RASC: Enhancing Observability & Programmability in Smart Spaces

Anna Karanika Kai-Siang Wang Han-Ting Liang Shalni Sundram Indranil Gupta

University of Illinois Urbana-Champaign

Abstract

While RPCs form the bedrock of systems stacks, we posit that IoT device collections in smart spaces like homes, warehouses, and office buildings—which are all "user-facing"—require a more expressive abstraction. Orthogonal to prior work, which improved the reliability of IoT communication, our work focuses on improving the *observability* and *programmability* of IoT actions. We present the RASC (Request-Acknowledge-Start-Complete) abstraction, which provides acknowledgments at critical points after an IoT device action is initiated. RASC is a better fit for IoT actions, which naturally vary in length *spatially* (across devices) and *temporally* (across time, for a given device). RASC also enables the design of several new features: predicting action completion times accurately, detecting failures of actions faster, allowing fine-grained dependencies in programming, and scheduling. RASC is intended to be implemented atop today's available RPC mechanisms, rather than as a replacement. We integrated RASC into a popular and open-source IoT framework called Home Assistant. Our trace-driven evaluation finds that RASC meets latency SLOs, especially for long actions that last O(mins), which are common in smart spaces. Our scheduling policies for home automations (e.g., routines) outperform state-of-the-art counterparts by 10%-55%.

1 Introduction

In the last decade, industry and users have moved away from managing individual IoT (Internet of Things) devices to programming and managing IoT device *collections*. IoT permeates our homes [32], workplaces [3], farms [67], Industry 4.0 [35], entertainment venues [65], etc. Smart buildings contain 2 B devices, expected to reach 2.5 B devices and a \$90 B market by 2027, and 4.12 B devices by 2030 [39, 40]. In a single deployment, 10s to 100s of IoT devices may be programmed and managed via automations [33] containing a combination of actions [20, 22], routines [26, 43], IFTTT (If This Then That) [26, 49], scripts, etc. An *action*

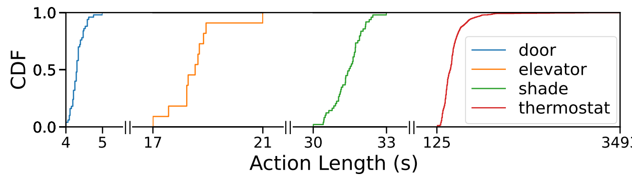


Figure 1: **IoT actions vary in length.** *Actions: door: close, elevator: up 1 floor, shade: open, thermostat: heat 68.x to 69.y °Fahrenheit.*

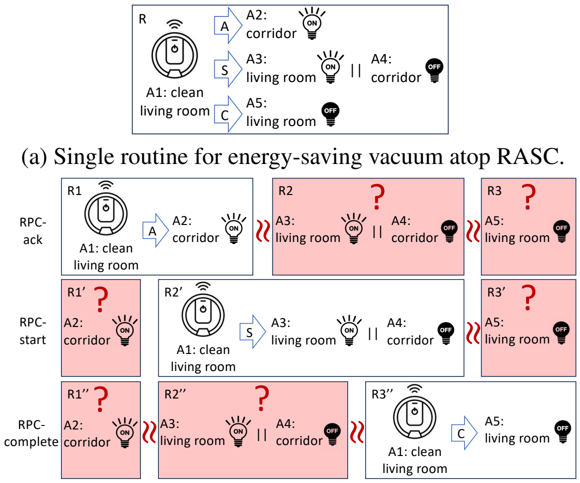
is a single command sent to a single device, from a home hub (e.g., Alexa, Google Home hub) or a user device (e.g., phone, tablet, etc.). For instance, the Google Home API [20] contains over 50 defined actions, e.g., `blinds.OpenClose`, `SetFanSpeedRelative`, `Cook`, `Dispense`, etc. The most common program written by users is a *routine*: a sequence of several actions, which can be triggered conditionally by time, sensors, or manually by the user.

RPCs and IoT Actions: Today, IoT actions predominantly rely on the traditional RPC (Remote Procedure Call) abstraction [8]. RPCs are simple, widely understood, and accepted, and have many stock implementations [14, 15, 58, 62]. An RPC consists of a single *Request* from a hub or personal device towards the IoT device (or its proxy web service), and a single *Reply* in the reverse direction.

However, for IoT settings, it is challenging to map the Reply to the appropriate point in the action's execution. This is because in user-facing settings like smart spaces, many actions are non-instantaneous, taking seconds or even minutes to execute. Physical devices like windows and doors, shades and awnings, kitchen appliances, printers, etc., all take several seconds to execute an action (e.g., Google Home API's actions `blinds.OpenClose`, `SetFanSpeedRelative`). That is, many actions in IoT settings are ***long actions***. Devices like sprinklers, ovens, HVAC systems (heating, ventilation, and air conditioning), etc., may take minutes or hours (e.g., Google Home API's actions `Cook`, `Dispense`, etc.). Even "fast" devices like locks, light bulbs, fans take a fraction of a second to finish an action. Together, these create a non-trivial time gap between starting and completing an action. Fig. 1 shows our measurements from devices in our office building.

Property	Goal	RPC-A	RPC-S	RPC-C	RASC
Observa-bility	Progress updates (Sec. 4.1)	✗	✗	✗	✓
	Failure detection (Sec. 4.2)	✗	✗	∼	✓
Program-mability	Diverse dependencies (Sec. 5.1)	✗	✗	✗	✓
	Dynamic scheduling (Sec. 5.2)	✗	✗	∼	✓

Table 1: **Desired Properties.** $\sim$ *means non-trivial design needed.*



(a) Single routine for energy-saving vacuum atop RASC.

(b) Ineffective RPC-based energy-saving vacuum routines. Pink routines need complex—or unavailable—triggers (?; e.g., "vacuum in living room" ∧ "cleaning/returning"). With suboptimal triggers, cross-routine conflicts (≀≀) arise or actions fire unintentionally.

Figure 2: **Routine example with dependencies on *Acknowledgment* (A), *Start* (S) and *Completion* (C) among actions.** *The vacuum requires light to navigate.*

Further, there is also a gap between the command's reception and start, due to ***preconditions*** for starting a command (e.g., coffee maker's reservoir must have water). Finally, the ***action may fail*** (e.g., main door lock is suddenly jammed) and create safety-violating situations.

Returning to RPCs, this diversity in the nature and length of actions in IoT spaces raises the question of *which* version of RPC to use. Specifically—*when* is the Reply sent back? Today's RPC implementations are *forced to choose* mapping of the Reply to one of three choices—either: (1) *Acknowledgment*: sent back by the device or its cloud service immediately when it receives the action (we call this *RPC-A*), but before the device has started executing the action, *or* (2) *Start* of the action: sent back by the device or its service when the action starts (we call this *RPC-S*), *or* (3) *Completion* of the action: sent back by the device or its service when the action is completed by the IoT device (we call this *RPC-C*). Many commercial smart home deployments today use RPC-A combined with state updates [4, 16], while RPC-C is commonly employed in robotics settings [10, 70].

Goals for an RPC Alternative: The inherent *one-shot* nature of an RPC is a mismatch with the *long-running* nature of IoT actions. We posit that the call-response for IoT device actions must adhere to two inter-related principles: (1) ***Observability***: the ability to track a device action; (2) ***Programmability***: the ability to write and efficiently run *expressive* programs.

Observability requires that the hub (or user device) receive ***progress updates*** at critical action execution points. If any key action part fails, internal mechanisms must ***detect failure fast***, and mitigate. Second, programmability implies that programs support expressive and ***diverse dependencies*** among actions (inside a program, and across programs), e.g., start the next action $A2$ earlier, *during* execution of a previous dependent action $A1$, after $A1$ has crossed key internal points (rather than waiting for $A1$ to finish). These dependencies require us to design ***dynamic scheduling*** for concurrent smart-space actions. Rescheduling is vital because varying action lengths and tail latency (Fig. 1) cause executions to deviate from original estimates and schedules. This intertwined nature of observability and programmability—summarized in Table 1's four goals—parallels challenges distributed systems [36, 48], Kubernetes [38], and SDNs [42].

A New Abstraction: This paper proposes *RASC* (Request-Ack-Start-Complete), a new abstraction for IoT devices, as an alternative to RPC. RASC does not replace RPC, but instead can be built atop RPC. RASC provides Replies at three critical action lifecycle points: **A**ck (the action is received by the device but not yet started), **S**tart (the device starts the action), and **C**omplete (the device finishes the action). Naturally, some of these replies may overlap (e.g., if the action is short), but each is essential for longer actions to satisfy Table 1's goals.

Consider a robotic vacuum that requires ambient light in its work area (for the robot's camera). Fig. 2a shows a routine (a program with a set of actions) named *R* expressed with diverse action dependency types. Because the vacuum needs light for its camera to navigate, initially the corridor light is switched on so that the vacuum can find its way to the living room (since the precondition for the action is that the vacuum reach the living room). After that, the living room light is kept on for vacuuming, but the corridor light is switched off. When the vacuum has completed and docks in the living room, the living room light is switched off. Similar routines exist involving lawn-mowers [52], tele-presence robots [64], etc.

Fig. 2b shows that to *correctly* implement *R* via RPC variants, *R* needs to be split into several routines. Each split is problematic because (i) there are no good triggers for the pink routines (e.g., vacuum location cannot be used in trigger clauses), and (ii) if users go for clauses provided by the APIs, which tend to be simple, they will not get the intended result.

Our system Rascal implements the RASC abstraction. Assuring Table 1 entails several challenges. First, we have to build a **progress/failure detector** atop RASC, that minimizes detection time while respecting device constraints. For poll-only devices, it must poll often enough to track progress and catch failures promptly without overwhelming devices. Second, atop RASC when we provide support for expressive routine dependencies, we must innovate **new dynamic scheduling algorithms** that reduce end-to-end latency (from routine arrival to completion). Third, a key principle in our design is backwards compatibility: to make Rascal immediately deployable atop today's ecosystems, IoT devices and their vendor services cannot be modified. This forces us to work within the constraints of existing RPC interfaces. This is challenging as (i) RASC is more expressive than RPCs, and (ii) RPCs may go either via the IoT cloud service, or directly to the device.

This paper makes the following contributions:

- We present a new expressive RPC-enhancing abstraction called RASC, for IoT settings.
- We build the Rascal system that implements RASC over existing RPC-based IoT APIs. To support action progress updates and failure detection, we propose new techniques for efficient device polling. We also propose new dynamic scheduling policies for diverse action dependencies.
- We integrate Rascal with the popular and open-source home automation system called Home Assistant [21].
- We measured action execution times for various devices in office buildings. We use these and sets of real routines to perform trace-driven evaluation. We find that (i) Rascal detects completion within 2-13 RPCs and 2s-16s over 90% of the time, and (ii) our routine scheduling policies outperform state-of-the-art by 10%-55%.

2 System Model

In today's deployments, devices communicate in one of five ways [23]: *Cloud Pull, Cloud Push, Local Pull, Local Push, Assumed State*. Here, *Cloud* means via the cloud service (device vendor's cloud service), while *Local* means locally directly from the device via Wifi, Bluetooth, Zigbee, etc. *Push* means the device updates the cloud service whenever its local state changes, while *Pull* means the cloud service has to poll the device for any state changes. For very old devices that allow neither pull nor push, the cloud service has to assume a state based on the last action. While the push variant naturally leans into providing updates for start and complete (and also intermediate states along the way), it is mostly adopted by sensor devices. Pull variants are widely prevalent—a cloud service or home hub continuously polls the device to detect updates. In fact, during our experiments, we have observed that at least 30% of sampled vendors supported by Home Assistant only offer pull options.

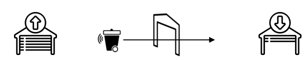
The execution of a single action on an IoT device has three components: (1) a *network component* (transmission of request and response messages among the mobile, hub, cloud, device), (2) a *contextual component* (virtual and physical) comprising preconditions (either physical or safety-related) needed for the device to execute the action, and finally (3) a *physical component* (device executing the action). RASC basically argues that the three stages are spatially and conceptually distinct. We assume IoT devices satisfy a few properties:

- *IoT devices and their vendor services cannot be modified.*
- *Only one action executes on a device at a time.* This is typical for today's devices, some of which maintain a queue (e.g., HP printers [25]) while others reject new requests when busy (e.g., Nuki locks [45]).
- *Sufficient polling rate.* To query the state of a pull-based device, we assume arbitrarily frequent pulls are allowed, e.g., Hue bulbs allow polling every second via Hue Bridge [24].
- *Failures/Speed:* IoT devices may be arbitrarily slow or fail to execute actions. The network may delay or drop packets, and IoT devices and their vendor services need not be synchronized. We assume the reliability of devices beyond our purview (cloud services, hub, phones, etc.).

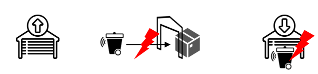
3 Design Goals

We discuss Table 1's key goals in detail. **Action progress updates.** This goal implies that the primitive should provide feedback at key points of the action execution: acknowledgment, action start, and action end (and if the device supports it, during the action). Doing so may require polling, and this polling needs to be efficient.

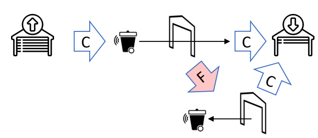
Action failure detection. When an action fails, the primitive should detect it quickly. This property may be implemented atop RPC-complete with the use of a timeout after the expected action length is over; however, that is not trivial since there is no fixed length for each action.



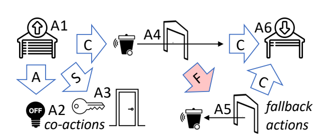
(a) *Routine description:* (i) the garage door opens; (ii) the trashcan is moved out into the driveway; (iii) the garage door closes.



(b) *Action failures:* the trashcan gets stuck below the garage door because of an obstacle and the garage door cannot close.



(c) *Fallback actions:* the trashcan moves back inside after the first movement's failure and the garage door closes.



(d) *Routine as a workflow:* diverse action dependencies allow for fine-grained control with *co-actions* and *fallback actions*.

Figure 3: **Routine with exception handling.** *Garage door opens, automatic trash can [47] goes out, garage door closes. Letter inside arrow indicates trigger event: S=Start, C=Complete, F=Failure.*

Diverse action dependencies. The primitive must support rich inter-action dependencies. For a given example (garage door opens and closes for a trashcan), Figs. 3(a,b) show two scenarios, and Figs. 3(c,d) show two variants. In Fig. 3d, $A2$ and $A3$ are *co-actions* of $A1$. The co-action concept allows (i) the light to turn off *before* the garage door starts opening, so that insects are not drawn indoors, and (ii) the inside door to lock only if the garage door starts opening; if $A1$ fails, no safety gap occurs. If $A4$ *fails* (Fig. 3b), the *fallback action* $A5$ returns the trashcan to its original spot) and, finally, $A6$ closes the garage door to prevent exposure to robbers. Implementing

Fig. 3d via RPC requires users stitching multiple routines (roughly four here–one extra for every new dependency type) guarded by complex predicates to emulate progress points, to block unintended triggers. This is prohibitive.

Dynamic action scheduling. When action lengths differ from the schedule, the primitive should reschedule subsequent actions. Previous smart home routine scheduling [2] (Fig. 3a) ignores unbounded action lengths and diverse dependencies (Figs. 2a, 3d), relying instead on static schedules. Action durations are context-dependent (e.g., floor area, dirt level for vacuum, etc.), and failures are unpredictable at compile or schedule time. The schedule must adapt to ensure: (a) *safety*, i.e., no actions trigger concurrently for the same device (which might cause action rejections), (b) upheld action dependencies, and (c) early starts for actions if predecessors finish ahead of schedule. RPC-A and RPC-S cannot handle this, and RPC-C only detects completion. RASC is more powerful.

4 Observability in RASC

The goals of action progress updates and failure detection (Table 1) can be served via a mechanism that *periodically polls* the device for its current state. This polling needs to (i) satisfy user-specified tolerances for detecting failures, (ii) balance detection latency vs. device/network load, and (iii) adapt to evolving action time distributions (i.e., wrong estimates).

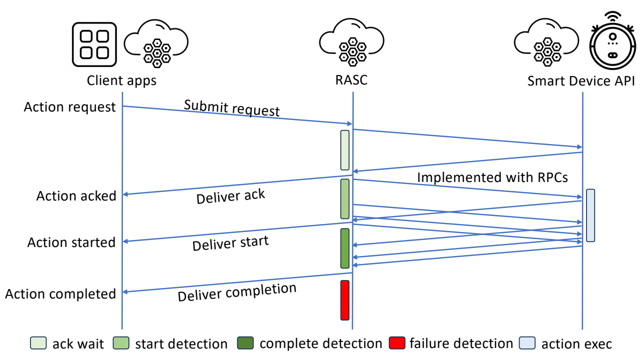


Figure 4: **RASC over RPCs.** *Frequent RPCs on right are polls.*

Lifetime of an Action. Fig. 4 shows how RASC runs backward-compatibly over the RPC layer. For a given action, Rascal cycles through four states (or stages or phases, which we use interchangeably): *ack wait, start detection, complete detection*, and *failure detection*. When the action is requested, Rascal is in the *ack wait* state. Immediately after the hub receives an ack (that the device or its web service has received the request), Rascal enters the *start detection* phase, wherein it starts tracking the state of the device (described in Sec. 4.1). Alternatively, if the IoT device is unresponsive, then Rascal enters the *failure detection state*. The user specifies a **tolerance threshold** Q_w in seconds (maximum time between a state change and its detection), and Rascal needs to meet it.

If the action is short, the action completion target state might be detected alongside or soon after the start. So during the start detection state, Rascal also checks if either of the start or completion states is already matched, and if so, short-circuits to the complete detection state.

In IoT settings, it is impossible to distinguish a failed device from one that is not executing actions at all. So Rascal detects the failure of *actions*, but does not declare a device as being failed. Rascal reports failures to the user (who may take subsequent actions like restarting the device).

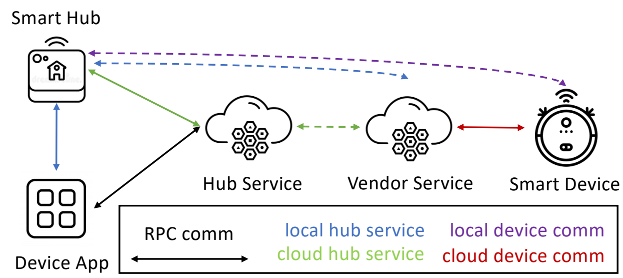


Figure 5: **Communication between a smart device and a client app.** *Each arrowed line is an RPC in today's stacks. This paper replaces dashed lines with the new RASC abstraction.*

IoT Architecture. Fig. 5 shows how RASC is built atop today's RPC mechanisms. RASC interfaces user devices and applications (running on the Hub Service, e.g., Alexa, Google Home), with the device or its vendor service (e.g., web microservices run by the device vendor, e.g., Philips Hue, GE Sync, etc.). In Fig. 5, RASC replaces only those RPCs at dashed arrows, while solid arrow RPCs remain. Every path from the hub/user device to the IoT device contains exactly one RASC call. RASC needs to modify only the cloud services and hubs but not the device or vendor services.

4.1 Adaptive Polling Strategy

Both progress updates and failure detection require *polling* that is (i) efficient (few messages) and (ii) responsive (detects changes quickly). For *push* devices (Sec. 2), this is straightforward—the device or vendor notifies the hub, which relays to Rascal. The challenge is *pull*-only devices. Polling too aggressively (e.g., every 500 ms) can overload devices and the network; polling too sparsely increases detection latency. Key questions arise: *What polling frequency is appropriate? Should polling become more frequent as it progresses?*

Problem Statement. We seek to minimize *detection time*: the gap between the action state change on a device and Rascal's next poll. Inputs are: (i) the **historical state-change time distribution** $p(t)$ for a (device, action) pair—`dist.pdf(t)`, (ii) a **state change time bound** U—`dist.ppf(0.99)` [1] which may occasionally be exceeded, and (iii) a poll **budget** k representing how many total polls are allowed for a given action (representing allowable bandwidth). We must select k poll timepoints in $(0,U]$ that minimize expected detection time.

Solution. Let the k poll times be $L_i, i = 1,...,k$. Then the expected detection time Q can be formulated as:

[1] Percent point function (inverse cdf); `ppf(y)` returns the value with cdf y.

$$Q = \int_0^{L_1} (L_1 - t)p(t)\,dt + \int_{L_1}^{L_2} (L_2 - t)p(t)\,dt + \cdots + \int_{L_{k-1}}^{L_k} (L_k - t)p(t)\,dt \tag{1}$$

$$Q = \sum_{i=1}^{k} L_i \int_{L_{i-1}}^{L_i} p(t)\,dt - \int_0^{L_k} t\,p(t)\,dt \tag{2}$$

Here, each term i represents the expected detection time of L_i if the change happens before L_i and after L_{i-1}. The highest L_k is always equal to U. The second term here is the average of $p(t)$, and thus a constant we can ignore to optimize Q.

Theorem 1 (Adaptive Poll Placement with Fixed Budget k)**.** *Given a time distribution $p(t)$ on $(0,U]$, a poll budget k, and a terminal tolerance $\varepsilon > 0$, polls $0 < L_1^* < \cdots < L_{k-1}^* < L_k^*$ that minimize expected detection time Q and satisfy $|L_k^* - U| \leq \varepsilon$ are given by the following recurrent relation:*

$$L_i^* = \begin{cases} \frac{1}{p(L_{i-1}^*)} \cdot \int_{L_{i-2}^*}^{L_{i-1}^*} p(t)\,dt + L_{i-1}^* & \text{for } i \in \{2,\ldots,k\}, \\ \text{value } \in (0,U] & \text{for } i = 1 \end{cases} \tag{3}$$

Proof Sketch (Full proof in Appendix A). Per (2), only the i-th and $(i+1)$-th terms depend on L_i. After setting the partial derivative w.r.t. L_i to 0, we rearrange and get the recurrence relation (3). Under $p > 0$ and continuity, fixing L_1 determines a unique sequence. Next, we fix $L_1 \in (0,U)$ and generate $L_2,\ldots,L_k$ via (3). Then $L_k(L_1)$ is strictly increasing in L_1. Hence, a unique $L_1^* \in (0,U)$ exists such that $L_k^* = U$. ☐

Next, to find the best sequence $\{L_i^*\}$ we use *binary search*. This leverages our empirical observations that there is a linear relationship between L_1 and L_k. Algorithm 1 initially sets the left and right boundaries to 0 and U, respectively. In line 18, L_1 is set to the midpoint of the current left and right values. Using the recurrence equation from (3), lines 20-29 calculate subsequent values based on L_1. We then check if L_k is close enough to U.[2] If it is, we return $\{L_i\}$. If not, we adjust the left and right boundaries and recursively search for the correct L_1.

After we obtain the placement $\{L_i^*\}$, we can examine the second derivative of each L_i, where $L_0 = 0$ is a constant:

$$L_i'' = \begin{cases} 2 \cdot p(L_i) - (L_{i+1} - L_i) \cdot p'(L_i) & \text{for } i \in [1,k-1], \\ 2 \cdot p(L_i) + L_i \cdot p'(L_i) & \text{for } i = k \end{cases} \tag{4}$$

If any of them is negative, indicating a maximum instead of a minimum, the placement is invalid and a larger k is required.

Fig. 6 shows a pictorial example of a PDF for shade up, and the resulting polls generated by Rascal's Algorithm 1.

Meeting Detection Tolerance. We need to meet the user-specified *detection tolerance* Q_w (gap between failure and its detection). More critical actions (e.g., involving locks, fire alarms, exhaust fans) use smaller values (e.g., < 1 s) while less critical actions (e.g., adjusting light brightness, the position of window shades, ovens) can use larger values. We account for this by ensuring that $\forall i : L_{i+1} - L_i \leq Q_w$ and $L_{i+1} - L_i >$ minimum polling interval allowed by the existing API (naturally, a $Q_w <$ min polling interval is unsupportable).

[2] The closeness is measured via the terminal tolerance ε. In our implementation, we set $\varepsilon = 10^{-5}$ (default in numpy np.isclose()).

Algorithm 1 **Adaptive Polling Algorithm**

1: **procedure** FINDPOLLS(dist, U, Q_w, slo)
2:     **return** RFindPolls(dist, U, 0, $ceil(U/Q_w)$,Q_w, slo)
3: **end procedure**
4: **procedure** RFINDPOLLS(dist, U, left, right, Q_w, slo)
5:     k^* = (left + right) / 2
6:     $\mathcal{L}$ = GetPollingInterval(dist, k^*, U, left, right)
7:     valid = examineQw(dist, $\mathcal{L}$, Q_w, slo)
8:     **if** left == right & valid **then**
9:         **return** $\mathcal{L}$
10:     **if** valid **then** ▷ Try to reduce polls
11:         **return** RFindPolls(dist, U, left, k^*+1, Q_w, slo)
12:     **if** N+1 >= right **then** ▷ Need more polls than *right*
13:         **return** RFindPolls(dist, U, k^*+1, $right \times 2$, Q_w, slo)
14:     **return** RFindPolls(dist, U, k^*+1, right, Q_w, slo)
15: **end procedure**
16: **procedure** GETPOLLINGINTERVAL(dist, k, U, left, right)
17:     $\mathcal{L}$ = zeros(k)
18:     L_1 = (left + right) / 2
19:     too_large = False
20:     **for** i in range(2, k + 1) **do**
21:         L_i = calculateByRecurrenceRelation()
22:         **if** $L_i > U$ **then**
23:             too_large = True
24:             **break**
25:     **if** isclose(L_k, U) **then**
26:         **return** $\mathcal{L}$
27:     **if** too_large **then**
28:         **return** GetPollingInterval(dist, k, U, left, L_1)
29:     **return** GetPollingInterval(dist, k, U, L_1, right)
30: **end procedure**

Given Q_w, we use binary search to determine the smallest k that meets Q_w—see lines 4-15 in Algorithm 1. Line 8 examines the $\mathcal{L} = \{L_i\}$ found by *GetPollingInterval* to make sure any interval between any two polls is smaller than Q_w. If $\mathcal{L}$ is valid and left = right, we return $\mathcal{L}$. If left does not equal right, this means we might be able to find $\mathcal{L}$ with a smaller k^*. The algorithm recursively updates k^* until it finds a k^* that respects Q_w and also cannot be smaller. The algorithm terminates when reducing k^* by 1 would violate Q_w.

Reducing Polling further via Confidence Intervals. We relax the worst-case detection time assumption by introducing a *service-level objective (SLO)*: a confidence level for meeting the detection window Q_w. When SLO=0.9, Rascal's placement may violate Q_w on at most 10% of events. This allows Rascal to avoid excessive polling in high probability areas of the distribution. Our evaluations (Sec. 7.1) find that SLO-awareness does not change average detection time. Algorithm 1 shows how to find polls under SLOs.

Theorem 2 (Meeting a Detection Tolerance SLO–Condensed version)**.** *Given a detection window Q_w, a service-level objective $SLO \in (0,1]$, and a binary oracle that detects whether a k-poll placement with coverage of at least SLO exists, binary search on the number of polls k returns the smallest k^* that*

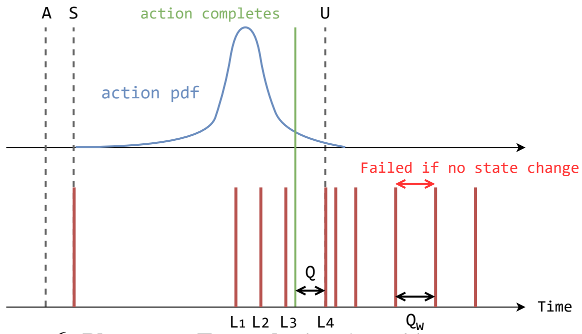


Figure 6: **Placement Example,** $k = 4$**.** *Red lines are RASC polls (after action start). Action completes between second and third poll.*

meets the SLO. (Full theorem and proof in Appendix A.)

Proof Sketch. The oracle predicate is monotone: if some placement meets the SLO with k polls, the SLO can still be met with any larger k' by adding polls. The search is bracketed because $k = 0$ is infeasible, while a sufficiently large k (e.g., evenly spaced polls with $k = \lceil U/Q_w \rceil$) guarantees feasibility. Binary search on a monotone predicate with a valid bracket returns the minimal feasible k^* that satisfies the SLO. □

4.2 Detection Beyond Upper Bound

Upper bound estimates for action completion may be violated due to human interference (e.g., propped elevator doors), environmental factors (e.g., colder weather slowing a heater), and out-of-order requests (e.g., the elevator moving from 1st to 3rd floor receives a 2nd-floor request). Consequently, if no state change occurs by time U, polling must continue. The key question is: *What is the optimal post-U polling technique?*

Our key observation is that past U, the chance of an imminent state change typically declines, so Rascal polls densely before tapering. We distinguish actions by progress: for those showing *some* partial progress (e.g., vacuum location changed), we estimate the remaining time via the observed rate (time to the current progress) and schedule the next poll at $min\{\text{estimate}, Q_w\}$. Otherwise, Rascal uses *exponential backoff*, doubling the inter-poll gap up to Q_w, to avoid overloading troubled devices while ensuring detection within Q_w (Fig. 6). If no change occurs within Q_w after the upper bound U we declare the action failed. In our experiments, overruns beyond U are rare, and late actions typically finish soon after U.

5 Building Atop the RASC Abstraction

While Sec. 4 addressed observability, we now turn to programmability: diverse dependencies and dynamic scheduling.

5.1 Diverse Action-Causality Expressiveness

Commercial IoT platforms rarely support cross-action dependencies: e.g., Alexa routines allow limited inter-routine chaining,[3] but not dependencies between actions. The RASC abstraction enables both cross-action and finer-grained dependencies across critical points *inside* an action's progress. Figs. 2a, 3d depict *co-action* and *fallback action* examples.

In Rascal each routine is represented internally as a DAG of device actions (Fig. 2a). When an action emits a progress event (A, S, or C), Rascal inspects its dependent children; any child whose dependencies are satisfied is immediately scheduled to run. Rascal execution is *event-driven*.

5.2 Dynamic Action Scheduling

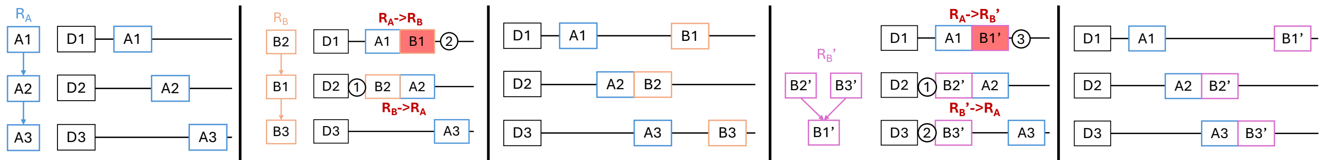
We describe how to schedule routines and actions when some of these actions take a shorter or longer time than expected.
Background. In real-world IoT deployments, action durations vary widely: ovens heat faster when pre-warmed, elevators stall if doors are obstructed, and HVAC cycles fluctuate with outside temperature. Due to these variable times, any static schedule that is made, which devices execute (to respect dependencies) will need to be *dynamically* adjusted at runtime.

The real-time systems community studied this problem through *resource reclamation algorithms* [18,37,57]. These algorithms optimize for *early completions*: if a task finishes ahead of its worst-case bound, subsequent tasks can advance to reclaim idle time while maintaining precedence. For example, *Restriction Vectors (RV)* [37] maintains a task timeline and, upon early task completion, pulls dependent tasks forward to begin once all predecessors have completed. This reclaims idle slack while preserving precedence constraints.
Challenges. We need to extend reclaiming algorithms to handle both early completions and actions *exceeding* their schedule. A heater lagging on a cold day or a door closing slower than usual delays subsequent tasks, and naive scheduling can break causal dependencies or stall routines.

Rascal's fine-grained causality model (Sec. 5.1) makes scheduling challenging: we must honor dependencies at arbitrary progress points (e.g., Ack, Start, Complete), increasing potential conflicts and correctness risks.

Correct IoT scheduling hinges on two properties: ***Safety*** (no device receives two action requests at once) and ***Serial Equivalence*** (overlapping routines execute with some serial order). Without them, final states can be unpredictable or inconsistent with any one routine. SafeHome [2] guarantees these only for fixed action durations. However, with unpredictable durations and fine-grained dependencies, maintaining these guarantees requires adaptation.
Overview. Rascal extends reclamation-style scheduling to IoT, building atop the RASC abstraction (Sec. 4). Rascal's scheduling occurs in two phases: (i) it schedules arriving routines (Sec. 5.2.1), and (ii) it dynamically adjusts the schedule for early state changes and delays (Sec. 5.2.2). We present ***DAG-TL***, to ensure serial equivalence for routines despite

[3] Alexa can call a routine from another but offers no richer dependencies.



(a) R_A. (b) R_B & invalid TL. (c) R_B & valid TL. (d) R'_B & invalid DAG-TL. (e) R'_B & invalid DAG-TL.

Figure 7: **Example Routines and Schedule for TL vs DAG-TL.** *Action symbols feature a letter that stands for the routine they are a part of and a number to denote which device they execute on, e.g., $A1$ is part of routine R_A and executes on device $D1$.*

variable action lengths, alongside two rescheduling policies, ***STF and RV***, which adapt to both early and late state changes while preserving Safety and Serial Equivalence.

5.2.1 DAG-TL: Ensuring Routine Serial Equivalence

We adapt the Timeline Scheduler (TL) from [2]. Upon routine arrival, the original TL places actions in the earliest gaps that preserve intra-routine action order and inter-routine serialization order. TL uses action-by-action backtracking to find gaps that maintain serialization at all times. For Rascal-style DAGs, the original TL has exponential worst-case time complexity.

To handle Rascal-style routines expressed as DAGs, our *DAG-TL* approach adopts an *aggressive backtrack strategy* that makes "big jumps" in the state space. Specifically, upon detecting a serialization conflict, it jumps to the routine's top-level action and shifts the entire DAG forward in a single step—placing it after the last conflicting time found by inspecting the intersection of the preceding and following routine sets in the serialization order. By pruning unsatisfied paths early, we avoid repeated local retries.

Illustrative example. Fig. 7 shows three routines: R_A, R_B and R'_B. After sequential R_A is scheduled (Fig. 7a), sequential R_B arrives. TL fills the earliest gaps: it tentatively places $B2$ (1), then places $B1$ (2), which reveals a serialization violation (Fig. 7b). TL then moves $B2$ after $A2$ (the next available gap) and completes scheduling R_B without further violations (Fig. 7c). Now consider non-sequential R'_B arriving after R_A (Fig. 7d). Here $B1'$ depends on $B2'$ and $B3'$. With a breadth-first traversal, DAG-TL encounters a violation at the third step ($B1'$); it schedules R'_B anew from the root and shifts the entire DAG so that all of R'_B follows R_A on any shared device (Fig. 7e). Naïve per-conflict backtracking would recheck and reschedule many prior actions and does not scale; DAG-TL's top-level backtrack reduces scheduling steps in practice.

5.2.2 Rescheduler

An action may finish later (over-time) or earlier (under-time) than initially scheduled. Rascal must adjust. We describe how actions are rescheduled to accommodate schedule deviations, while minimizing routine completion time and continuing to ensure safety, serial equivalence, and performance.

Problem Statement. Each routine R is a DAG of actions. An action a has: (i) a device $dev(a)$, (ii) a (possibly data-driven) duration estimate $len(a)$, and (iii) dependency edges to its parents in the routine $Pred_{\text{DAG}}(a)$. Multiple routines may run concurrently and touch overlapping device sets.

Our goal is to maintain a *feasible schedule* that (1) is **Safe** (no device runs two actions at once), (2) is **Serially Equivalent** [2] across routines (the final state equals some serial execution of whole routines), and (3) **adapts** when actions finish early/late while preserving (1)–(2).

When to Trigger the Rescheduler. Late actions are handled by Rascal *proactively*, to ensure safety. When a high percentage of the action's state change length upper bound U (Sec. 4.1) has elapsed (95% in our implementation), Rascal extrapolates to estimate the new state change time (e.g., oven took 20 min to warm up from 200°F to 400°F, then Rascal estimates it will take 5 min more to get to the target 450°F).

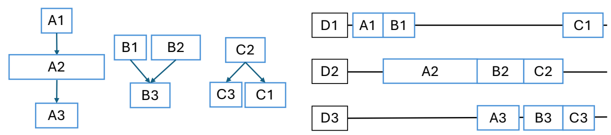
Early-state-changing actions are handled *reactively*, as early state changes do not violate safety. If the difference between expected and actual state change exceeds a threshold (1 s in implementation), we invoke the rescheduler.

Constraints on the Rescheduler. Rescheduling must respect two primary constraints: (1) dependency on prior actions in a routine, and (2) the (immutable) serialization order established (Sec. 5.2.1) across active routines. We handle these via preprocessing and two specialized rescheduling algorithms.

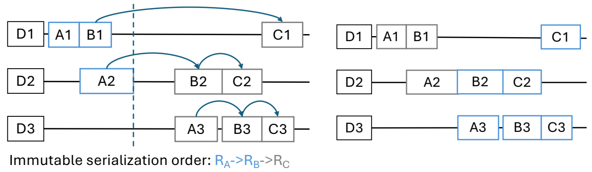
Preprocessing. Let I be the set of potentially impacted actions. We add an action A' to I when a deviation on action A satisfies any one of three conditions: (i) A' is in the same routine R and is a descendant of A in R's action DAG; (ii) A' is scheduled to start after A on the same device D that deviated; (iii) A' belongs to a routine R' that is serialized after R.

To preserve the *established* serial order among routines, Rascal records each routine's *postset*: the routines that, on any shared device, have executed after it up to the rescheduler's trigger time. It then constructs an immutable serialization order by repeatedly: (1) selecting all routines with empty postsets, (2) placing them at the front of the order (breaking ties by arrival time), and (3) removing them from the remaining postsets. We iterate until all routines are placed.

Rascal deschedules all actions in I, and reschedules them while still preserving the established serial order. For the routines in Fig. 8a and the original DAG-TL schedule in Fig. 8b, Fig. 8c shows the case where $A2$ completes early: affected actions (grey) are descheduled, the immutable order is $R_A \rightarrow R_B$, and routines that have not yet started (e.g., R_C) are left unchanged. Finally, Rascal inserts cross-routine, same-device dependencies to maintain serial equivalence.

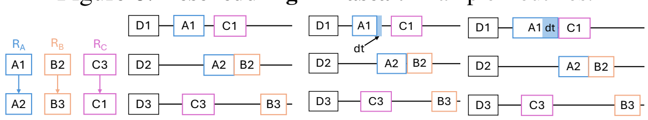


(a) Routine graphs. (b) Original schedule.



(c) Preprocessing phase. (d) Rescheduling phase.

Figure 8: **Rescheduling in Rascal:** Example Routines.



(a) Routine graphs. (b) Original DAG-TL schedule. (c) A_1 completes early $\rightarrow$ only RV. (d) A_1 completes late $\rightarrow$ Push + RV.

Figure 9: **RV rescheduling.** *For (d), upon A_1's late completion detection by an estimated dt, C_1, A_2, B_2, B_3 are first pushed forward by dt before RV is performed.*

Rescheduling. We reschedule descheduled actions using two policies. First, we augment the Shortest Task First (STF) algorithm (known to be optimal [61]) to honor inter- and intra-routine dependencies (Algorithm 2). We prioritize actions by estimated length; we repeatedly (i) pick the head, (ii) place it in the earliest common free slot across required devices, and (iii) update downstream dependencies, until the queue empties. In Fig. 8, STF respects the immutable serialization order, not placing $C2$ before $B2$ despite $C2$ being shorter (Fig. 8d).

Our second policy adapts *Restriction Vectors (RV)* from real-time systems [37]. The classical RV pulls a task forward once all its parents finish to reclaim slack. In Rascal, we apply this directly when progress points (e.g., *Start*, *Complete*) occur earlier than scheduled. Unlike classical real-time settings, IoT actions may also finish *later*, which is acceptable; to remain correct, we extend RV as follows: (i) shift *all* subsequent, not-yet-started actions forward by the observed delay to preserve safety; then (ii) apply RV to the shifted schedule to exploit remaining slack. Thus, RV stays correct under delays while still harvesting early-completion gains (see Fig. 9).

Theorem 3 (Action Safety)**.** *For any device D, Rascal: (i) never initiates >1 action to D concurrently, and (ii) initiates an action only if D is idle. Thus, ack-ed actions on D execute in isolation.*

Proof. *DAG-TL* assigns each action to the earliest idle slot on D that respects dependencies. *RV* may move actions earlier, but only when D is idle and all parents are complete. *STF* inserts each action into the earliest idle slot consistent with serialization. Neither rule permits overlapping initiations. Thus, the non-overlap invariant is preserved under both schedulers and all rescheduling steps. □

Algorithm 2 Shortest Task First (STF) with Serializability

Require: Impacted actions $\mathcal{A}$; duration $len(a)$; device $dev(a)$; routine DAG predecessors $Pred_{\text{DAG}}(a)$; immutable routine order $\prec$; device next-free times $next\_free[d]$ from current schedule.
Ensure: Updated schedule without violating device safety or serial equivalence.

Phase 0: Add serialization edges (once).
for all pairs of routines $R \prec R'$ **do**
  for all devices d shared by R and R' **do**
    Add edge from last action of R on d to first action of R' on d
Phase 1: Initialize readiness and earliest feasible starts.
for all $a \in \mathcal{A}$ **do**
  $Pred(a) \leftarrow Pred_{\text{DAG}}(a) \cup Pred_{\text{serial}}(a)$
  $indeg[a] \leftarrow |Pred(a)|$; $EST[a] \leftarrow 0$; $FIN[a] \leftarrow$ unset
$Ready \leftarrow \{a \in \mathcal{A} \mid indeg[a] = 0\}$
for all $a \in Ready$ **do**
  $start\_cand[a] \leftarrow next\_free[dev(a)]$
Phase 2: List scheduling (single priority queue).
while $Ready \neq \emptyset$ **do**
  $a^\star \leftarrow \arg\min_{a \in Ready} (start\_cand[a], len(a))$ ▷ Earliest start, then shortest task
  $s \leftarrow start\_cand[a^\star]$; $f \leftarrow s + len(a^\star)$
  Place $a^\star$ on timeline of $dev(a^\star)$ at $[s, f)$
  $FIN[a^\star] \leftarrow f$; $next\_free[dev(a^\star)] \leftarrow f$
  Remove $a^\star$ from $Ready$
  for all successors b of $a^\star$ in $Pred(\cdot)$ **do**
    $indeg[b] \leftarrow indeg[b] - 1$
    $EST[b] \leftarrow \max(EST[b], FIN[a^\star])$
    if $indeg[b] = 0$ **then**
      $start\_cand[b] \leftarrow \max(EST[b], next\_free[dev(b)])$
      Insert b into $Ready$
return updated schedule

Theorem 4 (Serial Equivalence)**.** *For any set of concurrent routines $\mathcal{R}$, the schedule $\mathcal{S}$ produced by Rascal is equivalent to executing $\mathcal{R}$ in some serial order $\pi(\mathcal{R})$. This holds for the baseline scheduler and for rescheduling under RV and STF.*

Proof. *DAG-TL* enforces a partial order $\prec$ on conflicting routines and produces a schedule $\mathcal{S}$ that extends $\prec$. *RV* advances or delays an action's start $s(a)$ only if $\prec$ is preserved. *STF* places each action a in the earliest idle slot $i(a)$ consistent with $\prec$. No step moves an action past the serialized prefix of a conflicting routine; thus, $\mathcal{S}$ is equivalent to some serial order $\pi(\mathcal{R})$ consistent with $\prec$. □

6 Implementation

We implemented Rascal as a new component (~8.5K Python LOC) for Home Assistant (HA) [19], a widely used open-source smart-home platform. Deploying Rascal requires no changes to the HA core, integrations, or devices and their APIs. Users simply install Rascal and provide lightweight per-device-class mappings that bind action execution points (i.e., start, completion) to existing device state attributes. This lets Rascal infer action progress and schedule polls without modifying devices. We also built a 500-LOC HA simulator and adapted the `hass-virtual` third-party component [69] for device simulation and Raspberry Pi-based emulation.

```yaml
- id: "1715802493830"
  alias: Heat when window starts closing and door is closed
  action:
    - parallel:
        - service: cover.close_cover
          target: { entity_id: cover.living_room_window }
        - service: cover.close_cover
          target: { entity_id: cover.balcony }
    - service: climate.set_temperature
      data: { temperature: 72 }
      depend_on: [start, complete]
      target: { entity_id: climate.main_thermostat }
```

Figure 10: **Example RASC routine with `depend_on` specification.** *`desc`, `trigger`, `condition` omitted for brevity. The main thermostat is requested to heat to 72°F after the living room window has started closing & the balcony door has closed.*

RASC API. We extend HA's action YAML with a single optional field, `depend_on`. This field is an *ordered list* of required progress events—one per parent, in the parents' order (e.g., `ack`, `start`, `complete`)—enabling fine-grained sequencing. An example routine with mixed dependencies appears in Fig. 10. If `depend_on` is omitted, we default to `complete`, preserving backward compatibility with HA's sequential semantics. At runtime, the YAML is compiled into a dependency DAG; DAG-TL monitors this DAG and triggers actions as soon as prerequisites are satisfied.

Action length PDF. We obtain each action's probability distribution function by storing historical durations for (ack→start) and (start→complete). Actions are keyed by $\langle device, action, transition\rangle$. Consequently, our dynamic polling strategy requires a brief training phase (several trials) before it can return accurate results.

Asynchronous poll placement computation. Algorithm 1's compute time increases with the upper bound, which can be large for long actions. To avoid polling delays in Rascal, we decouple computation from execution and make it asynchronous: upon an action's completion, poll placement is computed anew and stored for its next initiation.

Action status change notification. On each status change, Rascal publishes an `RASC_RESPONSE` event on HA's event bus; applications subscribed to this topic receive the update.

7 Experimental Evaluation

We address the following research questions:

1. How efficient is Rascal's polling technique? (Sec. 7.1)
2. Does action length estimation converge? (Sec. 7.2)
3. If the action length distribution evolves, can Rascal reconverge its learnt distribution quickly? (Sec. 7.2)
4. When actions are interrupted, can Rascal distinguish them from a scenario where they actually fail? (Sec. 7.3)
5. Under realistic routines, how do DAG-TL+{RV, STF} compare to baselines [2] in latency & overhead? (Sec. 7.4)

Trace Collection. To drive our trace-driven emulation, we collected traces of (I) device action times that form a diverse set of devices in an office building, and (II) a set of diverse

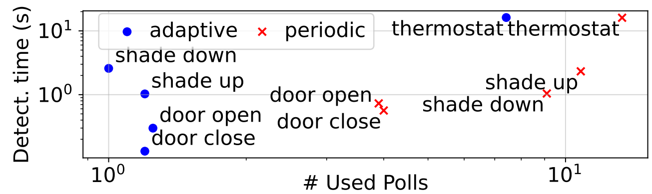


Figure 11: **Rascal's Adaptive Polling vs. Baseline Periodic Polling.** *Rascal is better (lower and to the left) in 4 of 5 cases.*

Action	Avg. Length	Detection Time		Computation Time		Speedup
		Adaptive	V-opt	Adaptive	V-opt	
Door close	3.19	0.19	**0.13**	**6.04e-04**	4.38e-02	59.37
Door open	3.06	**0.19**	0.30	**6.43e-04**	3.82e-02	72.56
Shade up	29.64	**0.31**	1.02	**3.96e-02**	1.97e+01	497.77
Shade down	27.45	**1.54**	2.58	**2.93e-02**	1.63e+01	554.99
Therm 68,69	432.17	**16.19**	–	**9.04**	2hr+	796.34

Table 2: **Rascal's Adaptive Polling vs. Baseline V-opt-calculated Polling, Detection vs. Computation Time. All times are in seconds. Bold font indicates the lowest value for each action/metric combination.** *Rascal has similar detection times, but V-opt's computation time is impractical, especially for long actions.*

routines, inspired by existing routine datasets including IoTBench [28, 60], described in Sec. 7.4. For (I), we manually collected a significant dataset of action completion times. We recorded multiple (>50) trials over 5 days, with 2 elevators, 2 projector screens, 2 lights, 2 doors, and 4 shades. Elevator data was collected while other users were using it, and thermostat traces are from an existing dataset [11].

7.1 Rascal Polling Efficiency

We compare Rascal's adaptive polling efficiency to a baseline periodic polling strategy, which polls every Q_w. Fig. 11 shows that compared to periodic polling, adaptive polling uses 44% to 89% fewer polls to detect the action completion. We set the Service Level Objective or *SLO*—the percentage of events Rascal promises to detect within Q_w (Section 1)—to 0.9 and Q_w equal to 2, 3, and 30 (seconds) for door, shade, and thermostat actions, respectively. Action `shade down`'s detection time is higher with adaptive polling but still below Q_w. This data point highlights the following tradeoff: periodic polling is costly and guarantees detection within Q_w always, whereas adaptive polling is very cost-efficient but may detect action progress within Q_w only with high probability ($\geq SLO$).

Table 2 compares Rascal's adaptive polling against the baseline called V-optimal (V-opt) [27], the classic dynamic-programming algorithm for histogram construction. V-opt selects bucket boundaries to minimize variance across bins. Both achieve comparable detection times, but Rascal computes poll placements 60-800× faster than V-opt. This efficiency gap makes V-opt impractical in real deployments; its computation time can exceed even the average action length (e.g., thermostat), while Rascal completes in milliseconds.

Fig. 12 shows the impact of device or API rate limits on Rascal's polling. The x axis shows the API/device-imposed minimum inter-arrival (MIA) gap between consecutive polls. The y axis shows the user-specified tolerance threshold (Q_w).

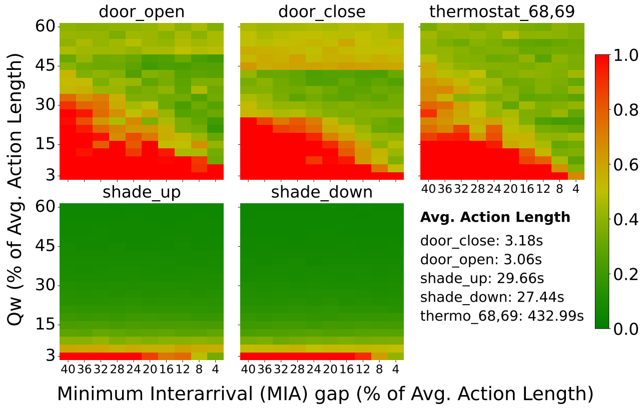
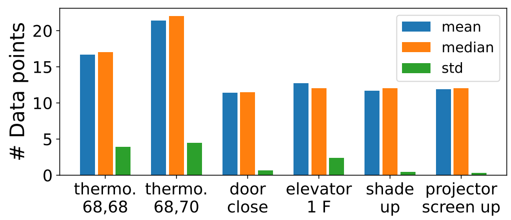
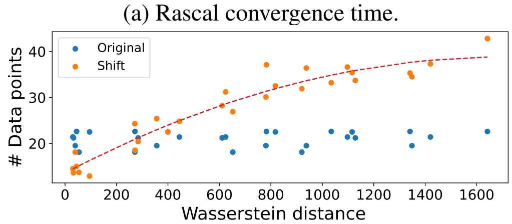


Figure 12: **Effect of Extreme Rate Limiting: Rascal's Detection time under varying Q_w (user-specified tolerance threshold).**



(a) Rascal convergence time.



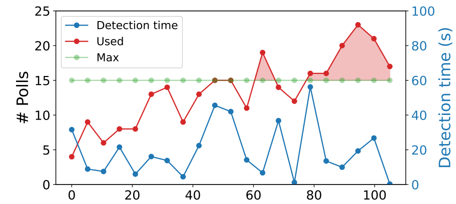
(b) Rascal convergence when action length drifts.

Figure 13: **Rascal Convergence.** *Rascal converges quickly, requiring just over 20 (new) data points.*

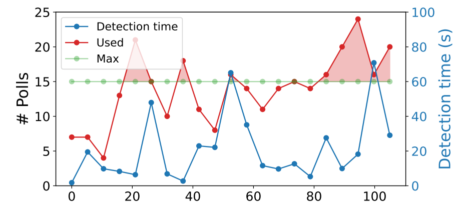
Both axes are normalized to average action length. The color is $min(\frac{detection\ time}{Q_w}, 1)$. Thus green and yellow indicate detection time threshold being met. We observe that (i) for a given MIA, too stringent Q_w values may not be satisfiable (red in plot); (ii) this Q_w value threshold (for meeting 100% detection: green in plot) drops quickly as MIA becomes less stringent (left to right); and (iii) detection time for uninterruptible mechanical actions (e.g., shade up & down) is more tolerant to rate limiting, because uninterruptible actions have tighter (and thus predictable) action length distributions.

7.2 Rascal Training

Convergence Speed. We measure how many action-length samples Rascal needs to reach a *stable* per–(device, action) distribution, i.e., mean and variance both differ by less than 5% upon consecutive samples. Fig. 13a shows that 10-20 samples suffice across devices: actions lasting tens of seconds (door, elevator, shade, projector screen) converge in ~ 10 samples, while longer actions (e.g., the 400 s thermostat) require ~ 20.
Data Drift. As devices get older, a single action's probability distributions for its progress points may start moving towards different ranges and probabilities. How quickly can Rascal's measurements catch up? Fig. 13b measures (for a thermostat device, at various temperature settings) the Wasserstein (earthmover) distance [30, 66]. This metric measures the distance between the real and learned distributions, for both the original distribution and the shifted distribution. We observe quick convergence, especially when the original distribution is < 500 (already a large value, much higher than the error in our measurements). For instance, the Wasserstein distance between the distributions from the door and the elevator is approximately 13.93. In reality, it is highly unlikely that a door would experience such significant degradation.



(a) The interruption happens at the 50% of the action.



(b) The interruption happens at the 80% of the action.

Figure 14: **Rascal performance (polls) vs. Interruption Length relative to action length (%). Action: thermostat from 68 to 69.** *Rascal generally uses fewer polls than max. Exceptions: when interruptions are long (right side of plots).*

7.3 Detecting Interrupted Actions

Interruptions to ongoing actions are common due to environmental factors and human interactions. Fig. 14 shows a real deployment where a thermostat change (68→69) is interrupted at 50% and 80% progress. The red area denotes extra polls beyond U. For early or brief interruptions, Rascal's overhead is low; with prolonged interruptions (right side of both plots), overhead rises, but detection time (blue line) stays low.

Fig. 15 shows a simulation where we focus on *False Positive rate*, i.e., the percentage of experiments where Rascal *falsely* detected an action failure. We observe the False Positive rate rises with longer interruptions, especially when they occur near the action's end, indicating Rascal reliably distinguishes true failures from interruptions in most cases.

7.4 Routine Scheduling

We evaluate Rascal's end-to-end performance: overheads and comparing DAG-TL scheduling (Sec. 5.2) to baselines.
Dataset of Routines. We generated a suite of routines spanning diverse devices and lengths—statistics of our routine dataset are shown in Fig. 16. We have two arrival workloads:

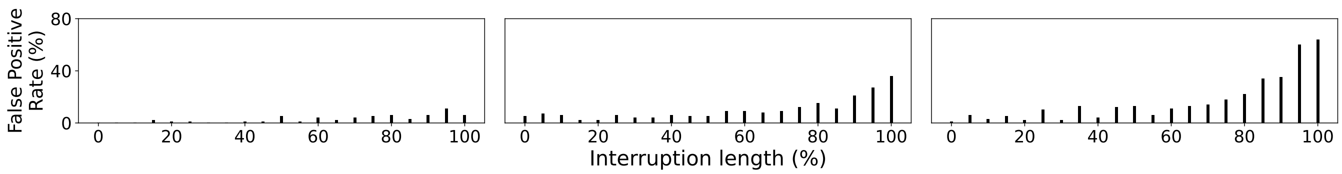
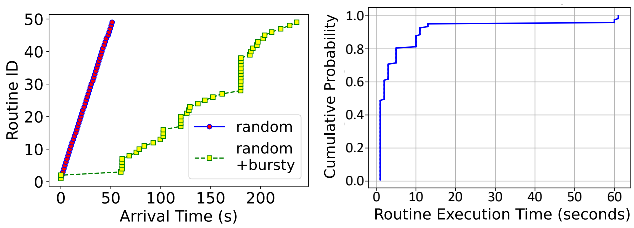


Figure 15: **Action Failure False Positive Rate during interruption. Left to right: interruption happens (resp.) at the 50%, 80%, and 90% points of action.** *False positives are generally low. Exception: when interruptions are long and towards the action's end.*



(a) The timeline of routine arrivals for the two datasets.

(b) The sum of action lengths per routine as a CDF plot.

Figure 16: **Our Routine Benchmark.**

Dataset	Polling Strategy	CPU (%)			Memory (MB)		
		avg	q50	q99	avg	q50	q99
random	adaptive	8.79	1.8	19.8	4.12	4.73	8.33
	periodic	9.6	8.6	24.06	**3.36**	**3.33**	**6.22**
	none	**2.41**	**1.1**	**10.1**	17.72	19.19	20.28
random + bursty	adaptive	3.55	1.75	**9.68**	**4.1**	**4.36**	**5.9**
	periodic	4.6	2.9	19.26	4.54	4.61	6.58
	none	**2.72**	**1.3**	22.39	17.87	18.21	19.07

Table 3: **Rascal's resource utilization under different polling strategies. Bold font indicates the lowest value for each dataset/metric combination.** *Rascal either incurs the lowest CPU or memory (among baselines), or is comparable to the lowest option.*

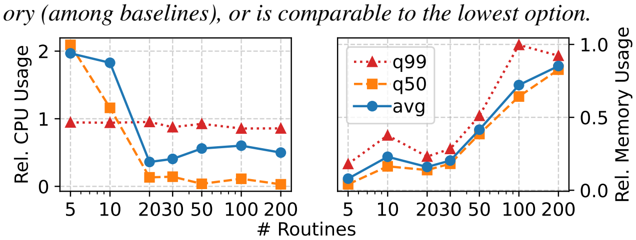


Figure 17: **Rascal's relative resource utilization across varying routine loads.** *Rascal's CPU overhead is consistently low, while memory usage scales sub-linearly with the # concurrent routines.*

(1) *random*, attempting to capture "background" routine arrival throughout the 24 hours of a day, and (2) *random+bursty*, a mix of 50% random and 50% bursty arrivals, intended to capture "peak" times of activity, such as morning, lunch, evening.

Overheads. Table 3 reports Rascal's process overhead under different polling strategies. Adaptive polling is consistently more efficient than periodic—up to 63% lower CPU and 43% lower memory. Disabling polling (*none*) yields only marginal or inconsistent gains (CPU q99 up to 68%, otherwise up to 33%), and memory can be worse. In fact, memory is higher with no polling than with Rascal, indicating our automation-tracking memory management outperforms Home Assistant's baseline, which overuses external libraries.

Fig. 17 illustrates Rascal's *relative* resource utilization, calculated as the ratio of utilization with *adaptive* polling to utilization with polling disabled (*none*). We vary concurrent routine load from smart home scale (up to 10) to smart buildings and campuses (up to 200). Both Rascal's 99th percentile CPU & all memory metrics stay below that of disabled polling. While mean (median) CPU usage peaks at $\sim 2\times$ at low loads (up to 20 routines) due to upfront Rascal overhead, it then drops, stabilizing at $\sim 0.6\times$ ($\sim 0.1\times$). Relative memory rises with load but plateaus with 200 routines at $\sim 0.8\times$. This is the resource vs. benefit tradeoff entailed by Rascal.

Fig. 18 shows Rascal's rescheduling overheads with STF (Shortest Task First). For both datasets, rescheduling time drops as action-length estimates increase, indicating that overestimation reduces overhead—even with more early completions. The random+bursty dataset achieves faster rescheduling time as the schedule is tighter at times and traversals are faster.

Rascal Rescheduler Vs. Baselines. We compare two Rascal reschedulers (STF and RV, Sec. 5.2.2: labeled in our plots as DAG-TL+STF and DAG-TL+RV respectively) against three baselines. Our three baselines are: (i) FCFS (First Come First Served): strict no-overlap scheduling by arrival; (ii) FCFS-Post: a faster FCFS variant allowing a routine to start once predecessor routines finish on all shared devices; (iii) Just in Time (JiT) algorithm [2], a greedy, opportunistic approach.

Figs. 19, 20 show that (i) Rascal's two variants (DAG-TL+RV and DAG-TL+STF) perform similarly, (ii) Rascal's slowest variant (DAG-TL+STF) finishes routines 11% faster than the fastest baseline (JiT), and (iii) Rascal's slowest variant's wait times are also lower—mean by 33% and q95 by 50%—than the strongest baselines (JiT and FCFS, respectively). Fig. 21a shows that Rascal reduces idle time (time a device is not utilized) by 27% at mean and 18% at 95th percentile compared to the fastest baseline (FCFS and JiT, respectively). In Fig. 21b, Rascal's slowest variant increases the routine end-to-end latency average time by only 25% and 95th percentile by 55%, compared to the fastest baseline (JiT and FCFS, respectively). Finally, Fig. 21c shows that Rascal's least efficient variant achieves 10% higher average parallelism rates over the most efficient baseline (JiT). Overall, Rascal's rescheduling improves metrics by 10–55% over baselines.

8 Discussion

Action lengths of some devices (e.g., shades) are more predictable than others, e.g., a thermostat (HVAC) may have seasonal variations, oven cook time depends on food, etc. Sec. 7 evaluated *representative* devices that span this spectrum. More expressive observability (e.g., progress bars) may be feasible atop RASC, but it must be *simple* and *accurate*.

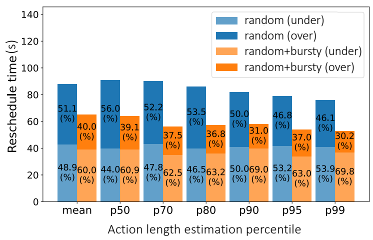


Figure 18: **Rescheduling Overhead. Bars show total rescheduling time over the dataset, split into under-/over-time handling.** *Rascal overhead stays low.*

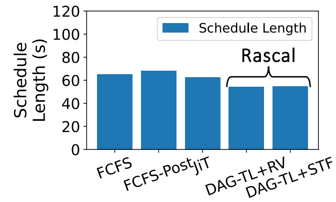


Figure 19: **Schedule length comparison.** *Rascal (right two) creates at least 11% faster schedules vs. Baselines (left three).*

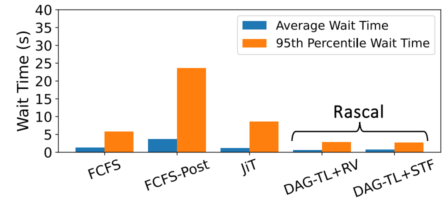
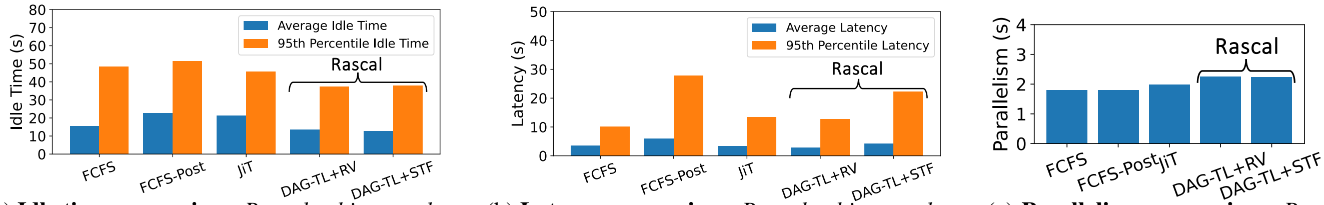


Figure 20: **Wait time comparison.** *Rascal (right two groups) decreases wait times by at least 33% and 50% for mean and q95 values, respectively, vs. Baselines (three left groups).*



(a) **Idle time comparison.** *Rascal achieves at least 27% and 18% lower mean and q95 values, resp.*

(b) **Latency comparison.** *Rascal achieves at least 25% and 55% lower mean and q95 values, resp.*

(c) **Parallelism comparison.** *Rascal achieves > 10% parallelism.*

Figure 21: **Routine Benchmark on Rascal (right two bars in each plot) vs. Baselines (left three bars).**

Current automation interfaces (e.g., Alexa [4], Google Home [16]) support limited causality expressiveness. Their IFTTT-style [26] rules struggle with conditional, multi-stage dependencies. Future work could integrate expressive, flow-based (and visual) systems like Node-RED [43] or Flogo [63].

9 Related Work

Push-based Frameworks. RASC shares semantic roots with ROS actionlib [51], gRPC streams [17], and MQTT [46], which provide goal-feedback-result lifecycles or push-based notifications. However, these frameworks typically require device-side protocol changes. RASC is minimal.

Scheduling and Action Dependencies. A range of systems provide abstractions for coordinating IoT and cyber-physical devices. Gaia [50], Bundle [68], dSpace [13], and DepSys [41] offer middleware or dependency-based programming, but at coarse granularity without action-level progress or dynamic scheduling. TransActuations [56], Rivulet [6], and IoTRepair [44] strengthen safety through transactional or rollback semantics, yet operate only at the level of instantaneous or fixed-length actions or routines. SafeHome [2] and Hades [5] handle routine conflicts but not fine-grained dependencies. In contrast, Rascal captures dependencies within as well as across routines, and our schedulers ensure safety and serial equivalence even as actions progress unpredictably.

Dynamic Rescheduling. Classical resource-reclaiming in real-time systems [18, 37, 57] captures slack from early tasks but assumes fixed DAGs and ignores late completions. Our adaptation of RV and STF extends these ideas to IoT action graphs with cross-routine serialization constraints, handling both early and delayed completions without violating safety.

Observability and Failure Detection. IoT systems often rely on coarse timeout-based detection or assume instantaneous actions [31], which is unsuitable for long-running tasks. SafeHome [2] and Hades [5] provide some exception handling but not progress-aware detection. Our adaptive polling provides fine-grained observability of start/complete events, enabling efficient and accurate failure detection without overloading the system. While pub/sub systems [1, 7, 9, 55] are often proposed as alternatives, they are best suited for settings with many subscribers across diverse topics. Rascal runs on a single central hub that subscribes to all topics, so pub/sub offers little benefit, and polling remains the core challenge it addresses. The histogram bucket problem [12, 29, 34, 53, 54] is analogous: both tasks require selecting partition ranges (bins vs. polling intervals) under uncertainty, differing only in the optimization criterion—e.g., minimizing error in density estimation versus balancing observability cost and timeliness.

Commercial Systems. Commercial platforms such as Alexa [4], Google Home [16], SmartThings [59], and Home Assistant [21] support limited forms of automation and do not guarantee conflict freedom or robust failure handling. RASC generalizes them for expressive DAG-based routines, with progress awareness and principled scheduling guarantees.

10 Conclusion

RASC provides an expressive RPC alternative for IoT device collections such as smart homes, smart buildings, etc. Our implementation, Rascal, supports *observability* by detecting action completion and failures quickly, and *programmability* via fine-grained causality dependencies, and assures safety and serial-equivalence. Rascal requires no modifications to existing devices. Trace-driven evaluation showed that Rascal detects completion within 2-13 RPCs and 2-16s over 90% of the time, even for actions lasting tens of minutes. Our routine scheduling outperforms state-of-the-art baselines by 10-55%.

Acknowledgments

We thank our shepherd Xiaowei Yang and the anonymous reviewers from this and previous submissions for their insightful feedback and guidance in shaping this paper.

This work was supported in part by NSF grant CNS 1908888, NSF grant CNS 2504595, a grant from the IBM-Illinois Discovery Accelerator Institute (IIDAI), and gifts from Microsoft and Capital One.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC*, pages 53–61, 1999.

[2] S. B. Ahsan, R. Yang, S. A. Noghabi, and I. Gupta. Home, SafeHome: Smart home reliability with visibility and atomicity. In *EuroSys*, 2021.

[3] K. Akpınar, T. Ballard, K. A. Hua, K. Li, S. Tarnpradab, and J. Ye. COMMIT: A multimedia collaboration system for future workplaces with the internet of things. In *MMSys*, 2017.

[4] Amazon Alexa. https://developer.amazon.com/alexa.

[5] E. Anceaume, G. Cabillic, P. Chevochot, and I. Puant. HADES: a middleware support for distributed safety-critical real-time applications. In *ICDCS*, pages 344–351, 1998.

[6] 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 *Middleware*, 2017.

[7] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS*, pages 262–272. IEEE, 1999.

[8] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM TOCS*, 2(1):39–59, 1984.

[9] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *TOCS*, 19(3):332–383, 2001.

[10] C. Cornelio and M. Diab. RECOVER: A neuro-symbolic framework for failure detection and recovery. In *IROS 2024*, 2024.

[11] Fraunhofer Center for Sustainable Energy Systems. Multifamily programmable thermostat data. https://catalog.data.gov/dataset/multifamily-programmable-thermostat-data, 2022.

[12] D. Freedman and P. Diaconis. On the histogram as a density estimator: L 2 theory. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 57(4):453–476, 1981.

[13] S. Fu and S. Ratnasamy. dSpace: Composable abstractions for smart spaces. In *SOSP*, pages 295–310, 2021.

[14] Google. gRPC: A high-performance, open source universal RPC framework. https://grpc.io/, 2025.

[15] Google. gRPC codebase. https://github.com/grpc/grpc, 2025.

[16] Google Home. https://store.google.com/us/product/google_home.

[17] gRPC Authors. gRPC Basics - Go: Server-side streaming RPC. https://grpc.io/docs/languages/go/basics/#server-side-streaming-rpc, 2026.

[18] I. Gupta, G. Manimaran, and C. S. R. Murthy. A new strategy for improving the effectiveness of resource reclaiming algorithms in multiprocessor real-time systems. *Journal of Parallel and Distributed Computing*, 60(1):113–133, 2000.

[19] HomeAssistantCore. https://github.com/home-assistant/core, 2024.

[20] Google Home. Google Home API: Supported starters, conditions, and actions. https://developers.home.google.com/automations/starters-conditions-and-actions.

[21] Home Assistant. https://www.home-assistant.io/.

[22] Home Assistant. Integrations. https://www.home-assistant.io/integrations/.

[23] Home Assistant: Classifying the internet of things. https://www.home-assistant.io/blog/2016/02/12/classifying-the-internet-of-things/, 2016.

[24] Home Assistant Community Forum User Vorta. Hue bridge rate-limits state change events to 1 per second per device. https://community.home-assistant.io/t/philips-hue-switch-delay/342266, 2022.

[25] HP Support. Fix hp print job stuck in queue. https://support.hp.com/us-en/help/printer/print-job-stuck.

[26] IFTTT - Automate business & home. https://ifttt.com/, 2010.

[27] Y. E. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. *ACM SIGMOD Record*, 24(2):233–244, 1995.

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

[29] Hosagrahar Visvesvaraya Jagadish, Nick Koudas, S Muthukrishnan, Viswanath Poosala, Kenneth C Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *VLDB*, volume 98, pages 24–27, 1998.

[30] L. V. Kantorovich. On the transfer of masses. *Doklady Akademii Nauk SSSR*, 37(7-8):227–229, 1942.

[31] M. Kashef, Y. Liu, K. Montgomery, and R. Candell. Wireless cyber-physical system performance evaluation through a graph database approach. *Journal of computing and information science in engineering*, 21(2):021009, 2021.

[32] C. Kelly. EE: Average UK smart home will have 50 connected devices by 2023. https://totaltele.com/ee-average-uk-smart-home-will-have-50-connected-devices-by-2023/, 2018.

[33] Jennifer Kent. The smart home in 2025: Outlook and opportunities. https://www.parksassociates.com/blogs/in-the-news/the-smart-home-in-2025-outlook-and-opportunities, January 2025.

[34] K. H. Knuth. Optimal data-based binning for histograms. *arXiv preprint physics/0605197*, 2006.

[35] G. Lampropoulos, Siakas K., and T. Anastasiadis. Internet of Things in the context of Industry 4.0: An overview. *IJEK*, 2019.

[36] J. Levin and T. A. Benson. Viperprobe: Rethinking microservice observability with eBPF. In *CloudNet*, pages 1–8. IEEE, 2020.

[37] G. Manimaran, C. S. R. Murthy, M. Vijay, and K. Ramamritham. New algorithms for resource reclaiming from precedence constrained tasks in multiprocessor real-time systems. *Journal of Parallel and Distributed Computing*, 44(2):123–132, 1997.

[38] O. Mart, C. Negru, F. Pop, and A. Castiglione. Observability in Kubernetes cluster: Automatic anomalies detection using Prometheus. In *HPCC/SmartCity/DSS*, pages 565–570. IEEE, 2020.

[39] Memoori. The Internet of Things in Smart Commercial Buildings 2022 to 2027. https://memoori.com/wp-content/uploads/2022/08/BIoT2022_Synopsis.pdf, 2022.

[40] Memoori. IoT devices in Smart Commercial Buildings 2025 to 2030. https://memoori.com/portfolio/iot-devices-smart-commercial-buildings-2025/, 2025.

[41] S. Munir and J. A. Stankovic. DepSys: Dependency aware integration of cyber-physical systems for smart homes. In *ICCPS*, 2014.

[42] T. Nelson, D. Yu, Y. Li, R. Fonseca, and S. Krishnamurthi. Simon: Scriptable interactive monitoring for SDNs. In *SIGCOMM*, pages 1–7, 2015.

[43] Node-RED. https://nodered.org/.

[44] M. Norris, Z. B. Celik, P. Venkatesh, S. Zhao, P. McDaniel, A. Sivasubramaniam, and G. Tan. IoTRepair: Systematically addressing device faults in commodity IoT. In *IoTDI*, pages 142–148, 2020.

[45] Nuki Developers Forum. Error “423 Locked” implies lock is busy and cannot accept another action. https://developer.nuki.io/t/error-423-locked-and-general-error-500/30457, 2024.

[46] OASIS Standard. MQTT Version 5.0. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, 2019.

[47] Protolabs. Cool idea award winner: SmartCan. https://www.protolabs.com/about-us/cool-idea-award/smartcan/.

[48] A. Quinn, J. Flinn, and M. Cafarella. You can’t debug what you can’t see: Expanding observability with the OmniTable. In *HotOS*, pages 163–169, 2019.

[49] Comment on "what do you use IFTTT for?". https://www.reddit.com/r/homeautomation/comments/li3shd/comment/gn13kg4/, 2021.

[50] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[51] ROS Wiki. actionlib. http://wiki.ros.org/actionlib, 2026.

[52] P. Santhoshini, K. H. Prabha, S.T. Jaibalaji, C.B. K. Vishwa, H. R. Kumar, and S. M. Sundar. Implementation of automatic lawn mower system for smart lawn maintenance. In *AISP*, pages 01–07. IEEE, 2024.

[53] J. D. Scargle, J. P. Norris, B. Jackson, and J. Chiang. Studies in astronomical time series analysis. VI. bayesian block representations. *The Astrophysical Journal*, 764(2):167, 2013.

[54] D. W. Scott. On optimal and data-based histograms. *Biometrika*, 66(3):605–610, 1979.

[55] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. *AUUG*, pages 243–255, 1997.

[56] A. Sengupta, T. Leesatapornwongsa, M. S. Ardekani, and C. A. Stuardo. Transactuations: Where transactions meet the physical world. In *ATC*, 2019.

[57] C. Shen, K. Ramamritham, and J. A. Stankovic. Resource reclaiming in multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):382–397, 1993.

[58] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. In *OSDI*, 2007.

[59] SmartThings. https://www.smartthings.com/.

[60] SmartThings open-source DeviceTypeHandlers and SmartApps code. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps.

[61] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, Upper Saddle River, NJ, 3rd edition, 2006. See Chapter 2, Section 2.4.2: Shortest Job First, pp. 82–83.

[62] The Apache Thrift Developers. Apache Thrift: A scalable cross-language RPC framework. https://github.com/apache/thrift, 2025.

[63] TIBCO Software Inc. Project Flogo: Open source event-driven ecosystem. https://www.flogo.io.

[64] K. M. Tsui, M. Desai, H. A. Yanco, and C. Uhlik. Exploring use cases for telepresence robots. In *HRI*, pages 11–18, 2011.

[65] L. Turchet, C. Fischione, G. Essl, D. Keller, and M. Barthet. Internet of Musical Things: Vision and challenges. *IEEE Access*, 6, 2018.

[66] L. N. Vaserstein. Markov processes over denumerable products of spaces, describing large systems of automata. *Problems of Information Transmission*, 5(3):47–52, 1969.

[67] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman. FarmBeats: An IoT platform for Data-Driven agriculture. In *NSDI*, 2017.

[68] P. A. Vicaire, E. Hoque, Z. Xie, and J. A. Stankovic. Bundle: A group based programming abstraction for cyber physical systems. In *ICCPS*, pages 32–41, 2010.

[69] Virtual devices for Home Assistant. https://github.com/twrecked/hass-virtual, 2023.

[70] Y. Zhang, P. Tokmakov, M. Hebert, and C. Schmid. A structured model for action detection. In *CVPR*, June 2019.

A Extended Analysis of the Adaptive Polling Algorithm

Theorem 1 (Adaptive Poll Placement with Fixed Budget k). Given a time distribution $p(t)$ on $(0,U]$, a poll budget k, and a terminal tolerance $\varepsilon > 0$, polls $0 < L_1^* < \cdots < L_{k-1}^* < L_k^*$ that minimize expected detection time Q satisfy $|L_k^* - U| \le \varepsilon$ are given by the following recurrent relation:

$$L_i^* = \begin{cases} \frac{1}{p(L_{i-1}^*)} \cdot \int_{L_{i-2}^*}^{L_{i-1}^*} p(t)\,dt + L_{i-1}^* & \text{for } i \in \{2,\dots,k\}, \\ \text{value } \in (0,U] & \text{for } i = 1 \end{cases} \quad (5)$$

Proof. Only the i-th and $(i+1)$-th terms of Q depend on L_i:

$$A_i = \int_{L_{i-1}}^{L_i} (L_i - t)\,p(t)\,dt, \qquad A_{i+1} = \int_{L_i}^{L_{i+1}} (L_{i+1} - t)\,p(t)\,dt.$$

By Leibniz's rule,

$$\frac{\partial A_i}{\partial L_i} = (L_i - L_i)p(L_i) + \int_{L_{i-1}}^{L_i} \frac{\partial}{\partial L_i}\big[(L_i - t)p(t)\big]\,dt = \int_{L_{i-1}}^{L_i} p(t)\,dt,$$

and

$$\frac{\partial A_{i+1}}{\partial L_i} = -\,(L_{i+1} - L_i)\,p(L_i).$$

The first-order optimality condition $\partial Q/\partial L_i = 0$ gives

$$\int_{L_{i-1}}^{L_i} p(t)\,dt \;-\; (L_{i+1} - L_i)\,p(L_i) \;=\; 0,$$

which rearranges to (5). Under $p > 0$ and continuity, these conditions determine a unique sequence once L_1 is fixed.

Fix $L_1 \in (0,U)$ and generate $L_2,\dots,L_k$ via (5). Then $L_k(L_1)$ is strictly increasing in L_1. Hence, there exists a unique $L_1^* \in (0,U)$ such that the generated sequence satisfies $L_k^* = U$. □

Theorem 2 (Meeting a Detection Tolerance SLO). Given a detection window $Q_w \in (0,U]$, an SLO slo $\in (0,1]$, and a placement $\mathcal{L} = \{L_1 < \cdots < L_k = U\}$, its covered set is

$$C(\mathcal{L}) = \bigcup_{i=1}^{k} \big((L_i - Q_w)^+, L_i\big] \cap (0,U], \quad (x)^+ = \max\{x, 0\},$$

and its coverage is

$$\mathrm{Cover}(\mathcal{L}) = \int_{C(\mathcal{L})} p(t)\,dt.$$

Let $A_k(Q_w)$ denote the best achievable coverage with k polls,

$$A_k(Q_w) = \max_{\mathcal{L}} \mathrm{Cover}(\mathcal{L}),$$
$$k^*(\mathrm{slo}, Q_w) = \min\{\, k \in \mathbb{N} : A_k(Q_w) \geq \mathrm{slo} \,\}.$$

Assume a binary search over k is run with initial bracket `left` $= 0$, `right` $= \lceil U/Q_w \rceil$, using an oracle `examineQw` that returns `true` exactly when there exists a placement $\mathcal{L}$ with $\mathrm{Cover}(\mathcal{L}) \geq \mathrm{slo}$. Then the search returns $k^*(\mathrm{slo}, Q_w)$ together with an SLO-feasible placement.

Proof. **Monotonicity.** If $k' < k$, any k'-poll placement can be extended to k polls by inserting additional polls; this cannot reduce $C(\mathcal{L})$, hence $A_k(Q_w) \geq A_{k'}(Q_w)$. The oracle predicate is therefore nondecreasing in k.
Bracketing. For $k = \lceil U/Q_w \rceil$, equal spacing $L_i = iU/k$ yields $\bigcup_i ((L_i - Q_w)^+, L_i] \supseteq (0, U]$, so $\mathrm{Cover}(\mathcal{L}) = 1 \geq \mathrm{slo}$. Thus the upper bracket is feasible, while $k = 0$ is not.
Minimality via binary search. Binary search on a nondecreasing predicate with a feasible upper bracket returns the smallest k for which the predicate holds; by definition, this is $k^*(\mathrm{slo}, Q_w)$. $\square$