Transactuations: Where Transactions Meet the Physical World

Aritra Sengupta, Tanakorn Leesatapornwongsa, and Masoud Saeida Ardekani,
Samsung Research; **Cesar A. Stuardo,** ***University of Chicago***

https://www.usenix.org/conference/atc19/presentation/sengupta

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Transactuations: Where Transactions Meet the Physical World

Aritra Sengupta
Samsung Research

Tanakorn Leesatapornwongsa *
Samsung Research

Masoud Saeida Ardekani †
Samsung Research

Cesar A. Stuardo ‡
University of Chicago

Abstract

A large class of IoT applications read sensors, execute application logic, and actuate actuators. However, the lack of high-level programming abstractions compromises correctness especially in presence of failures and unwanted interleaving between applications. A key problem arises when operations on IoT devices or the application itself fails, which leads to inconsistencies between the physical state and application state, breaking application semantics and causing undesired consequences. Transactions are a well-established abstraction for correctness, but assume properties that are absent in an IoT context. In this paper, we study one such environment, smart home, and establish inconsistencies manifesting out of failures. We propose an abstraction called *transactuation* that empowers developers to build reliable applications. Our runtime, *Relacs*, implements the abstraction atop a real smart-home platform. We evaluate programmability, performance, and effectiveness of transactuations to demonstrate its potential as a powerful abstraction and execution model.

1 Introduction

Building reliable IoT applications that interact with the physical world on top of existing solutions is difficult. Current IoT solutions (e.g., Smartthings [14] and OpenHAB[12]) provide simple abstractions that allow developers to easily read sensors and actuate actuators. However, they lack high-level abstractions for writing reliable and fault-tolerant applications that can tolerate different types of failures that might happen. Therefore, application programmers need to implement tedious and error-prone code for not only handling all kinds of failures happening in the physical world, but also to guarantee consistency between operations on application states (called soft states hereafter) and states of IoT devices (called hard states). For instance, an actuation to turn on an alarm might fail while the alarm state in an application might have been set to true.

The use of serverless functions as a de facto platform for running IoT applications has exacerbated the reliability issues of these applications even further. This is because serverless computing infrastructure can terminate running applications at any point [2]. This again leaves incomplete operations on some hard states (e.g., lock all doors) inconsistent with an operation on soft state inside the application (e.g., set the home state to `safe` after all doors are locked).

Transactions seem like the right mechanism for addressing the above issues. Interestingly though, a transactional abstraction cannot fix these issues because of intrinsic properties of IoT devices (and their associated hard states). A transactional abstraction is ideal for ensuring isolation and all-or-nothing guarantees among soft states. Moreover, a transactional system can easily rollback soft states without other transactions or users noticing effects of a rolled back transaction. However, rolling back a hard state has consequences. The state might have already been observed by a user and rolling it back may be undesirable. Or even worse, some states cannot be rolled back (e.g., undoing actuation of a water dispenser).

This paper proposes an abstraction called *transactuation*. Transactuations hide the complexity of handling various failures and allow developers to easily maintain soft states to be consistent with respect to reads and writes to hard states – states of sensors and actuators. Objectively, transactuations allow a developer to specify dependencies among operations on soft and hard states along with a *sensing/actuating policy* which specifies the conditions under which soft states can commit despite failures.

We provide a runtime system called *Relacs* that implements the abstraction for the smart home environment. Relacs transforms an application into a serverless function, and reliably executes the application in the cloud while enforcing transactuation specific semantics. We note that while the focus of this paper is on smart homes, the transactuation abstraction is not particularly specific to smart homes, and can be applied to other IoT environments as well.

*Work done at Samsung Research America. Now at Microsoft Research.
†Work done at Samsung Research America. Now at Uber Technologies.
‡Work done at Samsung Research America.

Concretely, this paper has the following contributions:

1. *Study of smart-home applications.* Using static analysis, we conduct a comprehensive study of smart-home applications written for two popular platforms [12, 14] and identify drawbacks of existing platforms in writing reliable and fault-tolerant applications (Section 3).

2. *Transactuations.* We present our abstraction that allows developers to simply write reliable IoT applications. Transactuations preserve the dependencies between operations on hard states and soft states, which when broken, break application semantics (Section 4).

3. *Relacs.* Our runtime, Relacs, enforces a serializable execution of transactuations without rolling back hard states (i.e., states of actuators) while enforcing the specified sensing and actuating policies (Section 4 and Section 5).

4. *Evaluation.* We evaluate representative smart-home applications to reveal the correctness issues due to lack of appropriate abstractions. Our evaluation further demonstrates that (a) Transactuations are an effective high-level abstraction for building reliable IoT applications and reduce lines of code significantly compared to manually handling failures. (b) Relacs guarantees reliable execution of transactuations while imposing reasonable overheads over a baseline that does not provide consistency between operations on hard states and soft states (Section 6).

2 Background & Model

In this section, we first review existing smart-home platforms and their programming models. We focus on smart homes as a case study of class of IoT environments that deal with real world state since many smart home applications and platforms are publicly available. We then discuss different types of failures that occur in IoT environments.

2.1 Smart-home Platforms

To setup a smart home, a user installs centralized gateways, called smart-home hubs or simply hubs, to connect in-home devices (e.g., light bulbs, outlet strip, and motion sensor) that typically communicate through low-energy wireless protocols (e.g., Zigbee [17], ZWave [16], and Bluetooth Low Energy [4]). The user then installs smart-home applications to create her desired home automation. For instance, to turn on a balcony light when motion is detected outside.

Currently, cloud-centric smart-home solutions (e.g., Smartthings [14]) are the most widely used architecture [28]. In this model, a hub is only responsible for collecting device events, and forwarding them to the cloud, where applications run. The applications running in the cloud then process events and send actuation commands back to the hub, which forwards the commands to corresponding devices. An alternative architecture is to run applications inside hubs. OpenHAB [12] follows this hub-centric approach.

```
1  preferences {
2    input(sensor, "capa.co2", req:true)
3    input(switches, "capa.switch", multi:true)
4    input(level, "number", req:true)
5  }
6  def initialize() {
7    state.active = false;
8    subscribe(sensor, "co2", handleLevel)
9  }
10 def handleLevel(evt) {
11   def co2 = sensor.currentValue("co2");
12   if(co2 >= level && !state.active) {
13     switches.each { it.on(); }
14     state.active = true;
15   } else if(co2 < level && state.active) {
16     switches.each { it.off(); }
17     state.active = false;
18   }
19 }
```

Listing 1: CO_2 vent application that turns exhaust fans on when CO_2 level is high and turns off otherwise.

2.2 Programming Model

In most smart-home platforms, an application is written in a trigger-action programming model [45] where an application comprises event handlers. Handlers can subscribe to changes in sensor/actuator states, updates to shared states, or timer-based events. Handlers can issue the following operations:

- Hard read: reading sensor/actuator values.
- Hard write: sending actuation commands to actuators.
- Soft read: reading application states from shared storage.
- Soft write: writing application states to shared storage.

In the remainder of this section, and for simplicity, we solely detail SmartThings [14] programming model. Yet, we note that other platforms have very similar constructs.

SmartThings uses capabilities, attributes, and commands to manage devices. Each device has one or more capabilities, and each capability has one or more associated attributes and commands. For example, a smart light bulb has two capabilities, switch and color. The switch capability allows an application to control the bulb status via `on`/`off` commands. The color capability has three attributes, color, hue and saturation that can be controlled via `setColor`, `setHue`, and `setSaturation` commands.

Listing 1 shows a SmartThings application, named CO_2 vent, written in the Groovy language [5]. It reads CO_2 level from sensors, and turns on an exhaust fan if the level is high. Similarly, it turns off an exhaust fan if the level is low. A developer first declares mapping of variable names to capabilities in the preference section (lines 1-5). Consequently, a variable is mapped to an array of devices with the same capability. For example, variable `switches` (line 3) gets mapped to an array of exhaust fans having the switch capability.

A developer then subscribes event handlers to value changes of some capabilities or timer schedules. In line 8, she subscribes an event handler called `handleLevel` to co2 capability. Observe that inside the handler, she can perform

hard read on sensor data (`sensor.currentValue()` in line 11) and soft read on shared states (reading `state.active` in line 12 and 15). Also, the developer can issue hard writes to list of actuators (line 13 and 16). She can also perform soft writes to application states (assignments to `state.active` in line 14 and 17).

2.3 Failures in IoT Environments

Previous work [20, 33] have shown a variety of failures in IoT environments. For instance, hubs can fail due to plug disconnection, hardware failure, and driver crash. IoT devices can fail due to battery drainage, plug disconnection, and failure in a sensor subsystem. Additionally, network loss occurs due to RF interference, concrete slab flooring and copper siding. These failures lead to permanent or intermittent unavailability of devices in an IoT environment.

Although, these failures are common, existing platforms do not provide a simple way to detect and handle them. A failed hard read can produce a null or stale value that a developer needs to handle or explicitly validate its timestamp (freshness). Detecting a failed hard write is even more difficult due to the asynchronous nature of IoT programming model. For instance, a developer needs to subscribe to an event triggered by a hard write, and periodically check if the event is fired. As shown in other systems [30, 35, 37], inserting failure detection and handling code for asynchronous environments is challenging and error prone. Moreover, due to inherent event-driven concurrency in applications, it is notoriously difficult to prevent interleaving and concurrency-related bugs in IoT platforms [40].

3 Problem Study

Existing smart-home solutions do not guarantee any consistency between soft reads/writes (i.e., reads/writes from/to shared storage) and reads/writes to hard states (i.e., sensor reads and actuation commands sent to actuators) in case of failures. Application developers need to carry the burden and ensure the correctness of an application when a failure occurs.

In this section, we present a systematic study of open source smart-home applications, using static analysis, in order to unearth various inconsistencies, that surface under failure, between operations on soft and hard states.

3.1 Inconsistency

Listing 2 shows a simplified code excerpt from a smart security application. This application associates a soft state named `alarmActive` with the status of an alarm. If the application detects an intruder when the alarm is not active, it activates the alarm and sets `alarmActive` to `true`. However, an inconsistency arises if the alarm is not activated properly. For example, RF interference may cause an actuation command

```
1    def intruderMotion(evt) {
2      ...
3      if (isIntruder(evt) && !state.
           alarmActive) {
4        alarm.strobe();
5        state.alarmActive = true;
6      }
7      ...
8    }
```

Listing 2: A simplified code excerpt from Smart Security application that detects an intruder using sensors, and activates an alarm if it has not been activated previously.

to be lost. This problem is so common that some brands (e.g., Fortrezz [15]) give warnings regarding RF interference, and explicitly ask consumers to not use the alarm in life supporting situations. Observe that even though `alarmActive` is set, the states of the physical world and application have diverged. Further, if the sensors detect the intruder again, the application will not retry to activate the alarm because as per the application's state the alarm is ringing. Clearly, the developer does not anticipate such a failure, and this divergence is irreversible without manual intervention. Such inconsistencies cause changes in application semantics and compromise correctness, and may severely affect smart-home users.

Moreover, stale hard reads may also break correctness of an application. For example, recent CO_2 level events might never get delivered to the CO_2 vent application in Listing 1. By reading a stale CO_2 level, the application may incorrectly turn off the exhaust fans.

Besides device failures, similar issues arise if an application crashes. For instance, an inconsistency arises if the smart security application fails between sending a command to set the alarm (line 4) and setting the active state to true (line 5).

Finally, applications may modify shared soft and hard states concurrently [40] which can cause canonical interleaving based inconsistencies [39].

As an example, the following quote from a disgruntled SmartThings customer [9] who got robbed during his vacation shows the impact of the inconsistency problem: "*More importantly, we were robbed when we were out on vacation. ... The logs show the motion of the robbers, but it never sounded the alarm ... I no longer trust it to do what it is supposed to do when it is supposed to do.*"

3.2 Dependency

In the previous section, we showed connections between hard states and soft states that are potential sources of inconsistencies due to hard read/write failure. We call these connections between two operations on hard states or two operations between soft and hard states that are semantically associated, a *dependency*. By identifying dependencies in an application, we can study the effects of failures on its correctness.

In order to systematically analyze smart-home applications, and understand how failures can affect them, we categorize

dependencies into four classes, using the following notations: we represent a hard read to device D as HR_D, and denote a hard write to device D with value V as $HW_D(V)$. A soft read from application state X is denoted as SR_X, and a soft write to state X with value V is represented as $SW_X(V)$.

1. $HR_D \rightarrow HW_{D'}(V)$: a dependency in this category captures the effect of a failure in a HR_D. The read might fail to return any value if device D is unavailable, or it might return a stale value. In either case, it implies that the application may exercise the dependency incorrectly, thus breaking its semantic. Such a dependency in an application can be because of a control dependence [27] or a data dependence.

If the dependency is a control-dependence [27], the value of HR_D controls the execution of $HW_{D'}(V)$. For example, in Listing 1, the dependency between lines 12 and 13, and also between lines 15 and 16 are control dependences. The hard read in line 11 flows into the control statements in lines 12 and 15. Therefore, a stale read at line 12 might incorrectly switch off the exhaust fans, and update the soft state even though the CO_2 levels are unsafe. A read value can also flow into a hard write via data dependencies. For example:
`a =` HR_{D_1}`; c = foo(a, b);` $HW_{D_2}(c)$.

2. $HR_D \rightarrow SW_X(V)$: this dependency affects the execution of a soft write. Analogous to $HR_D \rightarrow HW_{D'}(V)$, this results in a missing soft write or an incorrect soft write, because of control and data dependences. In turn, the incorrect soft write leads to unexpected program behavior when the state is read elsewhere. In our running example, this dependency exists between lines 12 and 14, and also lines 15 and 17.

3. $HW_D(V) \rightarrow SW_X(V')$: a hard write to soft write dependency is more subtle since $SW_X(V')$ is not a control or a data dependence on a $HW_D(V)$. Nevertheless, we observe that semantically tying a soft state with a hard state — meaning the soft state is an indicator of the hard state — is a common practice in many smart-home applications. Developers use this technique mainly to save battery: by associating a soft state with an actuation, developers can use the soft state elsewhere in the code instead of reading hard states.

For example, in the CO_2 Vent application, the developer implicitly creates a $HW_{switches}(ON) \rightarrow SW_{active}(true)$ dependency between lines 13 and 14, and also between lines 16 and 17. Thus, a failure in turning on `switches`, even if temporary, leaves a permanent inconsistency. Any subsequent change in the CO_2 level, even above the `level`, precludes turning on the exhaust fans.

To find a $HW \rightarrow SW$ dependency in the code, we compute the postdominance relation [23]: a code point b postdominates a code point a, if b is executed on every path from a to the end of the analyzed entity, which in our case, is an event handler. After computing postdominance instances, we manually look at all instances to confirm if the pair is semantically tied. Accordingly, we infer a case for semantic error if the soft state is read elsewhere in the application.

4. $SW_X(V) \rightarrow HW_D(V')$: this dependency has the same semantic effect as $HW_D(V) \rightarrow SW_X(V')$.

Note that all dependencies with soft reads (i.e., $SR_X \rightarrow *$), are not directly related to device failures. However, we still statically compute all such control and data dependences as an incorrect soft read can produce unintended behavior. Concretely, a soft read can be on a state determined by an incorrect, inconsistent, or missing soft write originating from the dependencies described above.

3.3 Analysis and Findings

We statically analyzed 147 SmartThings applications [19] and 35 OpenHAB applications chosen from IoTBench [10] by adding phases to the Groovy compiler. The AST visitors, `GroovyClassVisitor` [8], allow us to build a call graph per entry point and an intermediate representation (IR) amenable to data and control-flow analysis.

We analyzed the applications using inter-procedural data and control-flow analysis to understand the dependencies and their implications. Our analysis yields two key benefits: (i) understand the implications and the extent of failures on a large set of smart-home applications, and (ii) mitigate or eliminate the problems with our programming abstraction, called transactuation.

On average, the studied applications have three triggers, and manage a diverse set of devices (4–5 capabilities). In order to get a holistic view of the home state, on average, the applications perform three hard reads. They also perform between seven to nine hard writes on average. This shows that many of these applications try to provide automation among a set of devices (e.g., turning on restroom light, preparing coffee, and playing music, when a user wakes up), instead of managing a single device. Additionally, our analysis revealed that developers regularly use soft states to share states not only among handlers, but also among different applications. These results indicate that smart-home applications are fairly complex, and their behavior could be complicated through the use of handlers triggered by events that read/write both hard and soft states.

More specifically, we observed that, on average, applications have 3–10 instances of $HR \rightarrow HW$, 1–2 instances of $HR \rightarrow SW$ and 1–2 instances of $HW \rightarrow SW$ dependencies. We inspected these dependencies to find their potential implications on systems lacking appropriate abstractions to capture failures. We categorized the implications as follows: (i) missing actuation, (ii) wrong actuation, (iii) inconsistent soft state, (iv) missing notification, and (v) wrong notification . These implications can lead to unwanted outcomes, some of which have serious consequences such as security threats, health hazards, and missing critical alerts, e.g., a fire alarm not rung. They may also cause inconveniences, e.g., erroneous automation, incorrect notifications, sirens not turned off. Out of all 182 applications, our analysis unearthed 67 SmartThings and 32 OpenHAB applications, that have unintended effects. Due

Application	Type	Consequence	Dependency	Correction/Mitigation
Smart Humidifer (ST)	Automation	Wrong status flag causes humidifier to never be turned on/off. Incorrect notification.	*HW*→ *SW* *SR*→ *	Correct status flag to retry turning humidifier on/off later. Notify glitch to user.
Thermostat Auto Off (ST)	Energy	Wrong status flag causes thermostat to never be turned on/off.	*HW*→ *SW*	Correct status flag to retry turning thermostat on/off later.
CO2 Vent (ST)	Safety	Wrong status flag causes exhaust fans to never be turned on/off.	*HW*→ *SW* *SR*→ *	Correct status flag to retry turning exhaust fans on/off later.
Elder Care (ST)	Safety	Missing elder inactivity notification	*HR*→ *HW*	Notify glitch to user.
Smart Care (ST)	Safety	Alarm not armed. Missing notification.	Bad interleaving *HW*→ *SW*	Notify glitch to user.
Alarm (OH)	Security	Wrong status flag causes sirens to never be turned off.	*HR*→ *SW* *SR*→ *HW*	Correct status flag to retry turning sirens off.
Fire Detection (OH)	Security	Wrong status flag causes fire alarm to never ring	*HW*→ *SW* *SR*→ *	Correct status flag to retry ringing alarm later.
Forgiving Security (ST)	Security	Alarm does not ring. Incorrect notification.	*HW*→ *SW* *SR*→ *	Notify glitch to user.
Lock It When I Leave (ST)	Security	Door not locked but home vacant.	*HR*→ *HW*	Notify user to lock manually.

Table 1: Critical undesirable consequences in smart-home applications if failures are not handled and how developers can correct or mitigate the problems. ST and OH are abbreviations for SmartThings and OpenHAB, respectively.

to space constraint, we only show a subset of them with unintended semantics and potential fixes in Table 1.

To address these implications, a developer needs to preserve the semantic invariants of the dependencies to avoid discrepancy between the physical and application realms. One key trait of these applications is that their semantics tolerate different numbers of failed hard reads and writes. For example, for *HR*→*HW* in the application that computes average humidity level and reacts accordingly, even if some hard reads are stale based on their timestamp (i.e., some humidity sensors fail), the application can proceed with correct semantics as long as some sensors function properly. On the other hand, for *HW*→*SW* in the application that locks all doors and set the home state to `safe`, the developer needs to ensure that the home state is not set, even if only one door fails to be locked. To summarize, the following two key aspects are missing in existing IoT abstractions: 1. identifying the inherent connection between application semantics and *number* of failed operations, and 2. recomputing application states to preserve invariants under failed hard reads/writes.

4 Transactuations

To address the issues discussed in the previous section, we introduce a new abstraction called *transactuation* that allows a developer to build a reliable smart-home application. Transactuations provide the following two guarantees: (1) preserve dependencies between reads/writes to hard states and soft writes (i.e., *HR*→*SW* and *HW*→*SW*) even in cases of hardware and communication failures. (2) ensure isolation among transactuations that execute concurrently.

The concept of transactuations is very similar to database transactions. Yet, due to the intrinsic nature of physical world, it is impossible to ensure similar transactional guarantees. We note that transactuations are not meant to replace transactions completely. Instead, they are designed to address a similar problem in a cyber-physical environment which inherently prevents us from making strong assumptions. Precisely, transactuations and transactions differ as follows:

1. *Atomic durability:* atomic durability [36] guarantees that either all updates inside a transaction eventually become durable, or none of them becomes durable. Since IoT devices can neither be locked nor rolled back (e.g., in case of some failures), transactuation cannot guarantee atomic durability of hard writes. More specifically, unlike a transaction, a transactuation only guarantees atomic durability of soft writes but not hard writes inside it. Thus, if a hard write fails, a transactuation still commits by forcing its soft states to be consistent with its hard states, as per developer specified policies (see Section 4.1).

2. *Isolation & Atomic visibility:* strong isolation models (e.g., serializability or snapshot isolation) requires a transaction to read a consistent snapshot of a system (e.g., the last committed state) and precludes a use of partially committed states. A transactuation executes on the latest known consistent snapshot of the physical world, in isolation from other concurrent transactuations. However, two concurrent transactuations can execute on different snapshots of the physical world in absence of any committing transactuation. Additionally, (internal) atomic visibility ensures that effects of all updates in a transaction become visible to another transaction atomically [36]. Transactuations are also capable of guaranteeing internal atomic visibility: effects of a transactuation become atomically visible to other transactuation. However, in a smart home domain, consumers will unavoidably observe the effect of a hard write operation the moment it gets

executed in an actuator. Thus, it is impossible to provide *external* atomic visibility. For instance, one cannot expect that a smart-home user to observe all door locks become locked instantaneously.

Transactuations, further add to the definition of consistency based on consistency between hard reads/writes and soft writes. Transactuations preserve two invariants as follows:

(D1) A transactuation guarantees that if it executes, the staleness of its hard reads is bounded, as per the developer specified tolerance. A developer leverages this invariant to ensure inconsistencies arising out of breaking $HR\rightarrow *$ dependencies are detected, and appropriate actions are taken.

(D2) If writes to soft states are committed, it implies that sufficient number of hard writes as per developer specification have successfully executed. A developer leverages this invariant to enforce consistency of $HW\rightarrow SW$ dependencies.

4.1 Abstraction & API

Transactuations contain three pieces of logic which a developer writes as lambda expressions. A lambda expression is a function that can be passed as an argument to another function [1, 7, 11]. In the rest of this paper, we refer to these lambda expressions as lambdas. A transactuation can have the following three lambdas: perform lambda, onSuccess lambda, and onFailure lambda.

perform lambda. A perform lambda contains the core logic of a transactuation. Inside a perform lambda, a developer can perform hard writes (`actuate(Device, Value)`), soft reads (`read(State)`), and soft writes (`write(State, Value)`) as shown in Listing 3.

To assign a perform lambda to a certain transactuation, a developer calls the `perform()` method and passes the lambda as an argument as shown in lines 5–15 of Listing 3. The method signature is `perform(performLambda, [sensorList, timeWindow, sensingPolicy], [actuatingPolicy])`.

A developer cannot explicitly issue a hard read inside a perform lambda. Instead, she has to specify a list of required hard states as an argument (i.e., `sensorList`) to `perform()` method. The required hard states are read before perform lambda is executed, and a list of available hard states are accessible as key-value pairs to perform lambda, using `sensors` parameter of a perform lambda (line 5). Disallowing explicit hard reads inside a transactuation prevents reading stale or null sensor values, which can break application semantics.

To preserve consistency between hard reads and soft writes in case of a sensor unavailability, a developer can use a time window along with a sensing policy. The time window specifies that the sensor list must be validated such that, after validation, the list of available sensors includes those that have received events close in time. Specifically, a time window defines the duration when the transactuation triggering event and read hard states remain valid. For instance, a window of

```
1  function handler(evt) {
2    let tx = Transactuation(evt);
3    // executes if all CO2 sensors received
4    // events in past 5s w.r.t. triggering event
5    tx.perform(func(sensors){
6      let co2 = sensors['co2'];
7      let active = read('active');
8      if (co2 >= threshold && !active) {
9    //if all fans can be on, set active to true
10       actuateAll('fans', 'on');
11       write('active', true);
12     } else if (co2 < threshold && active) {
13       ...
14     }
15   }, ['co2'], 5, 'all', 'all');
16   // executes if both policies are met
17   tx.onSuccess(func(evt) {
18     let txs = Transactuation(evt);
19     txs.perform({
20       actuate('msg', 'CO2 is high');
21     }, 'none', 'none');
22     txs.execute();
23   });
24   // executes if either one policy is not met
25   tx.onFailure(func(evt) {
26     let txf = Transactuation(evt);
27     ...
28   });
29   tx.execute(); }
```

Listing 3: CO2 Vent written with transactuation. The code presented here is in synchronous style but our implementation uses asynchronous Node JS.

10 seconds has the following intent: a hard state passes validation if its most recent event and the transactuation triggering event are not more than 10 seconds apart.

A sensing policy is an acceptable level of hard-read failures that a transactuation can tolerate. It specifies that under what condition a perform lambda can be executed over a returned list of window-validated sensors. The perform lambda in turn may or may not execute depending on the sensing policy. Transactuations support three sensing policies:

- *All*: ensures that the perform lambda executes only if all hard states in the sensor list pass validation. Consider an application that reads presence sensors of every user and turns on cameras if no one is present. For privacy, *all* sensors need to pass validation. If even one presence sensor fails, it should not risk turning on the cameras since it violates privacy.
- *Any*: guarantees the execution of the perform lambda as long as at least one hard state in the sensor list passes validation. For example, an application that computes average humidity level from multiple sensors to control fans, executes accordingly with correct semantics, even if some sensors fail, but not all.
- *None*: states that the perform lambda executes over the returned validated list of hard states regardless of how many hard states are unavailable.

Observe that a time window along with a sensing policy helps preserve $HR\rightarrow *$ dependency as per the developer's intention to preserve invariant (D1). To preserve invariant (D2),

a developer needs to specify an *actuating policy*. The actuating policy is an acceptable level of hard-write failures that is tolerable. To meet an actuating policy in case of a failure, soft writes inside a transactuation roll back to their initial values, and onFailure lambda executes. Similar to a sensing policy, an actuating policy supports the following semantics:

- *All*: states that modifications to soft states commit if all hard writes successfully finish. An example of this policy is an application that locks all doors and sets home state to `safe`. If even one door fails, the home state should not be set.
- *Any*: guarantees that soft state modifications inside a lambda commits if at least one hard write succeeds. For example, an application that actuates all sirens and sets the flag `ringing`. Even if only one siren rings, the flag should be set.
- *None*: states that soft writes commit despite of failures.

onSuccess lambda. An onSuccess lambda executes if the perform lambda of a transactuation succeeds (i.e., sensing and actuating policies are met). A developer can assign an onSuccess lambda to a transactuation via `onSuccess()` as shown in line 17 of Listing 3.

onFailure lambda. An onFailure lambda executes if a transactuation cannot meet its sensing or actuating policies. It is assigned to a transactuation via `onFailure()` as depicted in line 25 of Listing 3.

When a developer has set up all the lambdas for a transactuation, she executes the transactuation by invoking `execute()` (line 29), which is an asynchronous call that executes the perform lambda in the background.

Listing 3 illustrates the CO_2 Vent rewritten with the transactuation abstraction. The perform lambda is parameterized with 5s time window. The transactuation only reads one hard state, `co2`. The lambda executes if the latest sensor update from `co2`, and the triggering event, which is also `co2` fall in the 5 second time interval. `switches`, which binds to an array of fans, requires the "all" policy if we want the soft writes to be consistent with the actuations. The soft state `active` will be set to true only if all fans can be turned on, otherwise, `active` remains unchanged.

4.2 Chaining transactuations

A transactuation can be chained to other transactuations by invoking it in their onSuccess and onFailure lambdas. As we shall see in the next section, the runtime guarantees to execute chained transactuations sequentially: if a transactuation τ_j is invoked in onSuccess lambda of τ_i, τ_j is guaranteed to see the updates τ_i makes. We call this ordered execution of transactuations as T-Chain. This is particularly relevant in an asynchronous runtime where high latency operations can finish in arbitrary order, executing outside the critical path such as in worker threads [25, 44]. Thus, if τ_j wants to use a soft state written by τ_i, τ_j needs to be invoked in onSuccess lambda of τ_i. In addition, if τ_j requires actuations of τ_i to complete before it, these two transactuations must form a T-Chain.

5 Relacs

In this section, we detail the design of our runtime, called Relacs, that execute smart-home applications, along with a supporting key-value store called *Relacs Store*.

5.1 Relacs Store

All soft and hard states inside a transactuation are stored in a key-value store called Relacs Store. It hides all complexities of working with sensors and actuators by allowing developers to not only perform read/write operations on soft states inside a transactuation, but also to issue hard reads/writes.

Conceptually, every state inside the Relacs Store maintains two values, *speculative* and *final*. A speculative value means that the state has been updated logically in the Relacs Store, but is not confirmed to be final (i.e., issued to an IoT device). For example, a transactuation that wants to unlock a door will have the speculative value of the door set to `unlocked`, before the actuation command succeeds. When Relacs receives an ack event confirming the success of an actuation command, it updates the final value and discards the speculative value. Along with setting the final value, the Relacs Store also logs the timestamp of the ack event for validating a time window of a transactuation reading that hard state. In Section 5.2, we explain how speculative states help Relacs to speculatively execute transactuations.

Since multiple hard writes on the same state can execute before the system receives an ack from the corresponding device, Relacs Store needs to record all versions of speculative values that have not been finalized yet. When reading a state, Relacs Store returns the latest speculative value, or the final value if no speculative value exists. For instance, consider the following transactuations: a transactuation τ_i sets a lamp color to red. While the lamp is changing its color, τ_j changes the lamp color to green. In this example, Relacs Store logs both speculative values. Thus, if τ_k tries to read the state of the lamp, Relacs Store returns green, even if the lamp has not completed executing the first actuation command to change its color to red.

5.2 Execution Model

A transactuation execution model comprises of the following three phases:

1. *Hard read phase*: to start executing a transactuation, the system first needs to determine if it can read the required hard states in the sensor list which satisfy the specified window and the sensing policy. If so, the system proceeds to the next phase. For a poll-based sensor, if Relacs fails to validate the

window, it polls the sensor to check if it can get a fresh value. For a push-based sensor, Relacs simply waits, as long as the window is valid, to receive an event from the sensor. Observe that the window is valid as long as the specified time window has not passed since the transactuation triggering event. If the window becomes invalid, and the list of received events fails staleness validation, it cannot execute the perform lambda, and proceeds to execute the onFailure lambda.

2. *Speculative Commit Phase*: since IoT devices cannot roll back, Relacs needs to make sure that a transactuation will definitely commit before performing real actuations. Therefore, it employs a speculative execution model where a perform lambda first executes speculatively, without performing any real actuation. Once the perform lambda finishes, it tries to speculatively commit like a normal transaction inside Relacs Store. Therefore, new speculative values are committed for modified soft and hard states. Additionally, committing new speculative values may trigger other handler functions subscribed to these states. Finally, Relacs starts executing the onSuccess lambda of the transactuation when it commits. Note that these lambdas triggered by speculative commit execute their transactuations speculatively.

3. *Final Commit Phase*: in the last phase, Relacs sends actuation commands that correspond to hard writes. A transactuation τ_i can start its final phase, when the following three conditions hold: first, all transactuations that precede τ_i in the T-Chain finally commit. Second, all transactuations updating states that τ_i read, finally commit. Third, no other finally committing τ_j conflicts with τ_i. More specifically, the readset of τ_i does not have any intersection with the writeset of some finally committing transactuation, and the writeset of τ_i does not intersect with both readset and writeset of some finally committing transactuation.

Relacs finally commits the transactuation when sufficient acks are received from actuators to satisfy its actuating policy. If the transactuation times out without satisfying its actuating policy, all soft writes inside the transactuation roll back to their initial state, and the transactuation finally commits. Next, onFailure lambda executes if it has been defined. Moreover, all speculative transactuations invoked by the failed transactuation abort (e.g., chained transactuations), and transactuations that bear data dependencies with the failed transactuation need to re-execute.

5.3 Relacs Runtime

Relacs is built atop serverless computing [32, 42]. The runtime comprises two classes of functions namely application functions and system functions. We explain these functions in detail here.

Application Functions. An application can comprise several handlers which are triggered when particular states in the Relacs Store change (publish-subscribe model), and each handler can comprise several transactuations. An application submitted to run by Relacs system is transformed into a set of application functions to run on serverless instances as follows:

1. For each handler, Relacs transforms the logic of an embedded transactuation (i.e., perform lambda) into a transaction that can execute transactionally inside the Relacs Store.

2. The logic inside onSuccess lambda and onFailure lambda are transformed into stand-alone serverless functions called *success* and *failure* functions, respectively, hereafter. If onSuccess lambda or onFailure lambda is comprised of transactuations with their own onSuccess lambda and onFailure lambda (T-Chain), the transformations are applied recursively.

3. Finally, every handler is transformed into a runnable stand-alone serverless function, called *handler* function.

System Functions. Relacs comprises a serverless function called *updater* function that is invoked whenever the state of a sensor or an actuator changes. Upon receiving a notification, the updater updates the hard state corresponding to the event in Relacs Store, and launches an instance of subscribed handler function(s).

Final-committer is a designated function to perform the final commits. It selects speculative transactuations that can finally commit without breaking the final commit rules, issues all of their actuation commands, and marks the actuations as issued. When a successful actuation receives a notification (ack) from an IoT device, the updater function updates its corresponding state in Relacs Store, and marks the actuation command as done transactionally.

In order to detect an actuation failure, Relacs has a *failure-detector* function that runs periodically, and checks whether an ack is received for an actuation command. If after certain threshold no ack is received, the failure detector marks the actuation as failed. If actuating policy is not met, the enclosing transactuation commits with rollback of soft writes, which triggers a *re-executor* function to re-execute transactuations that have data dependencies with the failed transactuation.

5.4 Fault Tolerance

A function in serverless computing is not guaranteed to complete, and can terminate at any arbitrary point of execution. Yet, Relacs guarantees applications to execute reliably despite failures as follows.

Relacs ensures that all transactuations are executed exactly-once even if an application function (handler, success, or failure) fails during its execution. To this end, Relacs maintains two logs: function log and transactuation log. Function log is a write-ahead log for application functions. The function name along with ID of the triggering event is recorded in the function log before the function executes. Transactuation log atomically records a transactuation name and the event ID during the speculative commit of a transactuation along with updates to soft/hard states.

A system function called *serverless checker* runs periodically, and inspects the function log to execute functions which have failed. In either case, the serverless checker invokes the failed functions again. This might lead to duplicated executions of transactuations that have executed. To prevent this, Relacs checks if a particular transactuation is in the transactuation log, and skips its execution if present. [1]

Currently, the updater failure is treated as an equivalent of sensor or actuator failure and it is handled by transactuation semantics. To address final committer failure, Relacs runs the final committer periodically to complete pending final commits by actuating unissued actuations. To preclude contention between the periodic and the regular final committer that can run concurrently, Relacs uses leases and ETAGS à la Tuba [21] in the final committer to ensure correctness.

5.5 Implementation

We implemented Relacs runtime and Relacs Store on top of Microsoft Azure. We used Azure Function (serverless computing) to implement the runtime, and used Azure Cosmos DB to build Relacs Store. All serverless functions were implemented with Azure Function. Application functions are triggered by HTTP calls and system functions are triggered on Cosmos DB updates or periodic timers. The parts of the protocol that need to update Relacs Store transactionally (including perform lambda) are transformed into Cosmos DB stored procedures [3].

Currently, Relacs has only been integrated with Samsung SmartThings. SmartThings allows a developer to build a web service that connects with devices in a home [18]. We built a gateway that forwards actuation commands from Relacs to actuators and also polls sensor data.

5.6 Discussion

As described, Relacs validates sensor failures through event timestamps and actuator failures through timeouts. For sensor validation, as explained, if validation fails and a device is pollable, Relacs polls the device within the window constraints. If a device is push-based but pollable, Relacs polls the device and if the validation fails again, it waits for its push-interval within the time window. However, if the device is purely push-based, Relacs cannot differentiate between inactivity and failure. We inspected 188 SmartThings-compatible devices and found that 113 of them are pollable. Likewise, actuation failures are detected with timeouts, first on initial ack from smart-home connector, followed by notification on final actuator state change. Again, if the ack message is lost, Relacs can incorrectly rollback soft states. However, transactuations can still help developers to prioritize home safety over convenience such as always setting a soft state to a conservative value; e.g., in Smart Security (Listing 2) to ensure that the alarm eventually rings.

[1] Note that any failure during the speculative commit results in a regular transactional abort and transactuation log is not updated. Hence the transactuation is retried when the function reexecutes.

6 Evaluation

In this section, we report our evaluation results on programmability, effectiveness of transactuations in enforcing correctness, and the overhead incurred by Relacs to provide transactuation semantics.

We selected 10 SmartThings applications from the applications that we statically analyzed. These applications are publicly available on SmartThings repository [19]. The applications cover the four most common categories—Security (Sc), Safety (Sf), Convenience (Cn), and Energy Efficiency (Ee). Instead of using the original version that runs on SmartThings cloud, we implemented the following three versions of the applications, that run on Azure Functions, using Javascript Node JS [44]. This allows us to compare an application with transactuations against an application without transactuations in an apple-to-apple fashion.

- *BE:* we wrote a best-effort version (BE) of the applications without the transactuation abstraction. The BE version follows the default semantics that ignores device failure, exactly-once execution, and isolation.
- *BE+Con:* since the BE version ignores potential failures in devices or applications, we implemented a *best-effort with consistency* (BE+Con) version of an application which adds code that keeps device states consistent with application states. More specifically, BE+Con introduces both sensor window validation and soft state rollback code. However, it ignores the isolation guarantee that transactuations provide.
- *TN:* we also implemented these applications with the transactuation abstraction (TN). 5 applications out of the evaluated 10 applications used T-Chain to establish order among hard and soft states.

Experimental setup. We set up SmartThings compatible devices and measured the round trip latency of four devices in a typical smart home: a door lock, a bulb, a power strip, and a smart power plug. The door lock has a significant latency of nearly 3.6s on average and maximum of nearly 9.8s, over 100 trials. The other devices incur an average latency of nearly 0.7s with the maximum at nearly 3.7s. Since we had a limited set of devices, we parallelized our experiments by simulating the devices using latency data on a Raspberry Pi Model 3 [13]. It comes with a 1.2 GHz 32-bit quadcore ARM Cortex-A53 processor and 1 GB RAM. In addition, the simulator also allowed us to easily inject failures for our experiments.

Application	#HR	#HW	Transactuation Policy	LOC		
				BE	BE+Con	TN
Rise And Shine (Cn1)	1 (*)	1	2 (none, none)	72	195	68
Whole House Fan (Cn2)	1 (*), 3	2 (*)	1 (none, none)	29	176	26
Thermostat Auto Off (Cn3)	1 (*)	2	1 (all, none), 1 (all, all), 1 (none, all)	70	198	68
Auto Humidity Vent (Ee1)	1 (*), 1	3(*), 1	1 (any, none), 1 (none, any), 1 (none, none), 1 (all, any)	49	170	100
Lights Off With No Motion (Ee2)	1 (*), 1	1 (*)	2 (all, all)	56	161	67
Cameras On When Away (Sc1)	2 (*)	2 (*)	1 (all, none), 1 (any, none)	31	149	88
Nobody Home (Sc2)	1 (*)	1	1 (all, none), 1 (any, none), 1 (none, none)	65	175	62
Smart Security (Sc3)	2 (*)	2 (*)	1 (all, all)	144	323	144
CO2 Vent (Sf1)	1	2 (*)	1 (all, all)	29	152	26
Lock It When I Leave (Sf2)	3 (*)	2 (*), 2	2 (none, none), 1 (all, none)	51	180	54

Table 2: Properties of each benchmark application including the number of hard reads and hard writes (* denotes an operation to an array of devices with a single command, for example, 2 (*) means 2 operations, each accessing a device group); the fault-tolerance policies for the TN configuration in a format of (sensing, actuating) (Col 4); and programability shown by LOC comparison among transactuation (TN), best effort (BE), and best effort with consistency (BE+Con) (Col 5).

6.1 Programmability

In order to evaluate the programmability and convenience of using transactuation in contrast to manually writing failure handling code, we compare lines of code (LOC) of applications, using CLOC [6].

Table 2 shows the programmability evaluation (LOC) along with the number of hard reads and writes, and transactuation policies we employ for each application. Observe that TN and BE versions are comparable in LOC despite no guarantees in the BE version, except in Ee1 where we introduce new soft states and four transactuations, each part of T-Chains, in order to ensure consistency. BE+Con version requires substantial code to explicitly handle failures. As mentioned earlier, BE+Con version validates sensor freshness similar to transactuation and may roll back soft states after determining the outcome of actuations for hard write to soft write dependencies. Finally, although transactuations require more code in order to create T-Chains, it automatically handles failure, and simplifies writing reliable applications considerably.

6.2 Correctness

Table 3 shows the applications that we evaluated with their inherent undesirable behaviors on transient or longer duration failures. The second column shows the undesirable behaviors, and the third column shows the outcome of using transactuations. The last column explains the mechanism transactuations use to resolve or mitigate the issue. We considered different types of failures that transactuations can address (i.e., unavailable sensors and failed actuations), and injected these failures by dropping event or actuation messages. Transactuation addresses these issues with three techniques. First, sensor staleness validation prevents the execution of perform lambda and executes onFailure lambda that can notify a user. Second, actuation losses are detected automatically and associated soft writes are rolled back to ensure consistency. Third, when one actuation depends on another, we used an intermediate soft state to chain two transactuations each having actuations. For example, in Sc3 (Smart Security) application, inconsistency between the alarm actuation and the soft write is resolved using roll back to eliminate the issue. However, some applications need to use multiple chained transactuations to correctly address actuation dependencies.

6.3 Overhead

To evaluate the overhead of transactuations, we measured execution time of the applications as follows. We started timing when an application began executing, and stopped when every soft write committed and all actuations completed. Our performance results are summarized in Figure 1. Each value is the mean of 30 runs, with 95% confidence intervals.

Failure-free. We first compare the execution times of TN and BE versions without any injected failures. The overhead of transactuations is attributed to (1) safeguarding against inconsistencies due to inherently concurrent execution, (2) providing fault tolerance, and (3) enforcing actuation orders of T-Chains. We note that the final committer function imposes significant overhead on Relacs since it is invoked[2] automatically by CosmosDB updates. For instance, we observed that its start may be delayed between zero to five seconds. The periodic final committer which we set to run every second helps to mitigate this overhead.

Figure 1a shows that, on average (geomean), the TN version incurs 1.5 times slowdown compared to BE. Observe that the

[2]Other functions except the re-executor are invoked by HTTP calls.

App	Undesirable consequence	Transactuation effect	Mechanism used
Cn1	Mode not set permanently	✓	Soft state rollback
Cn2	Incorrect behavior	Issue detected and user notified	Sensor staleness validation
	Fans not ON irreversibly	✓	Soft state rollback
Cn3	Thermostat not OFF	✓	Soft state rollback
	Incorrect mode	✓	Soft state rollback
Ee1	Incorrect energy and operation time reported	✓	Soft state rollback and chaining
	Incorrect behavior	Issue detected and user notified	Sensor staleness validation
Ee2	Incorrectly turning lights ON/OFF	Issue detected and user notified	Sensor staleness validation
Sc1	Incorrect behavior	Issue detected and user notified	Sensor staleness validation
	Actuation failure	✓	Chaining
Sc2	Incorrect mode set	Issue detected and user notified	Sensor staleness validation
	Home mode change w/o notification	✓	soft state rollback
Sc3	Intruder motion not detected	Issue detected and user notified	Sensor staleness validation
	Alarm not active irreversibly	✓	soft state rollback
Sf1	Incorrect behavior	Issue detected and user notified	Sensor staleness validation
	Exhauts not ON irreversibly	✓	soft state rollback
Sf2	Door unlocked but home vacant	Issue detected and user notified	Sensor staleness validation
	Door locked at arrival	✓	Chaining

Table 3: Applications with undesirable consequences on induced failures. Column 3 shows failure avoidance or mitigation when written with transactuations. Column 4 shows the internal mechanism used by the transactuations. A checkmark implies that transactuation automatically resolves the issue.

speculative commit duration (TN.SC) is significantly smaller than the final commit duration (TN.FC). Figure 1a also breaks down the final commit time into actuation time (TN.FC.ACT) and the final-committer triggering overhead (TN.FC.TRIG). As mentioned earlier, the triggering overhead is significantly large, especially, in the case of a long T-Chain like Ee1 (4 transactuations).

With failure. In this scenario, we conducted two experiments. In each experiment, we used a dummy application that issued a dummy actuation, and updated a dummy soft state. In the first experiment, the dummy actuation turned on a smart switch (low-latency actuation). In the second one, it actuated a door lock (high-latency actuation). We introduced an artificial data dependency (RAW) by forcing all benchmark applications to read the dummy soft state before executing their core logic. Lastly, we injected a failure to the dummy actuation to trigger failure detection and handling in the dummy application and re-execution of the benchmark applications to repair the broken data dependency. Because devices have different actuation latencies, the timeout thresholds to declare failed actuations are specific to each device. More specifically, we used the maximum observed latency for each device (i.e., 4s for the smart switch and 10s for the door lock).

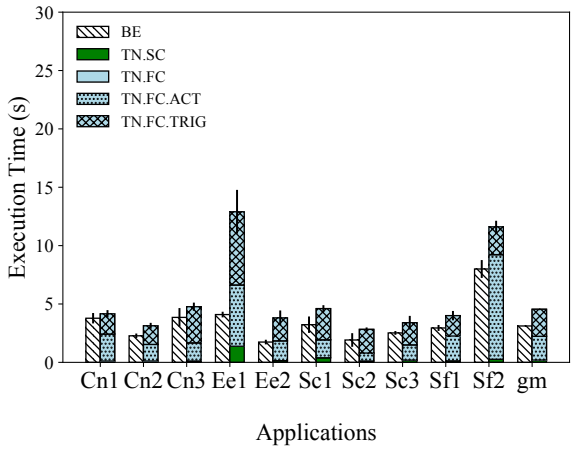
Figure 1b compares the execution time of the failure-free case against the two failure experiments. The additional overhead we observe here is the failure detection overhead which includes the timeout (TN.FD.TO) and the overhead of triggering the re-executor function (TN.FD.TRIG). Similar to the final committer, the re-executor is invoked automatically by Cosmos DB when actuations are marked as `failed`, thus it incurs similar overhead. Observe that the failure experiments have two stacked bars of speculative commits. The second bar shows the re-execution of transactuations with broken dependencies.

As expected, introducing a failure results in longer execution times for the applications. This slowdown is caused by the timeout threshold plus the re-executor triggering overhead (~2s). Moreover, the difference between the middle and right bars for each application is the difference in timeout thresholds for low and high latency actuations (~6s).

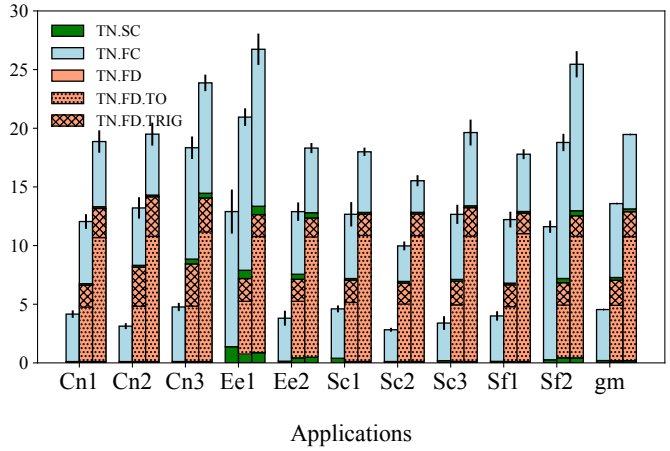
7 Related Work

Checking Correctness. Soteria [22] employs model checking to identify contradicting interactions between IoT applications. For example, water leak detection turns off a water valve while smoke detection attempts to turn on a fire sprinkler. Prior work like DeLorean [24] models absolute and relative time to find timing bugs in event driven programs, e.g., door open at unsafe times. In contrast, our work tackles a different problem, the lack of reliability and isolation, using a dynamic technique. IoT analyses also use dynamic taint analyses like techniques to detect source of security breaches [46] and dynamic program slicing to explain behaviors [40]. We use static dependence analysis to report potential problems.

Programming abstractions. Using speculative execution for improving latency and performance is a common technique in many transactional and replicated systems. These can be classified into two categories: systems [34, 41, 47]



(a) Execution times for BE and TN versions in failure-free case. We break down the execution time of TN into speculative commit (TN.SC) and final commit (TN.FC). TN.FC is shown as actuation time (TN.FC.ACT) and as overhead to trigger the final-committer function (TN.FC.TRIG).



(b) Execution time comparison for failure-free and failure cases. For each application, we show 3 bars, failure-free case (the left bar), low-latency actuation failure case (the middle bar), and high-latency actuation failure case (the right bar). For the failure cases, the breakdown includes failure detection time (TN.FD) which is subdivided into timeout detection (TN.FD.TO) and re-execution triggering overhead (TN.FD.TRIG).

Figure 1: The execution time of 10 applications chosen from SmartThings repository and their geomean (gm) for BE and TN versions of applications in failure-free and failure scenarios.

that hide the effects of speculation from applications, and work [29, 31, 43] that expose speculation results to applications. While certain applications in the latter case can benefit by reading speculative values, they need to handle possible side effects of acting on misspeculated values. With Relacs, effects of speculatively committed transactuations are exposed to other transactuations. Yet, no transactuation can finally commit, and actuate devices until all transactuations that it speculatively read from finally commit.

Planet [43] provides a mechanism to speculate on partial state of a transaction in distributed environments. The abstraction allows a developer to continue based on a predictive outcome, and later receive a confirmation or an apology. In contrast, we target a different environment and problem, and provide a simplified way to address device failure handling.

Execution semantics and conflict detection. IOTA [40] defines a calculus for programs in IoT domain. They also define an execution semantics to eliminate races on actions against the same physical event. Similar races can be resolved in our system by reordering transactuations according to programmer annotations similar to Zave et al. [48]. IOTA also shows offline analyses to detect device conflicts. Conflict detection in a home can include static model checking [38] or dynamic analyses [48] to detect feature interactions [38] and accesses to the same device [26]. They detect commands due to single event or concurrent independent events to the same device, e.g., simultaneous turning on and off on a device. The execution semantics of our system provides isolation naturally and can easily be enhanced to report device interactions by intersecting read-write sets of transactuations dynamically.

8 Conclusion

In this paper, we identified a fundamental problem that arises due to failures in IoT systems that interact with the physical world. We analyzed smart-home applications, and showed how application semantics is broken due to different failures that occur in an IoT environment. We introduced an abstraction, called transactuation, that allows a developer to build reliable IoT applications. Our runtime, called Relacs, enforces the semantic guarantees of transactuations. Our evaluation demonstrated programmability, performance, and effectiveness of the transactuation abstraction on top of our runtime.

9 Acknowledgment

We would like to thank our shepherd, Gernot Heiser and anonymous reviewers for their insightful and valuable feedback. We would also like to thank Nitin Agrawal, Arani Bhattacharya, Juan Colmenares, Iqbal Mohomed, Marc Shapiro, Pierre Sutra, Ahmad Bisher Tarakji, and Ashish Vulimiri for their suggestions and helpful discussions.

References

[1] Arrow functions. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions.

[2] AWS Lambda Retry Behavior. https://docs.aws.amazon.com/lambda/latest/dg/retries-on-errors.html.

[3] Azure Cosmos DB server-side programming: Stored procedures, database triggers, and UDFs. https://docs.microsoft.com/en-us/azure/cosmos-db/programming.

[4] Bluetooth Low Energy. https://www.bluetooth.com.

[5] CO2 Vent. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps/dianoga/co2-vent.src.

[6] Count Lines of Code. http://cloc.sourceforge.net.

[7] Expressions. https://docs.python.org/2/reference/expressions.html.

[8] Groovy ast interface. http://docs.groovy-lang.org/docs/groovy-2.4.0/html/api/org/codehaus/groovy/ast/package-summary.html.

[9] Inconsistent Behavior. https://community.smartthings.com/t/inconsistent-behavior/35284.

[10] IoTBench-test-suite. https://github.com/IoTBench/IoTBench-test-suite/tree/master/openHAB.

[11] Lambda Expressions. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[12] OpenHAB: Empowering the Smart Home. https://www.openhab.org.

[13] Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[14] SmartThings. http://www.smartthings.com/.

[15] SSA1 / SSA2 Instruction Manual. https://support.smartthings.com/hc/en-us/article_attachments/200715310/ssa_manual_14may2011_-_new_address0.pdf.

[16] Z-Wave Alliance. http://www.z-wavealliance.org.

[17] ZigBee Alliance. http://www.zigbee.org/.

[18] Web Services SmartThings. https://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/index.html, 2018.

[19] SmartThings Smart Apps. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps, 2019.

[20] Masoud Saeida Ardekani, Rayman Preet Singh, Nitin Agrawal, Douglas B. Terry, and Riza O. Suminto. Rivulet: A Fault-tolerant Platform for Smart-home Applications. In *Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference (MIDDLEWARE '17)*, Las Vegas, NV, December 2017.

[21] Masoud Saeida Ardekani and Douglas B. Terry. A Self-Configurable Geo-Replicated Cloud Storage System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[22] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated IoT Safety and Security Analysis. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC '18)*, Boston, MA, July 2018.

[23] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A Simple, Fast Dominance Algorithm. *Rice University, CS Technical Report 06-33870*, January 2001.

[24] Jason Croft, Ratul Mahajan, Matthew Caesar, and Madan Musuvathi. Systematically Exploring the Behavior of Control Programs. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '15)*, Santa Clara, CA, July 2015.

[25] James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.Fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the 2017 European Conference on Computer Systems (EuroSys '17)*, Belgrade, Serbia, April 2017.

[26] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An Operating System for the Home. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[27] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[28] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, September 2013.

[29] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[30] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC '14)*, Seattle, WA, November 2014.

[31] Pat Helland and Dave Cambell. Building on Quicksand. In *Proceedings of the 4th Conference on Innovative Data Systems Research (CIDR '09)*, Pacific Grove, CA, January 2009.

[32] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16)*, Denver, CO, June 2016.

[33] Timothy W. Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. The hitchhiker's guide to successful residential sensing deployments. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys '11)*, Seattle, WA, November 2011.

[34] Manos Kapritsos, Yang Wang, Vivien Quéma, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, October 2012.

[35] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. Examining Programmer Practices for Locally Handling Exceptions. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*, Austin, TX, May 2016.

[36] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 2013 European Conference on Computer Systems (EuroSys '13)*, Prague, Czech Republic, April 2013.

[37] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[38] Chieh-Jan Mike Liang, Börje F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. SIFT: Building an Internet of Safe Things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN '15)*, Seattle, WA, April 2015.

[39] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS '08)*, Seattle, WA, March 2008.

[40] Julie L. Newcomb, Satish Chandra, Jean-Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. IOTA: A Calculus for Internet of Things Automation. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (ONWARD '17)*, Vancouver, Canada, October 2017.

[41] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 191–205, 2005.

[42] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, Boston, MA, July 2018.

[43] Gene Pang, Tim Kraska, Michael J. Franklin, and Alan Fekete. PLANET: Making Progress with Commit Processing in Unpredictable Environments. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*, Snowbird, UT, June 2014.

[44] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, November 2010.

[45] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical Trigger-Action Programming in the Smart Home. In *Proceedings of the 2014 SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*, Toronto, Canada, April 2014.

[46] Qi Wang, Wajih Ul Hassan, Adam M. Bates, and Carl A. Gunter. Fear and Logging in the Internet of Things. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium, (NDSS '18)*, San Diego, CA, Februay 2018.

[47] Benjamin Wester, James A. Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating Latency in Replicated State Machines Through Client Speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, April 2009.

[48] Pamela Zave, Eric Cheung, and Svetlana Yarosh. Toward user-centric feature composition for the Internet of Things. *arXiv preprint arXiv:1510.06714*, October 2015.