function

	Motion	Temperature	Smoke	Leak	Presence	Contact	Light	Alarm
Poll	2.75 mW	6.1 mW	13.69 mW	15.81 mW	18.95 mW	20.88 mW	2.64 mW	2.79 mW
Actuate	N/A	N/A	N/A	N/A	N/A	N/A	10.7 mW	6.6 mW
Retry	4.01 mW	14.04 mW	15.75 mW	17.55 mW	20.64 mW	23.13 mW	6.52 mW	8.58 mW
softwareRestart	3.54 mW	3.83 mW	3.04 mW	2.99 mW	3.35 mW	3.44 mW	5.8 mW	7.07 mW
hardwareRestart	4.05 mW	4.24 mW	3.49 mW	3.58 mW	3.82 mW	3.92 mW	7.69 mW	7.88 mW

Another consideration in power consumption is the power cost to individual devices, which may rely on a battery or have their lifespan shortened by high power usage. To evaluate our device overhead, we collect the baseline consumption of the Arduino board with no devices powered and compare that to both when the device is powered but not doing anything and when our functions run on the device. As shown in Table 8, there is a very low cost each function incurs beyond the idle operations of the Arduino Board and devices. As can be seen in the table, the functions all have similar costs to the simple Poll and Actuate functions, and when comparing the several Watt cost of each poll cycle this cost is insignificant. It is worth noting that rollback has been left out of this table as its cost to Arduino and devices is identical to actuate, which only has a cost for light and alarm devices. Also, it is not possible to separate what is consumed by the device and what is consumed by the Arduino Board for computation using our method of power evaluation, but since both of these costs are incurred by the function call directly, their sum can be considered the direct cost incurred by a function call.

Comparing energy cost to other fault-handling solutions has the same limitations as latency comparisons. Nontheless, we compare IoTRepair's energy consumption to a fault-tolerance solution that aims for energy efficiency [43]. This article aims to use the min-cut algorithm to minimize the overhead of saving state information to non-volatile-memory while still saving enough for good fault tolerance. They implement their system on a set of controller benchmarks and observe an energy overhead ranging from 19.9% to 65.8% with an average of 41.5%, and observe a 14.9% average overhead on Mediabench benchmarks. With our functions costing only less than 1,000 milliwatts each time they are called, the total overhead of IoTRepair across one poll cycle is a fraction of a percent. As time passes with no faults in the system, the overhead during the short periods when there are faults will shrink even more, so IoTRepair clearly beats out this method in energy efficiency.

Overall, it is clear that there is a negligible energy cost to IoTRepair. This is intuitive as compared to the lengthy time IoT devices are meant to be deployed for; a single restart or actuation is a negligible and infrequent cost.

7 LIMITATIONS AND DISCUSSION

The evaluation of the effectiveness of IoTRepair's fault handling is limited by the coarse-grained definition of fault causes and the limited set of parameter values we evaluated. This is necessary for an evaluation with limited resources, as otherwise the set of combinations would be infeasible to execute. With coarse-grained definitions, some scenarios are not observed, most obviously a Software/Hardware Fault that cannot be corrected by a restart. For example, a hardware fault caused by a damaged circuit will likely persist or reappear after a restart. This is not essential to evaluate, however, as this then makes the faulty behavior similar to that of a Power Failure, with the only exception being the fault may be high-variance or spike. Conversely, there may be instances where Communication Disruption is removed by a restart. This is also similar to other evaluated cases, those being Software/Hardware Faults, with the difference it may take longer due to failed communication. Additionally, random elements such as random packet loss during

communication disruption can influence factors such as the effectiveness and latency of retry and restart. We also assumed a perfect fault identification module; so we assumed that no Byzantine Failures occurred where the fault-identification module missed faults entirely or falsely reported faults. When a device is never reported faulty, most of the time IoTRepair will simply remain idle and have no extra negative effect. However, if a falsely reported fault causes a rollback, then the silent failure may cause an incorrect checkpoint to be selected as the device's presumed state. This can also occur when a device is falsely identified, but in addition unnecessary function calls can disrupt the device's service or unnecessarily change many device states as part of a rollback. Obviously this is a major issue; the selected fault-identification module should report only faults it is absolutely sure of. Future work to mitigate this issue could be to include confidence levels for the functions, where certain functions will be disabled if the fault cannot be identified with sufficient confidence.

We restrict our evaluation to a moderate set of IoT devices and applications. It allows more tractable outputs to evaluate our fault-handling functions and the effect of faults. More sophisticated systems with a diverse set of devices and apps, especially in the settings of industrial IoT and automobiles, would allow for a more thorough evaluation. For instance, the increase in the number of devices would lead to more cascading behavior, as well as increasing the probability of device correlation and allowing some of our underutilized functions such as device replication to be more effective. This would also allow for suppression of entire apps to be effective, as the primary use of this technique is to stop cascading behavior.

There are also limitations in the core assumptions and design of IoTRepair. First, we assume a centralized control system, and so our current design would not work on a distributed system where control is shared between cooperating nodes. A future work could be modifying the design in a way that allowed a distributed installation. A relatively trivial method would be to force such systems to designate one node as the only one that could issue sensing and actuation commands generated by IoTRepair to devices and other nodes, while a more complex design would have to use scheduling or data locks to prevent faulty behavior through unnecessary or repeated actions.

Another design limitation is in our design of Checkpoint/rollback, which can change the states of devices that were not affected by a fault. The reason for this is our assumption that the identification module cannot provide the exact time a fault occurred. This makes tracking exactly what devices were affected nearly impossible, as we cannot track where cascading behaviors began. If we change this assumption, then a future work could detect what devices were affected by the faulty behavior. This can be accomplished either through analyzing history logs if the control node has access to cause/effect relationships in actuations or through analyzing the application logic.

8 CONCLUSION

With this extension on the previous IoTRepair paper, we show that our system can be used to mitigate faults in a physical system with minimal latency and acceptable power overhead. We have shown that a flexible fault handler is a necessary next step to improve reliability, as Internet of Things systems continue to grow in complexity. As future work, we hope to adapt our design to other systems that manage complex environments, such as Smart Cities.

REFERENCES

[1] Omar Alfandi, Safa Otoum, and Yaser Jararweh. 2020. Blockchain solution for IoT-based critical infrastructures: Byzantine fault tolerance. In *IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–4. DOI: https://doi.org/10.1109/WF-IoT.2018.8355149

[2] Apple. 2018. Apple: Homekit. Retrieved from https://developer.apple.com/homekit/.

[3] Masoud Saeida Ardekani, Rayman Preet Singh, Nitin Agrawal, Douglas B. Terry, and Riza O. Suminto. 2017. Rivulet: A fault-tolerant platform for smart-home applications. In *ACM Middleware Conference (ACM 18)*. ACM, 41–54. DOI: https://doi.org/10.1145/3135974.3135988

[4] Arduino. 2019. Arduino Mega 2560 Rev3. Retrieved from https://store.arduino.cc/usa/mega-2560-r3/.

[5] Idris M. Atakli, Hongbing Hu, Yu Chen, Wei Shinn Ku, and Zhou Su. 2008. Malicious node detection in wireless sensor networks using weighted trust evaluation. In *Spring Simulation Multiconference (SpringSim'08)*. SpringSim, 836–843. DOI: https://doi.org/10.1145/1400549.1400686

[6] Jiwon Choi, Hayoung Jeoung, Jihun Kim, Youngjoo Ko, Wonup Jung, Hanjun Kim, and Jong Kim. 2018. Detecting and identifying faulty IoT devices in smart home with context extraction. In *48th IEEE/IFIP Conference on Dependable Systems and Networks (DSN'18)*. IEEE, 610–621. DOI: https://doi.org/10.1109/DSN.2018.00068

[7] Prafulla Kumar Choubey, Shubham Pateria, Aseem Saxena, S. B. Vaisakh Punnekkattu Chirayil, Krishna Kishor Jha, and P. M. Sharana Basaiah. 2015. Power efficient, bandwidth optimized and fault tolerant sensor management for IOT in smart home. In *IEEE International Advance Computing Conference (IACC'15)*. IEEE, 366–370. DOI: https://doi.org/10.1109/IADCC.2015.7154732

[8] Jie Ding, Mahyar Nemati, Chathurika Ranaweera, and Jinho Choi. 2020. IoT connectivity technologies and applications: A survey. DOI: https://doi.org/10.1016/10.1109/ACCESS.2020.2985932

[9] Eiman Elnahrawy and Badri Nath. 2003. Cleaning and querying noisy sensors. In *2nd ACM International Conference on Wireless Sensor Networks and Applications (WSNA'03)*. Association for Computing Machinery, 78–87. DOI: https://doi.org/10.1145/941350.941362

[10] Lei Fang and Simon Dobson. 2013. Unifying sensor fault detection with energy conservation. In *Lecture Notes in Computer Science (Lecture Notes in Computer Science,* Vol. 8221*)*. Springer, Berlin, Germany, 176–181. DOI: https://doi.org/10.1007/978-3-642-54140-7_18

[11] Google. 2018. Google: Android Things. Retrieved from https://developer.android.com/things/.

[12] Jun Han, Albert Jin Chung, Manal Kumar Sinha, Madhumitha Harishankar, Shijia Pan, Hae Young Noh, Pei Zhang, and Patrick Tague. 2018. Do you feel what I hear? Enabling autonomous IoT device pairing using different sensor types. In *IEEE Symposium on Security and Privacy (SP'18)*. IEEE, 836–852. DOI: https://doi.org/10.1109/SP.2018.00041

[13] Timothy W. Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I. Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. 2011. The hitchhiker's guide to successful residential sensing deployments. In *Conference on Embedded Networked Sensor Systems (SenSys'11)*. Association for Computing Machinery, 232–245. DOI: https://doi.org/10.1145/2070942.2070966

[14] Iotivity. 2018. IoTivity. Retrieved from https://iotivity.org/.

[15] Asad Javed, Keijo Heljanko, Andrea Buda, and Kary Främling. 2018. CEFIoT: A fault-tolerant IoT architecture for edge and cloud. In *IEEE 4th World Forum on Internet of Things (WF-IoT'18)*. IEEE, 813–818. DOI: https://doi.org/10.1109/WF-IoT.2018.8355149

[16] Asad Javed, Keijo Heljanko, Andrea Buda, and Kary Främling. 2020. IoTEF: A federated edge-cloud architecture for fault-tolerant IoT applications. *J. Grid Comput.* DOI: https://doi.org/springer:10.1007/s10723-019-09498-8

[17] Krasimira Kapitanova, Enamul Hoque, John A. Stankovic, Kamin Whitehouse, and Sang H. Son. 2012. Being SMART about failures: Assessing repairs in SMART homes. In *ACM Conference on Ubiquitous Computing (UbiComp'12)*. Association for Computing Machinery, 51–60. DOI: https://doi.org/10.1145/2370216.2370225

[18] T. Kavitha and D. Sridharan. 2009. Security vulnerabilities in wireless sensor networks: A survey. *J. Inf. Assur. Secur.* 5 (Nov. 2009), 31–44.

[19] Palanivel A. Kodeswaran, Ravi Kokku, Sayandeep Sen, and Mudhakar Srivatsa. 2016. Idea: A system for efficient failure management in smart IoT environments. In *ACM MobiSys (ACM'14)*. ACM, 43–56. DOI: https://doi.org/10.1145/2906388.2906406

[20] Fanxin Kong, Meng Xu, James Weimer, Oleg Sokolsky, and Insup Lee. 2018. Cyber-physical system checkpointing and recovery. In *International Conference on Cyber-Physical Systems (ICCPS'18)*. IEEE, 22–31. DOI: https://doi.org/10.1109/ICCPS.2018.00011

[21] Ajay D. Kshemkalyani and Mukesh Singhal. 2011. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, Cambridge, UK.

[22] Ayaskanta Mishra, Sayan Karmakar, Ankush Bose, and Ankita Dutta. 2020. Design and development of IoT-based latency-optimized augmented reality framework in home automation and telemetry for smart lifestyle. *J. Reliab. Intell. Environ.* 6 (Feb. 2020), 169–187. DOI: https://doi.org/10.1007/s40860-020-00106-1

[23] Sirajum Munir and John A. Stankovic. 2014. FailureSense: Detecting sensor failure using electrical appliances in the home. In *IEEE 11th International Conference on International Conference on Mobile Ad Hoc and Sensor Systems*. IEEE, 73–81. DOI: https://doi.org/10.1109/MASS.2014.16

[24] 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 Trans. Sensor Netw.* 5, 3 (May 2009), 1–29. DOI: https://doi.org/10.1145/1525856.1525863

[25] Michael Norris, Berkay Celik, Prasanna Venkatesh, Shulin Zhao, Patrick McDaniel, Anand Sivasubramaniam, and Gang Tan. 2020. IoTRepair: Systematically addressing device faults in commodity IoT. In *IEEE/ACM 5th International Conference on Internet-of-Things Design and Implementation (IoTDI'20)*. IEEE, 142–148. DOI: https://doi.org/10.1109/IoTDI49375.2020.00021
[26] OpenHAB. 2018. OpenHAB: Open Source Automation Software. Retrieved from https://www.openhab.org/.
[27] G. Padmavathi and M. Shanmugapriya. 2009. A survey of attacks, security mechanisms and challenges in wireless sensor networks. Retrieved from https://arxiv.org/ftp/arxiv/papers/0909/0909.0576.pdf.
[28] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. 2005. Sympathy for the sensor network debugger. In *ACM International Conference on Embedded Networked Sensor Systems (SenSys'05)*. USENIX Association, 255–267. DOI: https://doi.org/10.1145/1098918.1098946
[29] Samsung SmartThings 2018. Samsung SmartThings Add a Little Smartness to Your Things. Retrieved from https://www.smartthings.com/.
[30] Aritra Sengupta, Tanakorn Leesatapornwongsa, Masoud Saeida Ardekani, and Cesar A. Stuardo. 2019. Transactuations: Where transactions meet the physical world. In *USENIX Annual Technical Conference (USENIX ATC'19)*. USENIX Association, 91–106. DOI: https://doi.org/10.1145/10.1145/3380907
[31] Abhishek B. Sharma, Leana Golubchik, and Ramesh Govindan. 2010. Sensor faults: Detection methods and prevalence in real-world datasets. *ACM Trans. Sensor Netw.* 6, 3 (June 2010), 1–39. DOI: https://doi.org/10.1145/1754414.1754419
[32] Chi-Sheng Shih, Jyun-Jhe Chou, Niels Reijers, and Tei-Wei Kuo. 2007. Designing CPS/IoT applications for smart buildings and cities. *IET Cyber-Phys. Syst.: Theor. Applic.* 1, 1 (Dec. 2007), 3–12. DOI: https://doi.org/10.1049/iet-cps.2016.0025
[33] SmartThings. 2017. Honeywell Z-Wave Thermostata. Retrieved from https://tinyurl.com/yyzknakl.
[34] SmartThings. 2018. Samsung SmartThings Developer Documentation. Retrieved from https://docs.smartthings.com/.
[35] Penn H. Su, Chi-Sheng Shih, Jane Yung-Jen Hsu, Kwei-Jay Lin, and Yu-Chung Wang. 2014. Decentralized fault tolerance mechanism for intelligent IoT/M2M middleware. In *World Forum on Internet of Things (WF-IoT'14)*. IEEE, 45–50. DOI: https://doi.org/10.1109/WF-IoT.2014.6803115
[36] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. 2004. Lessons from a sensor network expedition. In *European Workshop on Wireless Sensor Networks (EWSN'04)*. Springer, 307–322. DOI: https://doi.org/10.1007/978-3-540-24606-0_21
[37] KaaIoT Technologies. 2018. KaaIoT. Retrieved from https://kaaproject.org/.
[38] Doug Terry. 2016. Toward a new approach to IoT fault tolerance. *IEEE Comput.* 49, 8 (Aug. 2016), 80–83. DOI: https://doi.org/10.1109/MC.2016.238
[39] Zhan Tu, Fan Fei, Matthew Eagon, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. 2018. Redundancy-free UAV sensor fault isolation and recoverys. DOI: https://doi.org/arXiv:0909.0576
[40] Vera. 2018. Vera Control: Smart Home Control. Retrieved from https://support.getvera.com/customer/en/portal/articles/1710500-plugin-development.
[41] Chao Wang, Christopher Gill, and Chenyang Lu. 2020. Adaptive data replication in real-time reliable edge computing for internet of things. In *IEEE/ACM 5th International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 128–134. DOI: https://doi.org/10.1109/IoTDI49375.2020.00019
[42] Wikipedia. 2018. Wink: A Simpler Smart Home. Retrieved from https://winkapiv2.docs.apiary.io/#reference/a-restful-service.
[43] Teng Xu and Miodrag Potkonjak. 2016. Energy-efficient fault tolerance approach for internet of things applications. In *35th International Conference on Computer-aided Design*. ACM, 1–8. DOI: https://doi.org/10.1145/2966986.2967034
[44] Juan Ye, Simon Dobson, and Susan McKeever. 2012. Situation identification techniques in pervasive computing. *Pervas. Mob. Comput.* 8, 1 (Feb. 2012), 36–66. DOI: https://doi.org/10.1016/j.pmcj.2011.01.004

Received October 2021; revised February 2022; accepted April 2022