OKAPI: In Support of Application Correctness in Smart Home Environments

Themis Melissaris
Princeton University
themis@cs.princeton.edu

Kelly Shaw
University of Richmond
kshaw@richmond.edu

Margaret Martonosi
Princeton University
mrm@princeton.edu

***Abstract*—Typical Internet of Things (IoT) and smart home environments are composed of smart devices that are controlled and orchestrated by applications developed and run in the cloud. Correctness is important for these applications, since they control the home's physical security (i.e. door locks) and systems (i.e. HVAC). Unfortunately, many smart home applications and systems exhibit poor security characteristics and insufficient system support. Instead they force application developers to reason about a combination of complicated scenarios—asynchronous events and distributed devices. This paper demonstrates that existing cloud-based smart home platforms provide insufficient support for applications to correctly deal with concurrency and data consistency issues. These weaknesses expose platform vulnerabilities that affect system correctness and security (e.g. a smart lock erroneously unlocked). To address this, we present OKAPI, an application-level API that provides strict atomicity and event ordering. We evaluate our work using the Samsung SmartThings smart home devices, hub, and cloud infrastructure. In addition to identifying shortfalls of cloud-based smart home platforms, we propose design guidelines to make application developers oblivious of smart home platforms' consistency and concurrency intricacies.**

I. Introduction

Increasing focus on Internet of Things (IoT) approaches has led to the development of many platforms, systems and applications to enable and connect IoT devices. As IoT systems are used more broadly in more safety-critical environments, their security and reliability are increasingly important. Unfortunately, current IoT devices and systems do not yet meet these expectations, as can be seen in the frequent news reports about their vulnerabilities, e.g. [1],[2],[3].

This paper focuses on smart home environments, although most of the relevant system characteristics are common in other IoT systems that perform monitoring and actuation as well. Figure 1 depicts an example of a smart home system. In smart home environments, "smart" embedded devices are wirelessly connected to a hub. Typically the hub coordinates interaction between devices and acts as a gateway to the Internet, but devices can also be directly connected to the Internet gateway. Via the Internet gateway, the Edge Home Network connects to the cloud, where all the processing for the devices will take place. In most current environments, smart home device state is stored (solely) in the cloud. This includes sensor/actuator transitions and application logic controlling or interacting with these devices.

In order to design smart home systems with efficiency and at scale, it is essential to leverage concurrency. However, relying on concurrency can in itself be a challenge and provide many opportunities for things to go wrong. For example, Figure 1 shows that events for the same devices may be transmitted concurrently, processed concurrently and update cloud state concurrently. This concurrency offers convenience in some ways but also exacerbates correctness challenges regarding atomicity, data consistency, and event ordering. For correct execution of applications that access or modify shared resources, state accesses need to execute atomically and with the correct consistency ordering in order to ensure that state reads and updates reflect the expected and correct values.

These correctness issues can be observed in existing IoT platforms. Table I presents popular IoT platforms used for home automation, control and orchestration. These platforms are compared based on their programmability, system support and the consistency guarantees they provide. The platforms are presented in decreasing order of popularity; the number of installations of the platforms' mobile applications, as reported by the Google Play Store [4], is used as an estimate of their user base. Alexa [5] and Google Home Assistant [6] are general purpose personal assistants that allow development of applications with any choice of cloud storage systems from Amazon Web Services [7] and Google Cloud Platform [8] respectively, which allow development of applications with both strong and weak consistency guarantees. Out of the smart home platforms presented in Table I, only Apple Homekit [9], Samsung SmartThings [10] and Vera Home [11] allow development of applications. Homekit uses a mobile phone to control smart home devices and stores their state in a common database on the phone, whereas Vera performs home automation using a programmable hub. We have demonstrated correctness violations in two of these widely used platforms, SmartThings and Vera. In particular, SmartThings lacks primitives to allow applications to perform atomic code execution, and it uses a weak consistency model for its event ordering. Similarly, Vera allows creation of plugins for its programmable hub using scripts that are not thread-safe and cannot guarantee mutual exclusion while accessing smart home devices [12].

As Table I notes, smart home platform implementations often do not provide application developers with the atomic operations and predictable consistency ordering required for correct concurrent executions. Even when such mechanisms are provided, it is complex for application developers to know when and how to use them. This paper proposes the OKAPI system which offers atomicity and strict event ordering as an external add-on functionality in support of correctness. The paper first focuses on understanding unexpected behavior of smart home applications by studying and measuring the impact of weak data consistency, lack of atomic execution and out of order event delivery in popular smart home architectures. All observations, experiments and measurements are carried out using a realistic smart some setup. We specifically analyze and diagnose consistency and ordering problems in the Samsung SmartThings platform [10], but our techniques and observations apply to other cloud enabled smart home platforms as well. We use these empirical measurements and experiences to motivate the need for OKAPI. Our specific contributions are as follows:

- Foremost, our work demonstrates the serious correctness concerns of programmer scenarios where application logic breaks due to i) use of weak consistency in the cloud storage, ii) race conditions caused by the absence of guarantees for atomic execution on shared resources and iii) insufficient support for in-order event delivery. We measure and analyze smart home applications and systems and present our findings.
- Furthermore, we propose OKAPI, a synchronization ser-

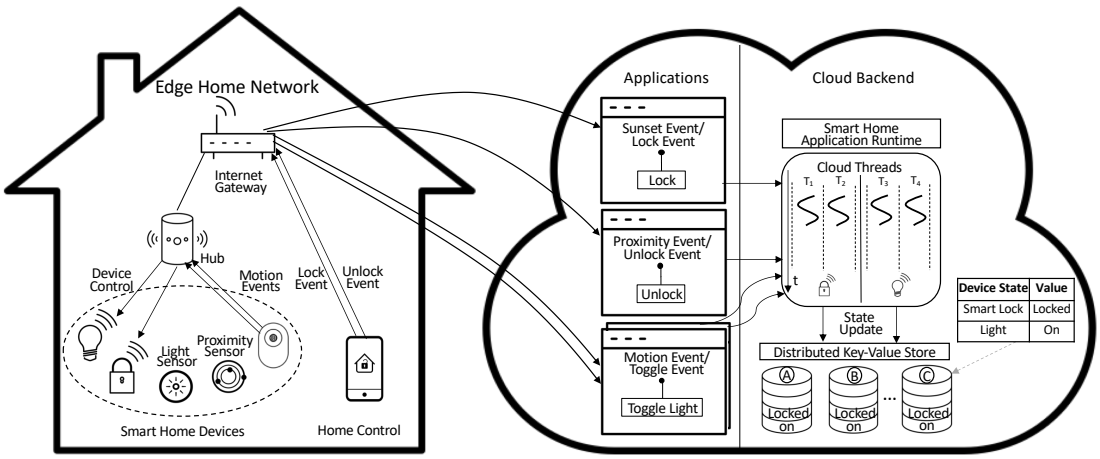

Fig. 1. Presentation of a typical smart home architecture. Events are generated in the smart home network by smart home devices and home controllers (e.g. smartphone app). Each event triggers the execution of an application in the cloud, potentially changing device state. Concurrent events are processed by the application runtime and data is persistently stored in a distributed key-value store.

TABLE I
IOT PLATFORM COMPARISON.

IoT Platform	Installations	Generality	Programmable	Atomic Primitives	Consistency Model
Google Home Assistant	50-100M	General	Yes	Yes	Configurable
Amazon Alexa	5-10M	General	Yes	Yes	Configurable
Apple HomeKit	-	Smart Home	Yes	Yes	Strong(phone)
Logitech Harmony	500-1000K	Smart Home	No	Not available	Not available
Samsung SmartThings	100-500K	Smart Home	Yes	No	Eventual
August Home	100-500K	Smart Home	No	Not available	Not available
Vera Home	50-100K	Smart Home	Yes	No	Strong

vice that provides atomicity and ordering guarantees to smart home applications. OKAPI can be implemented on a server external to the smart home platform provider. We evaluate OKAPI on the Samsung SmartThings platform by investigating its latency and throughput penalties.

- Finally, we discuss the impact of our approach on smart home applications and propose guidelines for designing smart home architectures with respect to correctness.

Section II identifies atomicity, ordering and consistency problems in smart home platforms using SmartThings as an example. Section III explains how these problems manifest themselves in the SmartThings platform and presents examples. Section IV presents the OKAPI solution for providing ordering and atomicity. Section V describes our evaluation methodology. Finally, Section VI presents the evaluation of OKAPI, and Section VII discusses related work, and Section VIII concludes.

II. ORDERING, ATOMICITY AND CONSISTENCY IN SMART HOME SYSTEMS

A. Smart Home Architectures

Figure 1 presents a typical smart home architecture, split between the Edge Home Network and the Cloud. In the Edge Home Network, "smart" embedded devices that can range from motion and light sensors to smart door locks and smart bulbs are wirelessly connected to a hub. These devices communicate via wireless protocols such as Zigbee [13], ZWave [14] or BLE [15] to a hub that coordinates interaction between devices and acts as a gateway to the Internet. Devices such as smart phones can be connected to the Internet gateway as well, serving as endpoints for the users' home control and orchestration. Any event generated by the smart home devices or the Home Control devices reaches the Internet gateway and is managed by the cloud component of the smart home architecture. In cases when actuation is required, events for control of devices reach the Edge Home Network.

In order to control the behavior of devices in the Edge Home Network and manage the influx of events, a smart home architecture features applications that run on a supporting cloud infrastructure. Smart home applications contain the logic behind home automation and are initiated by external triggers, events that are sent from smart home and Home Control devices. The types of events and the order in which they arrive affect the action that smart home applications perform. As most of the smart home architectures are event driven, the runtime system maintains a thread for each event received by an application, called Cloud Threads. The Cloud Threads execute the application logic, access and modify device state in the distributed storage layer and actuate physical devices within the Edge Home Network.

The distributed storage layer is an important component of cloud-based smart home architectures, responsible for applications' persistent storage and data consistency. Typically, a distributed Key-Value Store is in the core of the cloud storage layer. Key-Value Stores allow representation of data in key-value pairs $< key, value >$, allowing retrieval and storage of data from these data structures. In the cloud, key-value stores are replicated across multiple machines, increasing the availability of data and simultaneously introducing the complexity of keeping data consistent across all copies.

B. Event Ordering

We introduce event ordering issues first here, followed by atomicity in the next section. If we revisit Figure 1, we observe multiple events generated by the motion sensor and the smartphone responsible for home control. Although events are generated by devices connected to the same network, the edge home network and many current IoT services do not offer strong guarantees regarding their ordering. In this case, multiple events generated by a single device can be

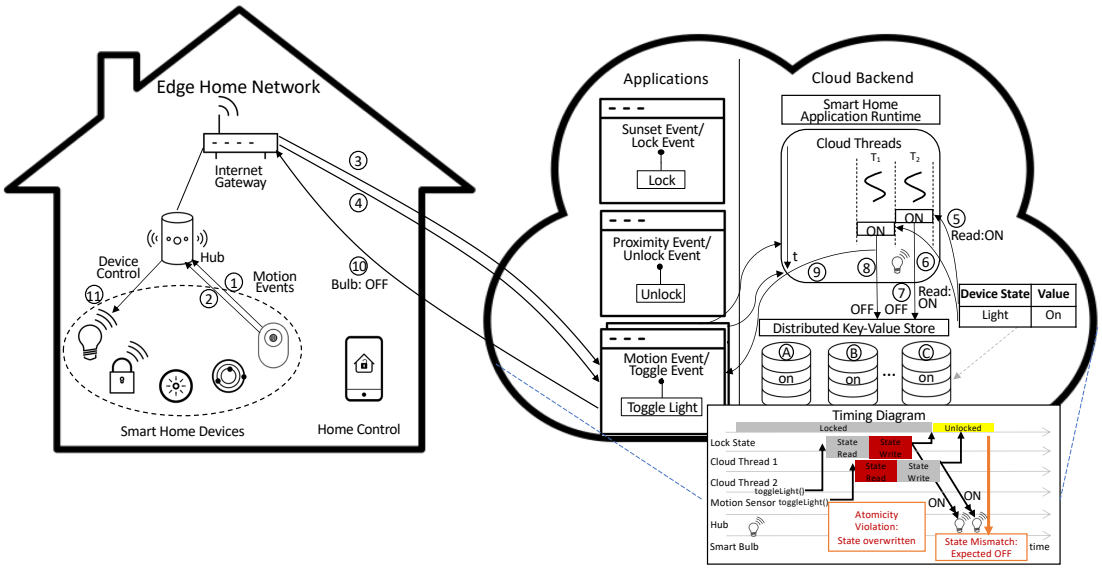

Fig. 2. Atomicity violation scenario in a smart home system. In some smart home environments, multiple events can perform writes to shared state unrestrictedly. As such, the lack of read-modify-write atomicity leads to incorrect or non-intuitive results.

reordered at multiple locations before arriving at the cloud service. Messages can be reordered by the wireless protocol connecting smart home devices with the smart home hub and also as they travel from the home's gateway to the cloud using transport protocols. In our example in Figure 1, both lock and unlock events are generated by the same phone. A possible reordering however can change the order of lock and unlock events. In that case, the device's view of state will become inconsistent with the state intended by the user's requests, and the door will be erroneously left in a state contrary to user's expectation. After the events arrive at the cloud platform, there are additional potential locations for reordering, due to arbitrary delays in accesses and updates in cloud storage as well as in the processing of events. This simple reordering scenario results from the fact that events can arrive out of order at the hub or home gateway within the edge home network or at the cloud and there is no mechanism to guarantee order of arrival identical with the order at the origin.

C. Atomicity Violations

Figure 2 depicts a scenario of a motion sensing application in a smart home system that shows the need for atomic operations. In it, two different events are created by a motion sensor due to physical triggers (labels 1,2 in Figure 2) and reach a smart home application in the cloud via the edge home network (3,4). The motion events are received by an event handler in an application that toggles a smart bulb, changing its state from on to off and vice versa. The state of the physical smart light bulb is a state variable that is stored in the cloud. In order to execute the application logic and perform accesses and modifications to the light bulb state, the smart home system runtime spawns a cloud thread per event. For the toggle operation, each cloud thread will access the current light bulb state in the key-value store (5,6), flip its value and perform an update to the cloud store (7,8). If the executions of both cloud threads corresponding to the motion events overlap and occur non-atomically, however, both cloud threads will read the initial state of the smart bulb instead of the second cloud thread viewing the state as altered by the first. The timeline of these accesses is presented by the timing diagram in Figure 2. The simultaneous accesses to cloud state create a race condition and will cause both cloud threads to read the state of the light bulb as ON and set it to OFF (9,10,11) which is counter to expectation.

Although often challenging, application developers can reason about and enforce atomicity while developing their applications. However, some smart home platform providers have designed their runtime systems in ways that do not allow use of atomic primitives by smart home applications due to cost considerations; when cloud threads do not restrict concurrent accesses to shared state, cloud thread execution times will remain shorter, which translates to lower resource usage and consequently cost reduction. Our example, however, indicates that having no mechanism to guarantee atomicity leads to correctness violations, which are critical in smart homes as the operation and physical security of users' homes might be compromised.

D. Weakly consistent cloud storage

Databases and distributed systems have long employed a variety of approaches for reasoning about state updates and ordering. These include the familiar ACID transaction model [16], as well as newer distributed systems concepts including eventual consistency [17] or causal consistency [18]. For cloud platforms and services supporting IoT and smart home applications, weaker consistency models like eventual consistency are commonly used to optimize for performance and trade off consistency for availability [19]. Weak ordering approaches on state updates—such as eventual consistency [17]—offer few formal promises regarding *when* a state update will complete; this burdens programmers trying to ensure reasonable operation on top of unpredictable foundations. A consistency model offering strong consistency guarantees would allow application developers to reason about update orders in which updates should be applied to avoid reading stale values and counter-intuitive behavior when interfacing with cloud storage.

Figure 3 presents a scenario of a sunset event (light sensor) and a proximity event (labels 1,2 in Figure 3) that are handled by two different applications (3,4), the first one causing a smart door lock to unlock and the second one to lock respectively. For each of these events, there are corresponding cloud threads that perform an update to an eventually consistent cloud storage, in the order of their arrival. An eventually consistent

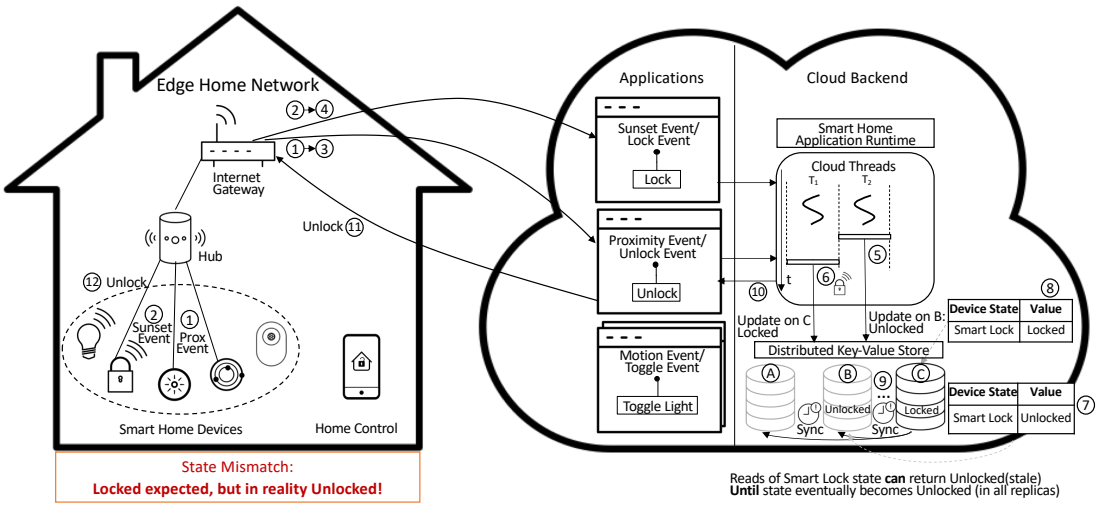

Fig. 3. Consistency violation scenario in a smart home system. Multiple events are received and processed by cloud applications, designed to control the state of a smart door lock, performing updates to cloud storage in the process. Due to weak consistency, stale values can be returned by the key-value store and cause unexpected application behavior.

cloud storage system means that the updates will be eventually applied in the same order to all replicas, but there is no guarantee as to when the updates will be visible throughout the system. In the example, the first update writes Unlocked into replica B (5,7) and subsequently, the second update writes Locked into replica C (6,8). In practice, that means that a read of the value of the Smart Lock state can return Unlocked, even if the more recent update set the value to Locked. This result can only happen during the window when the Locked update is not yet visible across all replicas (9). During that window, any application that uses the lock state to control smart home devices can access a stale lock state from one of the replicas for which the latest update is not yet visible. In this scenario, the application will Unlock the smart lock as this is the stale value returned by a read after the updates to the key-value store (10,11,12). Due to the staleness of the data, the application's control of the physical lock can be counter to the user's expectation.

III. Testing Real-World Smart Home Systems

Section II and Figures 2 and 3 offer simple scenarios where network ordering, atomicity and weak consistency affect application correctness. Despite the seemingly naive aspects to these scenarios, our work indicates they actually occur in real-world systems. In this section, we demonstrate how these scenarios can appear in the context of smart home systems and other platforms that build on distributed storage. We use the SmartThings smart home platform as a case study and present measurements that provide concrete evidence of these behaviors, along with the testing methodology.

A. SmartThings Smart Home Architecture

SmartThings home automation platform allows development of applications in the cloud, called SmartApps, that interact with devices in the user's home. The SmartThings platform follows the smart home architecture described previously; here, we describe how network reordering, atomicity and consistency violations appear on SmartThings.

There are two ways of persisting application state in the SmartThings cloud platform, *state* and *atomicState*. These storage mechanisms have different expectations about when their modifications are propagated to persistent storage in the cloud backend.

To understand updates to persistent data, one must understand the SmartThings execution model. SmartApps are not continuously running in the cloud. Their execution is triggered when an event arrives at the cloud (initiated by either a physical device or a mobile application), resulting in an event handler in the SmartApp executing on a cloud server. Multiple instances of the same SmartApp may execute simultaneously and independently in the cloud. This design means that when event handlers execute, their only context is the persistent state stored in the backend and that multiple event handlers may be accessing that state at the same time.

Given the potential overlap of reads and writes from different event handlers, it is difficult for programmers to develop correctly functioning applications without understanding the consistency guarantees of a smart home platform. On SmartThings, even if implemented correctly, an application is not guaranteed to execute correctly as we will see next. The cloud storage component of SmartThings is built with Cassandra [20], a widely-used, eventually-consistent, distributed, key-value store. For the two types of SmartThings state, *state* and *atomicState*, there are different expectations for when updates are sent to persistent storage. When an event handler executes, modifications made to *state* objects are only updated in persistent storage *after* the event handler's execution completes. In contrast, modifications to *atomicState* are reflected in the cloud backend not "more or less immediately" [19], meaning updates to persistent storage do not wait for the event handler's execution to complete.

While the name *atomicState* implies atomicity can be achieved, this is not how it is actually implemented. As the SmartThings platform does not provide any form of atomic read-write-modify capabilities, *it is not possible to achieve atomicity*. As a consequence, two simultaneous executions of event handlers accessing the same state (*state* or *atomicState*) can result in classic race conditions; both event handlers may read the state before either updates the state and their subsequent serialized updates result in the first update performed on that state being obliterated by the second event handler's update to that same state. The previous scenario demonstrates why the *atomicState* primitive constitutes a bad design choice. If race conditions are to be prevented, another mechanism must

be created to provide atomic updates to the state.

B. Experimentation on smart home deployments

In order to validate the existence of network reordering, atomicity and consistency violations in smart home systems, we design tests for the respective properties that we want to test, as presented in Table II. For all the tests, there is a single device that is generating the events that trigger the executions of the cloud threads. (See Section V for details on the experimental setup.) All tests are non-deterministic due to concurrency and therefore multiple trials are performed. For the atomicity test, a shared persistent variable x is incremented every time an event is handled. Since the cloud storage does not successfully implement mutual exclusion and executions of event handlers can overlap, concurrent executions can lead to values being erroneously overwritten due to race conditions. When both cloud threads read the value $x = 1$, this is an indication of an atomicity violation that leads to a race condition. Requests generated by a device can be generated in order but can be handled out of order by the application in the cloud. The network ordering test identifies this condition, when the cloud threads read the values in a different order than the order the corresponding events were generated. Finally, we want to test the impact of a weakly consistent key-value store on our application. However, it is impossible to test the impact of weak consistency in the absence of an atomicity mechanism, as we won't be able to distinguish consistency violations from atomicity violations. For this reason, we perform the weak consistency test on a standalone Cassandra key-value store deployment, the same key-value store that the SmartThings platform uses. In this test, we want to identify if a value is stale, if a read returns previous state (value initialized to 0) after a more recent update has set the value. That is possible when a write update is performed on a different replica than the one the read accesses.

We present event reordering and atomicity violation statistics for our tests on the SmartThings platform. As Figure 4 shows, our measurements indicate that event reorderings and race conditions are present and that they increase as the event generation rate on the device increases. For the consistency test, we measure stale values at a frequency of 1.4% over the number of the tests in our Cassandra deployment. Without improved support for the ordering and atomicity with which state is observed and updated and storage systems with stronger guarantees to build upon, even simple smart home systems will suffer from severe correctness and concurrency problems.

(a) Atomicity Test

Cloud Thread 1	Cloud Thread 2	Result	Outcome
x=x+1	x=x+1	$x_1 = 1, x_2 = 2$	OK
write(x)	write(x)	$x_1 = 2, x_2 = 1$	OK
read(x)	read(x)	$x_1 = 1, x_2 = 1$	Race condition

(b) Network Reordering Test

Cloud Thread 1	Cloud Thread 2	Result	Outcome
write(x=1)	write(x=2)	$x_1 = 1, x_2 = 2$	OK
read(x)	read(x)	$x_1 = 2, x_2 = 1$	Reordering

(c) Weak Consistency Test

Cloud Thread	Result	Outcome
write(x=1)	$x = 1$	OK
read(x)	$x = 0$	Stale value (initially $x = 0$)

TABLE II
TESTS FOR DETECTING ATOMICITY VIOLATIONS, NETWORK REORDERING AND WEAK CONSISTENCY.

IV. OKAPI: API FOR RESTORING SMART HOME CONSISTENCY AND ATOMICITY

To avoid the erroneous scenarios discussed earlier and to insure correctness in smart home applications, support for atomic

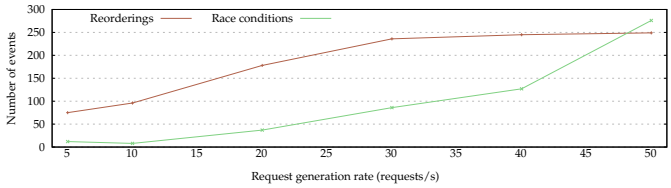

Fig. 4. Statistics for occurrences of event reorderings and race conditions. A test application executes the Atomicity and Network Reordering tests respectively using shared persistent variables. Results are presented in increasing request generation rate and out of a total of 1000 events. Lower is better.

operations and a mechanism for preventing event reordering is needed. However, cloud platforms like SmartThings are often either proprietary or restrictive in terms of platform modifications; application developers are unable to add the required functionality to the cloud backend. Therefore, we propose OKAPI, a synchronization service that creates consistency and atomicity mechanisms for use in Smart Home applications. Developers can use OKAPI to create the guarantees not provided by the platform's cloud backend but needed by their applications.

OKAPI's design consists of two components: an OKAPI server responsible for guaranteeing atomicity and ordering and an OKAPI synchronization layer that initiates a two-phase communication protocol with the OKAPI server on the application side. Applications use the OKAPI synchronization layer to send requests to the OKAPI server. The OKAPI server then disallows concurrent writes to shared state stored in the cloud, effectively serializing access to critical sections of code from concurrent event handlers.

OKAPI's design is flexible and allows adoption across various scenarios with the purpose of restoring consistency and atomicity in the smart home. When the platform does provide atomicity and ordering guarantees, consumers can still choose to deploy OKAPI in case an application does not use platform-available concurrency features correctly. OKAPI may be deployed in different ways. If a user's platform allows programmable hubs, the OKAPI server could be implemented on the hub. Alternatively, OKAPI can be hosted as a synchronization service on a third party server. Due to privacy concerns about third party servers and given the rise of personal cloud systems, individuals could also potentially deploy OKAPI in their personal cloud infrastructure.

A. OKAPI Usage

OKAPI relies on two mechanisms to provide atomicity and ordering functionality, the OKAPI Server and the OKAPI Synchronization Layer. To invoke these, applications annotate their critical sections accordingly and OKAPI does the enforcement.

OKAPI Server: The OKAPI synchronization server is used to implement atomic locks and their corresponding acquire and release operations. For each block of code within a SmartApp that requires mutual exclusion, the server maintains a lock. The OKAPI server receives and handles external lock acquire and release requests from SmartApps trying to access a critical section in the code, e.g. a SmartApp performs updates to shared *atomicState* in the cloud backend. Second, in such a scenario where the application needs to perform atomic updates, a mechanism is required in order to block an executing event handler while it waits to be granted sole access to the critical section by the synchronization server.

OKAPI Synchronization Layer: The OKAPI Synchronization layer provides Smart Home applications with an interface to utilize the atomicity and ordering features the OKAPI Server provides. Mutual exclusion is achieved by sending lock

acquire and release requests to the remote synchronization service via synchronous HTTP requests. A synchronous HTTP request blocks the execution of the event handler issuing the request until a response to the HTTP request is received. In this way, concurrent execution of event handlers is prevented, enabling mutual exclusion of event handlers accessing shared state in the application's critical section. When the response is received from the OKAPI server, the event handler's execution can proceed, knowing it currently has sole access to the shared *atomicState*. When the event handler finishes accessing the shared state, it sends another synchronous HTTP request to the remote server releasing the lock to the shared data. The event handler then waits until a response is received from the remote server to guarantee release of the critical section. For the Synchronization layer protocol to be correct, updates to the shared state need to be propagated to persistent storage before the release of the lock at the remote server. In SmartThings, an update to *atomicState* needs to complete before the lock is released from the OKAPI server, as code outside the critical section could otherwise execute concurrently. Tests across thousands of events did not reveal any updates to *atomicState* that happened out of the order of execution specified by the application or before the release of the lock at the OKAPI server. Based on our experimentation, we regard all *atomicState* updates as synchronous in the application logic.

Timeout Handling and Deadlock Prevention: If no response is returned within a specified time limit for each synchronous HTTP request, the HTTP request times out. The HTTP request timeout for the SmartThings platform is 10s. If a timeout occurs, an additional synchronous HTTP request is generated by the application to gain access to the critical section. During the execution, the OKAPI server can potentially run into a deadlock, e.g. when a remote device gets a lock but does not explicitly unlock it due to loss of network connection. Since the synchronization protocol is based on HTTP, the OKAPI server uses the HTTP timeout value as expiration for exclusive access to the critical section. In addition to HTTP timeouts, the SmartThings applications' execution time is limited to 20 seconds. This limit is imposed by the SmartThings Application Runtime in order to optimize allocation of resources; OKAPI is not limited by the application execution time.

B. Enforcing atomicity

For atomicity, Figure 5 sums up OKAPI's two-phase communication between Smart Home applications and the remote synchronization server. In this example, one phone generates two requests corresponding to the unlock() and lock() methods for a smart lock. The unlock() event handler in Execution 1 sends a synchronous HTTP request to the remote server (sync()) requesting the lock for its shared data to be acquired. The remote server executes its acquire() method for the lock and sends a response to Execution 1 indicating it has access to the shared data and can proceed. Execution 1 is able to read and write the shared data, with those updates being sent to the cloud backend, and send a lock message to the physical device. Execution 1 then sends a synchronous HTTP request releasing the lock which causes the remote server to release the lock. While Execution 1 has the lock, a second synchronous HTTP request is sent to the remote synchronization server by the lock() event handler, but the lock() event handler is blocked until it receives a response to its request after Execution 1's lock release is complete.

C. Ordering

Overall, OKAPI provides *Per-Object Sequential Consistency* [21], [22] at the application level. According to this guarantee,

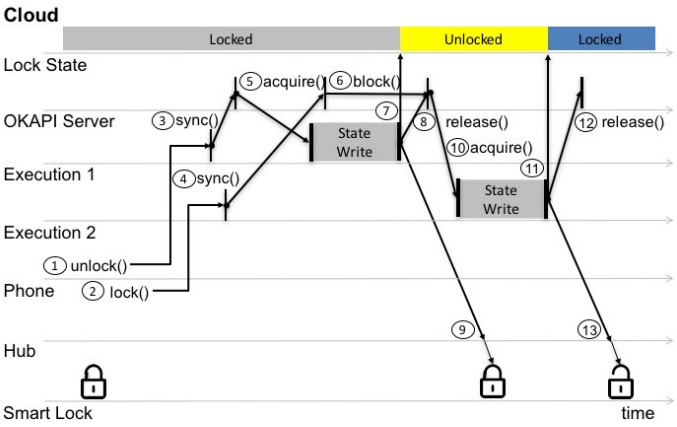

Fig. 5. OKAPI two-phase protocol enforcing atomicity.

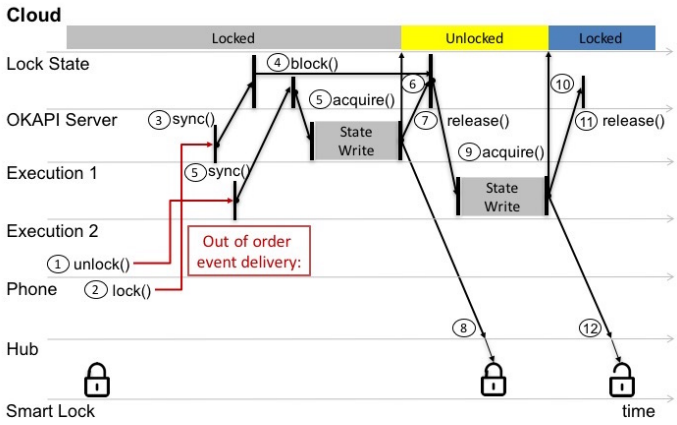

Fig. 6. OKAPI two-phase protocol enforcing atomicity and in-order event delivery in Smart Home applications.

for each shared state object (e.g. shared Smart Lock state), there exists a legal, total order over all events which is reflected in each clients order.

We achieve ordering by building upon the atomicity enforcement in the OKAPI synchronization server. The ordering mechanism is implemented in the remote server and provides a serial order of events per client and a total order of events per shared state object. In practice, OKAPI allows a single copy of an object to progress over time. Clients will always see a newer value of shared state as they interact with it, but these values can be different across clients.

To provide ordering guarantees, each client provides metadata in the form of a sequence number embedded in each request which allows the remote server to order and to serialize these requests. We introduce a sequencing scheme in the events received by the application to achieve ordering of events per client. Sequencing of events can also be achieved by modifying the smart home platform or hub, if the implementations are open source. Figure 6 summarizes OKAPI's enforcement of both atomicity and ordering. For one of the two requests arriving out of order at the Smart Home application, the synchronization server "stalls" its acquisition of the lock (i.e. execution of its acquire() method at the remote server) until the sequence number of said request matches the next sequence number allowed to proceed. Once that *acquire* operation is complete, the protocol proceeds as in Figure 5.

V. Evaluation Methodology

Experimental setup—Measurements: To concretely study event reorderings in Smart Home environments,we introduce

components for the client and physical device that enable us to automate event triggering and collect information about messages sent by the cloud backend.

Typically, a mobile application client sends a message to the cloud to trigger an event. To remove user involvement and automate experiments, we exploit part of the SmartThings API that creates an endpoint that is accessible over the web. HTTP requests sent to these endpoints trigger event handler executions just like messages sent from a mobile application to the cloud. A computer generates endpoint requests in order to automate event triggering and therefore simulate a client. We use the same methodology to interface with a 3-node Cassandra cluster for our consistency experiment, but we replace the SmartThings API with the Cassandra API.

Since the cloud backend and hub are proprietary on SmartThings, we cannot add instrumentation to these parts of the system, we must use them as black boxes. One of OKAPI's contributions is that we can overlay needed atomicity and ordering functionality over an IoT cloud platform that does not offer these capabilities.

To track message order, messages sent by our client are tagged with sequence numbers that propagate across the system. The cloud generates sequence numbers for each message it receives and sends a reply message to the client. The OKAPI synchronization service is a Go HTTP server which responds to requests providing access to the critical section of the Smart Home application. The HTTP server handles HTTP requests concurrently and is able to provide atomicity locally using mutexes. Given atomicity, one of the concurrent HTTP requests that are not processed gets hold of the mutex and participates in the two-phase protocol with the Smart Home application. The remaining concurrent requests are blocking, waiting to get hold of the mutex.

OKAPI runs on a Ubuntu 14.04 LTS virtual machine, with 1 CPU Core, 30 GB SSD storage and 2GB memory and is hosted on Linode [23]. Utilizing this OKAPI synchronization server for the evaluation incurs an additional latency, corresponding to the round trip time between the SmartApps and the OKAPI server. This additional latency is reflected in the results.

Latency & Throughput Evaluation: We evaluate OKAPI request latency by measuring from the time a request was generated until the Smart Home application responds. In addition, we measure the throughput in events processed per second. For our testing purposes, a client sends requests to the Smart Home application at a controlled rate. We vary the request generation rate to test the overhead of OKAPI under both low and high contention for the critical section. The critical section remains constant and short-lived across our experiments and consists of test code as presented in Table II. We compare an unmodified version of a test application (denoted as Unmodified) with a version that enforces atomicity (denoted as Atomic) and a version that enforces both atomicity and ordering (denoted as Atomic + Ordering).

VI. Evaluation

With respect to correctness, when the Atomic approach is used, no race conditions exist, and when the Atomic + Ordering approach is used, no race conditions or reorderings occur. This section evaluates the performance of OKAPI's (i) Atomic and (ii) Atomic + Ordering approaches versus the Unmodified approach. We evaluate OKAPI in terms of the latency and throughput overhead.

Throughput overhead evaluation: Figure 7 presents the throughput in requests/s for a single client simulating a mobile application across different rates of incoming requests. The

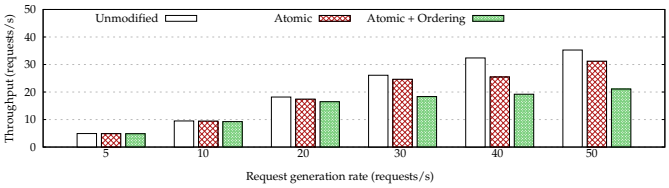

Fig. 7. Presentation of the average throughput of our test SmartThings application across three methods. We compare the (i) Unmodified (ii) Atomic and (iii) Atomic + Ordering approaches. Results are presented in increasing request generation rates. Higher throughput is better.

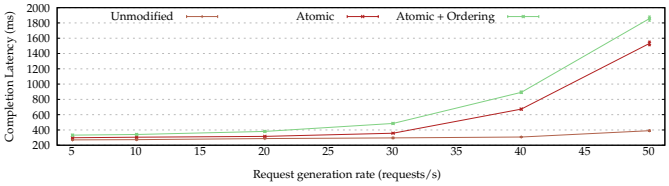

Fig. 8. Presentation of the average request completion latency of our test SmartThings application across three methods. We compare the (i) Unmodified (ii) Atomic and (iii) Atomic + Ordering approaches. Results are presented in increasing request generation rates. Lower latency is better.

overhead for request generation rate between 5-10 requests/s is 0.4%-0.7% and 0.8-1.6% for the Atomic and the Atomic + Ordering respectively. When the number of requests generated increases, the number of executions concurrently performing *acquire* to obtain access to the critical section increases as well. Reorderings cause the OKAPI synchronization server to delay responses to out of order requests, thus reducing throughput. Figure 4 presents the frequency of reorderings as a function of the event generation rate. The overhead of OKAPI is 11.5% for the Atomic approach and 40.2% for the Atomic + Ordering approach compared to Unmodified.

Latency overhead evaluation: Figure 8 presents the completion latency (ms) for requests generated by a single client across different request generation rates. Both OKAPI approaches use an external synchronization service, which adds one more hop in the network and increases the request completion latency. The overhead for the Atomic and Atomic + Ordering approaches increases when the request generation rate increases, demonstrating a linear behavior for rates below 30 requests/s and remaining below 21.0% and 64.1% respectively. The communication latency between the SmartApp and the OKAPI server is included in these results. Beyond that request generation rate, the latencies increase significantly and application developers need to account for the responsiveness of their smart home applications. Throughout our experiments, there are no timeouts for the request generation rates we evaluated. However, when we increased request generation rate to 100 requests/s in order to stress test the Atomic + Ordering approach, 65.6% of requests time out causing the average latency to increase significantly.

Discussion: Typical smart home system workloads don't typically exceed 2 requests/s [24], even while considering scenarios with multiple users performing multiple requests. In addition, latency values between 300-600ms are typical from the moment the event is triggered until an event notification reaches the application. To stress the system, we performed our evaluation using request rates higher than 5 requests/s (300 requests/minute). The corresponding average latency for the request rates 5 requests/s and 10 requests/s is 298 ms and 305 ms for the Atomic and 332 ms and 340 ms for the Atomic + Ordering version of OKAPI respectively. Under these conditions, OKAPI enforces atomicity and ordering with minimal impact on throughput and with latency that is within

typical values for smart home systems.

VII. Related Work

Prior work has explored event-driven architectures, focusing on exposing race conditions [25], as well as security deficiencies [26]. While we also focus on event-driven platforms, our approach concentrates on atomicity and ordering and considers the whole Smart Home environment, including hubs, devices and the cloud platforms. Related work [27], [28] has investigated misuse of smart home application privileges and has created solutions for data protection and access control. Ensuring the correct execution of the individual devices as well as the interaction of these devices requires checking that applications are written correctly. In our work, we focus on the atomicity and ordering problems that arise from these platforms. The detailed study of security mechanisms and tools that analyze applications' permissions can help in statically analyzing Smart Home application code and identify and pinpoint code blocks with atomicity and ordering issues. [29] investigates the security of smart locks, presents attacks against them and performs a security analysis of consumer products. The authors build a solution to mitigate the vulnerabilities found and suggest the use of eventual consistency for systems that use smart locks as it offers a nice intersection between availability and consistency. OKAPI focuses on providing ordering of events and atomicity in Smart Home application execution, concluding that stronger guarantees are required from a consistency standpoint. In another line of work, [30], [31] investigate the use of happens-before graphs, adding new constraints that model which operations can occur before other actions. By creating these graphs, it is possible to detect what new ordering constraints are necessary to prevent these race conditions. The use of happens-before graphs has also been used by [32] to formalize the concurrency semantics of the Android programming model and to perform race detection on real Android applications. In [33], a systematic exploration of Kernel concurrency bugs is performed using schedule exploration. In addition the authors of [34] perform a study of concurrency bug characteristics in real systems. The authors of [35] provide a bug finding analysis on formally verified distributed systems and [36] leverages model checking to discover errors in particular schedules of events. Such techniques could help detect whether reads or writes to a distributed platform could trigger possible race conditions and facilitate a more fine grained use of OKAPI. This paper presents OKAPI, a framework that provides ordering and atomicity to Smart Home applications. OKAPI is platform independent and provides stronger consistency and ordering guarantees, even if cloud platforms hosting Smart Home applications have more relaxed guarantees. The framework utilizes an external synchronization service and a two phase protocol for communication between the Smart Home applications and the remote service to provide atomic execution and in order event processing.

VIII. Conclusion

This paper proposed and evaluated OKAPI, a platform-independent synchronization service that allows insertion of atomicity and ordering guarantees into smart home applications. Consistency deficiencies and event reorderings and race conditions are challenges for commercial smart home platforms. OKAPI offers a methodology towards eliminating them in smart home applications. OKAPI can be used as a service to provide atomicity and ordering for Smart Home applications without depending on cloud providers' design decisions.

IX. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1739674/1739701 and by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] M. Heller, "BlueBorne Bluetooth vulnerabilities affect billions of devices," http://searchsecurity.techtarget.com/news/450426377/BlueBorne-Bluetooth-vulnerabilities-affect-billions-of-devices.
[2] N. Lomas, "Update bricks smart locks preferred by Airbnb," https://techcrunch.com/2017/08/14/wifi-disabled/.
[3] S. Hilton, "Dyn Analysis Summary of Friday October 21 Attack," https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/.
[4] "Google Play App Store," https://play.google.com/store/apps.
[5] "Amazon Alexa webpage," https://developer.amazon.com/alexa.
[6] "Google Personal Assistant webpage," https://assistant.google.com.
[7] "Amazon web services," https://aws.amazon.com/.
[8] "Google Cloud Platform," https://cloud.google.com/.
[9] "Developing for Homekit," https://developer.apple.com/homekit/.
[10] "Samsung SmartThings Smart home Monitoring kit," https://www.smartthings.com/.
[11] "Vera Control Smart Home Hub," http://getvera.com/.
[12] "Vera forum," http://forum.micasaverde.com/.
[13] "Zigbee wireless mesh network technology," http://www.zigbee.org/.
[14] "Z-Wave wireless control technology," http://www.z-wave.com/.
[15] "Bluetooth, a global wireless standard for simple, secure connectivity," https://www.bluetooth.com/.
[16] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.
[17] W. Vogels, "Eventually Consistent," *Commun. ACM*, 2009.
[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563
[19] "SmartThings Architecture Documentation," http://docs.smartthings.com/en/latest/architecture/.
[20] "Apache Cassandra Database," http://cassandra.apache.org.
[21] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979. [Online]. Available: https://doi.org/10.1109/TC.1979.1675439
[22] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd, "Existential consistency: Measuring and understanding consistency at facebook," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 295–310. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815426
[23] "Linode Cloud Hosting," https://www.linode.com/.
[24] A. Kamilaris, V. Trifa, and A. Pitsillides, "Homeweb: An application framework for web-based smart homes," 2011.
[25] J. Davis, A. Thekumparampil, and D. Lee, "Node.fz: Fuzzing the server-side event-driven architecture," in *EuroSys '17*, 2017.
[26] J. Davis, G. Kildow, and D. Lee, "The case of the poisoned event handler: Weaknesses in the node.js event-driven architecture," in *EuroSec'17*, 2017.
[27] E. Fernandes, J. Jung, and A. Prakash, "Security Analysis of Emerging Smart Home Applications," in *S&P '16*, 2016.
[28] E. Fernandes, J. Paupore, A. Rahmati *et al.*, "FlowFence: Practical Data Protection for Emerging IoT Application Frameworks," in *Proceedings of the 25th USENIX Security Symposium*, August 2016.
[29] G. Ho, D. Leung, P. Mishra *et al.*, "Smart locks: Lessons for securing commodity internet of things devices," in *ASIA CCS '16*, 2016.
[30] D. Lustig, M. Pellauer, and M. Martonosi, "PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models," *MICRO*, 2014.
[31] M. Martonosi, "Check Research Tools and Papers."
[32] P. Bielik, V. Raychev, and M. Vechev, "Scalable race detection for android applications," *SIGPLAN Not.*, 2015.
[33] P. Fonseca, R. Rodrigues, and B. B. Brandenburg, "Ski: Exposing kernel concurrency bugs through systematic schedule exploration," in *OSDI '14*, 2014.
[34] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *ASPLOS '08*, 2008.
[35] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *EuroSys '17*, 2017.
[36] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev, "Stateless model checking of event-driven applications," *SIGPLAN Not.*, 2015.