Home, SafeHome: Smart Home Reliability with Visibility and Atomicity

Shegufta Bakht Ahsan[†], Rui Yang[†], Shadi A. Noghabi[*], Indranil Gupta[†]
{sbahsan2,ry2, indy}@illinois.edu[†], shadi@microsoft.com[*]
[†]*University of Illinois at Urbana Champaign.* [*]*Microsoft Research.*

Abstract

Smart environments (homes, factories, hospitals, buildings) contain an increasing number of IoT devices, making them complex to manage. Today, in smart homes where users or triggers initiate routines (i.e., a sequence of commands), concurrent routines and device failures can cause incongruent outcomes. We describe SafeHome, a system that provides notions of atomicity and serial equivalence for smart homes. Due to the human-facing nature of smart homes, SafeHome offers a spectrum of *visibility models* which trade off between responsiveness vs. incongruence of the smart home state. We implemented SafeHome and performed workload-driven experiments. We find that a weak visibility model, called *eventual visibility*, is almost as fast as today's status quo (up to 23% slower) and yet guarantees serially-equivalent end states.

1 Introduction

The disruptive smart home market is projected to grow from $27B to $150B by 2024 [60, 61]. There is a wide diversity of devices—roughly 1,500 IoT vendors today [45], with the average home expected to contain over 50 smart devices by 2023 [25]. Smart devices cover all aspects of the home, from safety (fire alarms, sensors, cameras), to doors+windows (e.g., automated shades), home+kitchen gadgets, HVAC+thermostats, lighting, garden sprinkler systems, home security, and others. As the devices in the home increase in number and complexity, the chances of interactions leading to undesirable outcomes become greater. This diversity and scale is even vaster in other smart environments such as smart buildings, smart factories (e.g., Industry 4.0 [41]), and smart hospitals [62].

Past computing eras—1970s' mainframes, 1990s' clusters, and 2000s' clouds—were successful because of good management systems [17]. What is desperately needed are systems that allow a group of users to manage their smart home as a single entity rather than a collection of individual devices [23]. Today, most users (whether in a smart home or a smart factory) control a device using *commands*, e.g., turn `ON` a light. Further, major smart home controllers have started to provide users the ability to create *routines*. A routine is a sequence of commands [7, 55, 64, 76]. Routines are useful for both: a) convenience, e.g., turn `ON` a group of Living Room lights, then switch on the entertainment system, and b) correct operation, e.g., `CLOSE` window, then turn `ON` AC.

Motivating Examples: Today's *best-effort* way of executing routines can lead to incongruent states in the smart home, and has been documented as the cause of many smart home incidents [26, 31, 38, 55, 67] [1]. First, consider a routine involving the AC and a smart window [27, 73]: $R_{cooling}$ = {`CLOSE` window; switch `ON` AC}. During the execution of this routine, if either the window or the AC fails, the end-state of the smart home will not be what the user desired—either leaving the window open and AC on (wasting energy), or the window closed and AC off (overheating the home). Another example is a shipping warehouse wherein a robot's routine needs to retrieve an item, package it, and attach an address label—all these actions are essential to ship the item correctly. In all these cases, lack of *atomicity* in the routine's execution violates the expected outcome.

Our next example deals with concurrent routines. Consider a timed routine R_{trash} that executes every Monday night at 11 pm and takes several minutes to run: R_{trash}={`OPEN` garage; `MOVE` trash can out to driveway (a robotic trash can like SmartCan [59]); `CLOSE` garage}. One day the user goes to bed around 11 pm, when she initiates a routine: $R_{goodnight}$={switch `OFF` all outside lights; `LOCK` outside doors; `CLOSE` garage}. Today's state of the art has no *isolation* between the two routines, which could result in $R_{goodnight}$ shutting the garage (its last command) while R_{trash} is either executing its first command (open garage), or its second command (moving trash can outside). In both cases, R_{trash}'s execution is incorrect, and equipment may be damaged (garage or trash can). Concurrency even among short routines could result in such incongruences—Figure 1 shows such an experiment. The plot shows that two routines simultaneously touching only a few devices cause incongruent outcomes if they start close to each other. In all these cases, *isolation semantics* among concurrent routines were not being specified cleanly or enforced.

Challenges: This discussion points to the need for a smart home to autonomically provide two critical properties: i) *Atomicity* and ii) *Isolation/Serializability*. Atomicity ensures that all the commands in a routine have an effect on the environment, or none of its commands do (e.g., if the window is not closed, the AC should not be turned on). Serializability says that the *effect* of a concurrent set of routines is equivalent to executing them one by one, in some sequential order, e.g., when R_{trash} and $R_{goodnight}$ complete successfully, doors are locked, garage is closed, lights are off, trash can is in the driveway, and no equipment is damaged.

Specifying and satisfying these two properties in smart homes needs us to tackle certain unique challenges. The first

[1] While security issues also abound, we believe such correctness violations are very common and under-reported as a pain point.

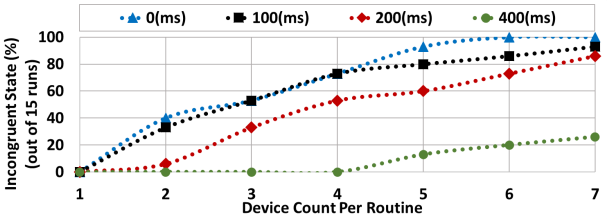


Figure 1: **Concurrency causes Incongruent End-state in a real smart home deployment.** *Two routines R1 (turn* ON *all lights) and R2 (turn* OFF *all lights) executed on a varying number of devices (x axis), with routine R2 starting a little after R1 (different lines). Y axis shows fraction of end states that are not serialized (i.e., all* OFF*, or all* ON*). Experiments with TP-Link smart devices [72].*

challenge comes from the human-facing nature of the environment. Every action of a routine may be *immediately visible to one or more human users*—we use the word "visible" to capture any action that could be sensed by any human user anywhere in the smart home. This requires us to clearly specify and reason about visibility models for concurrent routines. Visibility models provide notions of serial equivalence (i.e., serializability) of routines in a smart home.

Second, a smart home needs to optimize *user-facing metrics*—latency to start the routine, and also latency to execute it. This motivates us to explore a new spectrum of visibility models which trade off the amount of incongruence the user sees *during* execution vs. the user-perceived latency, all while guaranteeing serial-equivalence of the overall execution. Our visibility models are a counterpart to the rich legacy of weak consistency models that have been explored in mobile systems like Coda [40], databases like Bayou [66] and NoSQL [74], and shared memory multiprocessors [1].

Third, in a smart home, *device crashes and restarts are the norms*—any device can fail at any time, and possibly recover later. These failure/recovery events may occur during a command, before a command starts, or after a command has completed. Thus, reasoning about device failure/restart events while ensuring atomicity+visibility models is a new challenge. Today's failure handling is either silent or places the burden of resolution on the user.

Fourth, *long-running (or just long) routines are common* in smart homes. A long routine is one that contains at least one *long command*. A long command exclusively needs to control a device for an extended period, without interruption. Examples include a command to preheat an oven to $400^{\circ}F$, or to run north garden sprinklers for 15 minutes. Long commands cannot be treated merely as two short commands, as this would still allow the device to be interrupted by a concurrent routine in the interim, violating isolation. Long commands need to be treated as first-class commands.

Prior Work: These challenges have been addressed only piecemeal in literature. Some systems [24, 52] use priority-based approaches to address concurrent device access. Others [8] propose mechanisms to handle failures. A few systems [9, 43, 47] formally verify procedures. Transactuation [55] and APEX [80] discuss atomicity and isolation, but their concrete techniques deal with routine dependencies and do not consider users' experience—nevertheless, their mechanisms can be used orthogonally with SafeHome. None of the above address atomicity, failures, and visibility together.

The reader may also notice parallels between our work and the ACID properties (Atomicity, Consistency, Isolation, and Durability) provided by transactional databases [50]. While other systems like TinyDB [44] have drawn parallels between networks of sensors and databases (DBs), the techniques for providing ACID in databases do not translate easily to smart homes. The primary reasons are: i) our need to optimize latency (DBs optimize throughput); ii) device failure (DB objects are replicated, but devices are not, by default); and iii) the presence of long-running routines.

Contributions: We present *SafeHome*, a management system that provides atomicity and isolation among concurrent routines in a smart environment. For concreteness, we focus the design of SafeHome on smart homes (however, our evaluations look at broader scenarios). SafeHome is intended to run at an edge device in the smart home, e.g., a home hub or an enhanced access point. SafeHome does not require additional logic on devices; instead, it works directly with the APIs which devices naturally provide (commands are API calls). SafeHome can thus work in a smart home containing devices from multiple vendors.

The primary contributions of this paper are:

1. A new spectrum of *Visibility Models* trading off responsiveness vs. temporary congruence of smart home state.
2. Design and implementation of the SafeHome system.
3. A new way to reason about failures by *serializing failure events and restart events into* the serially-equivalent order of routines.
4. New *lock leasing* techniques to increase concurrency among routines, while guaranteeing isolation.
5. Workload-driven experiments to evaluate new visibility models and characterize tradeoffs.

SafeHome is best seen as the first step towards a grand challenge. A true OS for smart homes requires tackling myriad problems well beyond what SafeHome currently does. These include support for [2]: users to inject signals/interrupts/exceptions, safety property specification and satisfaction, leveraging programming language and verification techniques, and in general full ACID-like properties. SafeHome is an important building block over which (we believe) these other important problems can then be addressed.

2 Visibility and Atomicity

We first define SafeHome's two key properties–Visibility and Atomicity–and then expand on each.

- **SafeHome-Visibility/Serializability:** For simplicity, in this initial part of the discussion we ignore failures, i.e., we assume devices are always up and responsive. SafeHome-Visibility/Serializability means the *effect* of the concurrent

execution of a set of routines, is identical to an equivalent world where the same routines all executed serially, in some order. The interpretation of *effect* determines different flavors of visibility, e.g., identicality at every point of time, or in the end-state (after all routines complete), or at critical points in the execution. These choices determine the *spectrum* of visibility/serializability *models* that we will discuss soon.

- **SafeHome-Atomicity:** After a routine has started, either all its commands have the desired effect on the smart home (i.e., routine *completes*), or the system *aborts* the routine, resulting in a rollback of its commands, and gives the user feedback.

2.1 New Visibility Models in SafeHome

SafeHome presents to the user family a choice in how the effects of concurrent routines are visible. We use the term "visibility" to capture all senses via which a human user, anywhere in the environment, may experience immediate activity of a device, i.e., sight, sound, smell, touch, and taste. Visibility models that are more strict run routines sequentially, and thus may suffer from longer end-to-end latencies between initiating a routine and its completion (henceforth we refer to this simply as *latency*). Models with weaker visibility offer shorter latencies, but need careful design to ensure the end state of the smart home is congruent (correct).

Today's default approach is to execute routines' commands as they arrive, as quickly as possible, without paying attention to serialization or visibility. We call this *status quo* model as the *Weak Visibility (WV)* model, and its incongruent end states worsen quickly with scale and concurrency (see Fig. 1). We introduce three new visibility models.

1. **Global Strict Visibility (GSV):** In this strong visibility model, *the smart home executes at most one routine at any time.* In our SafeHome-Visibility definition (Sec. 2), the *effect* for GSV is "at every point of time", i.e., every individual action on every device. Consider a 2-family home where one user starts a routine $R_{dishwash}$ ={`dishwasher:ON; /*run dishwasher for 40 mins*/ dishwasher:OFF;`}, and another user simultaneously starts a second routine R_{dryer}= {`dryer:ON; /*run dryer for 20 mins*/ dryer:OFF;`}. If the home has low amperage, switching on both dishwasher and dryer simultaneously may cause an outage (even though these 2 routines touch disjoint devices). If the home chooses GSV, then the execution of $R_{dishwash}$ and R_{dryer} are serialized, allowing at most one to execute at any point of time. Because routines need to wait until the smart home is "free", GSV results in very long latencies to start routines. In GSV, a long-running routine also starves other routines.

2. **Partitioned Strict Visibility (PSV):** PSV is a weakened version of GSV that allows concurrent execution of non-conflicting routines, but limits conflicting routines to execute serially. For instance, for our earlier (GSV) example of

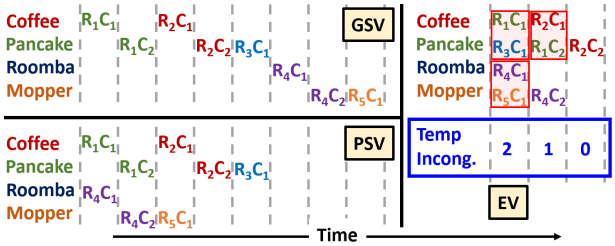


Figure 2: **Example routine execution in different visibility models:** *a) GSV b) PSV, c) EV. R_rC_c represents the c^{th} command of the r^{th} routine. In EV, red boxes show a pair of incongruent commands and the blue box shows the total number of temporary incongruences.*

$R_{dishwash}$ and R_{dryer} started simultaneously, if the home has no amperage restrictions, the users should choose PSV–this allows the two routines to run concurrently, and the end state of the home is (serially-) equivalent to the end state if the routines were instead to have been run sequentially (i.e., dishes washed, clothes dried). However, if the two routines *were* to touch conflicting devices, PSV would execute them serially.

3. **Eventual Visibility (EV):** This is our most relaxed visibility model which specifies that only *when all the routines have finished (completed/aborted), the end state of the smart home* is identical to that obtained if *all routines were to have been serially executed in some sequential (total) order*. In the definition of SafeHome-Visibility, the *effect* for EV is the end-state of the smart home after all the routines are finished.

EV is intended for the relatively-common scenarios where the desired final outcome (of routines) is more important to the users than the ephemerally-visible intermediate states. Unlike GSV, the EV model allows conflicting routines (touching conflicting devices) to execute concurrently–and thus reduces the latencies of both starting and running routines.

Consider two users in a home simultaneously initiating the same routine $R_{breakfast}$ = `{ coffee:ON; /*make coffee for 4 mins*/; coffee:OFF; pancake:ON; /*make pancakes for 5 mins*/; pancake:OFF; }`.

Both GSV and PSV would serially execute these routines because of the conflicting devices. EV would be able to pipeline them, overlapping the pancake command of one routine with the coffee command of the other routine. EV only cares that at the end both users have their respective coffees and pancakes.

Common Example – 3 Visibility Models: Fig. 2 shows an example with 5 concurrent routines executed for our three visibility models. This is the outcome of a real run of SafeHome running on a Raspberry Pi, over 5 devices connected via TP-Link HS-105 smart-plugs [69]. The routines are:
R_1: makeCoffee(Espresso); makePancake(Vanilla);
R_2: makeCoffee(Americano); makePancake(Strawberry);
R_3: makePancake(Regular);
R_4: startRoomba(Living room); startMopping(Living room);
R_5: startMopping(Kitchen);

GSV takes the longest execution time of 8 time units as it serializes execution. PSV reduces execution time to 5 time

	GSV	PSV	EV	WV
Concurrency	At most one routine	Non-conflicting routines concurrent	Any serializable routines concurrent	Any routines concurrent
End State	Serializable	Serializable	Serializable	Arbitrary
Wait Time: time to start routine	High	High for conflicting routines, low for non-conflicting routines	Low for all routines (modulo conflicts)	Low for all routines
User Visibility	Congruent at all times	Congruent at end, and at start/complete points of routines	Congruent at end	May be incongruent at anytime or end (Fig. 1)

Table 1: **Spectrum of Visibility Models in SafeHome.**

units by parallelizing unrelated commands, e.g., R_1's coffee command and R_4's Roomba command at time $t = 0$. EV is the fastest, finishing all routines by 3 time units. Average latencies (wait to start, wait to finish) are also fastest in EV, then PSV, then GSV. The figure shows that EV exhibits "temporary incongruence"–routines whose intermediate state is not serially equivalent. EV guarantees a temporary incongruence of zero when the last routine finishes.

Table 1 contrasts the properties of the four visibility models. Table 2 summarizes the examples discussed so far.

2.2 SafeHome-Atomicity

SafeHome-Atomicity states that after a routine has started, either all its commands have the desired effect on the smart home (i.e., routine *completes*), or the system *aborts* the routine, resulting in a rollback of its commands, and gives the user feedback. Due to the physical effects of smart home routines, we discuss three deviations from traditional notions of atomicity.

First, we allow the user to tag some commands as *best-effort*, i.e., optional. A routine is allowed to complete successfully even if any best-effort commands fail. Other commands, tagged as *must*, are required for routine completion—if any *must* command fails, the routine must abort. This tagging acknowledges the fact that users may not consider all commands in a routine to be equally important. For instance, a "leave-home-for-work" routine may contain commands which lock the door (must commands) and turn off lights (best-effort commands)—even if the lights are unresponsive, the doors must still lock. The user receives feedback about the failed best-effort commands, and she is free to either ignore or re-execute them.

Second, aborting a routine requires undoing past-executed commands. Many commands can be rolled back cleanly, e.g., command `turn Light-3 ON` can be undone by SafeHome issuing a command setting Light-3 to `OFF`. A small fraction of commands is impossible to physically undo, e.g., `run north sprinklers for 15 mins`, or `blare a test alarm`. For such commands, we undo by restoring the device to its state before the aborted routine (e.g., set the sprinkler/alarm state to `OFF`). Alternately, a user-specified undo-handler can be used.

Finally, we note that when a routine aborts, SafeHome provides feedback to the user (including logs), and she is free to either re-initiate the routine or ignore the failed routine.

3 Failure Handling and Visibility Models

Smart home devices could fail or become unresponsive, and then later restart. SafeHome needs to reason cleanly about failures or restarts that occur during the execution of concurrent routines. We only consider fail-stop and fail-recovery models of failures in the smart home (Byzantine failures are beyond our scope).

Because device failure events and restart events are visible to human users, our visibility models need to be amended. Consider a device D which routine R touches via one or more commands. D might fail *during* a command from R, or *after* its last command from R, or *before* its first command from R, or *in between* two commands from R. A naive approach may be to abort routine R in all these cases. However, for some relaxed visibility models like Eventual Visibility, if the failure event occurred anytime after completing the device's last command from R, then the event could be serialized to occur *after* the routine R in the serially-equivalent order (likewise for a failure/restart before the first command to that device from R, which can be serialized to occur before R).

Thus a key realization in SafeHome is that we need to *serialize failure events and restart events alongside routines themselves*. We can now restate the SafeHome-Visibility property from Sec. 2, to account for failures and restarts:

SafeHome-Visibility/Serializability (with Failures and Restarts): The *effect* of the concurrent execution of a set of routines, occurring along with concurrent device failure events and device restart events, is identical to an equivalent world where the same routines, device failure events, and device restart events, all occur sequentially, in some order [2]

First, we define the failure/restart event to be the event when the edge device (running SafeHome) *detects* the failure/restart (this may be different from the actual time of failure/restart). Second, failure events and restart events *must* appear in the final serialized order. On the contrary, routines may appear in the final serialized order (if they complete), or not appear (if they abort). We next reason explicitly about failure serialization for each of our visibility models from Sec. 2.1.

1. Failure Serialization in Weak Visibility: Today's Weak Visibility has no failure serialization. Routines affected by failures/restarts complete and cause incongruent end-states.

2. Failure Serialization in Global Strict Visibility: Because GSV intends to present the picture of a single serialized home to the user, if *any* device failure event or restart event were to occur while a routine is executing (between its start and finish), the routine must be aborted. There are two subflavors herein: (A) *Basic GSV or Loose GSV (GSV)*: Routine aborts only if it contains at least one command that touches failed/restarted device; (B) *Strong GSV (S-GSV)*: Routine aborts even if it does not have a command that touches

[2] This idea has analogues to distributed systems abstractions such as view/virtual synchrony, wherein failures and multicasts are totally ordered [15]. We do not execute multicasts in the smart home.

Example Routines	Scenario and Possible Behavior	SafeHome Feature	
"cooling"={window:CLOSE; AC:ON;}	If executed partially, can leave window open and AC on (wasting energy) or the window closed and AC off (overheating home).	Atomicity	
"make coffee"= {coffee:ON; /*make coffee for 4 mins*/ ; coffee:OFF;}	Coffee maker should not be interrupted by another routine. E.g, user-1 invokes make coffee, and in the middle, user-2 independently invokes make coffee.	Long running routines & mutually exclusive access	
R_1={dishwasher:ON; (dishwasher runs for 40 mins); dishwasher:OFF;} R_2={dryer:ON; (dryer runs for 20 mins); dryer:OFF;}	If home has low amperage, simultaneously running two power-hungry devices may cause outage (GSV).	Global Strict Visibility (GSV)	
R_1={coffee:ON; /*make coffee for 4 mins*/; coffee:OFF;} R_2={lights:ON, fan:ON}	Two routines touching disjoint devices should not block each other (PSV).	Partitioned Strict Visibility (PSV), closest to [55]	
"breakfast"={coffee:ON; /*make coffee for 4 mins*/; coffee:OFF, pancake:ON; /*make pancakes for 5 mins*/; pancake:OFF; }	Two users can invoke this same routine simultaneously. The two routines can be pipelined thus allowing some concurrency without affecting correctness (EV). (Both GSV and PSV would have serialized them.)	Eventual Visibility (EV)	
"leave home"={lights:OFF (Best-Effort); door:LOCK;}	Requiring all commands to finish too stringent, so only second command is Must (required). If light unresponsive, door must lock, otherwise routine aborts.	Must and Best-Effort commands	
"manufacturing pipeline" with k stages and $\{R_1, R_2, ..., R_k\}$ routines	If any stage fails, entire pipeline must stop immediately.	Strong GSV serialization (S-GSV)	Failure Serialization
"cooling"={window:CLOSE; AC:ON;}	If *anytime* during the routine (from start to finish), the AC fails or window fails, the routine is aborted.	Loose GSV serialization (GSV)	
"cooling"={window:CLOSE; AC:ON;}	If window fails after its command *and* remains failed at finish point of routine, routine is aborted.	PSV serialization	
"cooling"={window:CLOSE; AC:ON;}	If window fails after it is closed (but before AC is accessed), routine completes successfully–window failure can be serialized after routine.	EV serialization	

Table 2: **Example scenarios in a smart home, and SafeHome's corresponding features.**

failed/restarted device. A routine R_{shade} on living room shades can complete, if master bathroom shades fail, in GSV but not S-GSV. In S-GSV, the final serialization order contains the failure/restart event but not the aborted routine R_{shade}. In GSV, the final serialization order contains both R_{shade} (which completes) and the failure/restart event, in arbitrary order.

3. Failure Serialization in Eventual Visibility: For a given set of routines (and concurrent failure events and restart events), the *eventual (final)* state of the actual execution is equivalent to the end state of a world wherein the final successful routines, failure device events, and failure restart events, all occurred in some serial order.

Consider routine R, and the failure event (and potential restart event) of one device D. Four cases arise:

1. If D is not touched by R, then D's failure event and/or restart event can be arbitrarily ordered w.r.t. R.
2. If D's failure and restart events both occur *before R first touches the device*, then the failure and restart events are *serialized before* R.
3. If D's failure event occurs *after the last touch of D by R*, then D's failure event (and eventual restart event) are *serialized after* R.
4. In all other cases, routine R aborts due to D's failure. R does not appear in the final serialized order.

These are applicable to each concurrent routine accessing D.

4. Failure Serialization in Partitioned Strict Visibility: This is a modified version of EV where we change condition 3 (from 1-4 in EV above) to the following:

*3**. If D's failure event occurs *after the last touch of D by R*, and *has recovered when R reaches its finish point*, then D's failure event and restart event are *serialized right after R*.

Example—Effect of Failure on Three Visibility Models: Consider the routine from Section 1, $R_{cooling}$:= {CLOSE window; switch ON AC;}. Suppose the "window" device

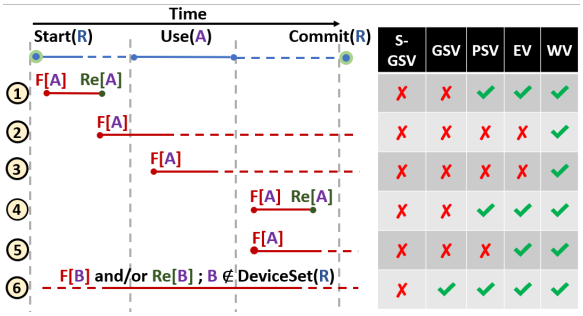


Figure 3: **Failure Serialization: 6 cases, and their handling in Visibility Models.** ✓ *- execute routine, X - abort routine. At F[A] /Re[A] the edge device detects the failure/restart (resp.) of device A.*

fails concurrently with the routine (between its start and finish times). GSV always aborts $R_{cooling}$ regardless of when the window failed. PSV aborts $R_{cooling}$ only if the window remains failed at $R_{cooling}$'s finish point. EV does *not* need to abort $R_{cooling}$ if window fails any time after $R_{cooling}$'s first command has completed successfully, even if window remains failed at $R_{cooling}$'s finish time. EV places the window failure event after $R_{cooling}$ in the serialization order, and the smart home's end state is equivalent. If the window fails and restarts before $R_{cooling}$'s first command, EV serializes the failure and restart before $R_{cooling}$, and executes $R_{cooling}$ correctly. Thus EV has the least chance of aborting a routine due to a failure.

Table 2 summarizes all our examples so far and Fig. 3 summarizes our failure handling rules.

4 Eventual Visibility: SafeHome Design

In order to maintain correctness for Eventual Visibility (i.e., serial-equivalence), SafeHome requires routines to lock devices before accessing them. Because long routines can hold locks and block short routines, we introduce *lock leasing* across routines (Sec. 4.1). This information is stored in the *Locking Data-structure* (Sec. 4.2). The *lineage table* ensures invariants required to guarantee Eventual Visibility (Sec. 4.3).

4.1 Locks and Leasing

SafeHome prefers Pessimistic Concurrency Control (PCC): SafeHome adopts pessimistic concurrency control among routines, via (virtual) locking of devices. Abort and undo of routines are disruptive to the human experience, causing (at routine commit point) rollbacks of device states across the smart home. Our goal is to minimize abort/undo only to situations with device failures, and avoid aborts because routines touch conflicting devices. Hence we eschew optimistic concurrency control approaches and use locking [3].

SafeHome uses *virtual locking* wherein each device has a virtual lock (maintained at the edge device running SafeHome), which must be acquired by a routine before it can execute any command on that device. A routine's lock acquisition and release do not require device access, and are not blocked by device failure/restart.

In order to prevent a routine from aborting midway because it is unable to acquire a lock, SafeHome uses *early lock acquisition*—a routine acquires, at its start point, the locks of all the devices it wishes to touch. If any of these acquisitions fails, the routine releases all its locks immediately and retries lock acquisition. Otherwise, acquired locks are released (by default) only when the routine finishes.

Leasing of Locks: To minimize chances of a routine being unable to start because of locks held by other routines, SafeHome allows routines to lease locks to each other. Two cases arise: 1) routine R_1 holds the lock of device D for an extended period *before* R_1*'s first access* of D, and 2) R_1 holds the lock of device D for an extended period *after* R_1*'s last access* of D. Both cases prevent a concurrent routine R_2, which also wishes to access D, from starting.

SafeHome allows a routine $R_{src}(=R_1)$ holding a lock (on device D) to *lease the lock* to another routine $R_{dst}(=R_2)$. When R_{dst} is done with its last command on D, the lock is returned back to R_{src}, which can then normally use it and release it. We support two types of lock leasing:

- **Pre-Lease:** R_{src} has started but has not yet accessed D. A lease at this point to R_{dst} is called a *pre-lease*, and places R_{dst} *ahead* of R_{src} in the serialization order. After R_{dst}'s last access of D, it returns the lock to R_{src}. If R_{src} reaches its first access of D before the lock is returned to it, R_{src} waits. After the lease ends, R_{src} can use the lock normally.
- **Post-Lease:** R_{src} is done accessing device D, but the routine itself has not finished yet. A lease at this point to R_{dst} is called a *post-lease*, and places R_{dst} *after* R_{src} in the serialization order. If R_{src} finishes before R_{dst}, the lock ownership is permanently transferred to R_{dst}. Otherwise, R_{dst} returns the lock when it finishes.

A prospective pre/post-lease is disallowed if a previous action (e.g., another lease) has already determined a serialization order between R_{src} and R_{dst} that would be contradicted by this prospective lease. In such cases R_{dst} needs to wait until R_{src}'s normal lock release. Further, a post-lease is not allowed if at least one device D is written by R_{src} and then read by R_{dst}. This prevents SafeHome from suffering dirty reads from aborted routines. We prevent scenarios like this–R_{src} switches on a light, and R_{dst} has a conditional clause based on that light's status, but R_{src} subsequently aborts. Cascading aborts are handled in [55], whose techniques can be used orthogonally with ours.

To prevent starvation, i.e., from R_{src} waiting indefinitely for the returned lock, leased locks are revoked after a timeout. The timeout is calculated based on the estimated time between R_{dst}'s first and last actions on D, multiplied by a leniency factor (we use $1.1\times$). Lock revocation before R_{dst}'s last access of D causes R_{dst} to abort.

[3] For the limited scenarios where routines are known to be conflict-free, optimistic approaches may be worth exploring in future work.

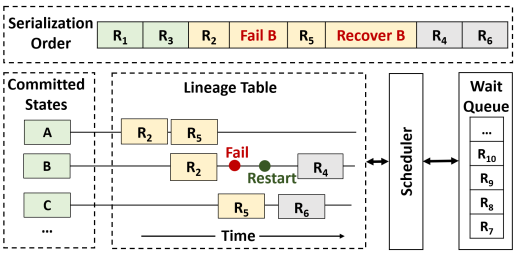


Figure 4: **SafeHome's Architecture for Eventual Visibility.**

4.2 Locking Data-structure

SafeHome adopts a state machine approach [58] to track current device states, future planned actions by routines, and a serialization order. SafeHome maintains, at the edge device (e.g., Home Hub or smart access point), a *virtual locking table data-structure* (Fig. 4). This contains:

- *Wait Queue:* Queue of routines initiated but not started. When a routine is added, it is assigned an incremented routine ID.
- *Serialization Order:* Maintains the current serialization order of routines, failure events, and restart events. For completed routines (shaded green), the order is finalized. All other orders are tentative and may change, e.g., based on lock leases. Failure and restart events may be moved flexibly among unfinished routines.
- *Lineage Table:* Detailed in Section 4.3, this maintains, for each device, a *lineage*: the *planned* transition order of that device's lock.
- *Scheduler:* Decides when routines from Wait Queue are started, acquires locks, and maintains serialization order.
- *Committed States:* For each device, keeps its last committed state, i.e., the effect of the last successfully routine. This may be different from device's actual state, and is needed to ensure serialization and rollbacks under aborts.

4.3 Lineage Table

The *lineage* of a device represents a temporal plan of when the device will be acquired by concerned routines. The lineage of

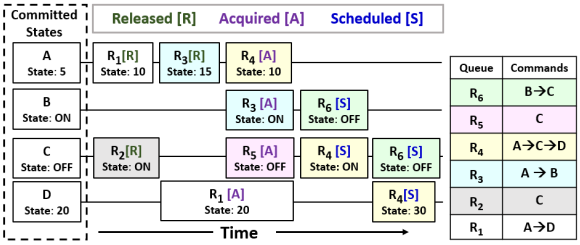


Figure 5: **Sample Lineage Table, with 6 routines. Some fields are omitted for simplicity.**

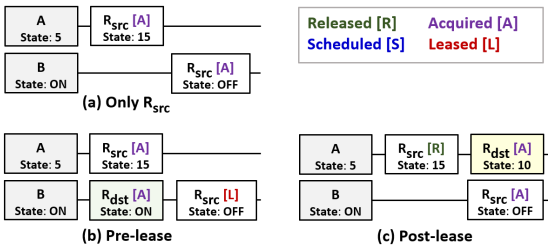


Figure 6: **Lineage table with Lock Leasing**. *a) Lineage before leasing with only R_{src}, b) Pre-lease to R_{dst} that only accesses device B, and c) Post-lease to R_{dst} that only accesses device A.*

a device starts with its latest committed state, followed by a sequence of *lock-access* entries (Fig. 5)–these are "stretched" horizontally. A width of a lock-access entry represents how long that routine will acquire that lock. Each lock-access entry for device D consists of: *i.* A routine ID, *ii.* Lock *status* (`Released, Acquired, Scheduled`) *iii.* Desired device state by the command (e.g., `ON/OFF`) and *iv.* Times: a start time ($T_{start}(R_i)$), and duration ($\tau_{R_i}(D)$) of the lock-access.

In the example of Fig. 5, a `Scheduled [S]` status indicates that the routine is scheduled to access the lock. An `Acquired [A]` status shows it is holding and using the lock. A `Released [R]` status means the routine has released the lock.

The duration field, $\tau_{R_i}(D)$, is set either based on known time to run a long command (e.g., run sprinkler for 15 mins), or an estimate of the command execution time. Our implementation uses a fixed $\tau_{R_i}(D) = \tau_{timeout}$ for all short commands (100ms based on our experience). $\tau_{R_i}(D)$ is also used to determine the revocation timeout for leased locks, along with a multiplicative leniency factor (1.1 in our implementation).

To maintain serializability, four key invariants are assured:

Invariant 1 (Future Mutual Exclusion: Lock-accesses in a device's lineage list do not overlap in time): No device is planned to be locked by multiple routines. Gaps in its lineage list indicate times the device is free.

Invariant 2 (Present Mutual Exclusion: At most one `Acquired` lock-access exists in each lineage list): No device is locked currently by multiple routines.

Invariant 3 (Lock-access [R]⟶[A]⟶[S]): In each lineage list, all `Released` lock-access entries occur to the left of (i.e., before) any `Acquired` entries, which in turn appear to the left of any `Scheduled` entries.

Invariant 4 (Consistent "serialize-before" ordering among lineages): Given two routines R_i, R_j, if there is at least one device D such that: lock-access$_D(R_i)$ occurs to the left of lock-access$_D(R_j)$ in D's lineage list, then for every other device D' touched by both R_i, R_j, it is true that: lock-access$_{D'}(R_i)$ occurs to the left of lock-access$_{D'}(R_j)$. Hence R_i is *serialized-before* R_j.

Transition of Lock-accesses: The status of lock-accesses changes upon certain events. First, when a routine's last access to a device ends, the `Acquired` lock-access ends, and transitions to `Released`. The next `Scheduled` lock-access turns to `Acquired`: i) either immediately (if no gap exists, e.g., R_4 after R_5 releases C in Fig. 5), or ii) after the gap has passed, e.g., R_4 after R_1 releases D in Fig. 5.

Second, when scheduling a new routine R (from the wait queue), a `Scheduled` lock-access entry is added to all device lineages that R needs (e.g., R_6 in Fig. 5 adds lock-accesses for B and C). Third, when a routine finishes (completes/aborts), all its lock-access entries are removed, releasing said locks. If the routine completed successfully, committed states are updated. For an abort, device states are rolled back.

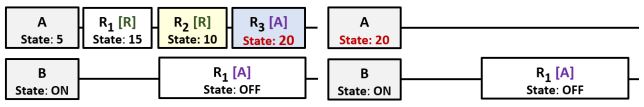
Leasing of Locks: Consider a pre-lease from R_{src} to R_{dst} (Fig. 6(b)). First, a new `Acquired` lock-access for R_{dst} is placed *before* (to the left of) the lock-access of R_{src} in the lineage table. Second, the lock-access of R_{src} is changed to "Leased (R_{dst})" status.

Figure 6(c) shows a post-lease: a new `Acquired` lock-access of R_{dst} is placed *after* (to the right of) the lock-access of R_{src} and the lock-access of R_{src} changes to `Released`.

Aborts and Rollbacks: For an aborted routine R_i, we roll back states of only those devices D in whose lineage R_i appeared. For a device D, there are two cases:

- *Device D was last* `Acquired` *by routine R_j ($\neq R_i$):* We remove R_i's lock-access from D's lineage. This captures two possibilities: a) R_i never executed actions on D (e.g., Fig. 5: device C when aborting R_4), or b) R_i leased D to another routine R_j, and since R_i is aborting, R_j's effect will be the latest (e.g., Fig. 5: device A when aborting R_1).
- *Device D was last* `Acquired` *by routine R_i* (e.g. device C when aborting R_5 in Fig. 5): We: 1) remove the R_i's lock-access from D's lineage, and 2) issue a command to set D's status to R_i's *immediately left/previous* lock-access entry in the lineage (if none exist, use Committed State), unless the device is already in this desired state.

Committing (Successfully Completing) a routine: When a routine reaches its finish point, it commits (completes successfully) by: i) updating Committed States, and ii) removing its lock-access entries. R_j might appear after R_i in the serialization order but complete earlier, e.g., due to lock leasing. SafeHome allows such routines to commit right away by using *commit compaction*–routines later in the serialization order will overwrite effects of earlier routines (on conflict-



(a) Before commit (b) After R_3 commits

Figure 7: **Commit with compaction.**

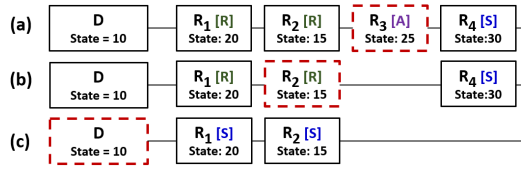


Figure 8: **Inferring the current device status**. *The dashed boxes point to the current device status in three different scenarios.*

ing devices). This is similar to "last writer wins" in NoSQL DBs [74]. Concretely, for all common devices we remove both R_i's lock-access, and all lock-accesses before it (Fig. 7).

Current Device Status: A device's current status is needed at several points, e.g., abort. Due to uncompleted routines, the actual status may differ from the committed state. The lineage table suffices to estimate a device's current state (without querying the device). Fig. 8 shows the three different cases. (a) If an `Acquired` lock-access entry exists, use it (e.g., R_3 in Fig. 8(a) with $D = 25$). (b) Otherwise, if lock-accesses exist with lock status `Released`, use the right-most entry (e.g., R_2 in Fig. 8(b) with $D = 15$). (c) Otherwise, use the Committed State entry (e.g., committed state $D = 10$ in Fig. 8(c)).

5 Scheduling Policies for Eventual Visibility

When a new routine arrives, SafeHome needs to "place" it in the serialization order, adhering to invariants of Sec. 4.3. This is the scheduling problem. We present three alternatives.

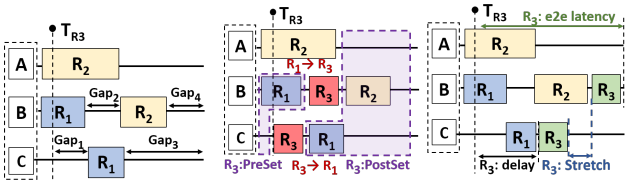
First Come First Serve (FCFS) Scheduling: Routines are serialized in order of arrival. When a routine arrives, its lock-access entries are *appended* to the lineage table. FCFS avoids pre-leases as they would violate serialization order. Post-leases are allowed.

FCFS is attractive if a user expects routines to execute in the order they were initiated. However, FCFS prolongs time between routine submission and start.

Just-in-Time (JiT) scheduling: JiT greedily places a new routine at the *earliest position (in the lineage) when it is eligible to start.* JiT triggers an *eligibility test* upon either: (i) each routine arrival, or (ii) on every lock release. The eligibility test greedily checks for routine R if it can now acquire all its locks, either right away, or via pre-leases or post-leases. For case (ii) we run the eligibility test only on those waiting routines that desire the released device. To mitigate starvation, we use a per-routine TTL (Time To Live)—when a waiting routine R's TTL expires, R is prioritized to start next (ties broken by arrival order).

Timeline(TL) Scheduling: This flexible policy uses estimates of lock-access durations, and *speculatively* places waiting routines into the lineage table based on these estimates. This means no routines need to wait (for an eligibility test) before being added to the lineage table. TL scheduling tries to place routines in the gaps in the lineage table without violating the lineage table invariants (Section 4.3). An example is shown in Figures 9a, 9b. Figure 9c shows that TL may "stretch" a routine's execution time due to lock waits during execution. To mitigate this, a new routine is delayed from starting (now) if this were to cause TL to stretch some running routine beyond a pre-specified threshold.

TL scheduling uses a *backtrack-based search strategy* to find the best placement for a new routine in the lineage table. Algo. 1 shows the pseudocode. We explain via an example. Fig. 9a depicts a lock table right before routine $R_3 = \{C \rightarrow B\}$ arrives at time T_{R3}, and has four gaps in the lineage. Starting with the first device in the routine (C for R_3): $\tau_{R_3}(C)$ (Line 3), the Timeline scheduler finds the first gap in C's lineage that



(a) (b) (c)

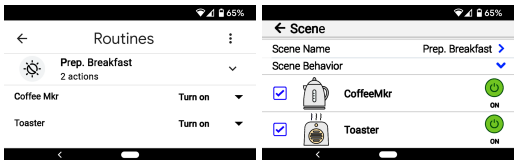
Figure 9: **Timeline Scheduler (TL) example** *a) before scheduling R_3 b) trying a potential (but invalid) schedule, c) scheduling R_3 at the first possible gap.*

Algorithm 1 Timeline scheduling of routine R

```
1:  function SCHEDULE(R, index, startTime, preSet, postSet)
2:      devID = R[index].devID
3:      duration = lock_access(R, devID).duration
4:      //return from recursion
5:      if R.cmdCount < index then
6:          return true
7:      end if
8:      //Find gap and pre- and post-set
9:      gap = getGap(devID , startTime, duration)
10:     curPreSet = preSet ∪ getPreSet(lineage[devID], gap.id)
11:     curPostSet = postSet ∪ getPostSet(lineage[devID], gap.id)
12:     if curPreSet ∩ curPostSet = ∅ then
13:         //Serialization is not violated
14:         canSchedule = schedule(R, index + 1, gap.startTime +
        duration , curPreSet, curPostSet)
15:         if canSchedule then
16:             lineage[devID].insert(R[index], gap)
17:             return true
18:         end if
19:     end if
20:     //backtrack: try next gap
21:     return schedule(R, index, gap.startTime + duration , preSet,
        postSet)
22: end function
```

```
{"RoutineName":"Prep. Breakfast", "CommandList":
[{"DevID":"CoffeeMkr", "Action":"ON", "Priority":"MUST"},
 {"DevID":"Toaster", "Action":"ON", "Priority":"MUST"}
]}
```


(a) JSON representation of SafeHome routine (part)



(b) G. Home routine [28] (c) TP-Link routine [70]

Figure 10: **Defining a routine "Prepare Breakfast"** *Two commands: i)Turn* `ON` *Coffee Maker and ii) Turn* `ON` *Toaster.*

can fit $\tau_{R_3}(C)$ (Line 9). This is Gap 1 in Fig. 9a. Next, the Timeline scheduler validates that this gap choice will not violate previously decided serializations. For the scheduled lock-accesses of R_3 so far, it builds two sets: a) *preSet:* the union of all (executing and scheduled) routines placed *before* R_3's lock-accesses ($\{R_1\}$ in Fig. 9b), and b) *postSet:* the union of all (executing and scheduled) routines placed *after* R_3's lock-accesses ($\{R_1, R_2\}$ in Fig. 9b). The preSet and postSet of R represent the routines positioned before and after R, respectively, in the serialization order. The gap choice is valid *if and only if* the intersection of the preSet and the postSet is empty. If true, the scheduler moves on to the next command of the routine. Otherwise (Fig. 9b), the scheduler backtracks and tries the next gap (Line 21). The process repeats.

6 SafeHome Implementation

We implemented SafeHome in 1200 core lines of Java. SafeHome runs on an edge device, such as a Home Hub or an enhanced/smart access point. Our edge-first approach has two major advantages: 1) SafeHome can be run in a smart home containing devices from a diverse set of vendors, and 2) SafeHome is autonomous, without being affected by ISP/external network outages [20, 78] or cloud outages [3, 29, 30].

SafeHome works directly with the APIs exported by devices—commands in routines are programmed as API calls directly to devices. SafeHome's routine specification is compatible with other smart home systems (Fig. 10). Our current implementation works for TP-Link smart devices [71, 72], using the HS110Git [68] device-driver. Other devices (e.g., Wemo [75]) can be supported via their device-drivers.

Fig. 11 shows our implementation architecture. When a user submits routines, they are stored in the *Routine Bank*, from where they can be invoked either by the user or triggers, via the *Routine Dispatcher*. The *Concurrency Controller* runs the appropriate Visibility model's implementation. Apart from Eventual Visibility (Sec. 5), we also implemented Global Strict Visibility (GSV), and Partitioned Strict Visibility (PSV), with failure/restart serialization. Our Weak Visibility reflects today's laissez-faire implementation.

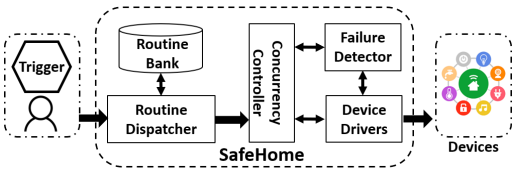


Figure 11: **SafeHome Architecture**

The *Failure Detector* explicitly checks devices by periodically (1 sec) sending ping messages. If a device does not respond within a timeout (100 ms by default), the failure detector marks it as `failed`. We also leverage *implicit* failure detection by using the last heard SafeHome TCP message as an implicit ack from the device, reducing the rate of pings.

7 Experimental Results

We evaluate SafeHome using both workloads based on real-world deployments, and microbenchmarks. The major questions we address include:

1. Are relaxed visibility models (like Eventual Visibility) as responsive as Weak Visibility, and as correct as Global Strict Visibility (Sec. 2.1)?
2. What effect do failures have on correctness and user experience (Sec. 3)?
3. Which scheduler policy (Sec. 5) is the best?
4. What is the effect of optimizations, e.g., lock leasing, commit compaction, etc. (Sec. 4)?

7.1 Experimental Setup

We wish to evaluate SafeHome for a variety of scenarios and parameters. Hence we run our implementation over an emulation, using both real-world workloads (Sec. 7.2) and synthetic workloads (Sec. 7.3 - 7.6).

Metrics: Because of the human-visible nature of SafeHome, our primary evaluation metrics are also human-visible:

End to end latency (or Latency): Time between a routine's submission and its successful completion.

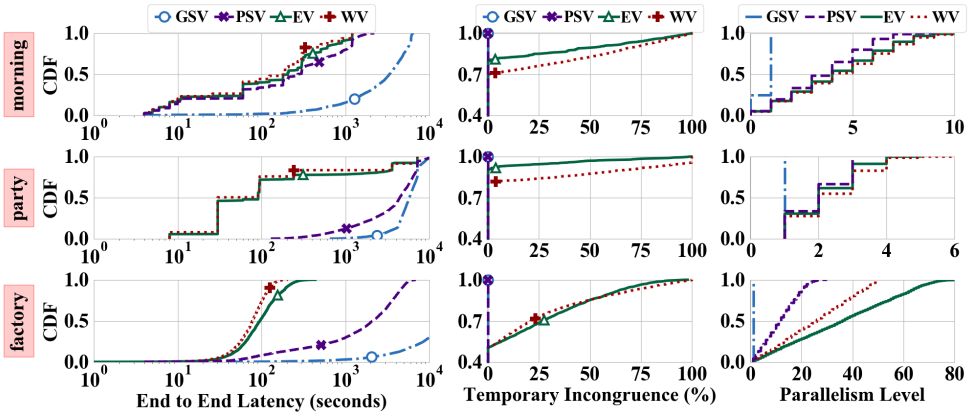
Temporary Incongruence: This metric measures how much the human user's actual experience differs from a world where all routines were run serially. We take worst case behavior. Before a routine R completes, if another routine R' changes the state of *any* device R modified, we say R has suffered a temporary incongruence event. The *Temporary Incongruence* metric measures the fraction of routines that suffer at least one such temporary incongruence event.

Final Incongruence: Final Incongruence measures the ratio of runs that end up in an incongruent state.

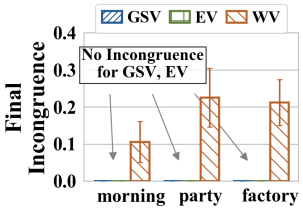
Parallelism level: This efficiency/utilization metric is the number of routines that are allowed by SafeHome to execute concurrently, averaged throughout the run. To avoid domination by durations when only 0 or 1 routines run, we only measure the metric at points when a routine starts/ends.

7.2 Experiments with Real-World Benchmarks

We extracted traces from three real homes (20-30 devices, multi-user families) who were using Google Home, over 2



(a) **Latency, Temporary Incongruence, and Parallelism for Three Scenarios.** *To identify lines we show one label symbol for each (plot has many more data points). Some GSV lines may be cut to show separation between other models.*



(b) **Final Incongruence.** *Run with 9 routines, 100 runs per scenario, and checks if final smart home state is equivalent to some serial ordering of routines (9! possibilities). Final Incongruence measures the ratio of end states that were not congruence out of 100 runs.*

Figure 12: **Experiment Results with Trace-Based Scenarios**

years. We also studied two public datasets: 1) 147 SmartThings applications [63]; and 2) IoTBench: 35 OpenHAB applications [39]. Based on these, we created three representative benchmarks: (We will make these available openly.)

Morning Scenario: This chaotic scenario has 4 family members in a 3-bed 2-bath home concurrently initiating 29 routines over 25 minutes touching 31 devices. Each user starts with a wake-up routine and ends with the leaving home routine. In between, routines cover bedroom & bathroom use, breakfast cook + eat, and sporadic routines, e.g., milk spillage cleanup.

Party Scenario: Modeling a small party, it includes one long routine controlling the party atmosphere for the entire run, along with 11 other routines covering spontaneous events, e.g., singing time, announcements, serving food/drinks, etc.

Factory Scenario: This is an assembly line with 50 workers at 50 stages. Each stage has access to local devices, to some devices shared with immediately preceding and succeeding stages, and to 5 global devices. Each stage's routine has device access probabilities: 0.6 for local devices, 0.3 for neighbor devices, and 0.1 for global devices. Routines are generated to keep each worker occupied (no idle time).

We trigger routines at random times while obeying preset constraints capturing real-life logic, e.g., "wake-up" routine before "cook breakfast" routine. In the morning scenario, each routine occurs once per run, and for the factory scenario routines are probabilistically generated (with possible repetition). We run 1000 trials to obtain each datapoint.

Results: From Fig. 12a (top row), in the morning scenario: 1) EV's latency is comparable to WV at both median and 95^{th} percentile, and 2) PSV has 15% worse 90^{th} percentile latency than EV. Generally, the higher the parallelism level (last column), the lower the latency. For instance, EV has a median parallelism level 3× higher than GSV, and median latency 16× better than GSV. Parallelism creates more temporary incongruences (middle column of figure). This is expected for EV. Yet, EV's (and GSV's) end state is serially equivalent while WV may end incongruently–this is shown in Fig. 12b. Thus *EV offers similar latencies as, but better final congruence than, WV.* Only if the user cares about temporary incongruence is PSV preferable.

In Fig. 12a (middle row), the party scenario shows similar trends to the morning scenario with one notable exception. PSV's benefit is lower, with only 11% 90th percentile latency reduction from GSV (vs. 77% in morning). This occurs because the single long routine blocks other routines. EV avoids this head-of-line blocking because of its pre- and post-leasing.

In Fig. 12a (bottom row), the factory scenario shows similar trends to morning scenario, except that: (i) EV's median latency is 23.1% worse than WV, and (ii) the parallelism level is higher in EV than WV. This is due to the back-to-back arrival of multiple routines. WV executes them as-is. However, EV may delay some routines (due to device conflicts)–when the conflict lifts, all eligible routines run simultaneously, increasing our parallelism level and latency.

7.3 Workload-Driven Emulation: Parameters

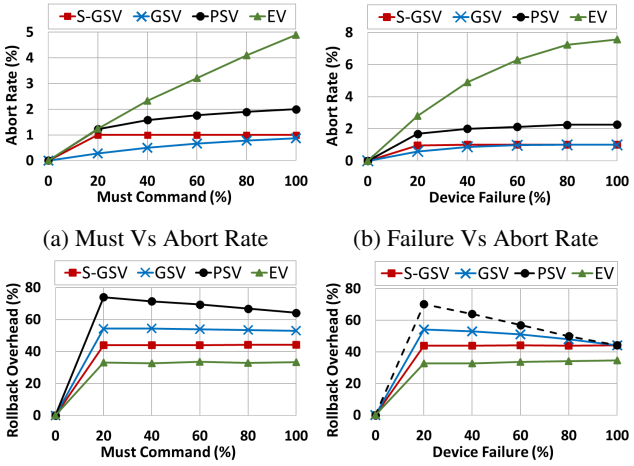
The rest of this section performs workload-driven experiments. Table 3 summarizes the parameters used. By default we run 100 routines, 25 devices, and an average of 3 commands per routine. Each routine has a 10% probability of being long-running. We run 1M trials to obtain each datapoint.

7.4 Atomicity Evaluation: Effect of Failures

Failures abort more routines in EV because it allows high concurrency, yet EV's intrusive effect on the user (due to aborts) is the lowest of all visibility models. Fig. 13a and 13b measure the fraction of routines aborted due to a failure. We induce fail-stop failures, where 25% of the total devices were marked as failed at a random point during the run. Yet Fig. 13c and 13d show that the *rollback overhead* of EV is smallest among all visibility models–this is the average fraction of commands rolled back, across aborted routines.

Name	default	Description
$\mathcal{R}$	100	Total number of routines
ρ	4	Number of concurrent routines injected
$\mathcal{C}$	3	Average commands per routine (ND)
α	0.05	Zipfian coefficient of device popularity
$\mathcal{L}_{\%}$	10%	Percentage of long running routines
$\lvert\mathcal{L}\rvert$	20 min.	Average duration of a long running command (ND)
$\lvert\mathcal{S}\rvert$	10 sec.	Average duration of a short running command (ND)
$\mathcal{M}$	100%	Percentage of "Must" commands of a routine
$\mathcal{F}$	0%	Percentage of the failed devices

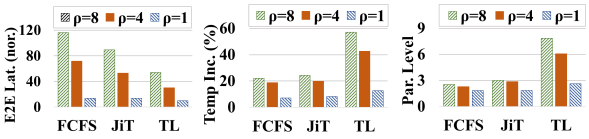
Table 3: **Parameterized Microbenchmark: Summary of Parameters.** *ND = Normal distribution.*



(a) Must Vs Abort Rate (b) Failure Vs Abort Rate

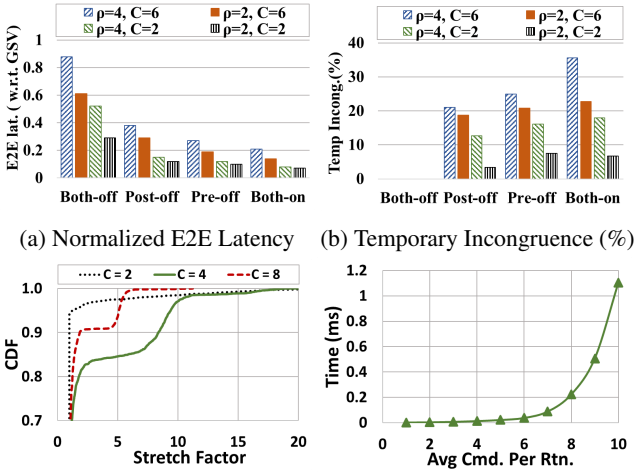
(c) Must Vs Rollback Overhead (d) Failure Vs Rollback Overhead

Figure 13: **Effect of Failures.** *Rollback Overhead = Intrusion on User. Parameters in Table 3.*



(a) E2E Latency (b) Incongruence (c) Parallelism

Figure 14: **Scheduling Policies.** *Parameters in Table 3. (a) E2E Latency normalized with routine runtime. (b) Temporary Incongruence. (c) Parallelism Level.*



(a) Normalized E2E Latency (b) Temporary Incongruence (%)

(c) CDF of Stretch Factor (d) Algo. 1 Insertion Time

Figure 15: **TL Scheduler under EV.** *Parameters in Table 3.*

PSV's rollback overhead is higher than EV as it aborts more at the routine's finish point (when checking up/down status of devices touched). EV aborts affected routines earlier rather than later. GSV and S-GSV have low abort rates because of their serial execution but have higher rollback overheads than EV. Thus, even *when execution is serial, the effect of failures can be more intrusive on the human*. We conclude that EV is the least intrusive model.

The plateauing in Figures 13a, 13b is due to saturation of parallelism level. The plateauing in Figs. 13c, 13d is due to saturation at abort-points–for GSV at 50%, with S-GSV lower at 40% since *any* device failure triggers the abort.

7.5 Scheduling Policies

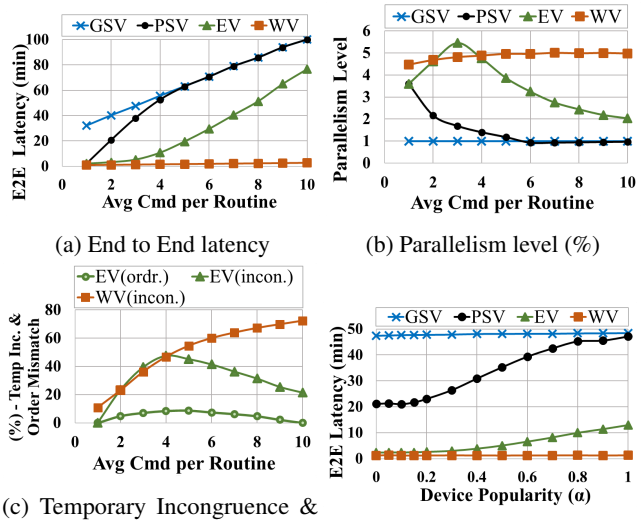
Fig. 14 compares FCFS, JiT, and Timeline (TL) scheduling policies (Sec. 5). In Fig. 14a with $\rho = 4$ concurrent routines, TL is 2.36× and 1.33× faster than FCFS and JiT respectively. The benefit of TL over FCFS is due to pre-leasing. The benefit of TL over JiT is due to opportunistic use of leasing. TL also has higher parallelism level (Fig. 14c) than FCFS (2.3× at $\rho = 4$) and JiT (2.0× $\rho = 4$).

7.5.1 Timeline-based Eventual Visibility (TL)

Fig. 15a and 15b show that disabling leasing reduces temporary incongruence but significantly increases latency. Turning off *both* pre and post leasing increases latency (from Both-on to Both-off) by between 3× to 5.5× (as concurrency level ρ and commands per routine $\mathcal{C}$ are varied). Post-leases are more effective than pre-leases: disabling the former raises latency by between 71% to 107%, while disabling the latter raises latency from between 29% to 50%. Post-leasing opportunities are more frequent than pre-leasing ones because the former does not require changing the serialization order (the latter does). These trends are true for all combinations of $\rho, \mathcal{C}$.

TL might also "stretch" routines (Fig. 9c). Fig. 15c shows *stretch factor*, measured as the time between a routine's actual start (not submission) and actual finish, divided by the ideal (minimum) time to run the routine. With routine size, stretch factor rises at first (at $\mathcal{C} = 2$ only 5% routines have stretch > 1, vs. 25% at $\mathcal{C} = 4$) but then drops (15% at $\mathcal{C} = 8$). Essentially the lock-table saturates beyond a $\mathcal{C}$, creating fewer gaps and forcing EV to append new routines to the schedule.

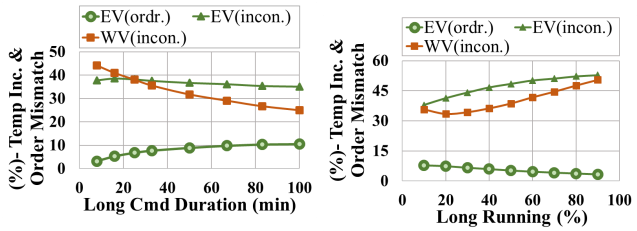
We used a Raspberry Pi 3 B+ [51] to run TL as the home hub (15 devices, 30 routines). Fig. 15d shows it takes only 1 ms to schedule a large routine with 10 commands. Surveys show typical routines today contain 5 commands or fewer [39, 63], hence our scheduler is fast in practice.



(a) End to End latency (b) Parallelism level (%)

(c) Temporary Incongruence & Order Mismatch (%) (d) Device Popularity vs. Latency

Figure 16: **Impact of Routine size ($\mathcal{C}$) and device popularity (α).** *PSV and GSV are always zero and omitted in (c).*



(a) (b)

Figure 17: **Impact of: (a) long routine duration ($|\mathcal{L}|$), and (b) percentage of long running routines ($\mathcal{L}_{\%}$).**

7.6 Parameterized Microbenchmark Experiments

Commands per routine ($\mathcal{C}$): Fig. 16a, 16b show GSV's latency rises as routines contain more commands. With smaller routines, PSV is close to EV and WV, but as routines contain more commands, PSV quickly approaches GSV. While EV has a similar trend, it stays faster than GSV and PSV. Parallelism level and temporary incongruence follow this trend. Finally, EV's peaking behavior and eventual convergence towards GSV (Fig. 16c) occur since beyond a certain routine size ($\mathcal{C}$=4), pre/post-leasing opportunities decrease.

Device popularity (α): Using a Zipf distribution for device access by routines, Fig. 16d shows that increasing α (popularity skew) causes EV's latency to stay close to WV. More conflict slows PSV quickly down to GSV.

Long running routines: As the long running routine length $|\mathcal{L}|$ rises (Fig. 17a), temporary incongruences decrease since the run is now longer, routines are spread temporally, and less likely to conflict. Increasing the number of long running routines ($\mathcal{L}_{\%}$) increases the chance of conflict, causing more temporary incongruence. (Fig. 17b). The *order mismatch*—how much the final serialization order differs from the submission order of routines, using swap distance: i) rises as routines get longer (Fig. 17a), ii) but falls with as more routines are longer (Fig. 17b), because post-leases dominate. Overall, order mismatch stays low, between 3%-10%.

8 Related Work

Support for Routines: Routines are supported by Alexa [6], Google Home [28], and others [4,54,57]. iRobot's Imprint [37, 76] supports long-running routines, coordinating between a vacuum [53] and a mop [16]. All these systems only support best-effort execution (akin to WV).

Consistency in Smart Homes: SafeHome can be used orthogonally with either: i) transactuations [55], which provides a consistent soft-state, or ii) APEX [80], which ensures safety by automatically discovering and executing prerequisite commands. These two systems maintain strict isolation by sequentially executing conflicting routines, making them both somewhat akin to PSV.

Abstractions: IFTTT [36] represents the home as a set of simple conditional statements, while HomeOS [24] provides a PC-like abstraction for the home where devices are analogous to peripherals in a PC. Beam [56] optimizes resource utilization by partitioning applications across devices. These and other abstractions for smart homes [12, 46, 48, 77, 79] do not address failures or concurrency.

Concurrency Control: Concurrency control is well-studied in databases [14]. Smart Home OSs like HomeOS, SIFT, and others [24, 43, 47, 52] explore different concurrency control schemes. However, none of these explore visibility models. Classical task graph scheduling algorithms [5,10,13,19,34,35, 42] do not tackle SafeHome's specific scheduling problem.

ACID Properties applied in Other Domains: There is a rich history of leveraging transaction-like ACID properties in many domains. Examples include work in software-defined networks to guarantee update consistency [21, 22] and for robustness [18]. ACID has also been applied in transactional memory [11, 32, 33, 49] and pervasive computing [65].

9 Conclusion

SafeHome is: i) the first implementation of relaxed visibility models for smart homes running concurrent routines, and ii) the first system that reasons about failures alongside concurrent routines. We find that:

(1) Eventual Visibility (EV) provides the best of both worlds, with: a) user-facing responsiveness (latency) only $0\% - 23.1\%$ worse than today's Weak Visibility (WV), and b) end state congruence identical to the strongest model Global Strict Visibility (GSV).

(2) When routines abort due to failures, EV rolls back the fewest commands among all models.

(3) Lock leasing improves latency by $3\times - 5.5\times$.

(4) Compared to competing policies (FCFS and JiT), Timeline Scheduling improves latency by $1.33\times - 2.36\times$ and parallelism by $2.0\times - 2.3\times$.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] Shegufta Bakht Ahsan, Rui Yang, Shadi Abdollahian Noghabi, and Indranil Gupta. Home, SafeHome: Ensuring a safe and reliable home using the edge. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, Renton, WA, July 2019. USENIX Association.

[3] Amazon Alexa outage. https://downdetector.com/status/amazon-alexa/news/235561-problems-at-alexa. Last accessed April 2020.

[4] Alexa routines show promise and limitations. https://www.timeatlas.com/create-alexa-routines/. Last accessed April 2020.

[5] Marcos Amaris, Giorgio Lucarelli, Clément Mommessin, and Denis Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro, editors, *Euro-Par 2017: Parallel Processing*, pages 220–231, Cham, 2017. Springer International Publishing.

[6] Amazon Alexa. https://developer.amazon.com/alexa. Last accessed April 2020.

[7] Amazon Alexa + SmartThings routines and scenes. https://support.smartthings.com/hc/en-us/articles/210204906-Amazon-Alexa-SmartThings-Routines-and-Scenes. Last accessed April 2020.

[8] 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 *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 41–54, New York, NY, USA, 2017. ACM.

[9] I. Armac, M. Kirchhof, and L. Manolescu. Modeling and analysis of functionality in eHome systems: dynamic rule-based conflict detection. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*, pages 10 pp.–228, March 2006.

[10] A.R. Arunarani, D. Manjula, and Vijayan Sugumaran. Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems*, 91, 09 2018.

[11] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. A programming language perspective on transactional memory consistency. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 309–318, 2013.

[12] Automate.io. https://automate.io/. Last accessed April 2020.

[13] Denis Barthou and Emmanuel Jeannot. Spaghetti: Scheduling/placement approach for task-graphs on heterogeneous architecture. In Fernando Silva, Inês Dutra, and Vítor Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, pages 174–185, Cham, 2014. Springer International Publishing.

[14] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[15] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, SOSP '87, pages 123–138, New York, NY, USA, 1987. ACM.

[16] Braava. https://www.irobot.com/braava. Last accessed April 2020.

[17] M. Campbell-Kelly. Historical reflections: The rise, fall, and resurrection of software as a service. *Commun. ACM*, 52(5):28–30, May 2009.

[18] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A distributed and robust sdn control plane for transactional network updates. In *2015 IEEE conference on computer communications (INFOCOM)*, pages 190–198. IEEE, 2015.

[19] L. Canon, L. Marchal, B. Simon, and F. Vivien. Online scheduling of task graphs on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 31(3):721–732, 2020.

[20] Comcast outage. https://webdownstatus.com/outages/comcast. Last accessed April 2020.

[21] Maja Curic, Georg Carle, Zoran Despotovic, Ramin Khalili, and Artur Hecker. Sdn on acids. In *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*, pages 19–24, 2017.

[22] Maja Curic, Zoran Despotovic, Artur Hecker, and Georg Carle. Transactional network updates in sdn. In *2018 European Conference on Networks and Communications (EuCNC)*, pages 203–208. IEEE, 2018.

[23] Scott Davidoff, Min Kyung Lee, Charles Yiu, John Zimmerman, and Anind K. Dey. Principles of smart home control. In *Proceedings of the 8th International Conference on Ubiquitous Computing*, UbiComp'06, page 19–34, Berlin, Heidelberg, 2006. Springer-Verlag.

[24] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 25–25, Berkeley, CA, USA, 2012. USENIX Association.

[25] EE: Average UK Smart Home will have 50 connected devices by 2023. https://www.totaltele.com/500103/EE-Average-UK-Smart-Home-will-have-50-connected-devices-by-2023. Last accessed April 2020.

[26] M. Ellis, 5 times smart home technology went wrong. https://www.makeuseof.com/tag/smart-home-technology-went-wrong/. Last accessed November 2019.

[27] Fenestra: Make your windows smart. http://www.smartfenestra.com/home. Last accessed April 2020.

[28] Google Home. https://store.google.com/us/product/google_home. Last accessed April 2020.

[29] Google Home outage. https://downdetector.com/status/google-home. Last accessed April 2020.

[30] SmartThings outage. https://downdetector.com/status/smartthings/news/224625-problems-at-smartthings. Last accessed April 2020.

[31] Google's smart home ecosystem is a complete mess. https://www.cnet.com/news/googles-smart-home-ecosystem-is-a-complete-mess/. Last accessed April 2020.

[32] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, 2008.

[33] Rachid Guerraoui and Michał Kapałka. Principles of transactional memory. *Synthesis Lectures on Distributed Computing*, 1(1):1–193, 2010.

[34] C. Hanen and A. Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. In *Proceedings 1995 INRIA/IEEE Symposium on Emerging Technologies and Factory Automation. ETFA'95*, volume 1, pages 167–189 vol.1, 1995.

[35] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, April 1989.

[36] IFTTT. https://ifttt.com/. Last accessed April 2020.

[37] Imprint™ link technology: Getting started. https://homesupport.irobot.com/app/answers/detail/a_id/21090/ /imprint%E2%84%A2-link-technology%3A-getting-started. Last accessed April 2020.

[38] Internet of Shit. https://twitter.com/internetofshit. Last accessed April 2020.

[39] IoTBench test-suite. https://github.com/IoTBench/IoTBench-test-suite/tree/master/openHAB. Last accessed April 2020.

[40] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[41] John Koon. Smart sensor applications in manufacturing (Enterprise IoT Insights). https://enterpriseiotinsights.com/20180827/channels/fundamentals/iotsensors-smart-sensor-applications-manufacturing. Last accessed April 2020.

[42] Gunho Lee. *Resource Allocation and Scheduling in Heterogeneous Cloud Environments*. PhD thesis, University of California at Berkeley, USA, 2012.

[43] C. M. Liang, B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu. SIFT: Building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, IPSN '15, pages 298–309, New York, NY, USA, 2015. ACM.

[44] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005.

[45] Mapping the Smart-Home Market. https://www.bcg.com/publications/2018/mapping-smart-home-market.aspx. Last accessed April 2020.

[46] Microsoft Flow. https://flow.microsoft.com. Last accessed April 2020.

[47] S. Munir and J. A. Stankovic. Depsys: Dependency aware integration of cyber-physical systems for Smart Homes. In *ICCPS '14: ACM/IEEE 5th International Conference on Cyber-Physical Systems (with CPS Week 2014)*, ICCPS '14, pages 127–138, Washington, DC, USA, 2014. IEEE Computer Society.

[48] OpenHAB. https://www.openhab.org/. Last accessed April 2020.

[49] Hany E Ramadan, Christopher J Rossbach, Donald E Porter, Owen S Hofmann, Aditya Bhandari, and Emmett Witchel. Metatm/txlinux: transactional memory for an operating system. *ACM SIGARCH Computer Architecture News*, 35(2):92–103, 2007.

[50] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[51] Raspberry Pi 3 Model B+. https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/. Last accessed April 2020.

[52] D. Retkowitz and S. Kulle. Dependency management in smart homes. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 143–156. Springer, 2009.

[53] Roomba. https://www.irobot.com/roomba. Last accessed April 2020.

[54] Routines not working. https://support.google.com/assistant/thread/3444653?hl=en. Last accessed April 2020.

[55] Aritra S., Tanakorn L., Masoud S. A., and Cesar A. S. Transactuations: Where transactions meet the physical world. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 91–106, Renton, WA, July 2019. USENIX Association.

[56] Chenguang S., Rayman P. S., Amar P., Aman K., and Ratul M. Beam: Ending monolithic applications for connected devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 143–157, Denver, CO, 2016. USENIX Association.

[57] Scheduled routines not reliable. https://support.google.com/assistant/thread/366154?hl=en. Last accessed April 2020.

[58] Fred Schneider. implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[59] Smartcan: Take control of your chores. https://www.rezzicompany.com/. Last accessed April 2020.

[60] Smart home. https://www.statista.com/outlook/279/109/smart-home/united-states. Last accessed April 2020.

[61] Smart home market worth $151.4 billion by 2024. https://www.marketsandmarkets.com/PressReleases/global-smart-homes-market.asp. Last accessed April 2020.

[62] Future of smart hospitals (ASME). https://aabme.asme.org/posts/future-of-smart-hospitals. Last accessed April 2020.

[63] SmartThings Smart Apps. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps. Last accessed April 2020.

[64] Stop shouting at your smart home so much and set up multi-step routines. https://www.popsci.com/smart-home-routines-apple-google-amazon. Last accessed April 2020.

[65] Oliver Storz, Adrian Friday, and Nigel Davies. Supporting content scheduling on situated public displays. *Computers & Graphics*, 30(5):681–691, 2006.

[66] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[67] The top 5 problems with smart home tech and how to troubleshoot them. https://www.nachi.org/problems-smart-home-tech.htm. Last accessed April 2020.

[68] TP Link device driver. https://github.com/intrbiz/hs110. Last accessed April 2020.

[69] TP Link HS105. https://www.tp-link.com/us/download/HS105.html. Last accessed April 2020.

[70] TP-link KASA android app. https://www.tp-link.com/us/kasa-smart/kasa.html. Last accessed April 2020.

[71] TP Link KASA HS100, HS220, KL130. https://www.kasasmart.com/. Last accessed April 2020.

[72] TP Link KASA HS105, HS110, HS200. https://www.kasasmart.com/. Last accessed April 2020.

[73] Velux: Smart home, smart skylights. https://whyskylights.com/. Last accessed April 2020.

[74] V. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009.

[75] Wemo. https://www.wemo.com/. Last accessed April 2020.

[76] What is imprint™ link technology? https://homesupport.irobot.com/app/answers/detail/a_id/21088/∼/what-is-imprint%E2%84%A2-link-technology%3F. Last accessed April 2020.

[77] Workflow. https://workflow.is/. Last accessed April 2020.

[78] Is Xfinity having an outage right now? https://outage.report/us/xfinity. Last accessed April 2020.

[79] Zapier. https://zapier.com/. Last accessed April 2020.

[80] Q. Zhou and F. Ye. Apex: Automatic precondition execution with isolation and atomicity in internet-of-things. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, pages 25–36, New York, NY, USA, 2019. ACM.