# Home, SafeHome: Smart Home Reliability with Visibility and Atomicity

Shegufta B. Ahsan[*]
sbahsan2@illinois.edu
University of Illinois at Urbana-Champaign

Rui Yang
ry2@illinois.edu
University of Illinois at Urbana-Champaign

Shadi A. Noghabi
shadi@microsoft.com
Microsoft Research

Indranil Gupta
indy@illinois.edu
University of Illinois at Urbana-Champaign

## Abstract

Smart environments (homes, factories, hospitals, buildings) contain an increasing number of IoT devices, making them complex to manage. Today, in smart homes when 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. isolation of the smart home. 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.

***CCS Concepts:*** • **Computer systems organization → Dependable and fault-tolerant systems and networks**; **Reliability**.

***Keywords:*** Smart Home, Routines, Reliability, Fault-tolerance

**ACM Reference Format:**
Shegufta B. Ahsan, Rui Yang, Shadi A. Noghabi, and Indranil Gupta. 2021. **Home, SafeHome: Smart Home Reliability with Visibility and Atomicity. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456261**

## 1 Introduction

The disruptive smart home market is projected to grow from $27B to $150B by 2024 [50, 67]. There is a wide diversity of devices—roughly 1,500 IoT vendors today [4], with the average home expected to contain over 50 smart devices by 2023 [43]. 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. Such diversity and scale are even vaster in other smart environments such as smart buildings, smart factories (e.g., Industry 4.0 [45]), and smart hospitals [68].

As smart environments become increasingly more complex and larger in scale, the chances of interactions leading to undesirable outcomes increases. What is desperately needed are systems that allow a group of users to *manage their smart home as a single entity* [24] rather than a collection of individual devices. Today, most users (whether in a smart home or a smart factory) control a device using *commands*, e.g., turn ON a light. Recently, major smart home controllers have started to provide users the ability to create *routines*. A routine is a sequence of commands [7, 37, 64, 82]. Routines are useful for both: a) convenience, e.g., turn ON a group of Living Room lights, then switch ON TV, and b) correct operation, e.g., CLOSE window, then turn ON AC. Routines enable a high degree of automation, and can be triggered either by a human or automatically, e.g., time-based, sensor-based, etc.

Today's ad-hoc way of executing routines fails to satisfy two natural expectations of human users. First, when a user initiates a routine, the user expects it to execute in its entirety, e.g., if the window is not closed, the AC should not turn on. In other words, routines should execute *Atomically*–either all the commands in a routine have an effect on the environment, or none of its commands do. Secondly, the execution of one routine must not be impacted by another concurrent routine, e.g., while a routine is taking out the garbage can, another routine must not simultaneously attempt to close the garage door. In other words, routines should be executed in *Isolation, i.e., with Serializability*—the *effect* of a set of concurrent routines, should be equivalent to executing the routines one by one, in some sequential order.

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

[*] Also with Amazon Inc. (Work done while student at UIUC).

*EuroSys '21, April 26–28, 2021, Online, United Kingdom*

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456261

access point—from where it can control devices. SafeHome does not require additional logic on devices—instead, it issues commands via the device's APIs. Thus SafeHome can work in homes containing devices from many vendors.

Assuring the properties of Atomicity and Isolation (Serializability) in smart homes needs us to tackle three unique challenges. First, this is a human-facing 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 user anywhere in the smart home. This requires us to clearly specify and reason about what we call *visibility models* for concurrent routines. Visibility models capture the various flavours of serializability and latency among routines, observed by the user.

Second, many commands can take extended periods of time (seconds or minutes) to execute. This may arise because of either: i) the nature of the device, e.g., oven; or ii) a user-specified action, e.g., running the water sprinkler for 15 minutes. Such commands cannot be treated merely as two short commands (start and stop), as this still allows the command to be interrupted by a concurrent routine in the interim, violating isolation. We introduce the notion of a *long running command* (or long command), a command which exclusively controls a device for an extended duration of time without interruption. A long running routine (or long routine) is one that contains at least one long command. We advocate that long commands/routines need to be treated as first-class citizens in the design of a smart home system. Particularly, long routines increase the likelihood of concurrency conflicts, creating tension between fast routine execution (responsiveness) and the assurance of serializability.

Third, in a smart home, *device crashes and restarts are the norm*—any device can fail at any point of time and possibly recover later. Devices are rarely replicated or have a fall-back mechanism. Therefore, *reasoning about device failure/restart events* while ensuring atomicity+visibility models is a new challenge. Today's failure handling in smart homes is either silent or places the burden of resolution on the user.

These challenges have been addressed piecemeal in smart home/IoT literature. Some systems [25, 61] use priorities to address concurrent device access. Others [8] propose mechanisms to handle failures. A few systems [9, 48, 52] formally verify procedures. Transactuation [64] and APEX [86] discuss atomicity and isolation, yet their concrete techniques deal with routine dependencies and do not consider users' visibility experiences—nevertheless, these mechanisms can be used orthogonally with SafeHome. None of the above simultaneously address atomicity, failures, and visibility.

The reader may also notice parallels between our work and the ACID properties (Atomicity, Consistency, Isolation, and Durability) provided by transactional databases [58]. While parallels between sensor networks and databases exist (TinyDB [49]), database ACID techniques do not translate easily to smart homes. The primary reasons are the human-facing aspect of the environment, device failures (DB objects are easily replicated), and presence of long routines.

The primary contributions of this paper are:

1. New *Visibility Models* that trade off responsiveness against temporary congruence of smart home state.
2. Introduction of *long-running* routines and new *lock leasing* techniques to increase concurrency among routines, while guaranteeing isolation.
3. A new way to reason about failures by *serializing failure and restart events into* the serially-equivalent order.
4. Design and implementation of the SafeHome system, and a workload-driven experimental evaluation.

## 2 Motivating Examples

**Atomicity:** Today's *best-effort* way of executing routines can lead to unwanted states in the home, as documented in many smart home incidents [27, 41, 53, 56, 64]. Consider a routine $R_{cooling}$ involving the AC and a smart window [28, 79]: $R_{cooling}$ = {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 these examples, lack of *atomicity* (all or nothing execution) in the routine's execution violates the expected outcome.

**Isolation/Serializability:** Conflicts among routines are common. A conflict occurs when more than one routine simultaneously touches the same given device. Auto-triggered routines might conflict among each other. Human users may start routines that conflict with either: 1) other auto routines, or 2) other users' routines, if the humans do not coordinate a priori (e.g., verbally). Above all, the presence of long commands, which take non-negligible time to execute, amplify the chance of such conflicts.

Consider a timed routine $R_{trash}$ that executes every Monday 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 [66]); CLOSE garage}. One Monday, 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 executing either: a) its first command (open garage), or b) its second command (move trash can out). In both cases, $R_{trash}$'s execution is incorrect, and equipment may be damaged (garage or trash can). Alternatively, executing routines one-at-a-time would mean the user has to wait several minutes to see the lights are turned off, even though the lights are not the

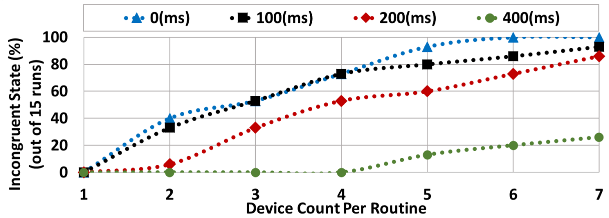


**Figure 1. Concurrency causes incongruent end-state in a real smart home deployment.** *Two routines R1 (turn* `ON` *all lights) and R2 (turn* `OFF` *all lights) executed on a varying number of devices (x axis), with routine R2 starting a little after R1 (different lines). Y axis shows fraction of end states that are not serialized (i.e., all* `OFF`*, or all* `ON`*). Experiments with TP-Link smart devices [78].*

conflicting device. We refer to such a non-serialized state (intermediate or end) of the home as an *incongruent state*. To provide isolation/serializability, when $R_{trash}$ and $R_{goodnight}$ complete successfully, doors should be locked, garage closed, lights turned off, trash can in the driveway, and no equipment damaged with minimal visible latency to the user.

Low latency is critical in user-facing scenarios like smart homes because: (i) studies routinely show that cutting even fractions of a second increases user engagement [18, 55, 60, 71], and (ii) for long routines, which run for many seconds or minutes, latency reductions of even a few percentage points are rather noticeable to the user.

Conflicts may occur even when routines contain no long commands. Fig. 1 shows that two routines simultaneously switching on/off a few devices cause incongruent outcomes when they start close to each other. In all the above cases, *isolation semantics* among concurrent routines were not being specified cleanly or enforced which can: (a) violate human expectations, and (b) cause device damage and failures [26].

# 3 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.

## 3.1 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.

Studies show [24, 46] that it is key in a smart home to optimize *user-facing responsiveness metrics*. Thus, we introduce a new *spectrum of visibility models* that trade off the amount of incongruence the user sees *during* execution, against the user-perceived latency—all while guaranteeing serial-equivalence of the overall execution.

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). Our weak(ened) visibility models can be considered a counterpart of the rich legacy of weak consistency models existing in mobile systems like Coda [44], databases like Bayou [72] and NoSQL [80], and shared memory multiprocessors [1].

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. 3), the *effect* for GSV is "at every point of time", i.e., every individual action on every device. Consider a home where two routines are started simultaneously: $R_{dishwash}$ =`{dishwasher:ON; /*run dishwasher for 40 mins*/ dishwasher:OFF;}`, and $R_{dryer}$=`{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 our previous (GSV) example of $R_{dishwash}$ and $R_{dryer}$ started simultaneously, if the home has no amperage restrictions, the users should choose PSV. This allows concurrent execution, and the end state is equivalent to serial execution, i.e., dishes are washed, clothes dried. However, if

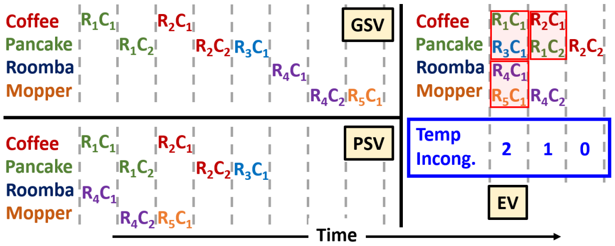


**Figure 2. Example Routine Execution in different Visibility Models (GSV, PSV, EV):** *$R_iC_j$ represents the $j^{th}$ command of the $i^{th}$ routine. In EV, red boxes show a pair of incongruent commands and the blue box shows the total number of temporary incongruences.*

the two routines happened to touch conflicting devices, PSV would have executed them serially.

3. **Eventual Visibility (EV):** This is our most relaxed visibility model that allows routines' commands to be interleaved and yet guarantees serial-equivalence. EV specifies that only *when all the routines have finished (completed or 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 identical instances of a routine ={`coffee:ON; /*make coffee for 4 mins*/; coffee:OFF; pancake:ON; /*make pancakes for 5 mins*/; pancake:OFF;`}. This scenario might occur if the two users do not coordinate a priori, or if one of the routines is auto-triggered. Both GSV and PSV execute these routines serially, 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 assures that at the end both users have their respective coffees and pancakes.

**Example for All Visibility Models:** Fig. 2 shows an example of 5 concurrent routines, $R_1 - R_5$, that we executed in a real SafeHome deployment (run on Raspberry Pi, with 5 devices as TP-Link HS-105 smart-plugs [75]).

$R_1$: *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 shows the longest execution time of 8 time units as it serializes execution. PSV reduces execution time to 5 time units by parallelizing unrelated commands, e.g., $R_1C_1$ and $R_4C_1$ at $t = 0$. 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 zero incongruence when the last routine finishes.

|  | **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 ones | Low for all routines (modulo conflicts) | Low for all routines |
| *User Visibility* | Congruent at all times | Congruent at end & at start/complete points of routines | Congruent at end | May be incongruent at anytime and/or end (Fig. 1) |

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

In summary, temporary incongruence refers to intermediate states that would never be reached via a purely serial execution. A majority of these are not "bad" states, with the user barely noticing a difference—for instance, in Fig. 2, the coffeemaker and pancake-maker running simultaneously (in EV, and PSV). Since different users' experiences may be subjectively different, hence our temporary incongruence metric is an objective "worst case" measure of the badness of non-serial intermediate states.

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

### 3.2 SafeHome-Atomicity

SafeHome-Atomicity states that for a routine, either: (a) all its commands have the desired effect on the home (i.e., the routine *completes*), or (b) the routine *aborts*, rolls back its commands, and gives feedback to the user. 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 ) and turn off lights (best-effort)—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 that sets Light-3 to `OFF`. Some long commands cannot be physically reversed, e.g., command `run north sprinklers for 15 mins`, or command `blare a test alarm`. For such commands, we undo by restoring the device to its state before the aborted routine (e.g., set the

| 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; /*run dishwasher for 40 mins*/; dishwasher:OFF;}<br>$R_2$={dryer:ON; /*run dryer 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;}<br>$R_2$={lights:ON, fan:ON} | Two routines touching disjoint devices should not block each other (PSV). | Partitioned Strict Visibility (PSV), closest to [64] | |
| "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 2nd 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.**

sprinkler/alarm state to OFF). Alternately, a user-specified undo-handler could be used.

A goal of SafeHome's atomicity and visibility is to ensure that aborts are caused *only* by failures, in all visibility models. In other words—without failures of commands or devices, routines always complete, even if they were to conflict. 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.

### 3.3 Safety Properties

Orthogonal to the atomicity and visibility properties is the notion of *Safety* [2]. While our current paper is not focused on safety, we briefly overview it for completeness.

SafeHome allows users to specify safety properties in the form of declarative predicates, which capture unanticipated states violating user intents. Once specified, SafeHome then assures that such unsafe states are never reached at any time, regardless of the visibility model. Safety properties are useful in avoiding scenarios which can create serializable, but unwanted, outcomes.

We envision that some safety predicates will come "baked" into the smart home, while others can be programmed and changed by the user. Safety properties could be specified in a myriad number of ways, e.g., [2, 48, 86]. Safety properties are useful in catching safety violations due to either: (a) badly programmed routines (GSV, PSV, EV), or (b) concurrent routines (PSV, EV). Examples of (a) and (b) follow. Example for (a): given the safety property "*if* (stove==ON) *then* (exhaust-fan==ON)", then a routine R1={stove:ON; exhaust-fan:ON} will be rejected by SafeHome, but the alternate routine R1′={exhaust-fan:ON; stove:ON} will be admitted. Example for (b): consider two routines : R1 ={... stove:ON; ... stove:OFF; ...} and R2 ={... exhaust-fan: OFF; ...}. If R2's exhaust-fan command were to execute in between R1's two stove commands, SafeHome catches a safety violation and prevents it from happening. Other interleavings that do not violate the safety property are allowed.

When SafeHome catches a potential safety violation, the offending command is not executed, and the routine containing it is aborted and rolled back. Safety properties are also checked any time the failure detector catches a device failure–in such cases, a report is bubbled up to the user as human intervention may be required, e.g., the CO detection device fails or fire alarm is low on battery.

In the rest of the paper we focus purely on atomicity and visibility/isolation, rather than the orthogonal safety requirements which literature has already explored [2, 64, 86]. We note that the presence of long-term safety properties does not obviate the need for (and challenges associated with) the properties of atomicity and visibility, which are relatively short-term and scoped to routines. Additionally, guaranteeing atomicity for a routine provides correct behavior when the routine's commands are not connected by safety conditions. Of course, while one could ostensibly carefully craft safety conditions scoped only to the run time of an individual routine, in order to achieve an atomicity-like behavior, this alternative would result in a large number of safety conditions, whose number would grow quickly with the number of routines. Keeping track of consistency across numerous safety conditions would be cumbersome for users. We envision a world where there are only a handful of long-term routine-independent safety properties in the home, and the rest of the correctness relies orthogonally on atomicity and visibility/isolation.

## 4 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. Our failure model is fail-stop/fail-recovery (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 $R$).

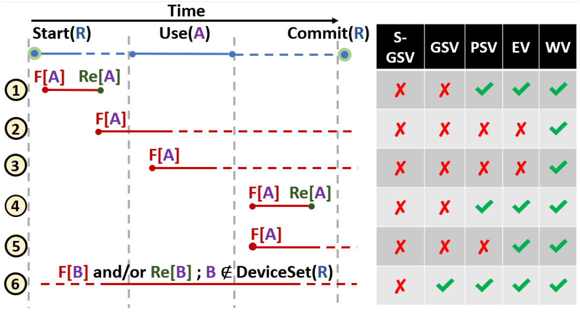
Thus a key realization in SafeHome is that we need to *serialize failure events and restart events alongside routines themselves.* We can now restate the SafeHome-Visibility property from Sec. 3, to account for failures and restarts:

**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 [1]

First, we define the failure/restart event to be the event when the edge device (running SafeHome) *detects* the failure/restart (this may differ from the actual time of failure or 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 successfully). We next reason explicitly about failure serialization for each of our visibility models from Sec. 3.1.

**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:** GSV presents the picture of a single serialized home to the user, thus, 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 sub-flavors: (A) *Loose GSV (GSV)*: A routine aborts only if it contains at least one command that touches a failed/restarted device; (B) *Strong GSV (S-GSV)*: A routine aborts even if it does not have a command that touches a failed/restarted device. A routine $R_{shade}$ on living room shades can complete, if master bathroom shades fail, in GSV but not in 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 an arbitrary order.



| | S-GSV | GSV | PSV | EV | WV |
|---|---|---|---|---|---|
| 1 | ✗ | ✗ | ✓ | ✓ | ✓ |
| 2 | ✗ | ✗ | ✗ | ✗ | ✓ |
| 3 | ✗ | ✗ | ✗ | ✗ | ✓ |
| 4 | ✗ | ✗ | ✓ | ✓ | ✓ |
| 5 | ✗ | ✗ | ✗ | ✓ | ✓ |
| 6 | ✗ | ✓ | ✓ | ✓ | ✓ |

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

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

**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 2, $R_{cooling}$={window:CLOSE; AC:ON;}. If the "window" device fails during $R_{cooling}$ execution, then 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 completes 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, making the smart home's end state 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.

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

[1] This idea has analogues to distributed systems abstractions such as view/virtual synchrony, wherein failures and multicasts are totally ordered [16]. We do not execute multicasts in the smart home.

# 5 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. 5.1). This information is stored in the *Locking Data-structure* (Sec. 5.2). The *lineage table* ensures invariants required to guarantee Eventual Visibility (Sec. 5.3).

## 5.1 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 any aborts arising from routines touching conflicting devices. Hence we eschew optimistic concurrency control approaches and use locking [2].

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.

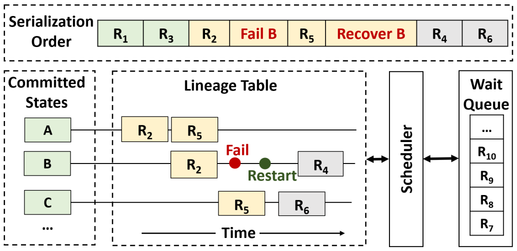


**Figure 4. SafeHome's Locking Table for Eventual Visibility.**

- **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, and thus prevents cascading aborts from scenarios like the following–$R_{src}$ switches on a light, and $R_{dst}$ has a conditional clause based on that light's status, but $R_{src}$ subsequently aborts. (If we were to allow cascading aborts, existing techniques [64] could be used orthogonally).

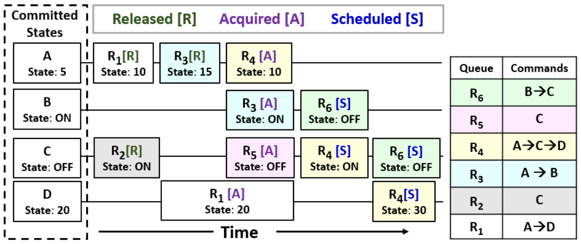
To prevent starvation, i.e., from $R_{src}$ waiting indefinitely for the returned lock, leased locks are revoked after a timeout. The timeout is calculated based on the estimated time between $R_{dst}$'s first and last actions on $D$ (a small multiplicative factor slightly $> 1$ could be used for leniency).

## 5.2 Locking Datastructure

SafeHome adopts a state machine approach [63] 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*, depicted in Fig. 4. This consists of:

- *Wait Queue:* Routines initiated but not started. Routines are assigned a new routine ID at add time.
- *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:* Maintains a per-device *lineage*: the *planned* transition order of that device's lock (Section 5.3).
- *Scheduler:* Decides when routines from Wait Queue are started, acquires locks, and maintains serialization order.

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



| Queue | Commands |
|---|---|
| $R_6$ | B→C |
| $R_5$ | C |
| $R_4$ | A→C→D |
| $R_3$ | A → B |
| $R_2$ | C |
| $R_1$ | A→D |

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

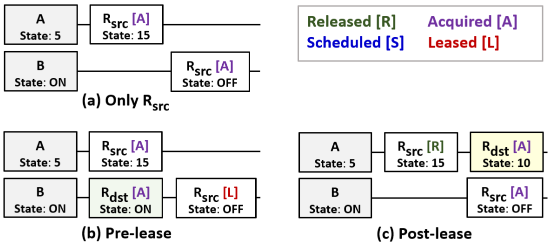


**Figure 6. Lineage Table with Lock Leasing**. *a) Lineage before leasing with only $R_{src}$, b) Pre-lease to $R_{dst}$ that only accesses device B, and c) Post-lease to $R_{dst}$ that only accesses device A.*

- *Committed States:* For each device, keeps its last committed state, i.e., the effect of the last successful routine. This may be different from device's actual state, and is needed to ensure serialization and rollbacks under aborts.

### 5.3 Lineage Table

The *lineage* of a device represents a temporal plan of when the device will be acquired by concerned routines. The lineage of a device starts with its latest committed state, followed by a sequence of *lock-access* entries (Fig. 5)–these are "stretched" horizontally. Width of a lock-access entry represents how long that routine will hold the lock. A 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 based on either known time to run a long command (e.g., run sprinkler for 15 mins), or an estimate of command execution time. Our implementation uses a fixed $\tau_{R_i}(D)$ for all short commands. $\tau_{R_i}(D)$ is also used to determine the revocation timeout for leased locks (a small multiplicative factor slightly $> 1$ could be used for leniency). At runtime, if a command's upper bound is violated, this is treated as a failure, and the routine is forced to abort and release its lock(s). We found the $95^{th}$ percentile latency for TP-Link Kasa switches [74] was 30 ms, thus $\tau_{R_i}(D)$ could be set as low as 100 ms.

To maintain serializability, we assure four key invariants:

**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$_{D'}(R_j)$. Thus $R_i$ 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.

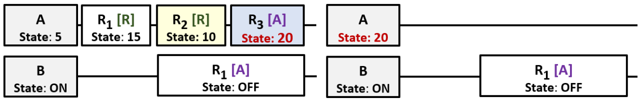
Second, when scheduling a new routine $R$ (from the wait queue), a `Scheduled` lock-access entry is added to all device lineages that $R$ needs (e.g., $R_6$ in Fig. 5 adds lock-accesses for B and C). Third, when a routine finishes (completes/aborts), all its lock-access entries are removed, releasing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

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

Fig. 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 $R_1$).



**(a)** Before $R_1$, $R_2$, $R_3$ commit **(b)** After $R_2$, $R_3$ commit (not $R_1$)

**Figure 7. Commit with Compaction.**

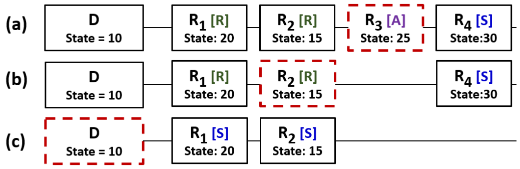


**Figure 8. Inferring the current device status**. *The dashed boxes point to the current device status in three different scenarios.*

- *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 conflicting devices). This is similar to "last writer wins" in NoSQL DBs [80]. 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 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)).

**Avoiding Deadlocks and Livelocks:** First, SafeHome avoids deadlocks because it maintains a (planned) linear serialization order among routines. To reiterate—when routine R *first* conflicts with routine R' on any given device, SafeHome decides the serialization order between R and R'. Any subsequent R-R' conflicts at any device D' will obey this serialization order.

Second, SafeHome avoids livelocks via the lineage table and serialization order, which together "plan" when routines will lock devices and execute commands. This prevents starvation and livelocks.

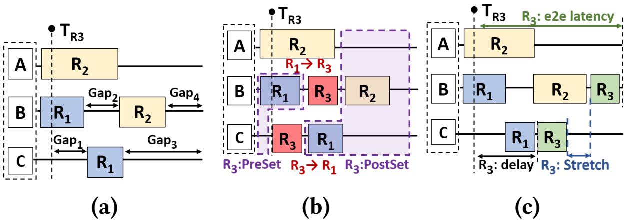


**Figure 9. Timeline Scheduler (TL) Example:** *a) Before scheduling $R_3$ b) Trying a potential (but invalid) schedule, c) Scheduling $R_3$ at the first possible gap.*

## 6 Eventual Visibility: Scheduling Policies

When a new routine arrives, SafeHome needs to "place" it in the serialization order, adhering to invariants of Sec. 5.3. This is a scheduling problem. We present three solutions.

**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 (unlike JiT) 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 (Sec. 5.3). An example is shown in Fig. 9a, 9b. Fig. 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

**Algorithm 1** Timeline Scheduling of Routine $R$

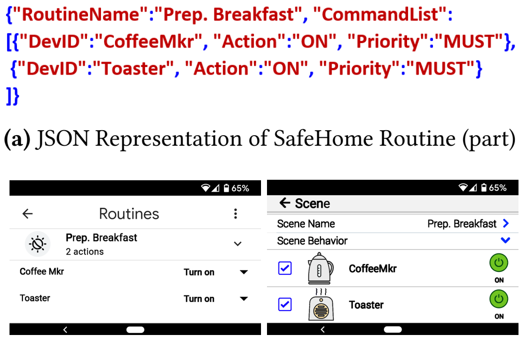
```
1:  function SCHEDULE($R$, index, startTime, preSet, postSet)
2:      devID = $R[index].devID$
3:      duration = $lock\_access(R, devID).duration$
4:      //return from recursion
5:      if $R.cmdCount$ < index then
6:          return true
7:      end if
8:      //Find gap and pre- and post-set
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
```

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 the finalized serialization. For the scheduled lock-accesses of $R_3$, it builds two sets: a) *preSet:* union of all (executing and scheduled) routines placed *before* $R_3$'s lock-accesses ($\{R_1\}$ in Fig. 9b), and b) *postSet:* union of all (executing and scheduled) routines placed *after* $R_3$'s lock-accesses ($\{R_1, R_2\}$ in Fig. 9b). preSet and postSet of $R$ are the routines positioned before and after $R$, respectively, in the serialization order. The gap choice is valid *if and only if* the intersection of preSet and postSet is empty. If true the scheduler moves to the routine's next command. Otherwise (Fig. 9b), the scheduler backtracks and tries the next gap (Line 21). This process repeats.

## 7 SafeHome Implementation

We implemented SafeHome in 2000 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 [21, 84] or cloud outages [3, 31, 32].

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 [77, 78], using the HS110Git [73] device-driver. Other devices (e.g., Wemo [81]) can be supported via their device-drivers.



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

**(a)** JSON Representation of SafeHome Routine (part)

Routines

Prep. Breakfast
2 actions

Coffee Mkr Turn on

Toaster Turn on

Scene

Scene Name Prep. Breakfast

Scene Behavior

☑ CoffeeMkr ON

☑ Toaster ON

**(b)** G. Home Routine [30] **(c)** TP-Link Routine [76]

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

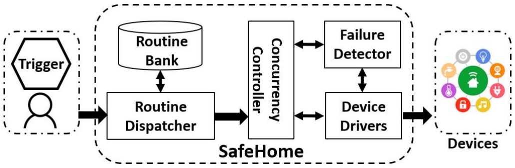


**Figure 11. SafeHome Architecture.**

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. 6), 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.

The *Failure Detector* explicitly checks devices by periodically (1 sec) sending ping messages. If a device does not respond within a short timeout, 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.

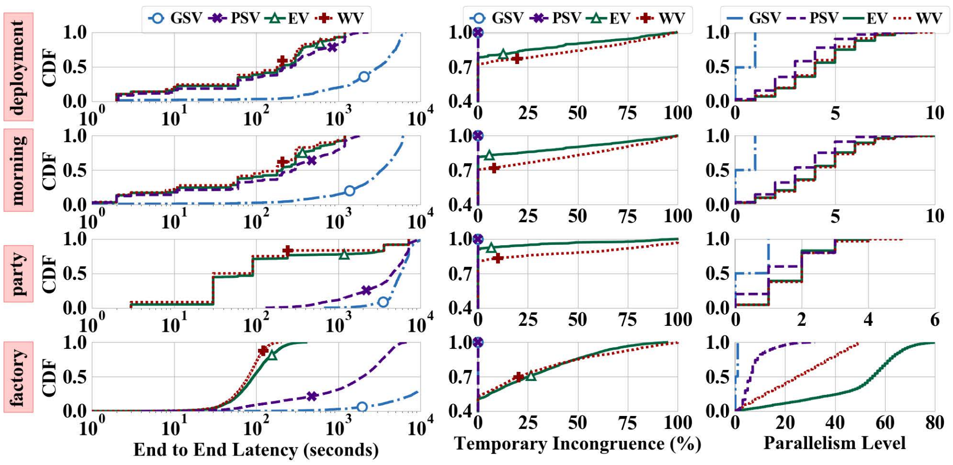
## 8 Experimental Results

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

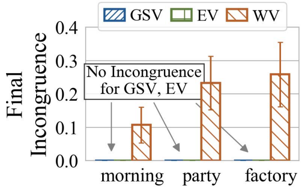
1. Are relaxed visibility models (Eventual Visibility) as responsive as Weak Visibility, and as correct as Global Strict Visibility (Sec. 3.1)?
2. What effect do failures have on correctness and user experience (Sec. 4)?
3. Which scheduler policy (Sec. 6) is the best?
4. What is the effect of lock leasing (Sec. 5)?

### 8.1 Experimental Setup

We wish to evaluate SafeHome for a variety of scenarios and parameters. Hence we run our implementation over an



**(a) Real-World Workloads:** *Latency, Temporary Incongruence, and Parallelism for One Deployment Scenario (Morning) and Three Simulated Scenarios (Morning, Party, Factory).*



**(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**.

emulation, using both real-world workloads (Sec. 8.2) and synthetic workloads (Sec. 8.3 - 8.5). We also validate the emulation with a real deployment.

**Metrics:** Our key metrics are:

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

*Temporary Incongruence:* We measure how much the human's experience differs from a world where all routines ran serially. We take worst case behavior. Before a routine $R$ completes, if *any* other routine $R'$ changes the state of *any* device $R$ modified, we say $R$ has suffered a temporary incongruence event. The plotted *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 auxiliary metric measures utilization. It is the number of routines that SafeHome allows concurrent execution of, averaged during periods of the run when more than 1 routine is active.

## 8.2 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 years. We also studied two public datasets: 1) 147 SmartThings applications [69]; and 2) IoTBench: 35 OpenHAB applications [42]. Based on these, we created three representative benchmarks (available as part of our software release):

**Morning Scenario:** This chaotic scenario has 4 family members in a 3-bed 2-bath home initiating 29 concurrent routines over 25 minutes touching 31 devices. Each user starts with a wake-up routine and ends with a leaving home routine. Intermediate routines cover bedroom & bathroom use, breakfast cook + eat, and sporadic actions, 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.

**Validating Simulation with Deployment:** Fig. 12a's top two rows show the morning scenario injected respectively into: a deployment with SafeHome controlling 31 TP-Link HS103 devices (top row), and simulation with identical setup and workload (second row). For all metrics–latency, temporary incongruence and parallelism level–we find that deployment and simulation are nearly identical. Across all visibility models, the area difference between the simulation and deployment CDF curves is at most: 4.8% for latency, 2.2% for incongruence, and 3.0% for parallelism level. Because simulation results so closely match deployment, we henceforth rely on simulation to perform flexible and long-duration experiments.

**Results:** From Fig. 12a (morning row), in the morning scenario: 1) EV's latency is comparable to WV at both median and 95th percentile, and 2) PSV has 15% worse 90th percentile latency than EV. Generally, the higher the parallelism level

| Name | default | Description |
|---|---|---|
| $\mathcal{R}$ | 100 | Total number of routines |
| $\rho$ | 100 | Number of concurrent routines at a time |
| $C$ | $[1, 4]$, or 4 (ND) | Commands per routine |
| $\alpha$ | 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 |
| $\mathcal{N}$ | 30 | Number of devices |
| $\mathcal{F}$ | 40% | Percentage of failed devices |

**Table 3. Parameterized Microbenchmark: Summary of Parameters (Section 8.3).** *(ND) = Normal distribution.*

(last column), the lower the latency. EV's median parallelism level is 3× higher than GSV, and median latency 16× better than GSV. Parallelism creates more temporary incongruences (middle column of figure), as expected for EV. Yet, EV's (and GSV's) end state is serially equivalent while WV may end incongruently–see 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.

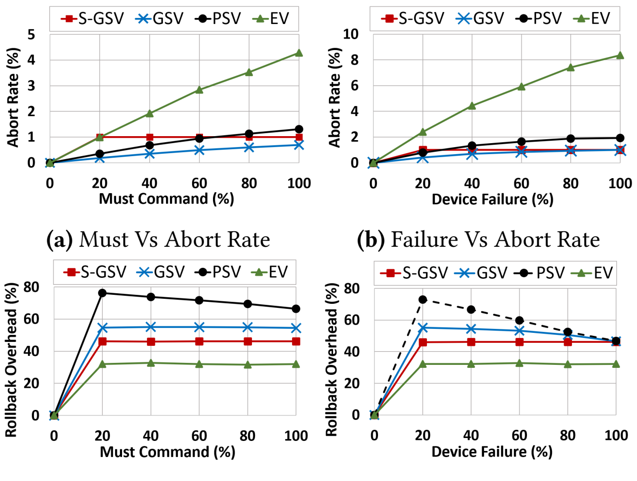
The party scenario (Fig. 12a (party row)) trends are similar to the morning scenario, with one exception. PSV's benefit is lower, with 11% 90th percentile latency reduction from GSV (vs. 77% in morning). This is because the single long routine blocks other routines. EV avoids such head-of-line blocking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

In Fig. 12a (factory 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.

## 8.3 Atomicity Evaluation: Effect of Failures

Next, we run parameterized workload-driven experiments (Table 3). Default values include 100 routines, and 30 devices. We pick the number of commands in a routine uniformly in the range $[1, 4]$. 10% of routines are long. Each datapoint is the average of 500K trials. Our open-source repository provides the exact configurations for all experiments (see end of this paper for a link).

Fig. 13a and 13b measure the fraction of routines aborted due to a failure. We induce fail-stop failures, where 40% of the total devices were marked as failed, each at a random point during the run. We observe that GSV has the lowest abort rate because it runs only one routine at a time. S-GSV's abort rate is higher because unlike GSV, even a failure of a device not touched by the current routine will cause an abort—this is also why S-GSV plateaus at 1%, wherein *any* failure aborts the currently running routine. Finally, PSV and EV have higher abort rates because they parallelly execute more routines, and thus a failure may affect more routines compared to GSV or S-GSV.



**(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.*

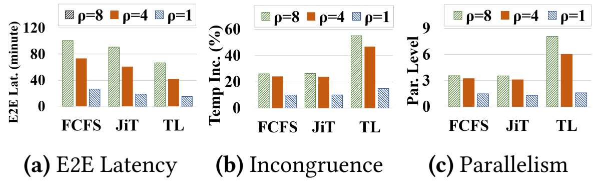
Yet Fig. 13c and 13d show that the *rollback overhead* of EV is smallest among all visibility models. We define rollback overhead as the average fraction of commands rolled back, averaged across aborted routines. This metric thus captures the effect of failures on intrusiveness experienced by the user, per aborted routine. To measure rollback overhead correctly, we fixed the number of commands/routine $C = 4$ (this gives us a fixed denominator). We note here that experiments with different values of $C$ gave the same trends.

From Fig. 13c and 13d, we observe that PSV's rollback overhead is higher than EV because PSV aborts more at the routine's finish point (when checking up/down status of devices touched). EV aborts affected routines earlier (closer to the failure occurrence) rather than at the routine's endpoint, thus avoiding wasted work. GSV and S-GSV had low abort rates because of their serial execution, but they both have higher rollback overheads than EV because they roll back more commands in the affected routines. The plateauing in Figs. 13c, 13d is due to saturation at abort points: for GSV at 55%, with S-GSV lower at 46% since any device failure triggers an abort of the current routine.

Overall, even *when execution is serial, the effect of failures, per aborted routine, can be more intrusive on the human (GSV, S-GSV, PSV).* We conclude that EV is the least intrusive model, because it rolls back the fewest commands per aborted routine.

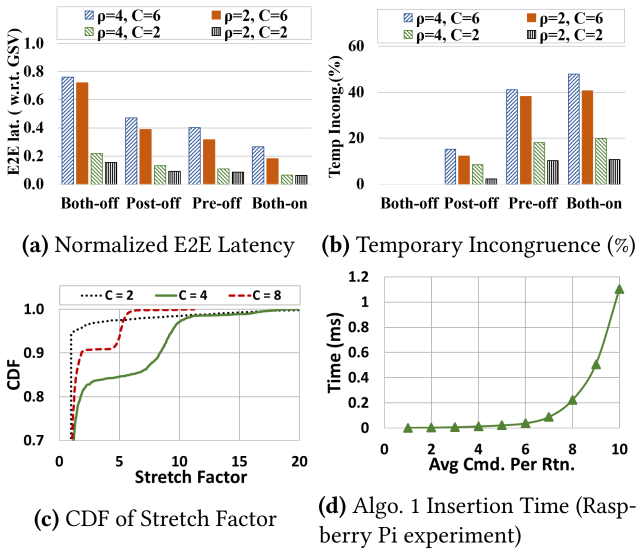
## 8.4 Scheduling Policies

Fig. 14 compares FCFS, JiT, and Timeline (TL) scheduling policies (Sec. 6). In Fig. 14a with $\rho = 4$ concurrent routines, for instance, TL is 1.8× and 1.4× faster than FCFS and JiT respectively. TL's benefit over: 1) FCFS is due to pre-leasing, 2) over JiT is due to opportunistic use of leasing. TL has more temporary incongruence but allows more parallelism (Fig. 14c) than FCFS (1.86× at $\rho = 4$) and JiT (1.94× $\rho = 4$).



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

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



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

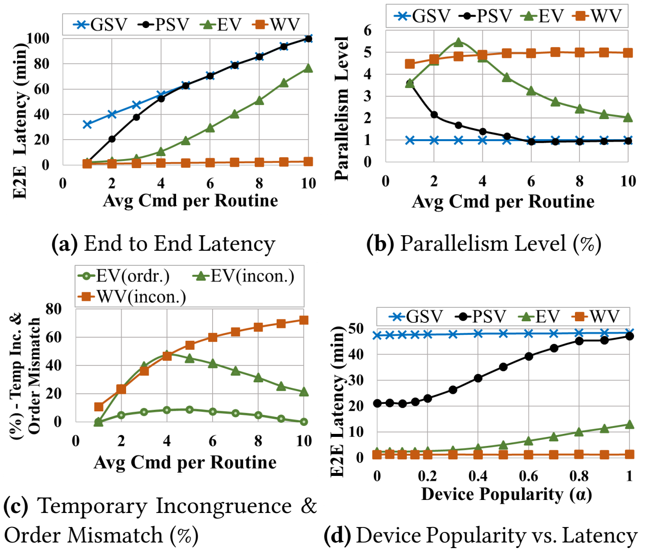
**(c)** CDF of Stretch Factor **(d)** Algo. 1 Insertion Time (Raspberry Pi experiment)

**Figure 15. TL Scheduler under EV.**

**TL Evaluation:** Fig. 15a and 15b show that disabling leasing reduces temporary incongruence but increases latency. Turning off *both* pre and post leasing increases latency (from Both-on to Both-off) by 1.5× - 4×. Post-leases are more effective than pre-leases: disabling the former raises latency by between 34% to 111%, while disabling the latter raises latency from between 5% to 74%. Post-leasing opportunities are more frequent than pre-leasing ones because only the former does not require changing the serialization order.

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

We used a Raspberry Pi 3 B+ [59] 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 [42, 69], 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 ($C$) and Device Popularity ($\alpha$).** *In (c), PSV and GSV are always 0 and omitted.*

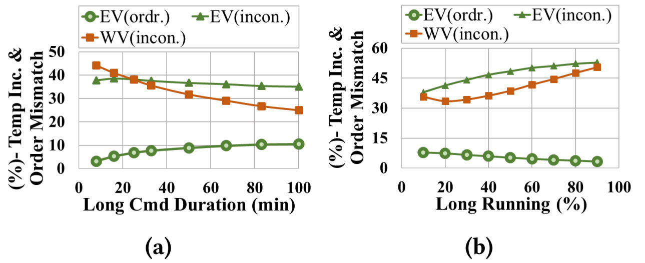


**Figure 17. Impact of: (a) Long Routine duration ($|\mathcal{L}|$), and (b) Percentage of Long Routines ($\mathcal{L}_{\%}$).** *Order Mismatch (ordr.) and Temporary Incongruences (incon.).*

### 8.5 Parameterized Microbenchmark Experiments

**Commands per routine ($C$):** 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. EV stays faster than GSV and PSV. Parallelism level and temporary incongruence follow this trend. EV's peaking and convergence towards GSV (Fig. 16c) occur since beyond a certain routine size ($C$=4), pre/post-leasing opportunities decrease.
**Device popularity ($\alpha$):** Using a Zipf distribution for device access by routines, Fig. 16d shows that increasing $\alpha$ (popularity skew) causes EV's latency to stay close to WV. More conflict slows PSV quickly down to GSV.
**Long running routines:** As routines become longer (Fig. 17a) temporary incongruences fall since the run is now longer, routines are spread temporally, and thus have fewer conflicts. Raising the number of long routines ($\mathcal{L}_{\%}$) increases conflicts and temporary incongruence. (Fig. 17b). We measure *order mismatch*—how much final serialization order differs from routines' submission order, by using swap distance. This

metric: i) rises as routines get longer (Fig. 17a), ii) but falls as more routines are longer (Fig. 17b), because post-leases dominate. Overall, order mismatch stays low, between 3%-10%.

## 9 Discussion

**Programming Long Routines:** While lock leasing allows natural concurrency of long routines, additional user intents need either: (i) careful design of routines, or (ii) safety properties (Section 3.3). An example of (i) is the user's intention to design a routine that locks doors at 10 pm and unlocks them at 8 am. If however the family expects a member to return late from work or party, then this single routine should be programmed as two separate routines—one routine at 10 pm and another routine at 8 am. This way locks are not "held" through the night, and a prodigal partying child can return home at midnight and unlock the doors.

**Safety Properties with Running Routines:** While we focused on atomicity and visibility, incorporation of safety raises a few open directions. One concern is the effect of aborts on safety. For instance, consider the safety predicate "*if* `stove==ON` *then* `exhaust-fan==ON`", and let $R$ be the current (uncommitted) routine which switched on the exhaust-fan. If another device touched by $R$ fails, then $R$ needs to abort, thus $R$'s exhaust-fan command needs to be undone and exhaust-fan switched off. Let $R'$ be the routine which (last) switched on the stove. Now, to satisfy the safety property when $R$ aborts, we have two design options—either: (a) merely switch off the stove, or (b) abort $R'$. Of course either $R'$ may have already committed or $R' = R$; in these common cases, only action (a) is feasible and straightforward. If $R'(\neq R)$ is still uncommitted though, the choice between action options (a) and (b) is a tradeoff of intrusiveness vs. user expectations of correctness. Particularly, intrusiveness arises because option (b) might cause cascading aborts. For simplicity, SafeHome's current design prefers option (a), and this avoids cascading aborts from arising due to safety violations.

**Future directions:** SafeHome opens up several exciting new directions that can now be addressed on top of the base system. SafeHome is the first step towards a smart home wherein "users control their lives, rather than control individual devices (the status quo)" [24, 46]. This means further ability for users to interrupt, override, and cancel running routines. Such "sub-atomicity" properties need careful reasoning and design. Users may also desire to have notions of priorities and deadlines for routines. If users require multiple visibility models to coexist simultaneously within the same home, careful reasoning is required in the system design about interactions among visibility models.

## 10 Related Work

**Support for Routines:** Routines are supported by Alexa [6], Google Home [30], and others [13, 29, 36]. iRobot's Imprint [40, 82] supports long-running routines, coordinating between a vacuum [62] and a mop [17]. All these systems only support best-effort execution (akin to WV).

**Consistency in Smart Homes:** SafeHome can be used orthogonally with either: i) transactuations [64], which provides a consistent soft-state, or ii) APEX [86], which ensures safety by automatically discovering and executing prerequisite commands. These two maintain strict isolation by sequential routine execution, i.e., they are akin to our PSV.

**Abstractions:** IFTTT [39] represents the home as a set of simple conditional statements, while HomeOS [25] provides a PC-like abstraction for the home where devices are analogous to peripherals in a PC. Beam [65] optimizes resource utilization by partitioning applications across devices. These and other abstractions for smart homes [12, 51, 54, 83, 85] do not address failures or concurrency.

**Concurrency Control:** Concurrency control is well-studied in databases [15]. Smart Home OSs like HomeOS, SIFT, and others [25, 48, 52, 61] explore different concurrency control schemes. However, none of these explore visibility models. Classical task graph scheduling algorithms [5, 10, 14, 20, 35, 38, 47] 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 [22, 23] and for robustness [19]. ACID has also been applied in transactional memory [11, 33, 34, 57], sensor networks [49], and pervasive computing [70].

## 11 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 conclude 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 equivalent 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) Timeline Scheduling is preferable over FCFS and JiT.

(4) Lock leasing improves latency by $1.5\times - 4\times$.

**Source Code:** *SafeHome* source code is available at: http://dprg.cs.uiuc.edu/downloads.php

## Acknowledgments

We thank our shepherd Pascal Felber and the anonymous Eurosys reviewers for their thoughtful comments and suggestions to improve the framing of our work. We also thank the excellent feedback from our MobiSys and NSDI reviewers.

This work was supported in part by research grant NSF CNS 1908888, by research grant NSF IIS 1909577, by a gift from Microsoft, and by a gift from Capital One.

## References

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

[2] S. B. Ahsan, R. Yang, S. A. Noghabi, and I. Gupta. Home, SafeHome: Ensuring a safe and reliable home using the edge. In *2nd USENIX Workshop on Hot Topics in Edge Computing*, HotEdge'19. USENIX Association, 2019.

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

[4] S. Ali and Z. Yusuf. Mapping the Smart-Home Market. https://www.bcg.com/publications/2018/mapping-smart-home-market.aspx. Last accessed October 2020.

[5] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In *European Conference on Parallel Processing*, Euro-Par'17, pages 220–231. Springer, 2017.

[6] Amazon Alexa. https://developer.amazon.com/alexa. Last accessed October 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 October 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, 2017.

[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. IEEE, 2006.

[10] A.R. Arunarani, D. Manjula, and Vijayan Sugumaran. Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems*, 91:407–415, 2019.

[11] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC'13, pages 309–318. ACM, 2013.

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

[13] T. B. Routines not working. https://support.google.com/assistant/thread/3444653?hl=en. Last accessed October 2020.

[14] D. Barthou and E. Jeannot. SPAGHETtI: Scheduling/Placement Approach for task-graphs on HETerogeneous architecture. In *European Conference on Parallel Processing*, Euro-Par'14, pages 174–185. Springer, 2014.

[15] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-Wesley Longman Publishing Co., Inc., 1987.

[16] 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. ACM, 1987.

[17] Braava. https://www.irobot.com/braava. Last accessed October 2020.

[18] J. Brutlag. Speed matters. https://ai.googleblog.com/2009/06/speed-matters.html, June 2009. Last accessed Jan 2021.

[19] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust sdn control plane for transactional network updates. In *2015 IEEE Conference on Computer Communications*, INFOCOM'15, pages 190–198. IEEE, 2015.

[20] 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.

[21] Comcast outage. https://webdownstatus.com/outages/comcast. Last accessed October 2020.

[22] M. Curic, G. Carle, Z. Despotovic, R. Khalili, and A. Hecker. SDN on ACIDs. In *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*, CAN'17, pages 19–24, 2017.

[23] M. Curic, Z. Despotovic, A. Hecker, and G. Carle. Transactional network updates in sdn. In *2018 European Conference on Networks and Communications*, EuCNC'18, pages 203–208. IEEE, 2018.

[24] S. Davidoff, M. K. Lee, C. Yiu, J. Zimmerman, and A. K. Dey. Principles of smart home control. In *Proceedings of the 8th International Conference on Ubiquitous Computing*, UbiComp'06, pages 19–34. Springer, 2006.

[25] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'12, pages 337–352. USENIX Association, 2012.

[26] ECO plugs. http://www.eco-plugs.net/. Last accessed October 2020.

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

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

[29] L. D. Freitas. Scheduled routines not reliable. https://support.google.com/assistant/thread/366154?hl=en. Last accessed October 2020.

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

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

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

[33] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'08, pages 175–184, 2008.

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

[35] 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. IEEE, 1995.

[36] A. Hennegar. Alexa routines show promise and limitations. https://www.timeatlas.com/create-alexa-routines/. Last accessed October 2020.

[37] S. Horaczek. 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 October 2020.

[38] J. Hwang, Y. Chow, F. D. Anger, and C. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[39] IFTTT. https://ifttt.com/. Last accessed October 2020.

[40] Imprint$^{TM}$ 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 October 2020.

[41] Internet of Shit. https://twitter.com/internetofshit. Last accessed October 2020.

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

[43] C. Kelly. 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 October 2020.

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

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

[46] V. Koshy, J. S. Park, T. Cheng, and K. Karahalios. "We just use what they give us": Understanding passenger user perspectives in smart homes. In *ACM CHI Virtual Conference on Human Factors in Computing Systems*, CHI'21. ACM, 2021.

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

[48] 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. ACM, 2015.

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

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

[51] Microsoft Flow. https://flow.microsoft.com. Last accessed October 2020.

[52] S. Munir and J. A. Stankovic. Depsys: Dependency aware integration of cyber-physical systems for smart homes. In *2014 ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '14, pages 127–138. IEEE, 2014.

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

[54] OpenHAB. https://www.openhab.org/. Last accessed October 2020.

[55] Website Optimization. The psychology of web performance. http://www.websiteoptimization.com/speed/tweak/psychology-web-performance/, May 2009. Last accessed Jan 2021.

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

[57] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. Metatm/txlinux: transactional memory for an operating system. *ACM SIGARCH Computer Architecture News*, 35(2):92–103, 2007.

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

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

[60] EHR Software Resource. Ehr and cognitive drift. https://www.wonderdoc.com/ehr-resources/cognitive-drift/, March 2015. Last accessed Jan 2021.

[61] D. Retkowitz and S. Kulle. Dependency management in smart homes. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, DAIS'09, pages 143–156. Springer, 2009.

[62] Roomba. https://www.irobot.com/roomba. Last accessed October 2020.

[63] F. B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[64] A. Sengupta, T. Leesatapornwongsa, Masoud S. Ardekani, and C. A. Stuardo. Transactuations: where transactions meet the physical world. In *2019 USENIX Annual Technical Conference*, ATC'19, pages 91–106. USENIX Association, 2019.

[65] C. Shen, R. P. Singh, A. Phanishayee, A. Kansal, and R. Mahajan. Beam: Ending monolithic applications for connected devices. In *2016 USENIX Annual Technical Conference*, ATC'16, pages 143–157. USENIX Association, 2016.

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

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

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

[69] SmartThings open-source DeviceTypeHandlers and SmartApps code. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps. Last accessed October 2020.

[70] O. Storz, A. Friday, and N. Davies. Supporting content scheduling on situated public displays. *Computers & Graphics*, 30(5):681–691, 2006.

[71] Pusher Team. How latency affects user engagement. https://blog.pusher.com/how-latency-affects-user-engagement/, February 2020. Last accessed Jan 2021.

[72] 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. ACM, 1995.

[73] TP-Link device driver. https://github.com/intrbiz/hs110. Last accessed October 2020.

[74] TP Link HS100. https://www.tp-link.com/us/download/HS100.html. Last accessed October 2020.

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

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

[77] TP-Link KASA HS100, HS220, KL130. https://www.kasasmart.com/. Last accessed October 2020.

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

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

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

[81] Wemo. https://www.wemo.com/. Last accessed October 2020.

[82] What is imprint$^{TM}$ link technology? https://homesupport.irobot.com/app/answers/detail/a_id/21088/\sim/what-is-imprint%E2%84%A2-link-technology%3F. Last accessed October 2020.

[83] Workflow. https://workflow.is/. Last accessed October 2020.

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

[85] Zapier. https://zapier.com/. Last accessed October 2020.

[86] 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. ACM, 2019.