IoTRepair: Systematically Addressing Device Faults in Commodity IoT

Michael Norris[1], Z. Berkay Celik[2], Prasanna Venkatesh[1], Shulin Zhao[1],
Patrick McDaniel[1], Anand Sivasubramaniam[1], and Gang Tan[1]

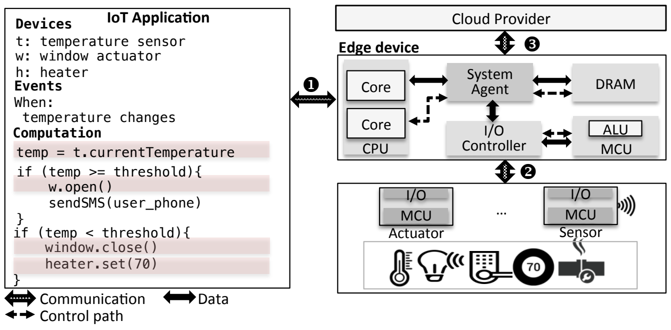
[1]Department of Computer Science and Engineering, Penn State University, State College, Pennsylvania
[2]Department of Computer Science, Purdue University, West Lafayette, Indiana

***Abstract*—IoT devices are decentralized and deployed in unstable environments, which causes them to be prone to various types of faults, such as power failure and network disruption. Yet, current IoT platforms require programmers to handle faults manually, a complex and error-prone task. In this paper, we present IoTRepair, a fault-handling system for IoT that (1) integrates with fault identification modules to track faulty devices, (2) provides a library of fault-handling functions for effectively handling different fault types, (3) provides a fault handler on top of the library for autonomous IoT fault handling, with deployed devices, user preferences, and developer configuration as input. Through an evaluation in a simulated lab environment, we find IoTRepair reduces the incorrect states on average 63.51%, which corresponds to less unsafe and insecure device states. Overall, through a systematic design of an IoT fault handler, we provide users flexibility and convenience in handling complex IoT fault handling, allowing safer IoT environments.**

I. Introduction

Internet of Things (IoT) has continued to increase in popularity in recent years, leading to a rush of new IoT devices and IoT environments with many diverse devices. IoT devices can now be used to provide services to users in home, industrial, city, and vehicular deployments. Figure 1 presents the typical architecture of an IoT device and an example IoT app. As shown on the right of the figure, an IoT device often consists of (i) a set of sensors such as location, temperature, and light sensors, (ii) an auxiliary Micro-Controller Unit (MCU) to read the raw sensor values, (iii) low power CPU cores, (iv) network interfaces to communicate the user-level events to end-users, and (v) a battery or a power supply unit to power all these components. The left side of the figure illustrates an example IoT app of how these devices are automated to interact with the environment. The app uses a temperature sensor to read the current temperature of the home, opening the window and notifying the user when it gets too hot and closing the window and turning on the heater when it gets too cold.

Faults in IoT Systems. IoT devices that interact with physical environments must maintain high dependability for practical deployment. However, they are prone to faults caused by factors such as power outages [1] and network disruption [2]; frequency of faults is also increased due to IoT device design constraints such as low computation capacity [3] and small batteries [4]. Additionally, devices face complex issues, such as disruptive environmental conditions like weather and collisions [5], user error during deployment in environment [1], and flaws in the device hardware and software [6]. A study shows that devices in smart homes can experience faults more than 4 hours a day due to power loss, network disruption, and hardware failure [4], and a more recent work shows that a temperature sensor experiences more than 15% faulty temperature readings [2].



```
Devices
t: temperature sensor
w: window actuator
h: heater
Events
When:
 temperature changes
Computation
temp = t.currentTemperature
if (temp >= threshold){
    w.open()
    sendSMS(user_phone)
}
if (temp < threshold){
    window.close()
    heater.set(70)
}
```

Fig. 1: IoT system architecture and an example IoT app.

Faults can manifest in several forms with different consequences. We divide faults into three categories. *Fail-stop* faults occur when a device stops functioning and is unresponsive to external requests. For example, the power loss of a device causes its sensor reading to stop and its actuations to fail. *Non-fail-stop* faults do not cause a device to function incorrectly and can appear in a variety of ways with different effects [7]. In detail, they happen when a device manifests an incorrect state, either producing incorrect sensor reads or failing to properly follow actuation commands. For example, a software error could cause the device to rapidly change between two states, which could trigger a large number of incorrect actuations from apps. *Cascading faults* happen when a faulty device in an app incorrectly triggers an event in another app, causing the initial fault to cascade through the system. For example, if a fault in the temperature sensor causes the app in Figure 1 to turn on the heater, a cascading fault happens when another energy-saving app turns off the air conditioner as a result.

Motivation for Automated IoT Fault Handling. As IoT systems are primarily autonomous, there is generally little interaction and oversight from users. When faults happen, asking users to handle faults manually would be impractical and lead to lengthened response time. Therefore, it is highly desirable to have a largely automated IoT fault identification and handling system. Such an automated system would reduce dependency on swift user interventions and increase the reliability of IoT services. Automatic identification of a variety faults is already present in IoT, but the ability to quickly handle the majority of faulty cases is still absent. Our motivation for IoTRepair is to provide a handling solution that will eliminate or mitigate the various effects of these faults in IoT platforms.

Fault Tolerance in IoT Systems. To understand state of the art in automated IoT fault detection and handling, we have studied both current major IoT platforms and past research. We aim at characterizing their fault identification and handling methods

	Fail-Stop Faults			Non-Fail-Stop Faults			
IoT Platform	Comm.	Power	Crit Error	Outlier	Stuck-at	High Var.	Spike
SmartThings [8]	⊘	⊘	⊘	⊗	⊗	⊗	⊗
OpenHab [9]	⊘	⊘	⊘	⊗	⊗	⊗	⊗
Vera [10]	⊘	⊘	⊘	⊗	⊗	⊗	⊗
Homekit [11]	⊖	⊖	⊖	⊗	⊗	⊗	⊗
Wink [12]	⊖	⊖	⊖	⊗	⊗	⊗	⊗
AndroidThings [13]	⊕	⊕	⊕	⊗	⊗	⊗	⊗
IoTivity [14]	⊖	⊖	⊖	⊗	⊗	⊗	⊗
KaaIoT [15]	⊖	⊖	⊖	⊗	⊗	⊗	⊗

⊘ Silent ⊖ Generic Error ⊕ Detailed Error ⊗ Undetected

TABLE I: IoT platforms' response to different device faults. *Undetected* means that the fault is not recognized by the platform, *silent* means no message sent to apps, *generic error* means message does not contain fault information, and *detailed* error means message contains fault information.

(See Table I). In general, current IoT platforms do not give the means for apps to handle non-fail-stop faults since these faults are not even detected by these platforms. For fail-stop faults, only Android Things gives enough information to handle faults effectively; however, developers still have to manually add code to handle these faults. Other platforms either provide only generic error messages, which make it hard for apps to differentiate types of fail-stop faults, or stay silent and rely on developer to attempt to implement ad-hoc detection. Finally, none of these platforms can handle cascading faults.

A large body of work on fault identification has been proposed targeting specific environments and fault types e.g., [16], [17], [18], [19], [20] for wireless sensor networks, and [21], [22], [23] for smart home devices. Yet, there is very little research on automated IoT fault handling after a fault is identified. One system [24] uses imperfect replication in redundancy-free UAV sensors to allow near-correct device operation during sensor failure, but it is specific to UAV sensors and does not repair faults. Most fault tolerance methods for commodity IoT use replication; for example, Rivulet [25] removes the edge device as a single point of failure by distributing communication relays to capable devices, though it does not handle the more common sensor and actuator faults. Transactuations [26] address transaction flaws in IoT and prevents physical device states from losing synchronization with stored variables. Yet, it does not repair faults or correct state errors that occur before the fault is detected. To the best of our knowledge, there exists no system that automates fault identification and handling for diverse IoT device faults.

Challenges in IoT Fault Handling. Compared to the fault handling in other computing platforms such as distributed systems and cloud services, IoT systems raise unique challenges [27].

- 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.
- 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.

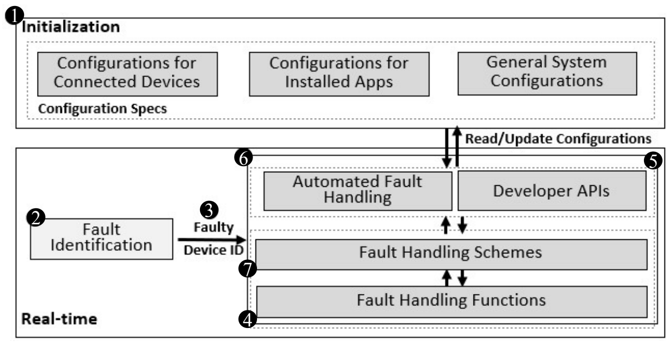


Fig. 2: Overview of IoTRepair architecture

- A fault-handling mechanism at installation and run-time 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 available, and the fault may require real-time response. IoTRepair provides automated fault handling functionality with user configurations and developers customizing with APIs to automate the fault-handling functions in apps.

II. IoTRepair Overview

We design and develop IoTRepair, a fault-handling system for IoT. IoTRepair integrates with fault identification modules to track faulty devices and provide a fault-handling library with a set of functions to handle diverse device faults. IoTRepair can be integrated into a cloud platform or edge device of an IoT system to ensure autonomous fault handling. Figure 2 presents IoTRepair architecture with components explained below:

- IoTRepair includes a *configuration file* created during an initialization phase and can later be customized by IoTRepair, users, and developers (❶). The file includes a set of parameters to manage and control IoTRepair functionality based on the device types and apps. For instance, parameters define whether application suppression is enabled and the upper bound on fault identification module identification time. These parameters enable IoTRepair to be flexible in addressing a variety of faults in diverse environments.
- IoTRepair integrates with a custom, separately installed fault-identification module to track faulty devices (❷). While there is a number of fault-identification techniques available, IoTRepair requires only the device ID be provided for each identified fault to automate fault handling (❸).
  IoTRepair includes a fault-handling library consisting of a set of *functions* to mitigate and repair different fault types, such as device restart, retry, and checkpoint (❹). Each function uses the parameters in the configuration file while attempting to handle a fault. Developers can also implement their fault handling logic by using provided APIs to modify the configuration file or adding try/catch block around device interactions in app source code (❺).
- IoTRepair automates fault handling through the configuration file, fault handling functions, and schemes to minimize the developer effort (❻). *Schemes* are organized lists of functions which invoke a set of fault-handling functions in a specific order (❼). Once a faulty device is identified, IoTRepair performs device suppression, which blocks polls to the faulty

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){...}`
`None 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 are arguments for device/app configuration parameters.

TABLE II: Fault handling functions prototypes.

device and commands sent to faulty devices. It then executes a scheme defined for a device in the configuration file.

Design Requirements. To fully provide the functionality above, IoTRepair requires the ability to query list of connected devices, poll device states, query installed apps, send actuation commands, create and manage a configuration file, expose library functions to apps, and hook fault identification messages. The installation space must be between the apps and devices so that events and actuations can be interrupted. IoTRepair also requires the ability to query types and capabilities at initialization at least to create configuration file.

Configuration File. There are three types of information in the configuration file: (i) general parameters, defining the upper bound on how long fault identification module takes to identify a fault, as well as specifications for checkpoint cleanup and replicated device detection, each specified by the user. (ii) device-based parameters, defining the list of devices running in the system and the parameters (e.g., which fault-handling scheme (detailed in Section III-D)) for each device; default configurations are generated for each device connected to the edge device by selecting from a list of default configurations that best matches the device type; they can be modified by the user, by the automated handler at runtime, or by developers by making API calls in their apps; (iii) app 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.

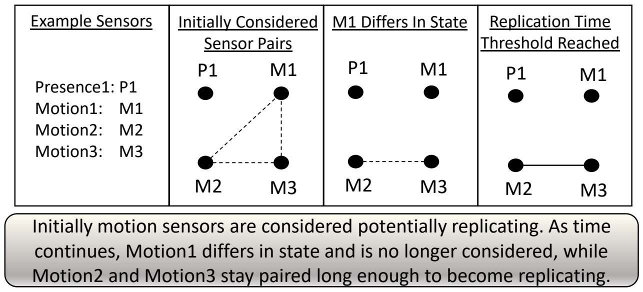
III. Fault-Handling Functions

We introduce a set of *functions* to handle faults in three groups (See Table II). We then discuss four built-in fault-handling schemes to automate fault handling.

A. Device-based Functions

Device-based functions implement *isolated* fault handling, which does not consider the overall state of the system (i.e., state of all connected devices), and acts based only on the state and configuration of the faulty device.

Activate Redundant Device. We define replicating devices to be of two types: (a) a duplicate of the primary device; (b) a device that provides similar capabilities; for example, a camera that can detect motion to replicate a motion sensor. IoTRepair currently only supports type (a) replication, yet imperfect replication could be easily integrated into IoTRepair through using device fingerprints [28] and physical equations [24].



Initially motion sensors are considered potentially replicating. As time continues, Motion1 differs in state and is no longer considered, while Motion2 and Motion3 stay paired long enough to become replicating.

Fig. 3: Example of automatic replication detection

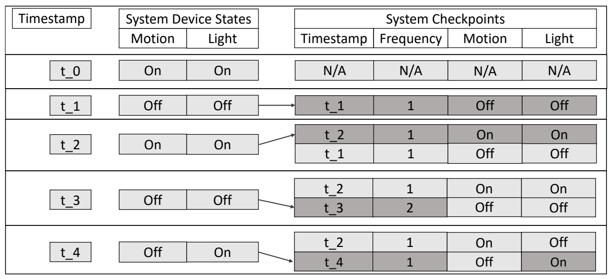
The function `ActivateRedundantDevice` allows the system to continue to run unaffected by a detected fault, so long as the configuration file specifies the replicating device ID if one exists. If there is, it redirects polls and actuation commands from the faulty device to the replicating device. Using this method, IoTRepair does not rely on native replication support from the platform. To ease the burden of writing replicating devices in the configuration file, IoTRepair 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, these devices are identified to be replicating. Figure 3 illustrates how two co-located motion sensors would be detected as replicating, as their states match consistently, while an unrelated presence sensor and distant motion sensor are discarded.

Retry Device State. `Retry` aims to prevent overhandling short, transient faults by delaying functions like restart that would waste time and energy, or rollback which would unnecessarily revert the system state. `Retry` continuously polls the faulty device's state until either the fault is resolved or the retry limit for this device in the configuration file is reached. Optionally if the faulty device is an actuator and the desired state is known, corrective actuations can be attempted in between polls which can resolve faults caused by temporary disruptions.

Software and Hardware Restart. Restarting can be useful to remediate software bugs or hardware failures. `SoftwareRestart` and `HardwareRestart` functions send a signal to the device to initiate a restart. This relies on the device having a built-in capability for the appropriate type of restart, so some device schemes may not be able to utilize this function. Many IoT market devices such as Honeywell Z-Wave Thermostat [29] and open-source platforms like Arduino implement restart. IoTRepair will wait for a response signal and monitor the device state, and if the restart does not complete, it retries a number of times based on configuration, and then the function confirms whether fault is resolved.

B. Environment-based Functions

Environment-based functions implement *linked* fault handling to mitigate cascading faults. The functions below consider



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. 4: Illustration of how Checkpoints are taken in a system.

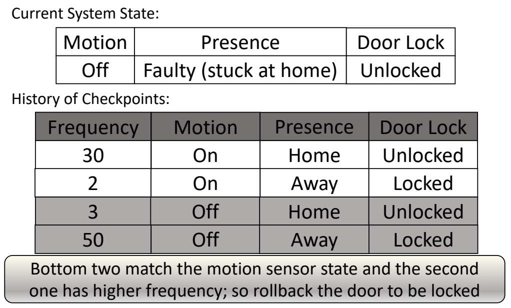
the state of all devices in an environment.

Checkpoint. `Checkpoint` stores a collection of all device states in a pending queue at the time it is called. The automated fault handler invokes the checkpoint function after every actuation where no subscribing application initiates an actuation based on the new state. Some time must pass before a checkpoint is considered valid and appended to the history log. The delay period is determined by the upper bound on the fault-detection time, which must be provided by the fault identification module or set to default. These restrictions prevent creating a checkpoint that triggers actuations or where a fault was present.

We divide device states into *sensor* states and *actuator* states. Sensor states are read-only states that collectively represent the state of the environment. Actuator states can be read and modified through actuation, hence they are the states that can be rolled back. We assume an app follows the well-known sensor-computation-actuator structure: modifying actuator states based on sensor states. 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 desirable actuator states by looking up the history according to the current sensor states.

Valid checkpoints are stored in a history log, which holds checkpoints and their frequencies. Frequency is essential when implementing fail-norm rollback (discussed soon) which rolls back to the checkpoint most likely to match the current physical environment. Only the most recent checkpoint for a given set of actuator states is stored in the log. Only storing the most recent sensor states for a set of sensor states keeps the log from exploding in size. For this reason, checkpoints are removed from history if they remain unused for a configured duration.

Figure 4 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 time-stamps $\mathtt{t}_1$ through $\mathtt{t}_4$, with initial time $\mathtt{t}_0$ 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 and updating after each actuation. The first two checkpoints at times $\mathtt{t}_1$ and $\mathtt{t}_2$ are new states, since there are no existing checkpoints that match the sensor states. For these new checkpoints, the time they occurred and current device states are recorded, and the frequency is set to 1. The checkpoint taken at $\mathtt{t}_3$ has the same device states as an existing checkpoint; so it updates the checkpoint time-stamp and the frequency. At time $\mathtt{t}_4$, a checkpoint that matches the sensor states is taken, but the actuator states do not match



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 match the motion sensor state and the second one has higher frequency; so rollback the door to be locked

Fig. 5: Example of Fail-norm Rollback.

those of an existing checkpoint; so the matching checkpoint's time-stamp is updated, actuator states are overwritten, and frequency is reset.

Rollback. `Rollback` performs a series of actuations to match the system state to a checkpoint selected based on configuration.

We have designed three techniques for choosing the best checkpoint to target for rollback. *Most Recent* targets the checkpoint with the latest timestamp. This technique is suitable for a system where faults can be detected quickly or system state changes slowly. *Fail-safe* only consider checkpoints where all actuators are in their configured fail-safe state. From this reduced list, a checkpoint whose sensor states match the current sensor states in the system is selected. As the states of faulty sensors cannot be trusted, their sensor states cannot be considered when determining if a checkpoint matches the current sensor states. If this causes more than one matching state, then the frequency is used as a tiebreaker. *Fail-norm* finds checkpoint matches in the same fashion as fail-safe, but does not reduce the list based on fail-safe configurations first.

Rollbacks can be dangerous, as a partial rollback can result in an invalid transition and enter an incorrect system state. For this reason, our rollback aborts if any actuator that needs to be changed is currently known to be faulty. We note that this captures the heterogeneous nature of IoT devices when compared to similar functions in distributed systems [30] and cyber-physical systems [31]. Additionally if rollback completes, faulty sensor values are set to their values in the checkpoint until the device is repaired or another rollback occurs.

Figure 5 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 5, 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.

C. Auxiliary Functions

Auxiliary functions are not directly used during fault handling but can be invoked by the automated handler or app developers to modify handler configuration.

Scheme	Function Ordering					
`Conservative`	1	2	3	4	5	6
`Transient-resistant`	1	3	4	5	6	∅
`Long-Restart`	1	2	5	3	4	6
`Time-sensitive`	1	5	2	3	4	6

TABLE III: Execution order of functions in schemes. (1) Replicate; (2) Retry; (3) Software Restart; (4) Hardware Restart; (5) Rollback; (6) Notify User.

Device Suppression. Device Suppression is not a library function, but rather a capability implemented in an edge device (e.g., hub). Through this capability, an 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 in other devices.

Application Suppression. Application suppression is also a capability implemented in the edge device. The edge suppresses all events sent to an app and all actuations triggered by an app for every app that subscribes to a faulty device and has application suppression enabled. This is useful for apps that may put the system into an unsafe state if the state of one of its dependent devices is unknown. For example, without suppression, an app may incorrectly open a window if the temperature gets too hot because a faulty presence sensor is falsely reporting that the user is home.

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, the values in the configuration file is updated. App developers can also call this function with the same parameters to customize the configuration file for their requirements.

D. Built-in Fault-Handling Schemes

We introduce a set of built-in schemes to automate the functions introduced above (See Table III). The schemes modify the execution order of the functions to address environmental requirements such as safety and security. We chose to use the device type as a primary determinant of each scheme as IoT devices 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 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. *Conservative* scheme fits in environments where there are no strict time requirements and energy conservation is the primary goal. *Transient-resistant* scheme aims at devices that are unlikely to experience 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. *Long-restart* scheme is used in devices that have excessively long software and hardware restart times, such as security cameras and a smart refrigerator. *Time-sensitive* aims to return the system to the desired state as the device impacts the safety, for instance, when the security system is unresponsive. This scheme fits industrial or vehicle environments where system integrity is the top priority.

IV. Implementation

We implemented 11 distinct IoT apps that automates 17 IoT devices in a simulated smart home. In this section, we begin by introducing the simulated IoT apps, and we then present fault injection and fault identification techniques required for evaluation of IoTRepair.

Implementation Setup. We deployed a set of Arduino devices connected to and run in parallel on an AVR Arduino Mega 2560 Rev 3 Board [32]. We use the sensor data to generate a realistic trace for 17 simulated devices that represents activity in a smart home, which then drives a trace-driven simulation.

IoT Devices and Applications. We deployed 8 Arduino devices, 6 sensors and 2 actuators to observe behavior for generating simulated devices. Table IV shows the description of applications. These apps are selected to cover all spectrum of home environment functionality, including green living, convenience, home automation, security, and safety.

Fault Injection. We use fault injection to evaluate the effectiveness of different fault handling methods. The injected fault overwrites a trace sensor value with the faulty value, either an incorrect real state for non-fail-stop faults or a null state for fail-stop. We ensure full coverage of device type and remediation effectiveness. The fault type, fault length, and injection time are injected within reasonable distributions because these values impact the number of incorrect states, not the way the faults are handled. We only consider single faults–when no more than one fault is in the system at any given time. Some examples include a device experiences a power failure, environmental damage, or communication interruption.

Fault Identification. We use a perfect identification module that identifies faults instantly and reports the faulty Device ID. This enables us to show the effectiveness of IoTRepair at mitigating the effects of injected faults.

V. Evaluation

We present our evaluation on the performance of IoTRepair's automated handler (Section V-A). Furthermore, we assess the effectiveness of IoTRepair on injected faults (Section V-B).

A. Fault Handling Latency

We evaluate the automated handler latency for different fault-handling schemes. Here latency refers to the running time of each scheme, which we average across possible executions. We examine the logic to track CPU operations and device interaction time from device manuals to get timing information. Time spent on CPU operations is then calculated based on the processing speed of a representative SmartThings IoT Hub [33].

We evaluate latency of each scheme introduced in Section III-D. We calculate the running time across all possible permutations of devices, function parameters, and fault types to obtain the average time each scheme takes to handle each fault type. The results are shown in Figure 6, where the average time to handle a fault is illustrated for each scheme, alongside the average time each scheme takes to rollback. Optimal assumes perfect knowledge for choosing functions. Using the proper scheme results in repairs or mitigation occurring over twice as fast. Faults underlined in red indicate IoTRepair cannot repair the fault, so rollback time is more important to mitigate errors. For these faults, the handling time indicates how long it takes

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

TABLE IV: IoT apps and their descriptions developed to evaluate IoTRepair.

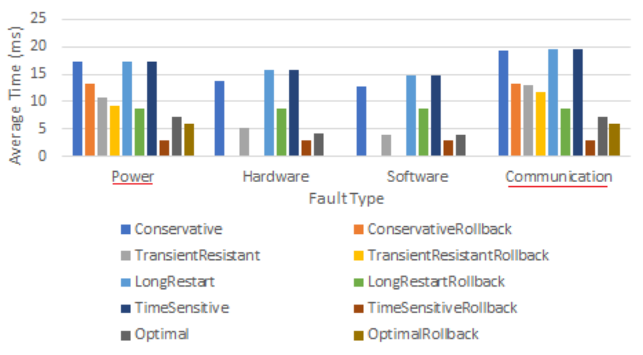


Fig. 6: Avg. time in msecs each scheme takes to handle different types of faults and to rollback.

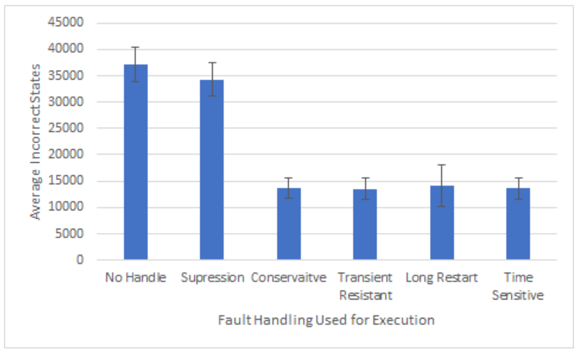


Fig. 7: Avg. Incorrect States for each fault handling type across 100 executions of the simulation.

the handler to recognize that the fault is unfixable and notify the user. We also found that IoTRepair is most effective at handling faults that last longer than ten msecs. However, suppression and replication help mitigate errors for shorter faults.

B. *Effectiveness*

We measure the number of incorrect device states that occur over different executions, which show the effectiveness of IoTRepair in mitigating faults. A device is in an incorrect state when it deviates from the state it was in at that time in the faultless execution. This metric is a conservative definition since a fault handler may correct a device to a state acceptable to users though it differs from the faultless execution. Yet, we use the incorrect-state metric since it does not require an acceptable-state definition, which is user and app-specific.

We execute the simulation: ⓐ when no faults are present, ⓑ when faults occur without a fault-handling system, ⓒ with device suppression, and ⓓ using IoTRepair. The baseline of the correct device states is obtained through ⓐ. We evaluate the effectiveness of the four fault-handling schemes discussed in Section III-D, and compare the number of incorrect states in each execution. We detail our results when a single fault occurs at a time and give a comprehensive analysis of injected faults and their effects in each execution in our extended paper [34]. Figure 7 presents the number of incorrect states over different executions. Most schemes had similar performance, but transient-resistant scheme was the most effective scheme for handling single faults. It shows a 63.51% decrease in incorrect states over no handler and a 60.43% decrease over device suppression; this represents significant mitigation of the effects of faulty devices.

VI. Limitations and Discussions

The evaluation of IoTRepair is limited for several reasons. The first and largest limitation is it is entirely simulation based, and while the simulation is based off of real Arduino devices, we cannot guarantee a physical deployment would have the same results. Second, for every run, every device uses the same scheme. This means improvements in one device can be canceled out by another device that performs better under a different scheme. Finally, we did not design our experiments to target the key distinguishing features of the schemes, such as adding frequent short transient faults to thwart the transient-resistant scheme. We plan to conduct more extensive experiments in a true physical deployment.

VII. Conclusions

We presented the current flawed state of fault handling in IoT and the need for a fault handling solution to prevent and mitigate device failures. We have designed and developed IoTRepair[1], a fault handler library that integrates with edge devices to meet the requirements for diverse IoT environments. IoTRepair includes runtime replication detection, fault history tracking, a set of fault-handling functions, and schemes automating the execution of functions in customized orders.

VIII. Acknowledgements

This work was supported by NSF grants CNS-1900873 and CNS-1629915. The authors would also like to thank the anonymous reviewers and the shepherd Marisol Garcia-Valls for their help in refining the paper.

[1]An extended version of this paper is available with substantially more methodology description, performance evaluation details, and commentary [34].

References

[1] T. Kavitha and D. Sridharan, "Security vulnerabilities in wireless sensor networks: A survey," *information Assurance and Security*, 2010.

[2] C.-S. Shih, J.-J. Chou, N. Reijers, and T.-W. Kuo, "Designing cps/iot applications for smart buildings and cities," *IET Cyber-Physical Systems: Theory & Applications*, vol. 1, no. 1, pp. 3–12, 2016.

[3] D. G. Padmavathi, M. Shanmugapriya *et al.*, "A survey of attacks, security mechanisms and challenges in wireless sensor networks," *arXiv preprint arXiv:0909.0576*, 2009.

[4] T. W. Hnat, V. Srinivasan, J. Lu, T. I. Sookoor, R. Dawson, J. Stankovic, and K. Whitehouse, "The hitchhiker's guide to successful residential sensing deployments," in *Conference on Embedded Networked Sensor Systems*, 2011.

[5] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler, "Lessons from a sensor network expedition," in *European Workshop on Wireless Sensor Networks*, 2004.

[6] E. Elnahrawy and B. Nath, "Cleaning and querying noisy sensors," in *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, 2003, pp. 78–87.

[7] K. Ni, N. Ramanathan, M. N. H. Chehade, L. Balzano, S. Nair, S. Zahedi, E. Kohler, G. Pottie, M. Hansen, and M. Srivastava, "Sensor network data fault types," *ACM Transactions on Sensor Networks (TOSN)*, 2009.

[8] "Samsung SmartThings add a little smartness to your things," https://www.smartthings.com/, 2018, [Online; accessed 9-August-2018].

[9] "OpenHAB: Open source automation software," https://www.openhab.org/, 2018, [Online; accessed 9-Aug-2018].

[10] "Vera Control: Smart home control," https://support.getvera.com/customer/en/portal/articles/1710500-plugin-development, 2018, [Online; accessed 15-August-2018].

[11] "Apple: Homekit," https://developer.apple.com/homekit/, 2018, [Online; accessed 15-August-2018].

[12] "Wink: A simpler smart home," https://winkapiv2.docs.apiary.io/#reference/a-restful-service, 2018, [Online; accessed 15-August-2018].

[13] "Google: Android things," https://developer.android.com/things/, 2018, [Online; accessed 15-August-2018].

[14] "IoTivity," https://iotivity.org/, 2018, [Online; accessed 15-Aug-2019].

[15] "KaaIoT," https://kaaproject.org/, 2018, [Online; accessed 15-Aug-2019].

[16] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," in *ACM International conference on Embedded networked sensor systems*, 2005.

[17] J. Ye, S. Dobson, and S. McKeever, "Situation identification techniques in pervasive computing," *Pervasive and mobile computing*, 2012.

[18] I. M. Atakli, H. Hu, Y. Chen, W. S. Ku, and Z. Su, "Malicious node detection in wireless sensor networks using weighted trust evaluation," in *Simulation multiconference*, 2008.

[19] L. Fang and S. Dobson, "Unifying sensor fault detection with energy conservation," in *Workshop on Self-Organizing Systems*, 2013.

[20] A. B. Sharma, L. Golubchik, and R. Govindan, "Sensor faults: Detection methods and prevalence in real-world datasets," *ACM Transactions on Sensor Networks (TOSN)*, 2010.

[21] K. Kapitanova, E. Hoque, J. A. Stankovic, K. Whitehouse, and S. H. Son, "Being smart about failures: assessing repairs in smart homes," in *ACM Conference on Ubiquitous Computing*, 2012.

[22] S. Munir and J. A. Stankovic, "Failuresense: Detecting sensor failure using electrical appliances in the home," in *IEEE Mobile Ad Hoc and Sensor Systems*, 2014.

[23] P. A. Kodeswaran, R. Kokku, S. Sen, and M. Srivatsa, "Idea: A system for efficient failure management in smart iot environments," in *ACM MobiSys*, 2016.

[24] Z. Tu, F. Fei, M. Eagon, X. Zhang, D. Xu, and X. Deng, "Redundancy-free UAV sensor fault isolation and recovery," *arXiv preprint:1812.00063*, 2018.

[25] M. S. Ardekani, R. P. Singh, N. Agrawal, D. B. Terry, and R. O. Suminto, "Rivulet: a fault-tolerant platform for smart-home applications," in *ACM Middleware Conference*, 2017.

[26] A. Sengupta, T. Leesatapornwongsa, M. S. Ardekani, and C. A. Stuardo, "Transactuations: where transactions meet the physical world," in *USENIX Annual Technical Conference*, 2019.

[27] D. Terry, "Toward a new approach to IoT fault tolerance," *IEEE Computer*, 2016.

[28] J. Han, A. J. Chung, M. K. Sinha, M. Harishankar, S. Pan, H. Y. Noh, P. Zhang, and P. Tague, "Do you feel what I hear? enabling autonomous iot device pairing using different sensor types," in *IEEE Symposium on Security and Privacy (SP)*, 2018.

[29] "Honeywell z-wave thermostat," https://tinyurl.com/yyzknakl, 2017, [Online; accessed 22-April-2019].

[30] A. D. Kshemkalyani and M. Singhal, *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.

[31] F. Kong, M. Xu, J. Weimer, O. Sokolsky, and I. Lee, "Cyber-physical system checkpointing and recovery," in *International Conference on Cyber-Physical Systems (ICCPS)*, 2018.

[32] "Arduino mega 2560 rev3," https://store.arduino.cc/usa/mega-2560-r3, 2019, [Online; accessed 19-April-2019].

[33] "Smarthings hub," https://www.smartthings.com/products/smartthings-hub, 2018, [Online; accessed 10-September-2019].

[34] M. Norris, Z. B. Celik, P. McDaniel, G. Tan, P. Venkatesh, S. Zhao, and A. Sivasubramaniam, "IoTRepair: Systematically addressing device faults in commodity IoT," *arXiv preprint arXiv:2002.07641*, 2020.