APEX: Automatic Precondition Execution with Isolation and Atomicity in Internet-of-Things

Qian Zhou
Department of Electrical and Computer Engineering
Stony Brook University
qian.zhou@stonybrook.edu

Fan Ye
Department of Electrical and Computer Engineering
Stony Brook University
fan.ye@stonybrook.edu

ABSTRACT

In Internet-of-Things there are situations where before users can execute commands on IoT devices, certain conditions must be met for sake of safety, correctness or efficiency. Thus, a series of other commands need to precede the user commands in a correct order to make those conditions true. Users have to consciously follow the order and manually send those commands one by one, which is error prone and inconvenient. We propose APEX, a system automatically deducing, satisfying all the preconditions of the user commands. It has two strategies. Evaluation on a 20-node testbed proves that our conservative strategy sustains high execution success rates despite resource contention, while in realistic enterprise environments, our aggressive strategy may execute significantly faster, saving up to 7s and reducing about 46% of conservative strategy's time cost.

CCS CONCEPTS

• **Networks** → **Application layer protocols**; **Cyber-physical networks**; • **Computer systems organization** → *Reliability*.

KEYWORDS

Internet of Things, building automation, isolation, atomicity

ACM Reference Format:
Qian Zhou and Fan Ye. 2019. APEX: Automatic Precondition Execution with Isolation and Atomicity in Internet-of-Things. In *International Conference on Internet-of-Things Design and Implementation (IoTDI '19), April 15–18, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3302505.3310066

1 INTRODUCTION

In Internet of Things there is a common constraint that before a command can execute on an IoT device, certain conditions (on possibly other devices) must be satisfied. Such a condition is called a *precondition*, and it exists for reasons including but not limited to: **1) safety.** E.g., before a fire sprinkler sprinkles water, the outlets within its spraying range should be powered down to prevent electric shocks; also, before a fireplace is turned on, the presence and condition of a smoke detector in the same room should be checked to ensure fire monitoring; **2) correctness.** E.g., a home theater should be wakened before being told to play a movie; or to store water in a kitchen sink, the sink valve needs to be closed before the tap is turned on; **3) efficiency.** E.g., an air conditioner may be set to work after all the doors and windows in the room are closed, for efficient cooling.

Unfortunately, there is no formal effort to address this automatic precondition execution problem in IoT. Users have to manually track and execute the commands making those preconditions true, which is a great burden and error prone. It leads to inconvenience and safety risks or inefficiency.

Precondition execution faces several challenges. First, a command can have multiple preconditions in multiple levels. E.g., before a sprinkler sprays, the alarm should be sounded besides of outlet power cut; and before the power cut, all the computers powered by the outlets should save their data and safely shut down.

Second, IoT is a multi-user environment where interleaving of commands from different users can happen, leading to faulty consequences. E.g., a user's commands closed a window and are about to turn on the air conditioner when others' commands arrive, opening the window thus destroying the former's precondition.

Third, execution failures are common in IoT, especially enterprise IoT, due to message losses, resource contention, actuator malfunctions, etc. If any execution failure occurs, causing partial execution, it is left to the user to cumbersomely undo the commands that have executed, to roll the system back to its initial state.

There are home automation solutions such as Logitech Harmony Smart Control [17], Apple Home [1] and IFTTT [14]. The first two allow a user to make her smartphone a universal remote control, and use one tap to trigger a preconfigured activity that operates one or multiple devices. IFTTT creates chains of simple conditional statements such that one action triggers another. These solutions solve problems which differ fundamentally from the automatic precondition execution one in IoT. i) They either simultaneously trigger multiple executions which are independent and thus can be done in parallel, or use an action as the *beginning* to trigger subsequent *post-actions*. Precondition execution is different—a user command is the *end*: it causes *pre-actions* to be performed to make its execution environment prepared, and itself is the last to execute. ii) They are mostly handling small amounts of home devices (e.g., up to 8 in Logitech) or web/app based services owned by one account, and they have no concern about multi-user contention which however is common in enterprise-scale IoT. iii) They do not consider recovery from partial execution. More details about existing work are presented in Section 10.

In this paper we propose **APEX**, an **A**utomatic **P**recondition **EX**ecution system which recursively discovers all the preconditions of a command issued by the user, and automatically generates,

IoTDI '19, April 15–18, 2019, Montreal, QC, Canada

ACM ISBN 978-1-4503-6283-2/19/04…$15.00
https://doi.org/10.1145/3302505.3310066

executes commands to make the preconditions satisfied. Our solution borrows concepts and principles from database transactions. However, the context of large scale IoT (wireless distributed system, frequent expensive actuations, etc.) gives rise to new challenges not existing in database environments, and transaction techniques cannot be trivially applied as is. To the best of our knowledge, we are the first to define this automatic precondition execution problem, and propose a solution applying isolation and atomicity properties [9, 10] of database transactions to solve it. In this solution, we identify new issues in IoT contexts, and adapt/extend database techniques to address them. Specifically,

- We formally define the problem of precondition execution in large scale IoT systems, and describe how preconditions may be set by vendors/users/admin.
- A mechanism of precondition execution piece/graph is proposed to generically represent preconditions, and a solution is devised to acquire the graph for running a user command.
- We design two execution strategies (conservative and aggressive) with mechanisms for deadlock prevention [6, 12], operation isolation and atomicity suitable for IoT contexts.
- We implement the two strategies and conduct experiments on a 20-node testbed in realistic enterprise environments. We find that conservative strategy sustains high execution success rates despite resource contention, while aggressive strategy executes faster, saving up to 7s and reducing about 46% of conservative strategy's time cost.

2 MODELS AND ASSUMPTIONS

Node Types. IoT nodes fall into three types: the backend, subject devices, objects. The backend is a cloud server at which a new subject/object gets registered for joining the system. It maintains the attribute profiles of registered nodes. It also stores, updates access rights of what functions a subject can invoke on an object. A subject is a user who uses a subject device (e.g., smartphone) to operate objects, i.e., IoT devices or "Things". Subject devices and objects constitute a *ground network*, where we assume network connectivity exists among the nodes and multi-hop routing is available.

Condition. Each object owns some state variables, e.g., an air conditioner has its target temperature, a window has its opening degree. A *condition* is a predicate on a state variable, denoted as a 4-tuple $\langle obj : variable\ opr\ value \rangle$. *obj* is an object's unique identifier; operator *opr* may be but not limited to $=, >, \leq, \in$ and their negation operators. Some condition examples are: $\langle air\_conditoner\_1 : target\_temp < 70 \rangle$; $\langle window\_2 : status = \text{`}closed\text{'} \rangle$.

Command. A (write) command alters a variable after reaching the object and being executed, making the corresponding condition met. E.g., command $[window\_2 : open()]$ sets condition $\langle window\_2 : status = \text{`}open\text{'} \rangle$ true. We assume a conversion mechanism (e.g., by enforcing certain naming conventions [24]) exists to map a command to the corresponding condition and vice versa. Also, note that some IoT commands write variables while others only read (e.g., get the current temperature). Read commands change no condition or precondition, and are less relevant to APEX system. *Thus, in this paper most commands are* ***write commands****. And in reality, the IoT objects handling write commands usually have actuators (e.g., doors have mechanical actuators, air conditioners have mechanical and thermal ones). They are mostly wall-powered and assumed to have sufficient energy for operations.*

Precondition. *If condition $\mathcal{A}$ must be true before a command can execute to set condition $\mathcal{B}$ true, then $\mathcal{A}$ is $\mathcal{B}$'s precondition.* We denote this relationship as $\mathcal{A} \Rightarrow \mathcal{B}$, e.g., $\langle outlet : status = \text{`}off\text{'} \rangle \Rightarrow \langle sprinkler : status = \text{`}spray\text{'} \rangle$ means the sprinkler should not sprinkle water before the outlet is off. There is a transitive law: *If $\mathcal{A}$ is $\mathcal{B}$'s precondition and $\mathcal{B}$ is C's precondition, then $\mathcal{A}$ is also C's precondition.* We assume the backend stores, updates precondition rules (like access rights), and a mechanism is available to generate preconditions for a condition based on those rules.

A condition may have zero, one or more preconditions. When there are multiple, they can be combined in logic AND, OR. Particularly, AND—all preconditions should be met—is the simplest but suffices most cases in reality. Our design supports it. OR can be translated into AND using De Morgan's laws [13].

A precondition itself is a condition, thus may have its own preconditions, leading to multi-level, recursive preconditions. We call a precondition generated based on a precondition rule *direct precondition* while one deduced from the transitive law indirect.

A command which makes a precondition true is a *precondition command*. Due to recursive preconditions, a series of precondition commands may need to be deduced, executed before executing the command issued by the subject. A subject command together with *all* of its precondition commands form a *combo*.

Consistency. Different from database transactions, "consistency" in our context is redefined as: if initially no precondition rule is violated in the ground network, after a combo executes, there should still be no violation. E.g., assume there is a precondition rule corresponding to $\mathcal{A} \Rightarrow \mathcal{B}$, then the coexistence of $\{!\mathcal{A}, \mathcal{B}\}$ is inconsistent (! for logic NOT), while $\{!\mathcal{A}, !\mathcal{B}\}$, $\{\mathcal{A}, !\mathcal{B}\}$, $\{\mathcal{A}, \mathcal{B}\}$ are consistent.

2.1 Properties of Internet-of-Things

We notice that a combo is similar to a database transaction to some extent, with a command in the former similar to a statement in the latter. This makes it natural to explore the adaptation of existing transaction techniques (e.g., ACID properties [10]) to IoT precondition execution. However, large scale IoT environments differ fundamentally from the database context:

1) Wireless Distributed System. Unlike a database system, the variables in IoT are distributed among objects. Combo execution needs multiple scattered objects to receive wireless messages. In this context: i) Message losses are common, and transmission latency is not negligible; ii) In database systems a locking-based strategy [3] is commonly used for concurrency-control, but here a centralized lock manager would incur long latency.

2) Frequent Expensive Actuations. After a variable is modified, the corresponding actuator brings the new value into effect, and this process is an actuation. E.g., a window's opening degree variable was 0 and now set to 5, then its motor spins till the degree gets to 5. IoT is quite different from database systems due to actuations: i) IoT actuations frequently incur expensive (in terms of time, energy cost, etc.) physical (e.g., open/close the window) or electronic (e.g., turn on/off the AC) operations. Database transactions may lead to observable external writes (e.g., a printer prints transaction records, or an ATM dispenses cash), which share some

similarities with IoT actuations but usually have much less expense and frequency; mostly merely inexpensive internal reads/writes are involved. ii) Actuation failures can be common in IoT, e.g., a stuck pulley stops the window reaching the desired opening degree.

3) Lower Requirement on Throughput despite Concurrent Commands. Database pursues both isolation and high throughput, but IoT needs only much lower throughput: 1) It is impossible and makes no sense to let objects with slow actuations (e.g., doors) react to multiple users' different commands within a very short time; 2) Even for a lamp (with quicker actuations in changing its brightness), it does not need to take, say, > 10 commands per second.

2.2 Failure Causes

Execution Failures. A command execution may fail mainly because of: **1) transmission/object failures.** Due to message losses, a command cannot arrive at the object; or the object encounters hardware or software malfunctions/bugs stopping it taking the command. In either case the variable is unaltered. **2) resource contention.** In this paper "variable" and "resource" are used interchangeably. Variables are read or written by commands. In IoT, multiple commands from different subjects may attempt writing the same variable simultaneously, then contention occurs. **3) actuation failures.** As mentioned, for most IoT write commands, execution is not over when the variable is altered. An actuation (by mechanical/electrical/thermal/optical components) bringing the altered variable into effect is needed, otherwise a failure may occur.

System Crashes, Disk Failures. Some hardware or software malfunctions/bugs cause loss of the content in main memory; a disk may also lose its content. They are out of this paper's scope.

3 DESIGN GOALS

Precondition Auto-Completion. When a subject issues a command, the system should automatically find out all its preconditions recursively, deduce the precondition commands, and execute those commands in the correct order before executing the subject one.

Isolation. Isolation ensures consistency despite concurrent commands from different subjects targeting the same variable. Specifically, two combos should be prevented from interleaving such that they do not destroy each other's preconditions. An example of interleaving is: a user's commands closed a window and are about to turn on the air conditioner when others' commands arrive, opening the window thus destroying the former's precondition.

Atomicity. Atomicity means combo execution appears atomic that either all its commands execute successfully or none does. It prevents the system from ending up with intermediate states or partial execution. A mechanism is needed to roll the system back to its initial state upon an execution failure. Note that rollback may not always be performed; we offer the solution and it can be partially, fully enabled or disabled based on the needs of the scenarios.

Robustness. Message losses should be detected and remedied. Execution failures (Section 2.2) due to transmission, object, contention or actuation failures should be announced to the backend and the subject, such that rollback may be conducted.

Non-Goals. 1) A full solution of command-to-condition conversion (and vice versa), precondition rule making and precondition generation according to the rules is out of the scope. 2) The system enforces commands to execute in the correct order and tries to achieve low latency, but there is no hard time guarantee. 3) Recovery from a system crash or disk failure, and achieving a property similar to "durability" in ACID are out of the scope. 4) We do not consider security attacks or privacy issues in this paper.

4 SYSTEM OVERVIEW

There are three steps in the system as follows:

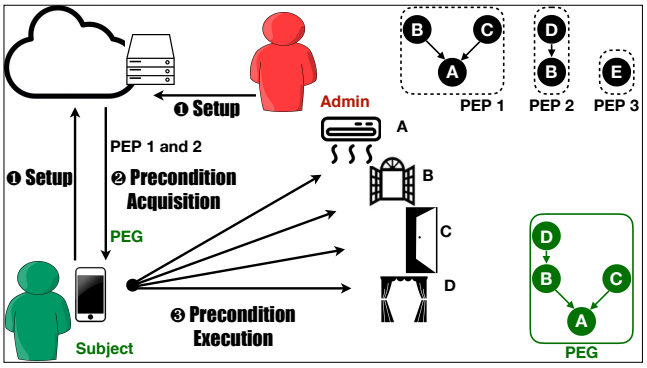


Figure 1: Vendor/subject/admin configures precondition rules, based on which PEP 1, 2, 3 are generated on the backend. The subject asks for her token to invoke a function on A, and receives from the backend the PEG generated from PEP 1 and PEP 2. When operating A, her device automatically deduces, executes precondition commands on D, C, B first according to the order specified by the PEG.

1) Precondition Setup. Precondition rules are offered by object vendors/subjects/admin and imported into the backend at registration. A condition's direct preconditions are generated based on the rules, represented as precondition execution pieces (PEP).

2) Precondition Acquisition. When a subject asks the backend for a token [30] of invoking a command, the relevant PEPs are assembled to a precondition execution graph (PEG) where a vertex is a precondition of the subject command and an edge specifies the order. The PEG is sent to the subject. When she issues a command, the precondition commands are deduced, forming a combo.

3) Precondition Execution. The subject device monitors and operates the relevant objects, ensuring that precondition commands execute in the order described by the PEG, and it executes the subject command when all the precondition commands are done.

Precondition Examples. *PEG depth* is defined as the number of vertices in its longest path. We show several real PEGs with various depths in Fig. 2. E.g., the depth 5 PEG means before a sprinkler sprays water, the alarm should be turned on and the outlets (in the same room) should be turned off, before which all the computers (powered by the outlets) should be turned off, before which the stable storages should be checked to ensure that the computers' important data are archived. The depth 3 PEG is the one in Fig. 1.

5 PRECONDITION SETUP

Here we describe how we think preconditions can be configured, while a full set of precondition rule syntax is left to future work.

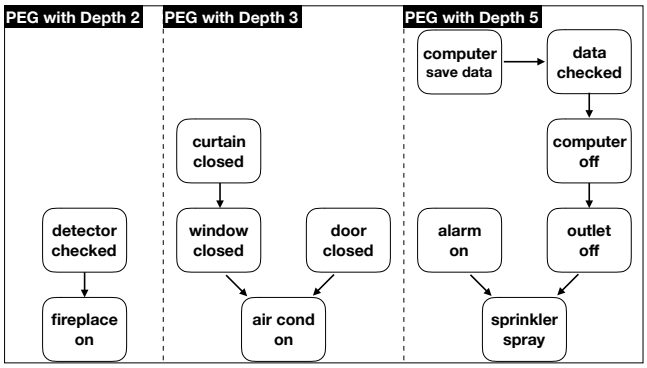


Figure 2: Examples of PEGs with different depths.

Precondition rules are made by IoT object vendors, subjects, admin, based on which the backend generates direct preconditions for a condition. Vendors do that to ensure their products are used in a safe or correct way, and the rules come with the products, and get imported into the backend at object registration. Subjects may set rules for personalized needs. The admin usually sets rules based on vendor rules with additional customized adjustments.

5.1 Ease of Rule Setup

Precondition setup can be challenging, so we use the following techniques to make it easy to conduct.

First, a condition can have multiple levels of preconditions and multiple preconditions per level. It is hard for a rule setter to handle all of them correctly. To solve this problem, we require each rule only to specify a condition's **direct preconditions**. E.g., in Fig. 2, the sprinkler vendor only requests that the outlets should be powered down before the sprinkler sprays water, and it is others' duty (e.g., subjects') to request the computers to safely shut down before the outlets are off. In this way, every rule setter only cares about the preconditions in the immediate previous step, even though multiple recursive preconditions exist.

Second, vendors face extra challenges because they do not have priori knowledge about which exact objects will be installed around theirs, thus they cannot directly refer to the exact IDs of those objects in requirements. E.g., a sprinkler vendor does not know the IDs of the outlets to be installed around her products. We use **attribute-based rules** using predicates to specify a category of objects sharing certain characteristics, regardless of the number of objects and their concrete IDs. E.g., $\langle type = \text{`}window\text{'} \wedge Room = X : status = \text{`}closed\text{'}\rangle \Rightarrow \langle type = \text{`}air\_cond\text{'} \wedge Room = X : status = \text{`}on\text{'}\rangle$, i.e., close the window(s) in Room X before turning on the AC in Room X. Later the backend will find out which exact objects match the predicates (because objects' profiles containing device types and installation sites were registered).

5.2 Precondition Execution Piece

A precondition execution piece (PEP) states all the direct preconditions of one condition. It is expressed as $\{prec\} \Rightarrow cond$, with *cond* specifying the condition and $\{prec\}$ a set of preconditions combined in AND. When a subject requests a token [29] to invoke certain functions, the backend will:

1) Deduce Possible Subject Conditions. Subject condition, i.e. the condition that will be set true by subject command, is obtained first. E.g., command $[air\_cond : set\_temp(75)]$ has condition $\langle air\_cond : target\_temp = 75\rangle$. A token may allow a subject to invoke a function with parameters in a range, then the possible subject condition is a range, e.g., $\langle air\_cond : target\_temp > 70 \wedge target\_temp < 80\rangle$.

2) Find Relevant PEPs. The backend searches PEPs for the subject condition, i.e., those whose *cond* have intersection with the subject one (e.g., the PEP with *cond* $\langle air\_cond : target\_temp > 75\rangle$ is relevant to the last subject condition example in Step 1). Next, it searches PEPs for each *prec* of the PEPs found, and does this recursively till all the relevant ones are found out.

Example. In Fig. 1, the subject requests a token for command $[air\_cond : set\_status(on)]$, the backend deduces its condition $\langle air\_cond : status = \text{`}on\text{'}\rangle$, which is PEP 1's *cond*. PEP 1 has $\{prec\} = \{\langle window : status = \text{`}closed\text{'}\rangle, \langle door : status = \text{`}closed\text{'}\rangle\}$. PEP 1's first *prec* is PEP 2's *cond*, so PEP 2 is also relevant. PEP 2 has $\{prec\} = \{\langle curtain : status = \text{`}closed\text{'}\rangle\}$. *curtain* and *door* have no preconditions. Totally, PEP 1 and PEP 2 are found relevant.

6 PRECONDITION ACQUISITION

The backend assembles the PEPs to a precondition execution graph (PEG) and sends it to the subject. When the subject issues the command, the PEG's precondition commands are deduced.

Algorithm 1 PEPs to PEG

```
1:  V ← {subject_condition}
2:  E ← ∅
3:  queue ← v
4:  while queue ≠ ∅ do
5:    cond ← queue.pop()
6:    prec_set ← cond.get_direct_precs_from_PEPs()
7:    for each prec in prec_set do
8:      if prec ∉ V then
9:        V.insert(prec)
10:       E.insert(ID_prec, ID_cond)
11:       queue.push(prec)
12:     else if ∃v ∈ V and is compatible with prec then
13:       v ← v ∩ prec
14:       E.insert(ID_v, ID_cond)
15:     else if ∃v ∈ V and is incompatible with prec then
16:       print error
17:       return {∅, ∅}
18:     end if
19:   end for
20: end while
21: return {V, E}
```

6.1 Generate PEG

A PEG is represented by a Directed Acyclic Graph $\{V, E\}$ (for vertex and edge set respectively) in which a vertex is a condition and an edge specifies the precondition relationship: $(\mathcal{A}, \mathcal{B})$ in E is an edge

from $\mathcal{A}$ to $\mathcal{B}$, which means $\mathcal{A} \Rightarrow \mathcal{B}$. Notice that the destination vertex is the subject condition and the others are preconditions.

We give a simple algorithm which incrementally builds a PEG. There are three possible cases when handling a precondition *prec*: 1) *prec* is new to V, which means its $obj : variable$ is different from that of any condition in V, thus *prec* can be simply added to V. 2) *prec* is compatible with an existing condition v in V, this happens when *prec* and v have the same $obj : variable$ and their predicates have intersection. E.g., for *prec* $\langle air\_cond : target\_temp < 75\rangle$ and v $\langle air\_cond : target\_temp < 80\rangle$, $prec \cap v = prec \neq \emptyset$. In this case no vertex is added to V but v is updated to $prec \cap v$. 3) *prec* is incompatible with an existing condition v in V, this happens when *prec* and v have the same $obj : variable$ but their predicates have no intersection. In this case *prec* and v cannot be met simultaneously, and an error for conflicting precondition rules is reported.

The algorithm ensures no two conditions in a PEG have the same $(obj : variable)$, thus no variable will be accessed more than once by a combo. By eliminating this redundancy, message overhead and latency in later execution can be reduced. In the example in Fig. 1, PEP 1 and PEP 2 are fed into the algorithm and the PEG is obtained.

6.2 Deduce Subject Condition & Precondition Commands

When the subject issues a command, her device uses the same command-to-condition mechanism as the backend does to convert the command to the subject condition, and based on that which PEG to use becomes known.

Now the PEG is known, the subject device converts each precondition in it to a precondition command. Those commands together with the subject command form the combo. E.g., condition $\langle air\_cond : target\_temp = 65\rangle$ is converted to command $[air\_cond : set\_target\_temp(65)]$. One covering a set (e.g., $\in$) or an interval (e.g., $>$) usually has multiple possible precondition commands. E.g., $\langle air\_cond : target\_temp \geq 70\rangle$ can be met by executing $[air\_cond : set\_target\_temp(x)]$ for any $x \geq 70$. Which x to use belongs to policy making by the admin and is out of the scope.

Example. In Fig. 1, the precondition commands are: D: $[curtain : close()]$; C: $[door : close()]$; B: $[window : close()]$.

7 PRECONDITION EXECUTION

In this step the subject device acts as a *pivot* to monitor and control all the relevant objects, ensuring precondition commands execute in the PEG order. We adopt two-phase locking (2PL) for isolation of interleaving but make its lock management distributed for fitting IoT contexts. Also, our system remedies message losses, and can detect execution failures listed in Section 2.2. Besides, it can recover the system to the initial state when an execution failure occurs.

We propose two execution strategies preferred in different situations: *conservative strategy* reserves all the variables in a combo before altering any of them, and has higher execution success rates and lower rollback cost; *aggressive strategy* starts execution on some variables before others are reserved, and runs faster.

7.1 Conservative Strategy

Conservative strategy enforces a rule: *once a combo starts executing any command, it will not encounter a contention failure.* This is for achieving high execution success rates and low rollback cost despite contention. We adopt conservative strict 2PL: the pivot reserves all the variables in the combo first, and then sends out commands to execute, and releases all the variables at once when all the commands execute successfully. During reservation, preemption [12] is used to prevent deadlocks: the pivot starting later gives way to the one starting earlier.

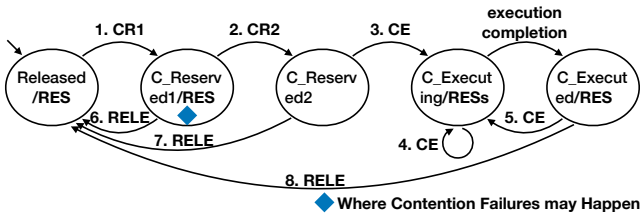


Figure 3: Conservative strategy's Moore state machine for an object. The input for transition 1–8 is the reception of a control message (CR1, CR2, CE or RELE) from the pivot, and the output is sending a response (RES) back to the pivot.

In Fig. 3, message 1, 2, 3 are for reservation and execution; 4, 5 rollback (Section 7.4); 6, 7, 8 release (Section 7.5). We introduce reservation and execution in this section.

$$S \rightarrow O : ID_{MSG}, ID_S, ID_O, Var, Ctl, T$$

$$O \rightarrow S : ID_{MSG}, Code, Val_{init}, Val_{curr}$$

Figure 4: Subject sends a control message to object and receives a response.

Message Formats. In Fig. 4, Subject S (the pivot) sends a control message to Object O, including: 1) ID_{MSG}, ID_S, ID_O: message ID, subject ID, object ID; 2) Var: the variable to operate; 3) Ctl: the control type, it can be CR1, CR2, CE or RELE; 4) T: a timestamp of the pivot's start moment—when it sends the first message for the combo. O sends a response (RES) back, where *Code* is a status code telling execution success or failure/cause. Val_{init} is Var's initial value before being modified, Val_{curr} is its current value.

Procedures (when no contention occurs):

1) *Reservation Stage.* (Fig. 5) Initially the pivot S is in this stage, and sends Conservative Reservation 1 (CR1) messages to all variables in the PEG to reserve them. After receiving a CR1, a variable $(ID_O : Var)$ in state **Released** (unlocked) updates its state to **C_Reserved1** , its holder ID to ID_S, its holder timestamp to T (see symbols in Fig. 4), and sends a CR1_RES back. The pivot waits till all the RESs arrive, and if each of them announces a success, it moves on to Conservative Execution Stage.

2) *Conservative Execution Stage.* (Fig. 5) In this stage the pivot executes commands on variables according to the PEG order. It sends Conservative Execution (CE) messages along with commands to the variables *valid* for execution, triggering execution on them. The variables valid for execution have either no preconditions or preconditions which are all true, and originally they are the former. E.g., in Fig. 1, Variable D and C are valid and receive CEs first.

Also, the pivot sends Conservative Reservation 2 (CR2) messages to all the variables *invalid* for execution (i.e. those whose preconditions are not yet true), informing them that it is in Conservative Execution Stage such that they can reject reservation requests from any other pivot while waiting for CEs. This is for meeting the rule of conservative strategy that once a combo starts executing any command, it will not lose in contention (Section 7.1.1).

Specifically (Fig. 3), a CR2 makes a variable in state **C_Reserved1** switch to **C_Reserved2** if the CR2's ID_S equals the variable's holder ID; a CE makes a variable in state **C_Reserved2** switch to **C_Executing** if the CE's ID_S equals its holder ID, and start execution. Note that we require all variables to switch in order **C_Reserved1** → **C_Reserved2** → **C_Executing** for a consistent style, though variables like D and C are valid from the very beginning and do not really need CR2s. For such cases we merge CR2 into CE (i.e., CR2+CE, Fig. 5), using one message to switch a variable to **C_Reserved2** and then immediately to **C_Executing** .

When execution is done, **C_Executing** goes to **C_Executed** . A CE_RES announcing execution completion is back. The pivot checks if this execution completion makes any variable which was invalid for execution now valid. If so, it sends a CR2+CE to the variable (CR2 is with CE in case that the last CR2 is lost or delayed). E.g., in Fig. 1, when the RES of Variable D is back, B becomes valid for execution. The pivot repeats this process until all variables in the PEG finish execution, then the combo execution is done.

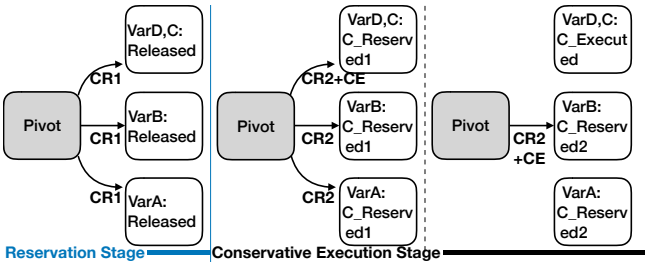


Figure 5: The pivot reserves the variables and executes commands according to the PEG in Fig. 1, using conservative strategy. The diagram of execution on VarA is omitted here.

Example. In Fig. 1 and Fig. 5, the pivot sends CR1s to reserve the 4 variables (A, B, C, D), after which is done it starts execution. D: [$curtain : close()$] and C: [$door : close()$] have no precondition commands, and are executed first; when D is finished, B: [$window : close()$] executes; after B and C are both completed, A: [$air\_cond : set\_status(on)$] executes.

7.1.1 Distributed Lock Management. Two pivots may try accessing the same variable and encounter contention. In database systems a centralized lock manager is commonly used to decide which transaction can access a variable. Even in IoT, it is still possible to use a centralized manager, e.g., the backend, but that incurs significantly worse availability and longer latency (see IoT Property 1 in Section 2.1). Here we propose a distributed lock management mechanism making pivots and objects both participate in management, achieving better availability and responsiveness.

Initially a variable is in state **Released** and unlocked, and it will be locked by any pivot requesting to reserve it, then the pivot becomes its holder. When a locked variable receives a reservation message (called *challenge message*, CR1 or AR (AR will be introduced in Section 7.2)) from a pivot which is not its holder (called *challenger*), contention begins. Arbitration is performed to make one pivot (between its holder and the challenger) win the contention and the other lose. It is a preemption process for preventing deadlocks. The arbitration principle is: *1) If neither pivot is in Conservative Execution Stage, the one which started earlier wins (based on a wound-wait timestamp-based preemption scheme); 2) Otherwise the one in Conservative Execution Stage wins (due to conservative strategy's rule).* Specifically,

1) If the variable is in **C_Reserved1** (when it is unsure whether its holder has entered Conservative Execution Stage): i) A challenge message with a timestamp newer than the holder's is rejected by the variable's object, a failure RES is sent to the challenger. The challenger loses and will wait or abort; ii) A challenge message with a timestamp older is forwarded to the holder, and the holder: a) loses if it is in Reservation Stage, will abort; b) wins if in Conservative Execution Stage, and a failure RES is sent to the challenger.

2) If the variable is in **C_Reserved2** or **C_Executing/ed** (at this time it is sure that its holder is in Conservative Execution Stage), a challenge message is always rejected by the variable's object, and a failure RES is sent to the challenger.

We justify the three mechanisms used in our distributed lock management method as follows:

1) Objects as Semi-Autonomous Managers. In our system a variable's holder (pivot) and owner (object) both arbitrate when facing contention. A natural way is to let two pivots negotiate who should give up the variable. However, in IoT the variable's object first receives challenge messages. Of course it can always relay the messages to the holder for arbitration, but that would incur additional traffic, message losses and latency. To reduce traffic and improve responsiveness, we make objects assist pivots in contention arbitration, as introduced above.

2) Conservative Strict 2PL. Conservative strategy adopts the 2PL variant which is conservative (C2PL) and strict (S2PL) [4, 21]. C2PL, when used in database systems, ensures that a transaction will not block waiting for other resources once it starts execution. But it is not frequently used due to its requirement on global knowledge of all resources needed at the beginning and its lower concurrency. In our context, however, it can be a good choice. 1) We have PEG, which is the global knowledge. 2) In many cases IoT allows for lower throughput due to its slow actuations (see IoT Property 3 in Section 2.1). Because of the long actuation time, conservative 2PL's extra latency (compared with non-conservative 2PL) becomes less a problem. 3) C2PL helps reduce the expensive rollback of actuations (see IoT Property 2): a pivot will not start execution/actuation before it holds all the variables needed, and will not lose in contention during execution, thus has a smaller chance to abort and roll back the actuations performed. Considering that IoT actuations are usually expensive (time-consuming, energy intensive, etc.), this conservative strategy is especially meaningful.

In certain cases conservative 2PL's longer latency may become significant, deteriorating the system responsiveness. E.g., in enterprise environments the network can be very congested during office hours, which makes C2PL take unacceptably long (Section 9.4). A non-conservative strategy may become preferred then (Section 7.2).

S2PL is widely used in databases because it prevents cascading abort and rollback [3]. In our context it is additionally important for isolation of interleaving and easier rollback: 1) If a pivot releases variables gradually before all commands in the combo are finished, other pivots will get a chance to access the released precondition variables, breaking the preconditions while it is still executing. S2PL stops this. 2) S2PL does not release a variable until the end, thus not need to re-reserve the variables when rollback is needed.

3) Preemption for Deadlock Prevention. Deadlocks may occur in Reservation Stage, when multiple pivots hold the resources others are reserving and reserve the resources others are holding.

Deadlocks can happen only if the four conditions are met [6, 12]: 1) mutual-exclusion 2) hold-and-wait 3) no-preemption 4) circular wait. We break the third condition in arbitration. Breaking any of the other three is less appealing. 1) Breaking "mutual-exclusion" condition is impossible in our context because a variable cannot be set to two values simultaneously. 2) Breaking "hold-and-wait" condition is possible: a pivot in Reservation Stage releases the variables it holds if there is any that it cannot get, and retries reserving all variables later. It prevents deadlocks, but the system may encounter starvation [25] that pivots keep getting partial resources and releasing them, making no progress. 3) Breaking "circular wait" condition also works. One way is to give each variable a global identity, and a pivot is required to reserve variables in the order of their identities. E.g., a pivot cannot reserve a larger ID variable if it fails in reserving a smaller ID one, thus it will not hold the variables wanted by the pivot beating it. This, however, requires a pivot to grab variables one by one. Our strategy sends out all CR1s at once, without blocking at any single RES, saving significant time.

7.2 Aggressive Strategy

Aggressive strategy adopts non-conservative strict 2PL: it does not wait for all the variables to be reserved before starting execution on some, hopefully one which is now locked by others will later be available when it is the time to use it. It has no reservation stage as in conservative strategy but has a similar execution stage. It has shorter latency but a higher chance of abort and rollback.

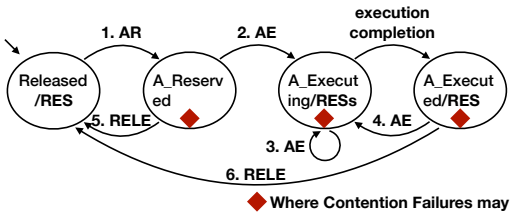


Figure 6: Aggressive strategy's Moore state machine for an object. The input for transition 1–6 is the reception of a control message (AR, AE or RELE) from the pivot, and the output is sending a response (RES) back to the pivot.

In Fig. 6, message 1, 2 are for reservation and execution; 3, 4 for rollback (Section 7.4); 5, 6 for release (Section 7.5). Aggressive and conservative strategies share the control message and response formats (Fig. 4), except that the *Ctl* here is AR, AE or RELE.

Procedures (when no contention occurs):

Aggressive Execution Stage. (Fig. 7) This is the only stage in aggressive strategy. The pivot S sends Aggressive Reservation (AR) and Aggressive Execution (AE) messages along with the commands to the variables valid for execution, reserving them and triggering execution on them. Also, it sends ARs to all the variables invalid for execution to reserve them, for later execution.

To be specific (Fig. 6), after receiving an AR, a variable in state **Released** updates its state to **A_Reserved** , its holder ID to ID_S, its holder timestamp to T (see symbols in Fig. 4); an AE changes a variable in state **A_Reserved** to **A_Executing** if the AE's ID_S equals the variable's holder ID.

When execution is done, **A_Executing** goes to **A_Executed** . An AE_RES announcing execution completion is back. Similar to conservative strategy, the pivot checks which variables are now valid for execution, and send AR+AE to them. AR is with AE in case that the last AR is lost or delayed.

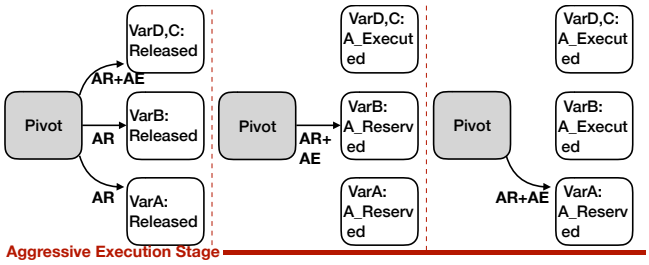


Figure 7: The pivot executes commands according to the PEG in Fig. 1, using aggressive strategy.

Distributed Lock Management. Aggressive and conservative strategies use the identical lock management method and arbitration principle, thus the system allows some pivots to perform aggressive execution and in the meanwhile others execute conservatively. Still, a challenge message is a CR1 or AR.

Since it has no Conservative Execution Stage which is exempted from contention failures, the pivot's timestamps are the only preemption criterion: no matter the variable is in state **A_Reserved** or **A_Executing/ed** : i) A challenge message with a timestamp newer than the holder's will be rejected by the variable's object, a RES announcing failure is sent to the challenger, the challenger loses and will wait or abort; ii) A challenge message with a timestamp older is forwarded to the holder, the holder loses and will abort.

Non-Conservative Strict 2PL. Aggressive strategy uses the 2PL variant which is non-conservative (non-C2PL) and strict (S2PL). S2PL is chosen for the same reasons as shown in conservative strategy. Non-C2PL eliminates conservative strategy's Reservation Stage. It reduces time cost at the expense of lower execution success rates and higher recovery cost, because now a contention failure can make the combo abort and roll back.

7.3 Execution Failure Detection

APEX detects main causes failing an execution: 1) The command is lost/omitted (transmission/object failures); 2) It arrives at the object but cannot access the variable (contention failures); 3) It modifies the variable which however cannot take effect (actuation failures).

Also, the system tries to remedy message losses. Contention failures were discussed in Section 7.1.1, here we focus on the others.

The pivot starts a timer for every control message sent out. An ACK coming back before timeout cancels the timer, otherwise a "message loss" is detected and the message is retransmitted for remedy. If a message is still not delivered successfully when the retry attempts are exhausted, a "transmission failure" is detected, which results from either a bad network condition or "object failure".

Executing states (**C_Executing** or **A_Executing**) in IoT may last relatively long due to time-consuming actuations, e.g., opening curtains, lifting up a garage door, booting a PC. We use periodic RESs streamed to the pivot for keeping it updated. By checking Val_{curr} in RESs the pivot gains knowledge of execution progress. If no enough progress has been made beyond a time limit (e.g., Val_{curr} does not change), an "actuation failure" is announced.

7.4 Rollback

We provide a rollback mechanism for atomicity, which can be enabled and disabled as needed. It applies to reversible, erasable commands (e.g., not including dropping food in a fish tank). When it is enabled and a command in a combo failing in executing, the pivot should undo every executed command.

To realize this, it executes a second command on the variable, setting it to its initial condition. Message 5 in Fig. 3 shows a CE switches a variable in state **C_Executed** back to **C_Executing** , executing a second command for rollback. It also works if the variable is in **C_Executing** , making it abort the command running and execute a new one. Commands should be undone in reverse order of execution (reverse order is obtained from the PEG). The situation for aggressive strategy (message 3, 4 in Fig. 6) is the same. After rollback is finished, the variable stays in an executed state.

7.5 Release

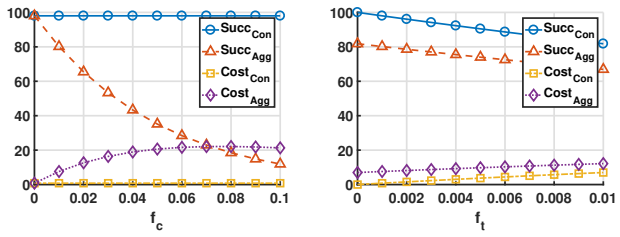
After a combo succeeds in execution, or it fails and the rollback is finished, every variable the pivot holds is either in a reserved or executed state. The pivot releases them in reverse order of execution with Release (RELE) messages (message 6, 7, 8 in Fig. 3; 5, 6 in Fig. 6).

8 SUCCESS RATE AND ROLLBACK COST

Rollback of time-consuming, energy-intensive actuations (e.g., by mechanical components) is expensive. To compare the two strategies's rollback cost, we ignore the variables unrelated to expensive actuations, and assume the remaining have similar expenses, then the cost is assessed in terms of the number of execution failures.

A command will succeed in execution stage only if it encounters no transmission, object, contention or actuation failure. Assume each command has independent failure probability f_t, f_o, f_c, f_a for the four causes respectively, and the probability that it succeeds is $s = (1-f_t)(1-f_o)(1-f_c)(1-f_a)$. Conservative strategy has $f_c = 0$ in execution stage, thus a higher chance to succeed and a lower chance of rollback than aggressive strategy.

The number of objects also affects rollback cost. Assume the PEG has a grid shape (Fig. 9 (a)) with d rows (*PEG depth*) and w columns (*PEG width*) of objects, each relating to expensive actuation. Variables of objects in Row i are the direct precondition variables of those in Row $(i+1)$, thus the former should execute before the latter. Variables of objects in the same row execute concurrently. The probability that a whole row succeeds in execution stage is s^w, and the combo (i.e., the whole PEG) success rate in execution stage is $Succ = s^{dw}$. i rows (iw variables) are to be recovered if and only if the first i rows execute successfully and $(i+1)$th fails. Thus the normalized expected value of rollback cost is $Cost = \Big(\sum_{i=1}^{d-1} iw(s^w)^i(1-s^w)\Big)/dw$. Note that i can be at most $(d-1)$, because if all d rows succeed, the whole combo succeeds and no rollback is needed. Simulation shows that when $fc = 0.01$, $f_t = 0.001$, $f_o = f_a = 0$, conservative strategy's *Succ* decreases roughly linearly with PEG depth and width, being 98% for the 5 by 4 PEG; *Cost* increases roughly linearly, being 1%. Aggressive strategy has the same pattern, with *Succ* being 80% and *Cost* being 8%.



(a) Impact of contention failures (b) Impact of transmission failures

Figure 8: Conservative strategy has higher success rates in execution stage, lower rollback cost than aggressive one.

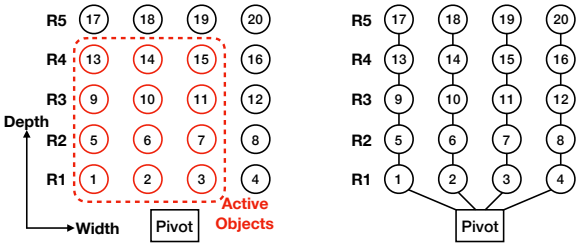
Simulation in Fig. 8 (a) uses $f_t = 0.001$, $f_o = f_a = 0$ and the 5 by 4 PEG. As is seen, conservative strategy's *Succ* and *Cost* do not vary with f_c, staying at 98% and 1%, while aggressive strategy's *Succ* drops to 35% at $f_c = 0.05$, meanwhile rollback cost rises to 21%. In Fig. 8 (b), $f_c = 0.01$, as f_t increases, both strategies get lower *Succ* and higher *Cost*. Still, conservative strategy performs better: its *Succ* and *Cost* are 82% and 7% when $f_t = 0.01$, while aggressive strategy's are 67% and 12%. As a result, conservative strategy shows its advantage in higher execution success rates and lower rollback cost, which becomes more remarkable under condition of high subject density and thus severe resource contention.

9 EXPERIMENTAL EVALUATION

We implement both the conservative and aggressive strategies, and conduct experiments on a testbed consisting of 20 objects, with each emulated by a Raspberry Pi 3. WiFi ad-hoc mode is used in the testbed for communication between objects.

The 20 objects are deployed in one room, while the PEG and network topology are software configurable. Particularly, the PEG we test has a grid shape and we change its size for different tests, by activating different numbers of rows and columns. Fig. 9 (a) shows a 4 by 3 (depth 4, width 3) PEG.

The network topology is configurable: we use application filters to filter out any packet from a non-neighbor node. We use two topologies in our tests: 1) a star topology in which every object is a neighbor of the pivot; 2) a linear topology (Fig. 9 (b)) where objects are chained, and there may be up to 4 lines connected to the pivot.



(a) PEG structure (b) Linear topology

Figure 9: Configurable PEG and network topology.

9.1 Rationality of Our Testbed

We make conscious efforts in the testbed design such that the results produced can reflect the performance in large scale IoT systems.

1) Identical Pis with Homogenous Communications. IoT can be heterogeneous in many ways: the objects can have different power sources (battery-powered, wall-powered), manufacturers, computing performances, storages, etc. We argue that using 20 Pis communicating via WiFi can emulate networking and power aspects well, which are much more relevant to APEX performance than computing and storage capacities.

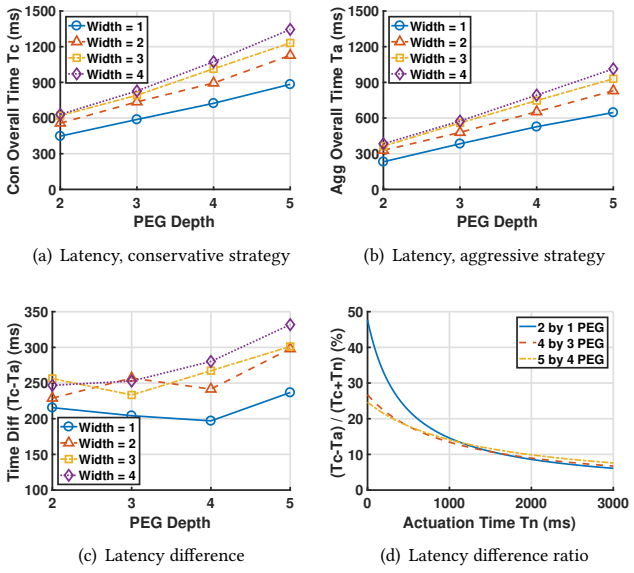
First, as mentioned, in this paper we focus on **write commands** because they alter variables, cause condition/precondition changes and also actuations, instead of commands merely reading variables. Objects which most of the time are read from but not written to are usually sensing nodes (e.g., temperature, smoke, motion sensors) and they are usually battery-powered. Our target—the objects with frequent write operations and actuations (e.g., door locks, lights, air conditioners)—require much more energy and are mostly wall-powered. It is reasonable to emulate them with wall-powered Pis.

Second, in real IoT, different radios (WiFi, Bluetooth, ZigBee, etc.) may co-exist. However, the exact percentage of each radio in the mixture would be hard to determine. We believe it is hard, if not impossible, to find universal proportions of heterogeneous radios from a "typical" IoT network. Thus our testbed uses WiFi only, so the results should not be interpreted literally. Even so, it is valuable in revealing the variation trends under different factor changes.

2) Star or Linear Topology of PEGs. We think PEGs with a star or linear topology make reasonable experimental setting. First, as shown in Section 4, real precondition examples with PEG depth 2–5 exist. We believe such depths are common in large scale IoT environments, and depth 5 (the largest depth we test) suffices most realistic cases. About PEG width, usually there are one to several objects in the same row, thus we use a width up to 4 as approximation. E.g., for the depth 5 PEG in Fig. 2, the first row R1 may be 2 sprinklers, R2: 4 outlets, R3: 6 computers, R4: 2 safe storages. Second, as for topology, in IoT, subjects are likely to interact with the objects nearby (e.g., control the air conditioners in their current rooms), and the objects involved in preconditions are probably also nearby (windows, curtains in those rooms). Those objects and the subject device (i.e. the pivot) are usually within 1 hop, given the transmission range (e.g., about tens of meters) of common wireless radios (e.g., WiFi, Bluetooth, ZigBee). Thus a star topology would be common. We also test linear topology, which we believe is harsher than common situations. Such results will give us a sense of how an occasional, worse case can be.

9.2 Impact of PEG Size on Latency

We test the latency from the pivot's transmission of the first reservation (CR1 or AR) message in a command combo to the reception of the last execution response, under different PEG sizes (depth: 2–5, width: 1–4). Star topology is used here for excluding the impact of hops. The experiments were conducted on weekends when most other adjacent offices/labs were closed, to avoid the interference of dynamic background traffic. The influences of hops and background traffic will be presented in next sections. We find that in this single-hop, silent environment, conservative and aggressive strategies cost 1.34 s and 1.01 s respectively to execute commands on 20 objects. Both are acceptably fast and the time saved by aggressive strategy is not remarkable. Conservative strategy might be preferred for its extra advantages, e.g., lower rollback cost.



(a) Latency, conservative strategy (b) Latency, aggressive strategy

(c) Latency difference (d) Latency difference ratio

Figure 10: Latency under different PEG sizes.

Conservative strategy's latency T_c (Fig. 10 (a)) increases with PEG depth roughly linearly, because one more depth level brings in one more precondition and precondition commands must execute successively. Also, PEG width increase leads to T_c growth which is slower than linear, because objects in the same row execute concurrently. Besides, when the number of objects is relatively large (e.g., 5 by 4 PEG, 20 objects), the latency may grow a little faster than linearly, due to more severe congestion. In Fig. 10 (b), aggressive strategy's latency T_a shows similar trends, but is smaller than T_c under the same condition. E.g., for 20 objects, conservative strategy costs 1.34 s; aggressive strategy costs 1.01 s, saving 0.33 s.

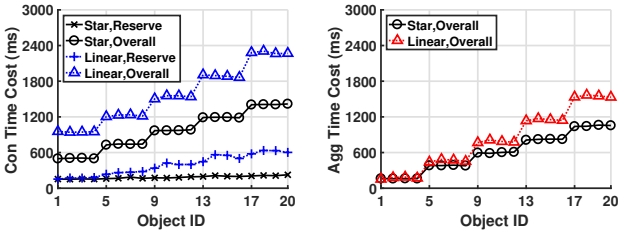
Fig. 10 (c) shows that the latency difference between conservative and aggressive strategies increases with PEG size, from 0.2 s to 0.33 s. Aggressive strategy saves time mainly because it has no reservation stage as in conservative strategy. As for the ratio of the

difference to conservative strategy's overall latency (Fig. 10 (d)), it mostly decreases with PEG size: since the difference is similar, a larger PEG size corresponds to longer overall latency, decreasing the percentage of the difference. E.g., for the 2 by 1 PEG, aggressive strategy saves 48% time, while for the 5 by 4 PEG, it saves 25%.

Actuation Time. For executions with time-consuming actuations (e.g., opening a window), execution time can remarkably increase, making overall latency much longer and the time saved by aggressive strategy less significant. Fig. 10 (d) presents the results for three PEGs, and all their latency difference ratios fall to 20% if actuation time T_n is around 0.5 s, and under 10% when $T_n > 2$ s. As a result, in situations with long actuation time, the time saved by aggressive strategy may become even negligible.

9.3 Impact of Hops on Latency

We evaluate the influence of hops by applying the linear topology in Fig. 9 (b) to the testbed. With this topology, an object in Row i is i hops away from the pivot, forming a typical multi-hop situation. E.g., Object 1–4 are 1 hop away, and Object 17-20 5 hops. PEG size is fixed at 5 by 4, and the network is largely silent to avoid the impact from background traffic. We find that in this multi-hop environment, conservative and aggressive strategies cost 2.26 s and 1.53 s respectively. Both are still responsive, and aggressive strategy now saves remarkably more time (0.73 s).



(a) Latency, conservative (b) Latency, aggressive

Figure 11: Latency in multi-hop situations.

Fig. 11 shows reservation latency (till reservation is done, applied to conservative strategy only) and overall latency (till execution is done). In Fig. 11 (a), linear topology where an object is 1–5 hops leads to higher latency in finishing reservation and execution than star topology where every object is 1 hop from the pivot. Object 20 is reserved at 0.6 s, about 0.38 s slower than star topology; it finishes execution at 2.26 s, about 0.92 s slower. Aggressive strategy (Fig. 11 (b)) costs 1.53 s before Object 20 finishes execution, about 0.52 s longer than star topology. Also, the time difference between conservative and aggressive strategies increases to 0.73 s under linear topology, which is 0.4 s larger than that under star topology.

9.4 Impact of Background Traffic on Latency

We run the testbed continuously in our lab surrounded by other offices/labs during weekdays to record the effect of background traffic at different times of the day on latency. The network is very congested (slow rate with large variance) in the daytime of weekdays due to large amounts of traffic from other companies and labs. We find that aggressive strategy saves significant time (> 7s) in this realistic environment with much background traffic, and might be preferred to conservative strategy for its better responsiveness.

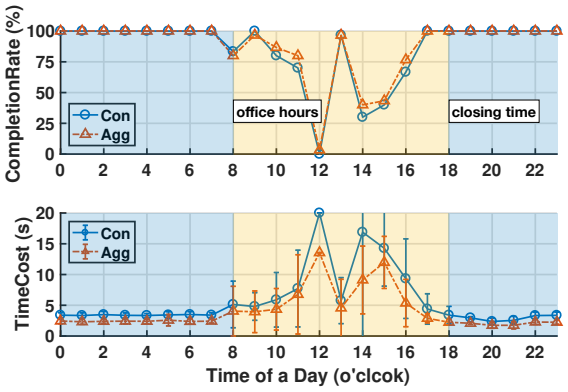


Figure 12: Latency at different times of the day.

Fig. 12 shows the completion rates (a combo is regarded as completed if all the 20 objects finish execution and respond within 20 seconds) and the latencies. From 6 PM to 8 AM, conservative/aggressive strategies have ~ 100% completion rates, and cost stably around 3.18/2.21 s respectively, with the latter saving 0.97 s.

In the daytime the completion rates drop, and there are three hours (12–1, 2–4 PM) when both strategies have rates below 50%. Through channel monitoring we find during those hours the traffic from adjacent companies/labs was at peak, and the network was saturated. Even 1 hop transmission frequently failed, let alone APEX's controlling 20 multi-hop objects. The countermeasures are orthogonal to APEX design, and are discussed in Section 11.

The daytime combos that finish execution have higher and more fluctuating latencies than the night ones. Also, most of the time, aggressive strategy has higher completion rates and shorter latency than conservative: it saves up to 7.76 s, 46% of conservative strategy's time. Note that the completion rate here is different from *Succ* in Section 8: *Succ* is the completion rate in execution stage; conservative strategy has higher completion rates in execution stage due to no contention failures, while it has lower overall completion rates because of one more stage (Reservation Stage) than aggressive strategy. Reservation is harder to finish and costs longer in a congested network.

9.5 Message Overhead

Message overhead is the number of messages sent by the pivot and objects, evaluated under the same conditions in Section 9.2, i.e. star topology in quiet environments. Retransmission occurs for any message whose ACK does not return within 200 ms.

Conservative strategy needs the pivot to send n CR1s, $(n - w)$ CR2s, n CEs, and the objects to send n CR1_RESs, n CE_RESs. Aggressive strategy needs $(n - w)$ ARs, n AEs from the pivot, and n AE_RESs from the objects. No individual CR2s/ARs are needed for the first row's w objects because they are merged into the CEs/AEs. Aggressive strategy needs $2n$ fewer messages. When using the 5 by 4 PEG, ideally conservative strategy needs 96 messages, and aggressive strategy needs 56 messages.

Table 1: Number of messages. n: # of objects in the PEG; w: # of objects in the first PEG row.

Num of Msgs	Pivot	Object	Total
Conservative	$3n - w$	$2n$	$5n - w$
Aggressive	$2n - w$	n	$3n - w$

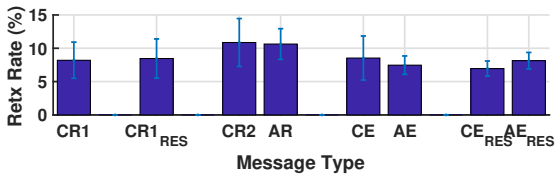


Figure 13: Retransmission rates of different message types.

The realistic message overhead is a little more expensive than the ideal case due to massage losses. Fig. 13 shows that our implementation has acceptably low retransmission rates for all of the message types: they are around or below 10%.

9.6 Summary on Strategies' Suitable Scenarios

According to the experiment results, we find that:

1) Conservative strategy has slightly longer latency than aggressive strategy when background traffic is moderate. Even for 20 objets in linear multi-hop topology, the difference is less than 1s. It might be preferred to aggressive strategy for its advantage in smaller rollback cost at the expense of a little extra latency.

2) When the network is severely congested, which can happen during office hours in enterprise environments, aggressive strategy may save remarkable time (up to 7s in our experiments). It might be more suitable because this latency reduction leads to noticeable responsiveness improvement to users.

10 RELATED WORK

Simultaneous Commands. Logitech Harmony Smart Control [17] allows a user to make her smartphone a one-touch universal remote control to operate up to 8 home entertainment devices (e.g., TV, PC, Xbox). The user connects the devices to a smart hub and predefines actions, e.g., switch to her favorite TV channel, and later uses one tap from smartphone to trigger it. Apple Home [1] allows a user to create a "scene" and predefine multiple actions inside. Then she uses one tap to trigger all the actions. E.g., a scene called "good morning" has actions including warming up the house, opening the blinds, and firing up the coffee maker. The objects and their commands in both cases must be independent such that they can safely execute simultaneously. APEX considers the constraints on execution orders among commands, and enforces such orders.

IFTTT. IFTTT [14] (If This Then That) creates chains of simple conditional statements such that one action triggers another. The actions are mostly on app or web based services. E.g., if a weather web shows it will snow tomorrow, then a mobile notification will be generated. Recently IFTTT has moved into home automation, e.g., Belkin [2] uses it for actions on smart home devices (WeMo). E.g., if the sun sets, then the WeMo switch will be turned on.

APEX is significantly different with existing work like IFTTT. i) It regards the user command as the end and searches conditions "backward", fulfills and holds them before the user command executes. To the contrary, IFTTT performs intuitive "forward" condition chaining and command execution. Note that APEX does not aim at replacing work like IFTTT, because "backward" and "forward" are complementary. ii) APEX uses a more generic model that each command has possibly multiple, recursive preconditions, and uses a DAG-like structure to represent them. It is more powerful than a model with one condition-command pair or multiple such pairs simply chained in a linear fashion. iii) APEX addresses more issues, including atomicity and isolation, specifically in large scale enterprise IoT contexts where large numbers of devices and complicated dependency relationships exist. We solve the issues arising from this new problem context.

Policy Expression & Conflict Detection. DepSys [19] detects the conflicts in a home setting, among multiple applications sharing physical world entities. SIFS [16] also checks the conflicts or policy violations between apps. Additionally, it simplifies user programming by allowing them to express high-level intent, and the system decides which operations to perform for satisfying the intent. Liang et al. [15] propose a solution to verify if the user specified IFTTT-style program logic violates their expectations—policies expressed as conditions in conjunction, and take a step in automated debugging for the violation to ease non-expert debugging. Dar et al. [7] exploit the concept of virtualized IoT services, and achieve a declarative way to specify applications' dependability requirements on devices (e.g., an assisted living application needs the data accuracy and availability of motion sensors to be highly dependable). Unlike the work on policy expression or conflict detection, APEX addresses problems in (precondition related) policy enforcement.

Policy Enforcement. CoMPES [11] is a cloud-based system composed of a set of virtual machines (VM). VMs have predefined policies of what commands should be executed under what conditions; they receive the telemetry from IoT devices on the ground to determine the current condition, and send the corresponding commands back, to enforce the policies. CityGuard [18] is a centralized smart city safety watchdog, to which the city services send requests of taking certain actions. The watchdog checks if an action would lead to device or environment conflicts (e.g., CO release above the threshold) based on predefined safety or performance policies, and only executes safe actions. APEX is quite different: i) It focuses on the enforcement of precondition rules, which has not been well addressed before; ii) Its execution is performed locally by subject devices, not via a centralized entity, and has resistance to a single point of failure, and shorter latency.

Concurrency-Control. Concurrency control for database, operating systems [3, 26] aims at faster processing for concurrent operations on the premise of correctness. ACID (Atomicity, Consistency, Isolation, Durability) [9, 10] is a set of database transaction properties for guaranteeing validity despite errors, failures, crashes, etc. Two-Phase Locking (2PL) [3, 27] achieves isolation. Strict 2PL [4, 21] which releases no write lock until the end of a transaction prevents cascading abort. Rigorous 2PL does not release any read or write lock until the end. APEX borrows the isolation, atomicity concepts, and adapts 2PL to realize isolation in a vastly different context of large scale IoT systems.

Deadlocks. According to the work of Coffman et al. [6] and Havender [12], if any of the four conditions (i.e., mutual-exclusion, hold-and-wait, no-preemption, circular wait) is not satisfied, it is impossible for a deadlock to happen. APEX prevents deadlocks during precondition execution using preemption.

11 DISCUSSION

Rule Syntax Standard. Different vendors may follow the same syntax standard when making rules, such that precondition relevant objects (e.g., air conditioners and windows) will work in harmony. Otherwise (e.g., air conditioner vendors mention window as "win" while window vendors use "window"), it is left to the admin to unify them when importing them into database.

Low Power Consumption. In this paper we focus more on the common IoT operations with expensive actuations (e.g., by mechanical, electrical components), and propose a conservative strategy to reduce the cost of rollback from aborted actuations. Those objects (e.g., doors, air conditioners) are wall-powered and have less concern about energy. However, for battery-powered objects, energy-saving solutions [20, 28] are necessary. We leave low-power automatic precondition execution as future work.

Transmission Failures. Severe transmission failures in congested networks lead to low completion rates of APEX. Possible countermeasures are: retry when the traffic is less intensive, or switch to other radios not interfered by WiFi traffic. In essence, this problem is on communication reliability, orthogonal to and interdependent from precondition enforcement.

Rollback Cost Metric. During analysis of rollback cost we simplify the case by assuming every object has the same expense. To be more accurate, a cost metric should first be defined, e.g., using time cost, energy, or weighted multiple factors. This relates to policy authoring or user preference that we do not address in this work.

Security. We consider benign subjects and objects only: a subject device strictly follows the PEGs obtained from the backend to control the process of precondition execution; and an object faithfully, correctly reacts to every received message (reservation, execution, rollback, release). An adversary who does not follow the PEGs or who has compromised objects can easily destroy benign combos' preconditions. IoT security [22, 23] and privacy [5, 8] are worthy topics which we plan to explore to reinforce APEX.

12 CONCLUSION

In this paper, we describe the design and evaluation of APEX, which automatically discovers the preconditions of a subject command, generates and executes commands to make those preconditions true. Also, it is able to prevent concurrent commands from interleaving, and roll the system back upon execution failures. We propose two execution strategies suitable under different scenarios, and our evaluation proves conservative strategy sustains higher execution success rates, lower rollback cost, while aggressive strategy executes faster, saving up to 7s, 46% of conservative strategy's time.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant number CCF 1652276.

REFERENCES

[1] Apple. [n. d.]. HomeKit. https://developer.apple.com/homekit/.
[2] Belkin. [n. d.]. Using IFTTT with WeMo. https://www.belkin.com/us/wemo/ifttt/.
[3] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. Concurrency control and recovery in database systems. (1987).
[4] Yuri Breitbart, Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Abraham Silberschatz. 1991. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering* 17, 9 (1991), 954–960.
[5] Alan Chamberlain, Andy Crabtree, Hamed Haddadi, and Richard Mortier. 2018. Special theme on privacy and the Internet of things. *Personal and Ubiquitous Computing* 22, 2 (2018), 289–292.
[6] Edward G Coffman, Melanie Elphick, and Arie Shoshani. 1971. System deadlocks. *ACM Computing Surveys (CSUR)* 3, 2 (1971), 67–78.
[7] Kashif Sana Dar, Amir Taherkordi, and Frank Eliassen. 2016. Enhancing dependability of cloud-based IoT services through virtualization. In *Internet-of-Things Design and Implementation (IoTDI), 2016 IEEE First International Conference on*. IEEE, 106–116.
[8] Chuhan Gao, Varun Chandrasekaran, Kassem Fawaz, and Suman Banerjee. 2018. Traversing the Quagmire that is Privacy in your Smart Home. (2018).
[9] Jim Gray et al. 1981. The transaction concept: Virtues and limitations. In *VLDB*, Vol. 81. Citeseer, 144–154.
[10] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)* 15, 4 (1983), 287–317.
[11] Jared Hall and Razib Iqbal. 2017. CoMPES: A Command Messaging Service for IoT Policy Enforcement in a Heterogeneous Network. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*. ACM, 37–43.
[12] James W. Havender. 1968. Avoiding deadlock in multitasking systems. *IBM systems journal* 7, 2 (1968), 74–84.
[13] Patrick Hurley. 2014. *A concise introduction to logic*. Nelson Education.
[14] IFTTT. [n. d.]. IFTTT. https://ifttt.com/discover.
[15] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically debugging IoT control system correctness for building automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. ACM, 133–142.
[16] Chieh-Jan Mike Liang, Börje F Karlsson, Nicholas D Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. 2015. SIFT: building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*. ACM, 298–309.
[17] Logitech. [n. d.]. Logitech Harmony Smart Control. https://www.logitech.com/en-us/harmony-universal-remotes.
[18] Meiyi Ma, Sarah Masud Preum, and John A Stankovic. 2017. Cityguard: A watchdog for safety-aware conflict detection in smart cities. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*. ACM, 259–270.
[19] Sirajum Munir and John A Stankovic. 2014. Depsys: Dependency aware integration of cyber-physical systems for smart homes. In *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*. IEEE, 127–138.
[20] Aaron N Parks, Alanson P Sample, Yi Zhao, and Joshua R Smith. 2013. A wireless sensing platform utilizing ambient RF energy. In *Power Amplifiers for Wireless and Radio Applications (PAWR), 2013 IEEE Topical Conference on*. IEEE, 160–162.
[21] Yoav Raz. 1992. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *VLDB*, Vol. 92. 292–312.
[22] Francesco Restuccia, Salvatore D?Oro, and Tommaso Melodia. 2018. Securing the Internet of Things in the Age of Machine Learning and Software-defined Networking. *IEEE Internet of Things Journal* (2018).
[23] Cong Shi, Jian Liu, Hongbo Liu, and Yingying Chen. 2017. Smart user authentication through actuation of daily activities leveraging WiFi-enabled IoT. In *Proceedings of the 18th ACM International Symposium on Mobile Ad Hoc Networking and Computing*. ACM, 5.
[24] Charles Simonyi. 1999. Hungarian notation. *MSDN Library, November* (1999).
[25] Andrew S Tanenbaum. 2009. *Modern operating system*. Pearson Education, Inc.
[26] Andrew S Tanenbaum and Albert S Woodhull. 1987. *Operating systems: design and implementation*. Vol. 2. Prentice-Hall Englewood Cliffs, NJ.
[27] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier.
[28] Pengyu Zhang. [n. d.]. Enabling Low-power Communication, Sensing, and Computation on Internet-of-Things. ([n. d.]).
[29] Qian Zhou, Mohammed Elbadry, Fan Ye, and Yuanyuan Yang. 2017. Flexible, Fine Grained Access Control for Internet of Things. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*. ACM, 333–334.
[30] Qian Zhou, Mohammed Elbadry, Fan Ye, and Yuanyuan Yang. 2018. Heracles: Scalable, Fine-Grained Access Control for Internet-of-Things in Enterprise Environments. *IEEE INFOCOM 2018* (2018).