NEW CONSISTENCY ORCHESTRATORS FOR EMERGING DISTRIBUTED SYSTEMS

BY

SHEGUFTA BAKHT AHSAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

ABSTRACT

We are gradually becoming more dependent on various distributed systems, e.g., smart home management systems, bank management, traffic monitoring, etc. Some run on datacenters while others run on edge devices, e.g., smart home applications. In all these environments, efficiently maintaining consistency across such a large number of components is a hard challenge.

Consistency ensures a coherent view across the disparate components of a distributed system. An inconsistent view might lead to various issues that directly impact user experience. For example, in a database management system, an inconsistent view of a primary replica might make the system slow, or show stale data to users. If a bank account has two primary-replicas, each of them might show their own version of the account balance – this leads to incorrect banking transactions. Similarly, in a smart home, failing to isolate concurrent routines (sequence of commands) might end the home in a state not consistent with the user's expectation. E.g., the outcome of two concurrent routines $R_1 = \{\text{Turn all light } \texttt{ON}\}$ and $R_2 = \{\text{Turn all light } \texttt{OFF}\}$ might end up in a state where some of the lights are `ON` while others are `OFF`.

Addressing this requires the disparate components of a distributed system to have coordination. Such coordination includes maintaining a consistent view of the failed nodes, leader election, consistent primary replica selection, coherence in smart home's current state, etc. *Orchestrators* are dedicated entities that use specialized protocols (e.g., Chubby, ZooKeeper etc.) to help coordinate the components.

Distributed systems can use i) generic *external orchestrators* such as Zookeeper, Chubby etc., or 2) build their own *internal orchestrator*. Unlike external orchestrators, internal orchestrators avoid external dependencies and are flexible and modifiable, e.g., making it relatively easier to provide complex consistency guarantees and providing consistent and reliable distributed data-structures.

In this thesis *we present new internal orchestrators for maintaining consistency in both edge-based and cloud-based distributed systems.* This thesis has the following contributions: for edge-based distributed systems, we develop a smart home orchestrator called *SafeHome* that offers a congruent end state by guaranteeing stronger properties, e.g., Isolation, Atomicity, Safety. This is an improvement over the best-effort philosophy used in today's smart homes which leads to incongruent states. Second, For cloud-based distributed systems, we reveal and analyze Service Fabric's consistent and scalable failure detector, which is the heart of its internal orchestration mechanisms. Third, we present a new way to decentralize Service Fabric's arbitration technique and design a consistent failure detector on top of it, which offers identical consistency guaranties as Service Fabric's centralized scheme. We also provide formal proof of correctness and time-bound for both central and distributed arbitrator schemes.

I dedicate my dissertation to my parents, sisters, and my beloved wife. Their eternal love, trust, and support kept me motivated throughout the long journey.

I would also like to dedicate my dissertation to the countless doctors, health workers, and volunteers who are selflessly working and risking their lives to control the global pandemic of the coronavirus. They are the real heroes!

ACKNOWLEDGMENTS

Fast and foremost, I would like to express my deepest gratitude to my advisor Professor Indranil Gupta for his invaluable guidance, support, and encouragement throughout the entire journey. I am also indebted to my previous advisor Professor Nitin Vaidya. Their constant supervision helped me to keep myself on the right track. They trusted me and gave me the freedom to explore my own research path, which helped a lot to grow myself as a confident and independent researcher.

I would like to sincerely thank the rest of my committee members, Professor Klara Nahrstedt, Professor Nikita Borisov, and Dr. Nitin Agrawal, for their invaluable feedback that played a big part in shaping my work in its current form.

I would also like to express my sincere gratitude to my current and former colleagues in the Computer Science Department with whom I spent several wonderful graduate years. In particular, I am grateful to Shadi Abdollahian Noghabi, Muntasir Raihan Rahman, Imranul Hoque, Anupam Das, Mainak Ghosh, Beomyeol Jeon, Le Xu, Rui Yang, Faria Kalim, Faraz Faghri, Sandeep Dasgupta, Güliz Seray Tuncay, Nirupam Roy, and Sanorita Dey. I am forever indebted to them for their help, guidance, and support.

I wish to show my gratitude to the excellent mentors whom I met during my internship at Huawei and Microsoft Azure. In particular, George Calcev, Lin Cai, Anmol Ahuja, Rishi Sinha, Mohammad Tanviruzzaman, Preetha Subbarayalu, Mert Coskun, and Gopal Kakivaya.

The current and former members of the Bangladeshi community here at University of Illinois at Urbana-Champaign have been a source of great joy, encouragement, and mental support for me. I am especially grateful to Mazhar Islam, Tanzila Alam, Reaz Mohiuddin, Tuba Yasmin, Ahsanur Rahman, Dulchi Fatema, Sajjadur Rahman, Saraf Tarannum, Priyanka Sarker, Anupam Aich, Tanvir Amin, Himel Dev, Songjukta Datta, Ahmed Khurshid, Ibtesam Nazim Ahmed, Abul Hassan Samee, Tanjida Kabir Choity, Mehedi Bakht, Shakil Bin Kashem, Nusrat Jahan Kazi, Kibria Roman, Nargis Akhter Shimu, Shahneela Chowdhury, Dr. Taher Saif, Nadeem Ahmed, Shahana Begum, Mohammad Sharif Ullah, and Rani Rahman.

Finally, I could not complete this difficult journey without the unconditional love and support of my family. I am indebted to my lovely and caring wife, Syeda Persia Aziz. She is one of the main reasons for my survival of the long Ph.D. journey. She was beside me during the entire time and guided me as a trusted friend. She kept me motivated when things were not going well. I am forever grateful for the inspiration my father Shegufta Bakht Mahmud has been– he is why I pursued the Ph.D. The endless effort and care of my mother Selina Akter helped me to become who I am. My elder sister Srabantee Shegufta and my younger sister Haimantee Shegufta are the most wonderful and supporting siblings one can ask for. I am also indebted to my father-in-law Syed Mahmudul Aziz, mother-in-law Afroza Rupa and brother-in-law Syed Arman for trusting me and encouraging me throughout the entire journey.

TABLE OF CONTENTS

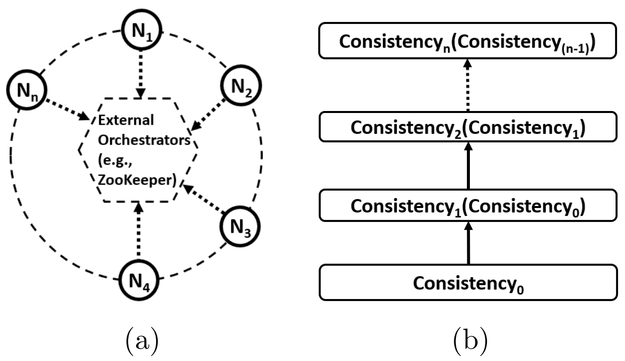
CHAPTER 1: INTRODUCTION

Distributed systems are the key to build highly scalable, widely available, fault-tolerant applications, e.g., stock management, flight tracking, smart home management, etc. These applications are quite diverse. For example, some services run on local edge nodes (e.g. various Internet of Things deployments, Smart Homes, etc.), whereas some services, e.g., Amazon Web Service (AWS), Microsoft Azure etc. span across globally-deployed datacenters.

For instance, in clouds, AWS is a broadly adopted platform, run across globally deployed data centers while offering over 175 fully-featured services and serving over millions of customers. Microsoft Service Fabric [1] is a widely adopted cloud-based *microservice* [2] orchestrator that hosts Azure SQL DB [3], Cosmos DB [4], Skype [5] and many others. Today, Azure SQL DB itself hosts 1.82 Million DBs containing 3.48PB of data and runs on over 100K machines across multiple geo-distributed datacenters. The smart home market is also expected to grow from \$27B to \$150B by 2024 [6, 7]. Today's smart home contains a wide diversity of devices–there are roughly $1,500$ IoT vendors [8], and the average home will contain over 50 smart devices by 2023 [9]. Smart devices cover all aspects of the home, from safety (fire alarms, sensors, cameras), to doors+windows (e.g., automated shades), home+kitchen gadgets, HVAC+thermostats, lighting, garden sprinkler systems, home security, and others. There are a plethora of smart home management apps [10, 11, 12, 13, 14, 15, 16, 17] to manage such a diverse set of devices.

Consistency is an essential element for coordinating both these classes of distributed systems. This means ensuring a coherent view of various properties (e.g., failed member, an elected leader, primary replica, etc.) across the group members. In the case of a microservice orchestrator (e.g., Service Fabric [1]), once a leader crashes, failing to maintain a consistent view of the new leader might mislead the existing microservices where some microservices might communicate with the old leader whereas others proceed with the new leader. Such a split-brain problem [18] hurts the performance and correctness of the system: e.g., in case of a database management system, multiple leaders can simultaneously elect different replicas as the *primary replica* for the same set of keys. In case of a bank account, such multiple primary-replicas might represent their own version of the account balance – which might cause incorrect transaction that negatively impacts user's expectation.

Similarly, in a smart home, the current best-effort approach used to handle concurrent Routines (a sequence of commands) might leave the smart home in an incongruent state. For example, if two routines $R_1 = \{$Light-1:`ON`, Light-2:`ON`, Light-3:`ON` $\}$ and $R_2 = \{$Light-1:`OFF`, Light-2:`OFF`, Light-3:`OFF` } run simultaneously, the outcome might be an incongruent state: some of these three lights are `ON`, while others remain `OFF`. This might not be consistent with user's expectation (either all lights `ON` or all `OFF`). Therefore, these distributed systems need to guarantee consistency in order to ensure correctness, performance and seamless user experience. In this thesis *we present new internal orchestrators for maintaining consistency in such distributed systems.*



N_1
N_2
N_3
N_4
N_n
External Orchestrators (e.g., ZooKeeper)
$\text{Consistency}_n(\text{Consistency}_{(n-1)})$
$\text{Consistency}_2(\text{Consistency}_1)$
$\text{Consistency}_1(\text{Consistency}_0)$
Consistency_0
(a)
(b)

Figure 1.1: (a) External orchestrator based approach. (b) Internal orchestrator based approach. The boxes represent the layers of a distributed systems. The built-in orchestrator ensures the lowermost layers consistency ($Consistency_0$). The n^{th} layer uses $(n-1)^{th}$ layer's consistency guaranties to form its own consistency properties.

Orchestrators are dedicated entities that use specialized protocols to help coordinating the disparate components of a distributed system. Despite being a conceptually centralized entity, orchestrators are typically deployed on multiple coordinating nodes to ensure fault tolerance. Distributed systems can use i) generic *external orchestrators* such as Zookeeper, Chubby etc., or 2) build their own *internal orchestrator.*

Such generic external orchestrators are deployed and maintained separately. They are attractive since the disparate components of a distributed system use well-defined APIs to communicate and sync with the orchestrators. These generic orchestrators are widely used and well tested.

Unlike external orchestrators, internal orchestrators are an inherent part of the distributed system that avoid external dependencies and provide the abstraction of an inherently consistent system (Fig. 1.1b). They are more flexible and modifiable than their external counterpart, which makes it relatively easier to provide complex consistency guarantees and to build consistent and reliable distributed data-structures (such as a distributed dictionary, queue, etc. [19]).

We focus on internal orchestrators in this thesis. The rest of this chapter describes our proposed internal orchestration approaches applicable to both cloud-based and edge-based distributed systems (Table 1.1).

System	**Scale**	**Consistency mechanism**
SafeHome [20] (Chap. 2)	Edge, home deployment	Ensures a safe and congruent end state of smart-devices using a consistent orchestrator.
Service Fabric (SF) [1] (Chap. 3)	Global, across multiple datacenters	Unveil, and analyze the internal orchestrator that relies on a consistent failure detector.
Distributed Arbitrator based Failure Detector [21] (Chap. 4)	Global, across multiple datacenters	Completely decentralize SF's internal orchestrator.

Table 1.1: Proposed consistency mechanisms across different type of distributed systems.

1.1 SAFEHOME: INTRODUCING SAFETY SCHEMES AND VISIBILITY MODELS FOR SMART HOMES

Current smart homes support a broad spectrum of home automation. They offer both local and remote controllability of smart devices. Users control a smart device using *commands* (e.g., turn `ON` a light). *Routines*, consists of a sequence of commands [22, 23, 24, 25], add a new dimension towards the home automation.

Routines, which are becoming an essential part of modern smart homes, are needed for both convenience (e.g., turn `ON` group of Living Room lights, then switch on entertainment system), and for correct operation (e.g., `close` window, then turn `ON` AC, then set to 70°). However, today's *best-effort* way of executing routines can lead to incongruent states in the smart home and has been documented as the cause of many smart home incidents [26, 27, 28, 29, 24].

The following simple scenarios illustrate how concurrent routines running in a best-effort method may end-up in an incongruent state:

First, consider a "movie-time" routine R_0={TV: `ON`; Sound System: `ON`; Living-room Light: `OFF` }. During the execution, if at least one of the commands fails, the end-state of the smart home might vary from user's expectation. Either all or no commands of the routine should be executed. In other words *atomicity* of the routine is not being ensured.

Second, consider two routines – R_1 that turns `ON` all lights in the living room, and R_2 that turns `OFF` all lights in the living room. Today if two different users run R_1 and R_2 concurrently, the outcome could be an incongruent end-state, with some lights on and some lights off. In other words, *isolation semantics* among concurrent routines are not being specified or enforced cleanly.

Third, consider a "prepare-food" routine – R_3={Exhaust Fan: `ON` ; Stove: `ON` } where the user expects the exhaust fan to be turned `ON` as long as the oven is running. If the exhaust-fan fails during the operation, the current best-effort approach does not offer any preventive mechanism. Sometimes this might lead to a safety hazard. In other words, the current best-effort approach does not *verify inter-device dependencies*.

The lack of atomicity, isolation, and safety that are inherent parts of today's best-effort strategy makes this approach inadequate to serve the demands of today's modern smart homes. Lacking these properties might leave the smart-home in an incongruent state, which might cause user inconvenience. Even worse, this might lead to safety hazards. Therefore, today's smart homes require a mechanism to handle concurrent routines safely.

Routines are akin to Transactions in the database world. Therefore, incorporating the existing well-studied transaction management systems is the first remedy that should naturally come into one's mind. However, Routines' discrete execution strategy separates them from that of Transaction.

SafeHome is best seen as the first step towards a grand challenge. A true OS for smart homes requires tackling myriad problems well beyond what SafeHome currently does. These include support for: users to inject signals/interrupts/exceptions, safety property specification and satisfaction, leveraging programming language and verification techniques, and in general full ACID-like properties [30]. SafeHome is an important building block over which (we believe) these other important problems can then be addressed.

1.1.1 Problem Statement and Challenges

This thesis focuses on providing atomicity, isolation, and safety for smart homes that are running concurrent routines across disparate smart devices.

Providing such guarantees across concurrent routines face unique challenges not found in other domains that ensure similar guarantees. *First*, every action of a routine is *immediately visible to the human user(s)*. This requires us to clearly reason about *visibility models for concurrent routines* in a smart home. Visibility models provide notions of serial equivalence (i.e., serializability) of routines. A smart home needs to optimize *user-facing metrics* especially latency to start the routine, and then latency to execute it.

Second, in a smart home, *device crashes and restarts are the norm* – any device can fail at any time, and possibly recover later. These failure/recovery events may occur during a command, or before a command starts, or after a command has completed. Thus, in a smart home, *reasoning about device failure/restart events that occur alongside concurrent routines*, is a new challenge.

Third, *long-running routines are common* in smart homes, e.g., a routine containing a command to preheat an oven to $400°F$), or to run north garden sprinklers for 15 minutes.

Long-running routines may hog devices, preventing other routines from starting or making progress. This creates a need for *correct and efficient resource sharing techniques* which reduce latency, without violating visibility properties.

1.1.2 Contributions

In this thesis we argue that, to ensure a congruent state, smart homes should provide the two fundamental properties used in database systems: i) **Atomicity:** once a routine has started, either all its commands have the desired effect on the smart home (i.e., routine completes), or the system aborts the routine, resulting in a rollback of its commands. ii) **Isolation:** effect of the concurrent execution of a set of routines is identical to an equivalent world where the same routines all executed serially, in some order. Additionally, it should monitor device-dependencies to ensure safety. We propose new visibility models that trade off responsiveness vs. temporary congruence of smart home state. We also propose a new way to reason about failures by serializing *failure events* and *restart events* into the serially-equivalent order of routines. Our scheme introduces a new pre/post lease based locking mechanism that enhance the concurrency without affecting the correctness. We also develop SafeHome, a smart-home orchestrator (2K lines of Java code) that runs on raspberry-Pi and integrate popular TP-Link devices [31].

1.1.3 Key Techniques

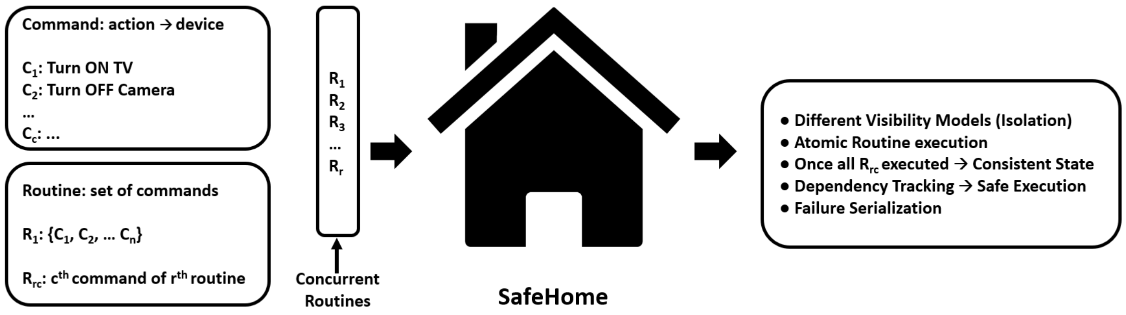

Command: action → device
C_1: Turn ON TV
C_2: Turn OFF Camera
...
C_c: ...
Routine: set of commands
R_1: {C_1, C_2, ... C_n}
R_{rc}: cth command of rth routine
R_1
R_2
R_3
...
R_r
Concurrent Routines
SafeHome
• Different Visibility Models (Isolation)
• Atomic Routine execution
• Once all R_{rc} executed → Consistent State
• Dependency Tracking → Safe Execution
• Failure Serialization

Figure 1.2: SafeHome [20] goals.

SafeHome [20] proposes a set of visibility models that provide different level of Isolation, guarantees Atomicity and ensures safe routine execution (Fig. 1.2). SafeHome is: i) the first implementation of relaxed visibility models for smart homes running concurrent routines, and ii) the first system that reasons about failures alongside concurrent routines. This work is currently under submission.

In our framework called SafeHome (Fig. 1.2), we introduce a new *Lineage Table* data-structure that applies an unique safe *lock-leasing* technique among the conflicting devices and thus maximizes concurrency while also maintains the atomicity and isolates the final outcomes (as per different visibility model requirements).

1.2 MICROSOFT SERVICE FABRIC– IN SEARCH OF AN INTERNAL ORCHESTRATOR

In this work, we reveal Microsoft Service Fabric's (SF) [1] internal orchestration strategy that ensures SF's inherent consistency guarantees (e.g., failure detection, leader election, primary replica selection etc.). Existing microservice frameworks (e.g., Akka [32], Bluemix [33], Nirmata [34] etc.) rely on external orchestrators. SF is the only microservice framework that supports *state-full microservice.* This uniqueness stems from its *internal orchestrator* that maintains an inherently consistent framework. It is easier to build and maintain consistent and reliable distributed data-structures (such as a distributed dictionary, queue, etc. [19]) on top of such consistent framework. Such data-structures are key to building stateful microservices.

A consistent failure detector is the core of SF's internal orchestration technique. SF runs across geo-distributed datacenters and consists of thousands of nodes. Therefore, the orchestrator used in SF requires a failure detector that is both consistent and scalable to geo-distributed datacenters. Such a failure detector is crucial to achieving Service Fabric's claimed guarantees efficiently.

1.2.1 Problem Statement and Challenges

This thesis reveals and analyzes Service Fabric, and in particular its consistent and scalable failure detector, which ensures a time-bounded consistency.

Service Fabric uses the consistent failure detector to develop a consistent membership protocol. This protocol ensures a timing guarantee: if node P fails, all of its existing group members consistently mark P as failed within a given time bound. Therefore, after waiting for that time-span, the resource and responsibilities hosted on P can be safely moved to other healthy nodes.

Building a consistent while efficient and scalable failure detector is a non-trivial task. Current failure detectors mostly sit in two extreme endpoints of the consistency-scalability spectrum. One end consists of the Gossip-style failure detection services [35] such as SWIM/Serf [36, 37], those are highly scalable, but weakly consistent and thus not the best choice for applications that require a higher form of guaranties (such as transaction). The other end consists of protocols that provide stronger consistent membership like virtual synchrony [38, 39]. However, these protocols do not scale to datacenters.

Another popular approach of maintaining a consistent view (e.g. consistent failure detection, leader election, primary replica selection, etc.) is to offload such jobs to external services, e.g. Zookeeper [40]. However, as explained earlier, such external services act as an additional layer, which increases the latency and might create a central point of failure.

To ensure efficiency, the internal orchestrator used in Service Fabric needs a consistent failure detector that sits around the middle of the consistency-scalability spectrum: provides strong consistency while also scalable to datacenters.

1.2.2 Contributions

In this thesis we reveal the internal orchestrator used in Microsoft Service Fabric (SF) [1]. SF relies on *ground-up consistency*, a new approach to ensure consistency that stems from the novel and unique *consistent failure detector*. This failure detector is both consistent and scalable to geo-distributed datacenter. SF has been in service for over 15 years and supports major Microsoft systems such as Azure hosts SQL DB [3], Cosmos DB [4], Skype [5] and many others. We are the first to explore and measure SF's unique internal orchestrator based approach and reveal it to the outside world. This work has been published in *EuroSys 2018*.

1.2.3 Key Techniques

One prevalent philosophy for building consistent and fault-tolerant applications is to develop them atop inconsistent components (Callas [41], Yesquel [42], Tapir [43] etc.). Popular systems such as Akka [32], HBase [44], Kafka [45], Kubernetes [46] etc. often outsource their consistency needs to external modules (e.g. Chubby [47], Zookeeper [40] etc., Fig. 1.3a).

However, Service Fabric follows a different technique where SF's lower most layer forms an internal orchestrator that relies on a unique consistent failure detector. This failure detector relies on a novel *Arbitrator Group* based approach. In SF, the inherent consistency stems from this internal orchestrator, and is propagated to the upper layers (Fig. 1.3b). Therefore, instead of outsourcing, SF's well designed and consistent layers internally solve hard distributed computing problems related to failure detection, consistency, leader election, failover, manageability etc.

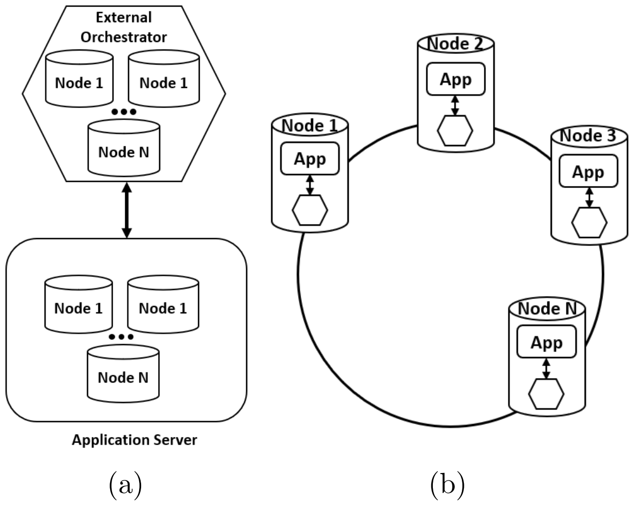


Figure 1.3: Different approaches of building consistent applications: a) Outsource consistency module to external orchestrators (e.g. Zookeeper [40]) etc. b) Build an internal orchestrator at the lower most layer and develop application on top of it (e.g. Service Fabric [1]).

1.3 DECENTRALIZING SERVICE FABRIC'S CENTRALIZED ARBITRATOR BASED FAILURE DETECTOR

This section briefly describes our work on decentralizing SF's centralize arbitration scheme.

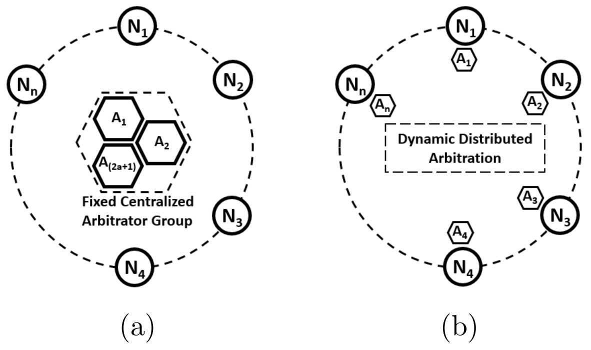


Figure 1.4: (a) Service Fabric's fixed centralized arbitrator group based *Consistent Failure Detector*. The arbitrator nodes A_1 to $A_{(2a+1)}$ forms the arbitrator group. (b) Introducing decentralized dynamic arbitration. each node N_n also acts as an arbitrator node A_n.

Service Fabric's unique consistent failure detector (the core of the internal orchestrator) relies on a *fixed set* of special nodes (called the *arbitrator group*, Fig. 1.4a). This centralized *arbitrator group* can become the single point of failure. Besides, this unique group might become a bottle-neck as it has to monitor and orchestrate the entire system. To avoid the dependency upon SF's centralized arbitrator group, the responsibilities assigned to it need to be distributed across the ring members (Fig. 1.4b).

1.3.1 Problem Statement and Challenges

In this thesis, we decentralize Service Fabric's centralized arbitrator scheme while ensuring similar consistency properties. We also propose a coherent node join protocol for the new scheme. We present theoretical analyses for both central and distributed arbitrator schemes.

Efficiently decentralizing such arbitrator group while still ensuring the same consistency guarantee is a challenge. Such distribution requires each pair of neighbouring nodes (P, Q) to form their own arbitrator sets (A_{PQ} and A_{QP}, respectively). Without loss of generality, if node P suspects node Q as failed, it consults with the arbitrator group A_{PQ}. To ensure correctness, this two distributed arbitrator groups A_{PQ} and A_{QP} must be consistent. Maintaining consistency across such symmetric arbitrator group is a challenge. Also, in this approach, the arbitrator group consists of neighboring nodes, which requires a new node join/leave protocol that maintains the arbitrator consistency.

1.3.2 Contributions

We devise a novel approach (called Fully Decentralized Arbitrator Based Failure Detector, Fig. 1.4b) that does not rely on the fixed arbitrator group but still provides the same consistency-guaranties. In the same work, we also provide the formal proof of correctness for both Service Fabric's Fixed Arbitrator based failure detector and our proposed Dynamic Arbitrator based failure detector. We have also designed an efficient node join protocol that works coherently with our Dynamic Arbitrator based failure detector scheme.

Besides showing the empirical analysis, we also provide formal proof of correctness and time-bound of both the Service Fabric's *centralized* fixed arbitrator based failure detector and our proposed *decentralized* dynamic arbitrator based failure detector. This work has been published in *InfoCom 2020*.

1.3.3 Key Techniques

- *Introducing Dynamic Distributed Arbitrator Group:* In this approach, nodes are arranged in a virtual ring. Instead of relying on a centralized arbitrator group, each neighbouring pair of nodes form their own local *arbitrator group*. Such pair of nodes need to maintain a consistent view of the *arbitrator groups*. Our proposed novel arbitrator hand-off strategy dynamically updates the arbitrator group while also guarantees the consistent view.

- *An efficient node join protocol:* We also propose an efficient node-join mechanism that complies with the dynamic arbitrator group.

1.4 THESIS ORGANIZATION

The rest of the thesis is organized as follows. Chap. 2 introduces safety schemes and visibility models for smart homes. Chap. 3 describes the design details of Service Fabric, where we mainly focus on its centralized arbitration based unique consistent failure detection scheme. Chap. 4 sketches the totally distributed arbitrator based failure detector, an improvement over the Service Fabric's centralized arbitration based failure detector. Chap. 5 shares the lesson learned throughout the three projects. Finally, we conclude by presenting our future directions in Chap. 6.

CHAPTER 2: HOME, SAFEHOME: SMART HOME RELIABILITY WITH VISIBILITY AND ATOMICITY

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

This chapter is organized as follows: Sec. 2.1 introduces and motivates the problem, Sec. 2.2 describes different visibility models, Sec. 2.3 introduces failure handling across different visibility models, Sec. 2.4 unwraps the Eventual Visibility model, Sec. 2.5 explores different scheduling policies for the Eventual Visibility model, Sec. 2.6 describes the implementation of SafeHome, Sec. 2.7 evaluates the SafeHome framework with a number of experimental results, Sec. 2.8 analyze the state-of-art related works and finally Sec. 2.9 concludes the project.

2.1 INTRODUCTION

The disruptive smart home market is projected to grow from $27B to $150B by 2024 [6, 7]. There is a wide diversity of devices—roughly 1,500 IoT vendors today [8], with the average home expected to contain over 50 smart devices by 2023 [9]. Smart devices cover all aspects of the home, from safety (fire alarms, sensors, cameras), to doors+windows (e.g., automated shades), home+kitchen gadgets, HVAC+thermostats, lighting, garden sprinkler systems, home security, and others. As the devices in the home increase in number and complexity, the chances of interactions leading to undesirable outcomes become greater. This diversity and scale is even vaster in other smart environments such as smart buildings, smart factories (e.g., Industry 4.0 [48]), and smart hospitals [49].

Past computing eras—1970s' mainframes, 1990s' clusters, and 2000s' clouds—were successful because of good management systems [50]. What is desperately needed are systems that allow a group of users to manage their smart home as a single entity rather than a collection of individual devices [51]. Today, most users (whether in a smart home or a smart factory) control a device using *commands*, e.g., turn `ON` a light. Further, major smart home controllers have started to provide users the ability to create *routines*. A routine is a sequence of commands [22, 23, 24, 25]. Routines are useful for both: a) convenience, e.g., turn `ON` a group of Living Room lights, then switch on the entertainment system, and b) correct operation, e.g., `CLOSE` window, then turn `ON` AC.

Motivating Examples: Today's *best-effort* way of executing routines can lead to incongruent states in the smart home, and has been documented as the cause of many smart home incidents [26, 27, 28, 29, 24] [1].

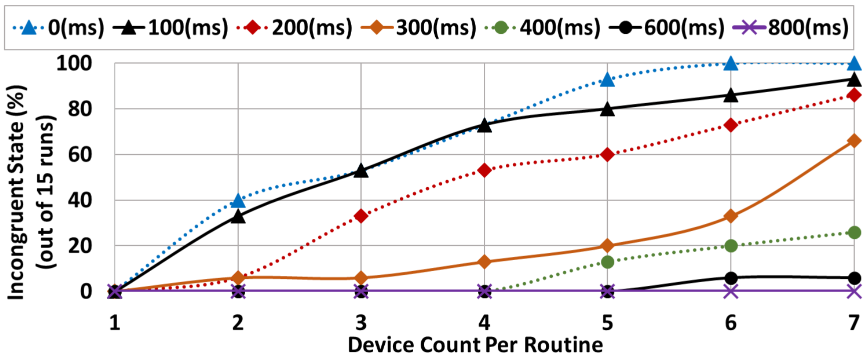


Figure 2.1: **Concurrency causes Incongruent End-state in a real smart home deployment.** *Two routines R1 (turn* `ON` *all lights) and R2 (turn* `OFF` *all lights) executed on a varying number of devices (x axis), with routine R2 starting a little after R1 (different lines). Y axis shows fraction of end states that are not serialized (i.e., all* `OFF`*, or all* `ON`*). Experiments with TP-Link smart devices [52].*

First, consider a routine involving the AC and a smart window [53, 54]: $R_{cooling} = \{$`CLOSE` window; switch `ON` AC$\}$. During the execution of this routine, if either the window or the AC fails, the end-state of the smart home will not be what the user desired—either leaving the window open and AC on (wasting energy), or the window closed and AC off (overheating the home). Another example is a shipping warehouse wherein a robot's routine needs to retrieve an item, package it, and attach an address label—all these actions are essential to ship the item correctly. In all these cases, lack of *atomicity* in the routine's execution violates the expected outcome.

Our next example deals with concurrent routines. Consider a timed routine R_{trash} that executes every Monday night at 11 pm and takes several minutes to run: R_{trash}=$\{$`OPEN` garage; `MOVE` trash can out to driveway (a robotic trash can like SmartCan [55]); `CLOSE` garage$\}$. One day the user goes to bed around 11 pm, when she initiates a routine: $R_{goodnight}$=$\{$switch

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

`OFF` all outside lights; `LOCK` outside doors; `CLOSE` garage}. Today's state of the art has no *isolation* between the two routines, which could result in $R_{goodnight}$ shutting the garage (its last command) while R_{trash} is either executing its first command (open garage), or its second command (moving trash can outside). In both cases, R_{trash}'s execution is incorrect, and equipment may be damaged (garage or trash can). Concurrency even among short routines could result in such incongruences—Figure 2.1 shows such an experiment. The plot shows that two routines simultaneously touching only a few devices cause incongruent outcomes if they start close to each other. In all these cases, *isolation semantics* among concurrent routines were not being specified cleanly or enforced.

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

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

Second, a smart home needs to optimize *user-facing metrics*—latency to start the routine, and also latency to execute it. This motivates us to explore a new spectrum of visibility models which trade off the amount of incongruence the user sees *during* execution vs. the user-perceived latency, all while guaranteeing serial-equivalence of the overall execution. Our visibility models are a counterpart to the rich legacy of weak consistency models that have been explored in mobile systems like Coda [56], databases like Bayou [57] and NoSQL [58], and shared memory multiprocessors [59].

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

Today's failure handling is either silent or places the burden of resolution on the user.

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

Prior Work: These challenges have been addressed only piecemeal in literature. Some systems [60, 61] use priority-based approaches to address concurrent device access. Others [62] propose mechanisms to handle failures. A few systems [63, 64, 65] formally verify procedures. Transactuation [24] and APEX [66] discuss atomicity and isolation, but their concrete techniques deal with routine dependencies and do not consider users' experience—nevertheless, their mechanisms can be used orthogonally with SafeHome. None of the above address atomicity, failures, and visibility together.

The reader may also notice parallels between our work and the ACID properties (Atomicity, Consistency, Isolation, and Durability) provided by transactional databases [67]. While other systems like TinyDB [68] have drawn parallels between networks of sensors and databases (DBs), the techniques for providing ACID in databases do not translate easily to smart homes. The primary reasons are: i) our need to optimize latency (DBs optimize throughput); ii) device failure (DB objects are replicated, but devices are not, by default); and iii) the presence of long-running routines.

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

The primary contributions of this paper are:

- A new spectrum of *Visibility Models* trading off responsiveness vs. temporary congruence of smart home state.
- Design and implementation of the SafeHome system.

- A new way to reason about failures by *serializing failure events and restart events into* the serially-equivalent order of routines.
- New *lock leasing* techniques to increase concurrency among routines, while guaranteeing isolation.
- Workload-driven experiments to evaluate new visibility models and characterize tradeoffs.

SafeHome is best seen as the first step towards a grand challenge. A true OS for smart homes requires tackling myriad problems well beyond what SafeHome currently does. These include support for: users to inject signals/interrupts/exceptions, safety property specification and satisfaction, leveraging programming language and verification techniques, and in general full ACID-like properties [30]. SafeHome is an important building block over which (we believe) these other important problems can then be addressed.

Assumptions:

SafeHome relies on the following assumptions:

- In SafeHome, commands are the indivisible executable entities. Commands are of two types: 1) **short command**– changes the state of a smart-device (e.g. `TV: Turn ON`) or 2) **long command**– changes the state of a smart-device and holds that state for a predefined amount of time (e.g. `Water-Sprinkler: Turn ON for` $\mathcal{T}_L = 15$ `minutes`). A routine consists of long command is referred to as a *long-running routine*.
- The long-running command duration $\mathcal{T}_L$ is specified while designing the routine. For the same smart-device, different routines might have different values of $\mathcal{T}_L$ (e.g., a *good morning* routine might define "`Speaker: play classical music for` $\mathcal{T}_L = 15$ `minutes`" while a *good night* routine might define "`Speaker: play rain sound for` $\mathcal{T}_L = 2$ `hours`"
- Command types and durations are known a priori.
- Routine consists of a set of commands which are executed sequentially.
- Routines are fixed when specified, and cannot be changed once submitted.
- Multiple routines cannot share devices (e.g. in R_1 a security camera is video footage while in R_2 another process is reading that footage in real time).

- Routines can be triggered by either user or automated event (light-sensor, motion sensor, etc.).
- Neither current smart homes, nor SafeHome supports interrupt, interception or pauses of routines. However, it has been considered as a part of the future work.
- Failures (and recoveries) of a device can occur at any time.
- The current version of SafeHome considers the fail-recovery model. Besides, this version does not handle byzantine failure.

2.2 VISIBILITY AND ATOMICITY

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

- **SafeHome-Visibility/Serializability:** For simplicity, in this initial part of the discussion we ignore failures, i.e., we assume devices are always up and responsive. SafeHome-Visibility/Serializability means the *effect* of the concurrent execution of a set of routines, is identical to an equivalent world where the same routines all executed serially, in some order. The interpretation of *effect* determines different flavors of visibility, e.g., identicality at every point of time, or in the end-state (after all routines complete), or at critical points in the execution. These choices determine the *spectrum* of visibility/serializability *models* that we will discuss soon.
- **SafeHome-Atomicity:** After a routine has started, either all its commands have the desired effect on the smart home (i.e., routine *completes*), or the system *aborts* the routine, resulting in a rollback of its commands, and gives the user feedback.

2.2.1 New Visibility Models in SafeHome

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

Today's default approach is to execute routines' commands as they arrive, as quickly as possible, without paying attention to serialization or visibility. We call this *status quo* model as the *Weak Visibility (WV)* model, and its incongruent end states worsen quickly with scale and concurrency (see Fig. 2.1). We introduce three new visibility models.

1. **Global Strict Visibility (GSV):** In this strong visibility model, *the smart home executes at most one routine at any time.* In our SafeHome-Visibility definition (Sec. 2.2), the *effect* for GSV is "at every point of time", i.e., every individual action on every device. Consider a 2-family home where one user starts a routine $R_{dishwash}$ ={`dishwasher:ON; (run dishwasher for 40 mins); dishwasher:OFF;`}, and another user simultaneously starts a second routine R_{dryer}= {`dryer:ON; (run dryer for 20 mins); dryer:OFF;`} . If the home has low amperage, switching on both dishwasher and dryer simultaneously may cause an outage (even though these 2 routines touch disjoint devices). If the home chooses GSV, then the execution of $R_{dishwash}$ and R_{dryer} are serialized, allowing at most one to execute at any point of time. Because routines need to wait until the smart home is "free", GSV results in very long latencies to start routines. A long-running routine also starves other routines.

2. **Partitioned Strict Visibility (PSV):** PSV is a weakened version of GSV that allows concurrent execution of non-conflicting routines, but limits conflicting routines to execute serially. For instance, for our earlier (GSV) example of $R_{dishwash}$ and R_{dryer} started simultaneously, if the home has no amperage restrictions, the users should choose PSV–this allows the two routines to run concurrently, and the end state of the home is (serially-)equivalent to the end state if the routines were instead to have been run sequentially (i.e., dishes are washed, clothes are dried). However if the two routines *were* to touch conflicting devices, PSV would execute them serially.

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

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

Consider the two users in a home simultaneously initiating the routine $R_{breakfast}$ = R={ `coffee:ON; /*make coffee for 4 mins*/; coffee:OFF; (wait for user to take coffee); pancake:ON; /*make pancakes for 5 mins*/; pancake:OFF; (wait for user to take pancake)` } . Both GSV and PSV would serially execute these routines. EV would be able to pipeline them, overlapping the pancake command of one routine with the coffee command of the other routine. EV only cares that at the end both users have their respective coffees and pancakes.

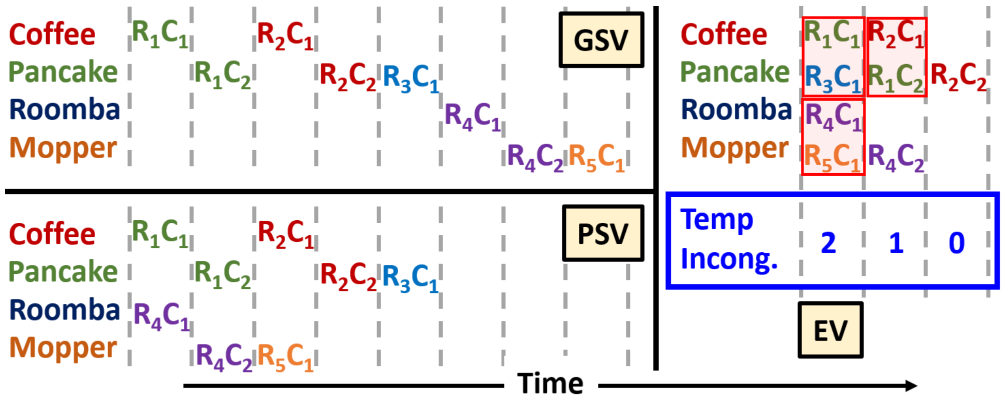


Figure 2.2: **Example routine execution in different visibility models:** *a) GSV b) PSV, c) EV. R_rC_c represents the c^{th} command of the r^{th} routine. In EV, red boxes show a pair of incongruent commands and the blue box shows the total number of temporary incongruencies.*

Common Example: 3 Visibility Models: Fig. 2.2 shows an example with 5 concurrent routines, executed for our three visibility models. This is the outcome of a real run of SafeHome running on a Raspberry Pi, over 5 devices connected via TP-Link HS-105 smart-plugs [69]. The routines are:

R_1: *makeCoffee(Espresso); makePancake(Vanilla);*

R_2: *makeCoffee(Americano); makePancake(Strawberry);*

R_3: *makePancake(Regular);*

R_4: *startRoomba(Living room); startMopping(Living room);*

R_5: *startMopping(Kitchen);*

GSV takes the longest execution time of 8 time units as it serializes execution. PSV reduces execution time to 5 time units by parallelizing unrelated commands, e.g., R_1's coffee command and R_4's Roomba command at time $t = 0$. EV is the fastest, finishing all routines by 3 time units. Average latencies (wait to start, wait to finish) are also fastest in EV, then PSV, then GSV. EV does exhibit "temporary incongruence"—shown are the number of devices whose intermediate state is not serially equivalent. EV guarantees a temporary incongruence of zero when the last routine finishes.

Table 2.1 contrasts the properties of the four visibility models. Table 2.2 summarizes examples discussed so far.

	GSV	**PSV**	**EV**	**WV**
Concurrency	At most one routine	Non-conflicting routines concurrent	Any routines concurrent	Any routines concurrent
End State	Serializable	Serializable	Serializable	Arbitrary
Wait Time	High	High for conflicting routines, low for non-conflicting routines	Low for all routines (modulo conflicts)	Low for all routines
User Visibility	Smart home congruent at all times	Smart home congruent at end, and at start/complete points of routines	Smart home congruent at end	Smart may be incongruent at any time and at end (Fig. 2.1)

Table 2.1: **Spectrum of Visibility Models in SafeHome.**

2.2.2 SafeHome-Atomicity

To remind the reader:

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

Due to the physical effects of smart home routines, we discuss three subtleties that are essential.

First, we allow the user to tag some commands as *best-effort*, i.e., optional—the routine is allowed complete successfully even if any best-effort commands fail. Other commands, tagged as *must*, are required for routine completion—if any *must* commands fail, the routine must abort. This tagging acknowledges the fact that users don't consider all commands within a routine to be equally important. A "leave-home-for-work" may contain commands that lock the door (must commands) and that turn off lights (best-effort commands)–even if the lights are unresponsive, the doors must still lock. The user receives feedback about failed such commands, and she may choose to initiate another routine to switch off lights.

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

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

2.3 FAILURE HANDLING AND VISIBILITY MODELS

Smart home devices could fail or become unresponsive, and then later restart. SafeHome needs to reason cleanly about failures or restarts that occur during the execution of concurrent routines [2]. We only consider fail-stop and fail-recovery models of failures of devices in the smart home [3].

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

Thus a key realization in SafeHome is that we need to *serialize failure events and restart events alongside routines themselves.* We can now restate the SafeHome-Atomicity property from Sec. 2.2, to account for failures and restarts:

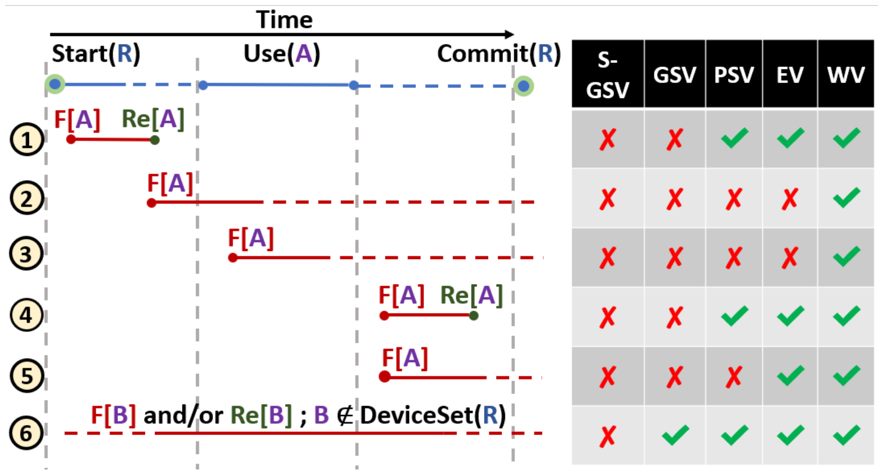
- **SafeHome-Visibility/Serializability (with Failures and Restarts):** The *effect* of the concurrent execution of a set of routines, occurring along with concurrent device failure events and device restart events, is identical to an equivalent world where the same routines, device failure events, and device restart events, all occur sequentially, in some order [4].

First, we define the failure/restart event to be the event when the edge device (running SafeHome) *detects* the failure/restart (this may be different from the actual time of failure/restart). Second, unlike routines–which may or may not appear in the final serialized order (if completed or aborted respectively)–failure events and restart events *must* appear in the final serialized order. Hence we reason explicitly about failure serialization for each of our visibility models from Sec. 2.2.1. Fig. 2.3 shows examples.

[2]Unlike transactional databases, where objects are always available due to replication, smart home devices have no replicas.

[3]Byzantine failures are beyond our scope.

[4]This idea has analogues to distributed systems abstractions such as view/virtual synchrony, wherein failures and multicasts are totally ordered [38, 70, 71]. Of course those are not applicable to the smart home.



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

Figure 2.3: **Failure Serialization: 6 cases, and their handling in Visibility Models.** ✓ *- execute routine,* ✗ *- abort routine. At F[A] /Re[A] the edge device detects the failure/restart (resp.) of device A.*

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

2. Failure Serialization in Global Strict Visibility: Because GSV intends to present the picture of a single serialized home to the user, if *any* device failure event or restart event were to occur while a routine is executing (between its start and finish), the routine must be aborted. There are two sub-flavors herein: (A) *Basic GSV or Loose GSV (GSV)*: Routine aborts only if it contains at least one command that touches failed/restarted device; (B) *Strong GSV (S-GSV)*: Routine aborts even if it does not have a command that touches failed/restarted device. A routine R on living room shades can complete, if master bathroom shades fail, in GSV but not S-GSV. In S-GSV, the final serialization order contains the failure/restart event but not the aborted routine R. In GSV, the final serialization order contains both R (which completes) and the failure/restart event, in arbitrary order.
3. Failure Serialization in Eventual Visibility: For a given set of routines (and concurrent failure events and restart events), the *eventual (final)* state of the actual execution is equivalent to the end state of a world wherein the final successful routines, failure device events, and failure restart events, all occurred in some serial order.

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

- If D is not touched by R, then D's failure event and/or restart event can be arbitrarily ordered w.r.t. R.

- If D's failure and restart events both occur *before R first touches the device*, then the failure and restart events are *serialized before R.*
- If D's failure event occurs *after the last touch of D by R*, then D's failure event (and eventual restart event) are *serialized after R.*
- In all other cases, routine R aborts due to D's failure. R does not appear in the final serialized order.

These are applicable to each concurrent routine accessing D.
4. Failure Serialization in Partitioned Strict Visibility: This is a modified version of EV where we change condition 3 (from 1-4 in EV above) to the following:
3.* If D's failure event occurs *after the last touch of D by R*, and *has recovered when R reaches its finish point*, then D's failure event and restart event are *serialized right after R.*
Examples: Table 2.2 summarizes several scenarios, and how SafeHome's features help and behave in each scenario.

2.4 EVENTUAL VISIBILITY: SAFEHOME DESIGN

In order to maintain correctness for Eventual Visibility (i.e., serial-equivalence), SafeHome requires routines to lock devices before accessing them. Because long routines can hold locks and block short routines, we introduce *lock leasing* across routines (Sec. 2.4.1). This information is stored in the *Locking Data-structure* (Sec. 2.4.2). The *lineage table* ensures invariants required to guarantee Eventual Visibility (Sec. 2.4.3).

2.4.1 Locks and Leasing

SafeHome prefers Pessimistic Concurrency Control (PCC): SafeHome adopts pessimistic concurrency control among routines, via (virtual) locking of devices. Abort and undo of routines are disruptive to the human experience, causing (at routine commit point) rollbacks of device states across the smart home. Our goal is to minimize abort/undo only to situations with device failures, and avoid aborts because routines touch conflicting devices. Hence we eschew optimistic concurrency control approaches and use locking [5].

SafeHome uses *virtual locking* wherein each device has a virtual lock (maintained at the edge device running SafeHome), which must be acquired by a routine before it can execute

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

Example Routines	**Scenario and Possible Behavior**	**SafeHome Feature**	
“cooling”=`{window:CLOSE; AC:ON;}`	If executed partially, can leave window open and AC on (wasting energy) or the window closed and AC off (overheating home).	Atomicity	
“make coffee” `{coffee:ON; /*make coffee for 4 mins*/ ; coffee:OFF;}`	Coffee maker should not be interrupted by another routine. E.g, user-1 invokes make coffee, and in the middle, user-2 independently invokes make coffee.	Long Running routines, Mutually Exclusive access to devices routines	
R_1=`{dishwasher:ON; (run dishwasher for 60 mins); dishwasher:OFF;}` R_2=`{dryer:ON; (run dryer for 80 mins); dryer:OFF;}`	If home has low amperage, simultaneously running two power-hungry devices may cause outage (GSV).	Global Strict Visibility (GSV)	
R_1= `{coffee:ON; /*make coffee for 4 mins*/; coffee:OFF;}` R_2=`{lights:ON, fan:ON}`	Two routines touching disjoint devices should not block each other (PSV).	Partitioned Strict Visibility (PSV), closest to [24]	
“breakfast”= R=`{coffee:ON; /*make coffee for 4 mins*/; coffee:OFF, pancake:ON; /*make pancakes for 5 mins*/; pancake:OFF;}`	Two users can invoke this same routine simultaneously. The two routines can be pipelined thus allowing some concurrency without affecting correctness (EV). (Both GSV and PSV would have serialized them.)	Eventual Visibility (EV)	
“leave home” `={lights:OFF (Best-Effort); door:LOCK;}`	Requiring all commands to finish too stringent, so only second command is Must (required). If light unresponsive, door must lock, otherwise routine aborts.	Must and Best-Effort commands	
`‘‘manufacturing pipeline’’ with k stages and` $\{R_1, R_2, ..., R_k\}$ `routines`	If any stage fails, entire pipeline must stop immediately.	Strong GSV serialization (S-GSV)	Failure Serialization
“cooling”=`{window:CLOSE; AC:ON;}`	If *anytime* during the routine (from start to finish), the AC fails or window fails, the routine is aborted.	Loose GSV serialization (GSV)	
“cooling”=`{window:CLOSE; AC:ON;}`	If window fails after its command *and* remains failed at finish point of routine, routine is aborted.	PSV serialization	
“cooling”=`{window:CLOSE; AC:ON;}`	If window fails after it is closed (but before AC is accessed), routine completes successfully–window failure can be serialized after routine.	EV serialization	

Table 2.2: **Example scenarios in a smart home, and SafeHome’s corresponding features.**

any command on that device. A routine’s lock acquisition and release do not require device access, and are not blocked by device failure/restart.

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

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

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

- **Pre-Lease:** R_{src} has started but has not yet accessed D. A lease at this point to R_{dst} is called a *pre-lease*, and places R_{dst} *ahead* of R_{src} in the serialization order. After R_{dst}'s last access of D, it returns the lock to R_{src}. If R_{src} reaches its first access of D before the lock is returned to it, R_{src} waits. After the lease ends, R_{src} can use the lock normally.

- **Post-Lease:** R_{src} is done accessing device D, but the routine itself has not finished yet. A lease at this point to R_{dst} is called a *post-lease*, and places R_{dst} *after* R_{src} in the serialization order. If R_{src} finishes before R_{dst}, the lock ownership is permanently transferred to R_{dst}. Otherwise, R_{dst} returns the lock when it finishes.

A prospective pre/post-lease is disallowed if a previous action (e.g., another lease) has already determined a serialization order between R_{src} and R_{dst} that would be contradicted by this prospective lease. In such cases R_{dst} needs to wait until R_{src}'s normal lock release. Further, a post-lease is not allowed if at least one device D is written by R_{src} and then read by R_{dst}. This prevents SafeHome from suffering dirty reads from aborted routines. We prevent scenarios like this–R_{src} switches on a light, and R_{dst} has a conditional clause based on that light's status, but R_{src} subsequently aborts. Cascading aborts are handled in [24], whose techniques can be used orthogonally with ours.

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

2.4.2 Locking Data-structure

SafeHome maintains, at the edge device (e.g., Home Hub or smart access point), a *virtual locking table data-structure* (Fig. 2.4). It contains:

- *Wait Queue:* Queue of routines initiated but not started. When a routine is added, it is assigned an incremented routine ID.

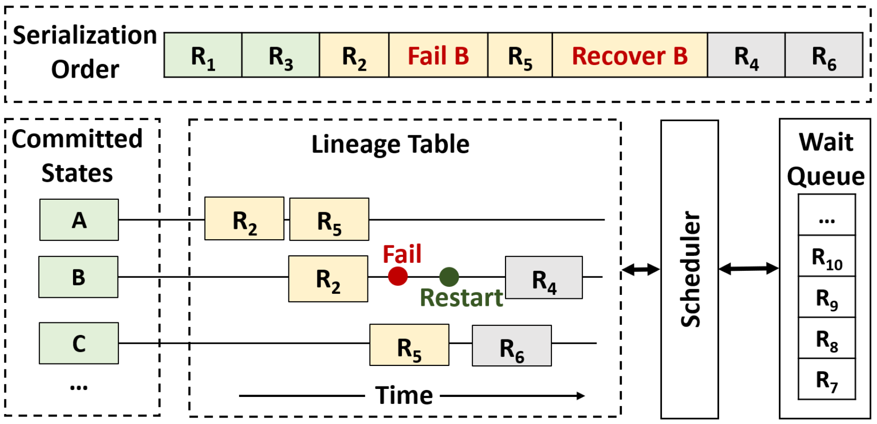


Figure 2.4: **SafeHome's Architecture for Eventual Visibility.**

- *Serialization Order:* Maintains the current serialization order of routines, failure events, and restart events. For completed routines (shaded green), the order is finalized. All other orders are tentative and may change, e.g., based on lock leases. Failure and restart events may be moved flexibly among unfinished routines.
- *Lineage Table:* Detailed in Section 2.4.3, this maintains, for each device, a *lineage*: the *planned* transition order of that device's lock.
- *Scheduler:* Decides when routines from Wait Queue are started, acquires locks, and maintains serialization order.
- *Committed States:* For each device, keeps its last committed state, i.e., the effect of the last successfully routine. This may be different from device's actual state, and is needed to ensure serialization and rollbacks under aborts.

2.4.3 Lineage Table

The *lineage* of a device represents a temporal plan of when the device will be acquired by concerned routines. The lineage of a device starts with its latest committed state, followed by a sequence of *lock-access* entries (Fig. 2.5)–these are "stretched" horizontally. A width of a lock-access entry represents how long that routine will acquire that lock. Each lock-access entry for device D consists of: *i.* A routine ID, *ii.* Lock *status* (`Released, Acquired, Scheduled`) *iii.* Desired device state by the command (e.g., `ON/OFF`) and *iv.* Times: a start time ($T_{start}(R_i)$), and duration ($\tau_{R_i}(D)$) of the lock-access.

In the example of Fig. 2.5, a `Scheduled [S]` status indicates that the routine is scheduled to access the lock. An `Acquired [A]` status shows it is holding and using the lock. A

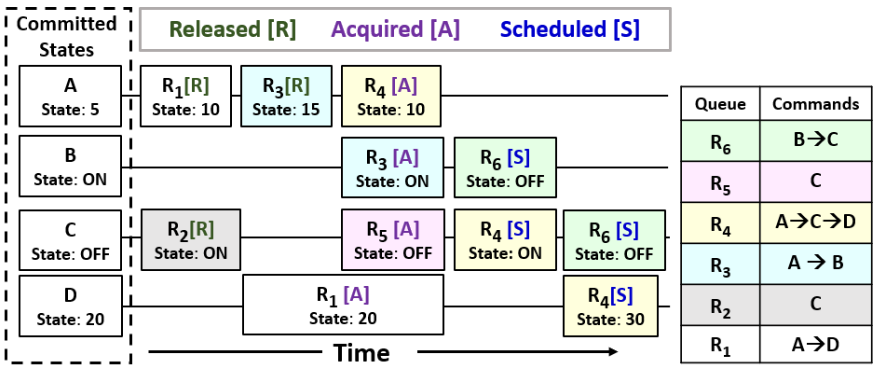


Figure 2.5: **Sample Lineage Table, with 6 routines. Some fields are omitted for simplicity.**

`Released [R]` status means the routine has released the lock.

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

To maintain serializability, four key invariants are assured:

Invariant 2.1 (**Future Mutual Exclusion: Lock-accesses in a device's lineage list do not overlap in time).** *No device is planned to be locked by multiple routines. Gaps in its lineage list indicate times the device is free.*

Invariant 2.2 (**Present Mutual Exclusion: At most one `Acquired` lock-access exists in each lineage list).** *No device is locked currently by multiple routines.*

Invariant 2.3 (**Lock-access [R]⟶[A]⟶[S]).** *In each lineage list, all* `Released` *lock-access entries occur to the left of (i.e., before) any* `Acquired` *entries, which in turn appear to the left of any* `Scheduled` *entries.*

Invariant 2.4 (**Consistent "serialize-before" ordering among lineages).** *Given two routines R_i, R_j, if there is at least one device D such that: lock-access$_D(R_i)$ occurs to the left of lock-access$_D(R_j)$ in D's lineage list, then for every other device D' touched by both R_i, R_j, it is true that: lock-access$_{D'}(R_i)$ occurs to the left of lock-access$_{D'}(R_j)$. Hence R_i is serialized-before R_j.*

Transition of Lock-accesses: The status of lock-accesses changes upon certain events. First, when a routine's last access to a device ends, the `Acquired` lock-access ends, and transitions to `Released`. The next `Scheduled` lock-access turns to `Acquired`: i) either immediately (if no gap exists, e.g., R_4 after R_5 releases C in Fig. 2.5), or ii) after the gap has passed, e.g., R_4 after R_1 releases D in Fig. 2.5.

Second, when scheduling a new routine R (from the wait queue), a `Scheduled` lock-access entry is added to all device lineages that R needs (e.g., R_6 in Fig. 2.5 adds lock-accesses for B and C). Third, when a routine finishes (completes/aborts), all its lock-access entries are removed, releasing said locks. If the routine completed successfully, committed states are updated. For an abort, device states are rolled back.

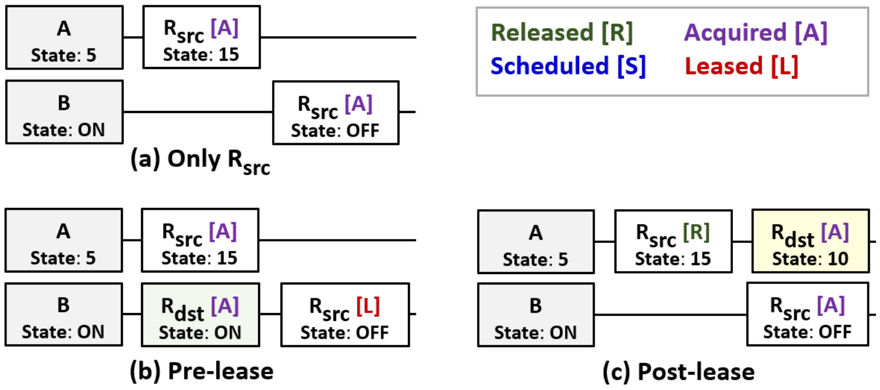


Figure 2.6: **Lineage table with Lock Leasing**. *a) Lineage before leasing with only R_{src}, b) Pre-lease to R_{dst} that only accesses device B, and c) Post-lease to R_{dst} that only accesses device A.*

Leasing of Locks: Consider a pre-lease from R_{src} to R_{dst} (Fig. 2.6(b)). First, a new `Acquired` lock-access for R_{dst} is placed *before* (to the left of) the lock-access of R_{src} in the lineage table. Second, the lock-access of R_{src} is changed to "Leased (R_{dst})" status.

Figure 2.6(c) shows a post-lease: a new `Acquired` lock-access of R_{dst} is placed *after* (to the right of) the lock-access of R_{src} and the lock-access of R_{src} changes to `Released`.

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

- *Device D was last* `Acquired` *by routine R_j ($\neq R_i$):* We remove R_i's lock-access from D's lineage. This captures two possibilities: a) R_i never executed actions on D (e.g., Fig. 2.5: device C when aborting R_4), or b) R_i leased D to another routine R_j, and since R_i is aborting, R_j's effect will be the latest (e.g., Fig. 2.5: device A when aborting R_1).

- *Device D was last* `Acquired` *by routine R_i* (e.g. device C when aborting R_5 in Fig. 2.5): We: 1) remove the R_i's lock-access from D's lineage, and 2) issue a command to set D's status to R_i's *immediately left/previous* lock-access entry in the lineage (if none exist, use Committed State), unless the device is already in this desired state.

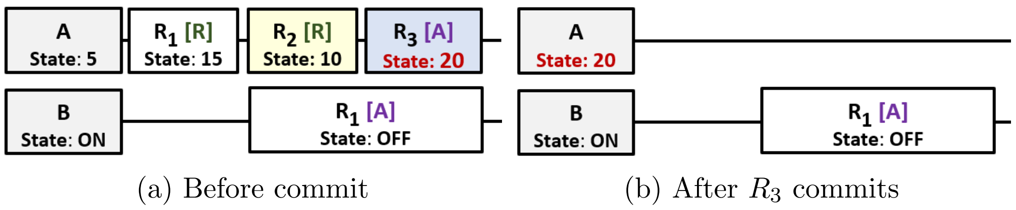


Figure 2.7: **Commit with compaction.**

Committing (Successfully Completing) a routine: When a routine reaches its finish point, it commits (completes successfully) by: i) updating Committed States, and ii) removing its lock-access entries. R_j might appear after R_i in the serialization order but complete earlier, e.g., due to lock leasing. SafeHome allows such routines to commit right away by using *commit compaction*–routines later in the serialization order will overwrite effects of earlier routines (on conflicting devices) [6]. Concretely, for all common devices we remove both R_i's lock-access, and all lock-accesses before it (Fig. 2.7).

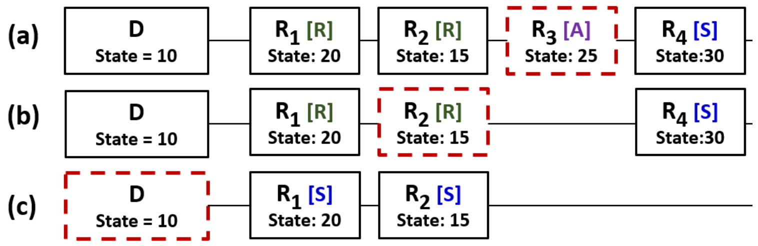


Figure 2.8: **Inferring the current device status**. *The dashed boxes point to the current device status in three different scenarios.*

Current Device Status: A device's current status is needed at several points, e.g., abort. Due to uncompleted routines, the actual status may differ from the committed state. The lineage table suffices to estimate a device's current state (without querying the device). Fig. 2.8 shows the three different cases. (a) If an `Acquired` lock-access entry exists, use it (e.g., R_3 in Fig. 2.8(a) with $D = 25$). (b) Otherwise, if lock-accesses exist with lock status `Released`, use the right-most entry (e.g., R_2 in Fig. 2.8(b) with $D = 15$). (c) Otherwise, use the Committed State entry (e.g., committed state $D = 10$ in Fig. 2.8(c)).

2.5 SCHEDULING POLICIES FOR EVENTUAL VISIBILITY

We now discuss how, for EV, SafeHome's Scheduler (Fig. 2.4) decides where a routine fits into the (eventual) serialization order. The Scheduler is a pluggable component, with

[6]Similar to "last writer wins" in NoSQL DBs [58].

minimal requirements that it satisfies all invariants from Sec. 2.4.3. We designed and implemented three scheduling policies.

2.5.1 First Come First Serve (FCFS) Scheduling

Routines are serialized based on their arrival time, i.e., routine ID. When the scheduler schedules a routine R, it adds lock-access entries for all of R's commands to the *end* of the corresponding lineages. Post-leases are allowed in FCFS. However pre-leases are inapplicable because they would violate the FCFS serialization order (Section 2.4.1).

FCFS is attractive if a user expects routines to execute in the order they were initiated. However, this serialization inflexibility results in long lag to start the routine. In contrast, there are many scenarios where users are willing to accept a completion order different from arrival, e.g., user submits a batch of routines, multiple users submit routines, etc.

2.5.2 Just-in-Time (JiT) scheduling

Just-in-Time (JiT) scheduling is a greedy approach where a routine is started (moved from wait queue to lineage) at the *earliest time when it is eligible to start.*

This *eligibility test* for a routine R checks if it will be able to acquire all its locks. That is, for each device accessed by R, the device is now either: a) "Released", or b) it is both "Acquired" *and* a pre-lease or post-lease can be availed by R. The eligibility test is performed for all routines in the wait queue. The eligibility test is triggered: (i) whenever a new routine arrives (eligibility test only for that routine), or (ii) whenever a lock-access, for some device D, is released by some executing routine. In case (ii) we minimize overhead of traversing the wait queue by running the eligibility test only on those waiting routines desiring to access device D.

JiT could cause starvation, e.g., a routine $R_{AB} = \{A = ON; B = 5;\}$ could be indefinitely starved by periodic invocations of two routines $R_A = \{A = OFF;\}$ and $R_B = \{B = 10;\}$. We mitigate starvation by using a TTL (time-to-live) for each routine in the wait-queue (initialized to 5 in SafeHome). The Routine R's TTL is decremented whenever a routine with a higher ID, and accessing a common device with R, is scheduled from the wait queue instead of R. When its TTL reaches 0, R *must* be scheduled earlier than higher ID routines. If multiple routines with TTL=0 exist, they are scheduled starting from the head of the queue.

2.5.3 Timeline Scheduling

It may be possible to start some routines even earlier than their eligibility test being satisfied. The Timeline Scheduling policy uses estimates of lock-access durations (τ_R), and attempts to place waiting routines into the gaps in the lineage table's timeline (using lock acquisitions, pre-leases, and post-leases). It finds the *earliest* possible schedule leveraging existing gaps so as: a) not to violate past/existing serialization order decisions, and b) does not prolong already-running routines beyond an acceptable threshold. This search is triggered for all waiting routines, upon every routine arrival or finish.

Algorithm 2.1 Timeline scheduling of routine R

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

Algo. 2.1 shows pseudocode. The key idea is to employ a recursive back-tracking strategy trying different gaps. We explain the algorithm by using the example in Fig. 2.9. Fig. 2.9a depicts a lock table right before routine $R_3 = \{C \rightarrow B\}$ arrives at time T_{R3}, and has four gaps in the lineage. Starting with the first device in the routine (C for R_3): $\tau_{R_3}(C)$ (Line 3), the Timeline scheduler finds the first gap in C's lineage that can fit $\tau_{R_3}(C)$ (Line 9). This is Gap 1 in Fig. 2.9a.

Next, the Timeline scheduler validates that this gap choice will not violate previously decided serializations. For the scheduled lock-accesses of R_3 so far, it builds two sets: a)

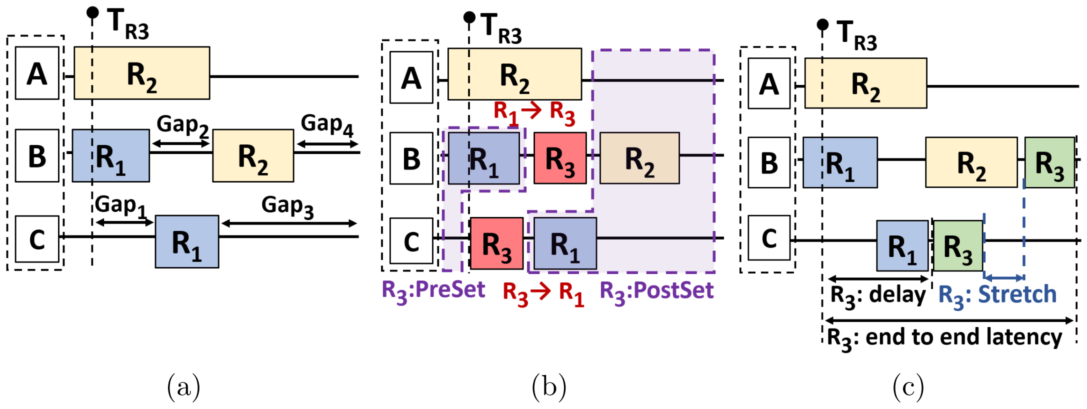


Figure 2.9: **Timeline Scheduler (TL) example** *a) before scheduling R_3 b) trying a potential (but invalid) option, c) scheduling R_3 at the first possible gap.*

preSet: the union of all (executing and scheduled) routines placed *before* R_3's lock-accesses ($\{R_1\}$ in Fig. 2.9b), and b) *postSet:* the union of all (executing and scheduled) routines placed *after* R_3's lock-accesses ($\{R_1, R_2\}$ in Fig. 2.9b). The preSet and postSet of R represent the routines positioned before and after R, respectively, in the serialization order. *The gap choice is valid if and only if the intersection of the preSet and the postSet is empty.* In this case, the scheduler moves on to the next command of the routine. Otherwise (e.g., as in Fig. 2.9b), the scheduler backtracks and tries the next gap (Line 21). This process then repeats.

2.5.3.1 Device State Dependencies and Safety in Timeline Scheduling

A Routine comprises a sequence of commands that collectively perform a compound task. E.g., the routine "prepare breakfast" starts brewing coffee and prepare the pancake. However, some routines require device-state dependencies to ensure a "safe/desirable environment" for safely/correctly performing that task. For example, the routine "cook food" first sets the exhaust fan's state `ON` and only then it sets the oven's state `ON`. Here the device status oven:`ON` depends on the exhaust fan:`ON` (Ex.Fan:`ON` $\rightarrow$ Oven:`ON`). In the current SafeHome design, such dependencies are marked by the user.

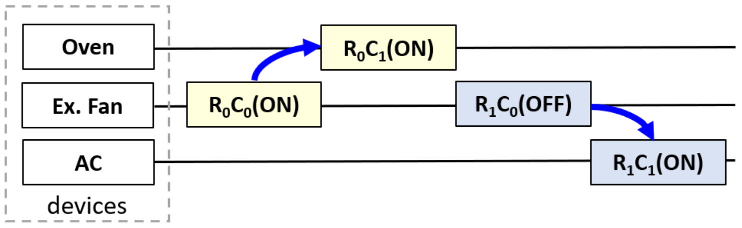
SafeHome's in-built safety mechanism should ensure that the concurrent routine executions never violate the device dependencies.

Consider the following two concurrent routines:

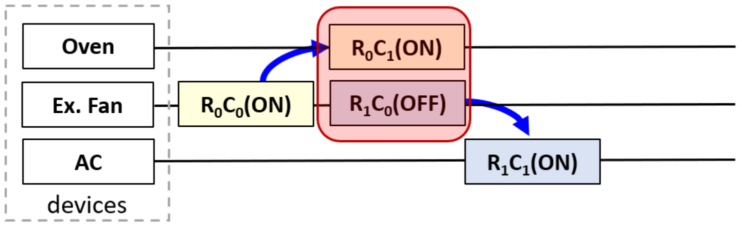
R_0: { Exhaust Fan : `ON`, Oven : `ON` }

R_1: { Exhaust Fan : `OFF`, AC : `ON` }

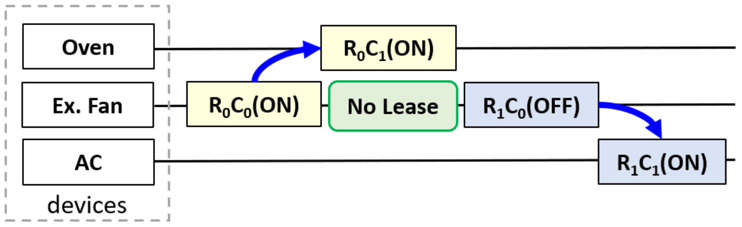
R_1 appears immediately after R_0 starts its execution. In both R_0 and R_1, the status of respectively the Oven and the AC depends on the status of the Exhaust fan. GSV and PSV's inherent serializability automatically ensures the device dependencies (Fig. 2.10a). However, TL's pre/post lease based opportunistic scheduling (Algo. 2.1) might violate the device dependency (Fig. 2.10b).



(a)



(b)



(c)

Figure 2.10: **Execution policy of R_0 and R_1 in different visibility models.** *R_rC_c represents the c^{th} command of the r^{th} routine. E.g., R_1C_1 = AC:*`ON`*. Blue arrows represent command-state dependencies. a) GSV and PSV– inherently preserve dependencies b) TL (without safety) – the red box shows dependency violation, c) TL (with safety)– Ex.Fan (R_0C_0) has dependent device (R_0C_1). Therefore the safety checker stalls/rejects the lease request (green box).*

For TL, SafeHome ensures the device dependencies by adding additional constraints on the pre/post leasing scheme (Algo. 2.1). In this modified approach, a lineage table approves pre/post lease for device D only if at that point D does not have any dependent device (Fig. 2.10c). Therefore, safety is ensured with the cost of increased end-to-end latency.

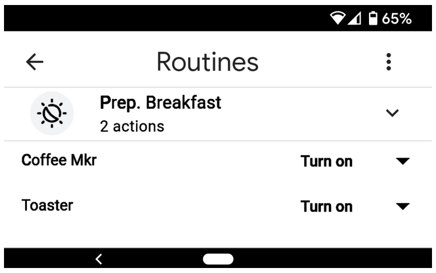
2.6 SAFEHOME IMPLEMENTATION

We implemented SafeHome in 1200 (core) lines of Java. SafeHome runs on an edge device, such as a Home Hub or an enhanced/smart access point. Our edge-first approach has two major advantages: 1) SafeHome can be run in a smart home containing devices from a diverse set of vendors, and 2) SafeHome is autonomous, without being affected by ISP/external network outages [72, 73] or cloud outages [74, 75, 76].

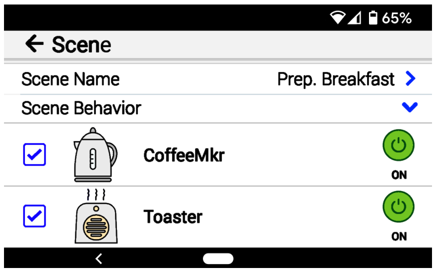
SafeHome works directly with the APIs exported by devices – commands in routines are programmed as API calls directly to devices. SafeHome's routine specification is compatible with other smart home systems (Fig. 2.11). Our current implementation works for TP-Link smart devices [31, 52], using HS110Git [77] device-driver. Other devices (e.g., Wemo [78]) can be supported via their device-drivers.

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

(a) JSON representation of SafeHome routine (part)



(b) G. Home routine [12]



(c) TP-Link routine [79]

Figure 2.11: **Defining a routine "Prepare Breakfast"** *Two commands: i) Turn* ON *Coffee Maker and ii) Turn* ON *Toaster.*

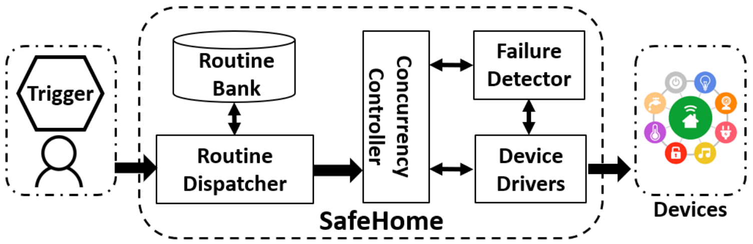


Figure 2.12: **SafeHome Architecture**

Fig. 2.12 shows our implementation architecture. When a user submits routines, they are stored in the *Routine Bank*, from where they can be invoked either by the user or triggers, via the *Routine Dispatcher*. The *Concurrency Controller* runs the appropriate Visibility model's implementation. Apart from Eventual Visibility (Sec. 2.5), we also implemented Global Strict Visibility (GSV), and Partitioned Strict Visibility (PSV), with failure/restart serialization. Our Weak Visibility reflects today's laissez-faire implementation.

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

2.7 EVALUATION

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

- Are relaxed visibility models (like Eventual Visibility) as responsive as Weak Visibility, and as correct as Global Strict Visibility (Sec. 2.2.1)?
- What effect do failures have on correctness and user experience (Sec. 2.3)?
- Which scheduler policy (Sec. 2.5) is the best?
- What is the effect of optimizations, e.g., lock leasing, commit compaction, etc. (Sec. 2.4)?

2.7.1 Experimental Setup

Metrics: Because of the human-visible nature of SafeHome, our main evaluation metrics are also human-visible (we also define additional metrics where applicable):
End to end latency (or Latency): This metric measures how long the human user has to wait from initiating/submitting a routine to its successful completion.
Temporary Incongruence: This metric measures how much the human user's actual experience differs from a world where all routines were run serially. We take worst case behavior. Before a routine R completes, if another routine R' changes the state of *any* device R modified, we say R has suffered a temporary incongruence event. The *Temporary Incongruence* metric measures the fraction of routines that suffer at least one such temporary incongruence event.
Final Incongruence: After executing a batch of routines, is the home's state serially-equivalent?

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

Setup: Because we wish to evaluate SafeHome for a variety of scenarios, and using multiple parameters and many routines and devices, our evaluation relies on workload-driven emulation. We use the same code from our implementation for this emulation.

2.7.2 Experiments with Real-World Benchmarks

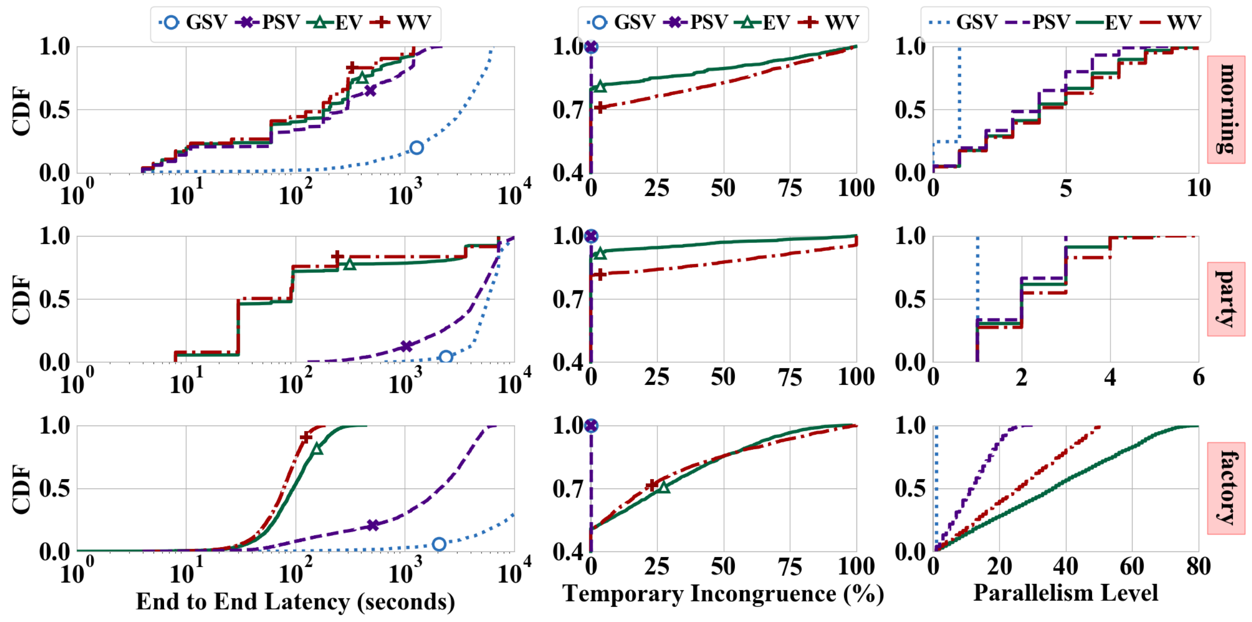


Figure 2.13: **Latency, Temporary Incongruence, and Parallelism for Three Scenarios.** *To identify lines we show one label symbol for each (plot has many more data points). Some GSV lines may be cut to show separation between other models.*

We extracted traces from three real homes (20-30 devices, multi-user families) who were using Google Home over 2 years. We also studied two public datasets: 1) 147 SmartThings applications [80]; and 2) IoTBench: 35 OpenHAB applications [81]. Based on these, we created the following three representative benchmarks. (We will make these available openly.)

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

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

Factory Scenario. This assembly line scenario has 50 workers working at 50 linear stages. Each stage's worker has access to some local devices, to some devices shared with immediately preceding and succeeding stages, and to 5 global devices. Each routine by a worker accesses devices probabilistically. Access probabilistics are 0.6 for local devices, 0.3 for neighbor-shared devices, and 0.1 for global devices.

Results: From Fig. 2.13 (top row), in the morning scenario: 1) EV's latency is comparable to WV at both median and 95th percentile, and 2) PSV has 15% worse 90th percentile latency than EV. Generally, the higher the parallelism level (last column), the lower the latency. For instance, EV has median parallelism level 3× higher than GSV, and median latency 16× better than GSV. Parallelism creates more temporary incongruences (middle column of figure). This is expected for EV. Yet, EV's (and GSV's) end state is serially equivalent while WV may end incongruently–this is shown in Fig 2.14. Thus *EV offers similar latencies as, but better final congruence than, WV.* Only if the user cares about temporary incongruence is PSV preferable.

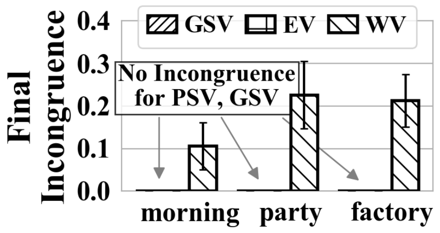


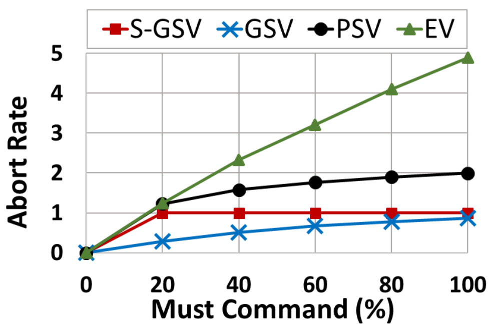
Figure 2.14: **Final Incongruence.** *Run with 9 routines, 100 runs per scenario, and checks if final smart home state is equivalent to some serial ordering of routines (9! possibilities).*

In Fig. 2.13 (middle row), the party scenario shows similar trends to the morning scenario with one notable exception. PSV's benefit is lower, with only 11% 90th percentile latency reduction from GSV (vs. 77% in morning). This is due to the single long routine blocking other routines. EV avoids this hogging because of its pre- and post-leasing.

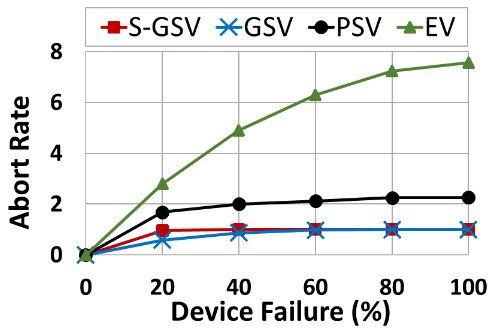
In Fig. 2.13 (bottom row), the factory scenario shows similar trends to morning scenario, except that: (i) EV's median latency is 23.1% worse than WV, and (ii) the parallelism level is higher in EV than WV. This is due to the back-to-back arrival of multiple routines. WV executes them as-is. However, EV may delay some routines (due to device conflicts)–when the conflict lifts, all eligible routines run simultaneously, increasing our parallelism level and latency.

2.7.3 Effect of Failures

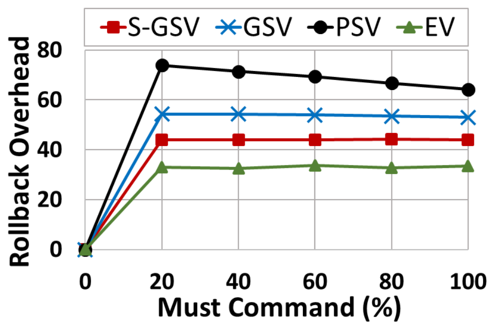
Failures abort more routines in EV because it allows high concurrency, yet EV's intrusive effect on the user (due to aborts) is the lowest of all visibility models. Fig. 2.15a and 2.15b measure the fraction of routines aborted due to a failure (fail-stop failures introduced at a random point during the run). Yet Fig. 2.15c and 2.15d show that the *rollback overhead* of EV is smallest among all visibility models–this is the average fraction of commands rolled back, across aborted routines. PSV's rollback overhead is higher than EV as it aborts more at the routine's finish point (when checking up/down status of devices touched). EV aborts affected routines earlier rather than later. GSV and S-GSV have low abort rates because of their serial execution but have higher rollback overheads than EV. Thus, even *when execution is serial, the effect of failures can be more intrusive on the human.* We conclude that EV is the least intrusive model.



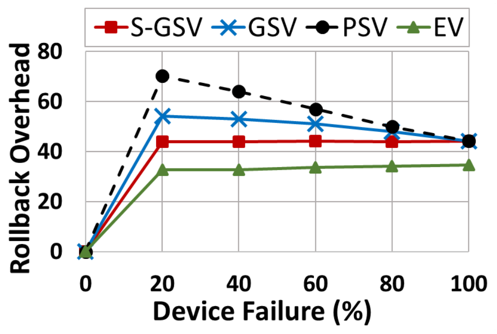
(a) Must Vs Abort Rate



(b) Failure Vs Abort Rate



(c) Must Vs Rollback Overhead

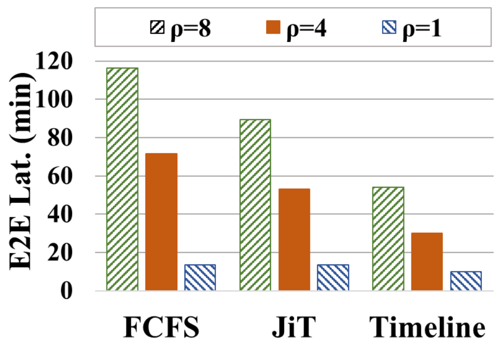


(d) Failure Vs Rollback Overhead

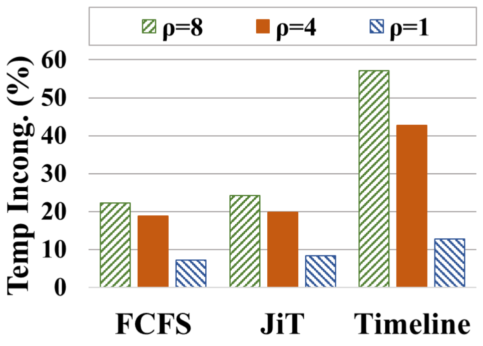
Figure 2.15: **Effect of Failures.** *Rollback Overhead = Intrusion on User. Parameters in Table 2.3.*

The plateauing in Figures 2.15a, 2.15b is due to saturation of parallelism level. The plateauing in Figs. 2.15c, 2.15d, is due to saturation at the average abort-point of a routine– for GSV around 50%, while S-GSV is lower at 40% since *any* device failure triggers the abort.

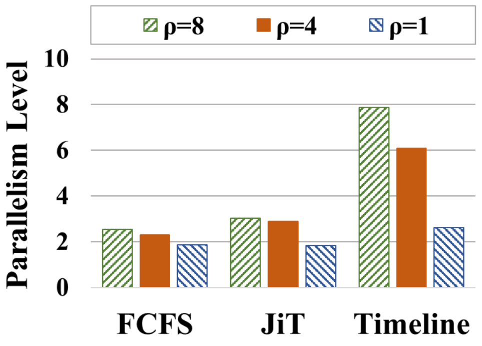
2.7.4 Scheduling Policies



(a) End to End Latency



(b) Temporary Incongruence



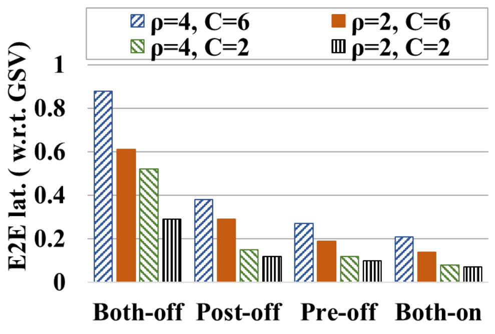
(c) Parallelism Level

Figure 2.16: **Scheduling Policies.** *Parameters in Table 2.3.*

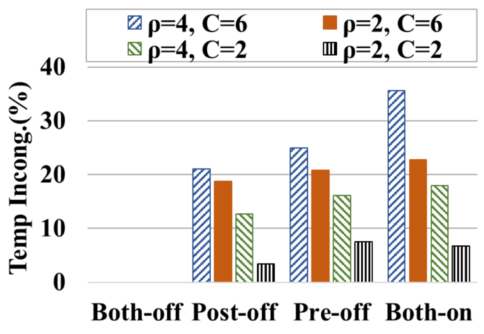
Fig. 2.16 compares FCFS, JiT, and Timeline (TL) scheduling policies (Sec. 2.5). TL has the lowest latency–in Fig. 2.16a with $\rho = 4$ concurrent routines injected, TL is 2.36× and 1.33× faster than FCFS and JiT respectively. The benefit of TL over JiT is due to leasing. The benefit of TL over FCFS is due to both opportunism and leasing. We also observed FCFS causing starvation. TL also has higher parallelism level (Fig. 2.16c) than FCFS (2.3× at $\rho = 4$) and JiT (2.0× $\rho = 4$).

2.7.5 Timeline-based Eventual Visibility (TL)

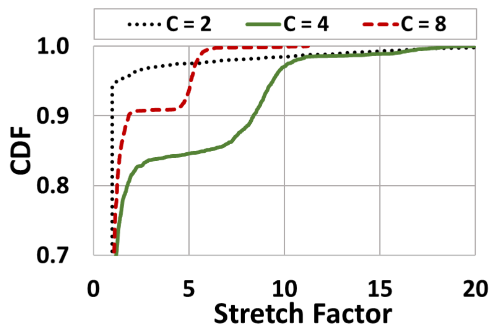
Fig. 2.17a and 2.17b show that disabling leasing reduces temporary incongruence but significantly increases latency. Turning off *both* pre- and post-leasing increases latency (from Both-on to Both-off) by between 3× to 5.5× (as concurrency level ρ and commands per routine $\mathcal{C}$ are varied). Post-leases are more important than pre-leases: disabling the former raises latency by 71% to 107%, while disabling the latter raises latency from 29% to 50%. Increasing $\rho, \mathcal{C}$ raise latency because they saturate the lock-table.



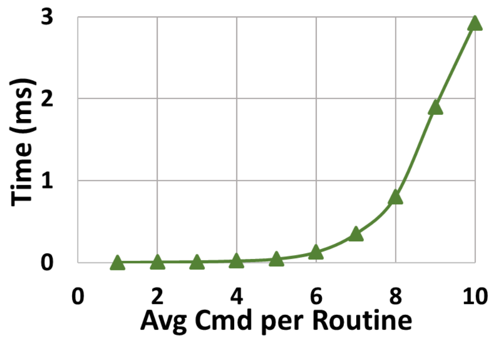
(a) Normalized E2E Latency



(b) Temporary Incongruence (%)



(c) CDF of Stretch Factor



(d) Algo. 2.1 Insertion Time

Figure 2.17: **Timeline policy for EV.** *Parameters in Table 2.3.*

TL might also "stretch" routines (Fig. 2.9c). Fig. 2.17c shows *stretch factor*, measured as the between a routine's actual start (not submission) and actual finish, divided by the ideal (minimum) time to run the routine. With routine size, stretch factor rises at first (at $\mathcal{C} = 2$ only 5% routines have stretch > 1, vs. 25% at $\mathcal{C} = 4$) but then drops (15% at $\mathcal{C} = 8$). Essentially the lock-table saturates beyond a $\mathcal{C}$, creating fewer gaps and forcing EV to append new routines to the schedule.

Fig. 2.17d shows that even when running on a Raspberry Pi 3 B+ [82], our scheduler inserts a new routine within 0.5 ms, at 7 commands/routine or less. Overhead then rises but stays below 3 ms. Our survey of smart home datasets [80, 81] revealed that typical routines contain 5 or fewer commands, which means our scheduler is fast in practice.

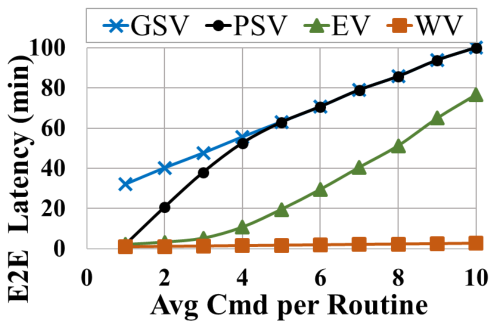
2.7.6 Parameterized Microbenchmark Experiments

We created a parameterized microbenchmark (Table 2.3) and we explore the effect of key parameters.

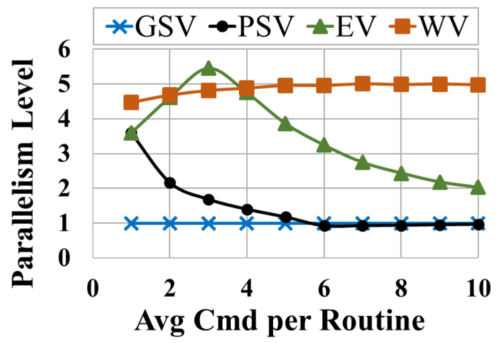
Impact of number of commands per routine ($\mathcal{C}$)**:** Fig. 2.18a, 2.18b show GSV's latency rises with more commands per routine. With smaller routines, PSV's latency is close to EV

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

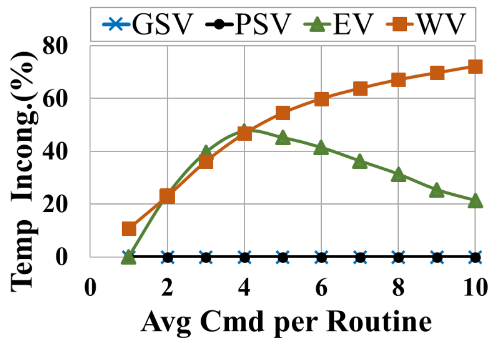
Table 2.3: **Parameterized Microbenchmark: Summary of Parameters.** *ND = Normal distribution.*



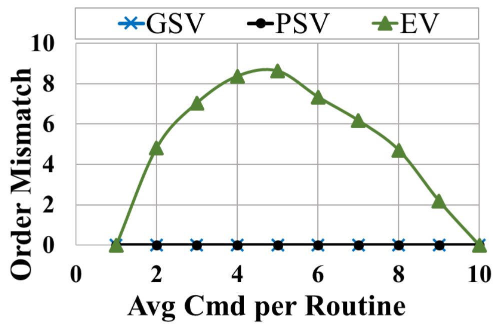
(a) End to End latency



(b) Parallelism level (%)



(c) Temporary Incongruence (%)



(d) Order Mismatch

Figure 2.18: **Impact of Routine size (Commands/routine $\mathcal{C}$).**

and WV, but as routines get larger, PSV quickly approaches GSV. A similar trend occurs with EV, but EV stays noticeably faster than GSV/PSV. Trends in parallelism level and temporary incongruence are also consistent with this trend. Finally, EV's peaking behavior and eventual convergence towards GSV (Fig. 2.18c, 2.18d) occur since beyond a certain routine size ($\mathcal{C}$=4), pre/post-leasing opportunities decrease.

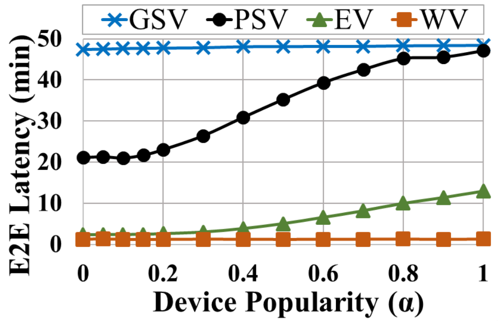
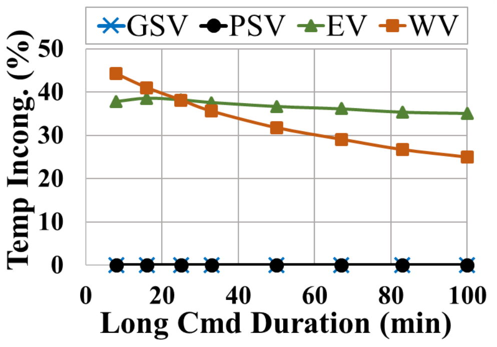
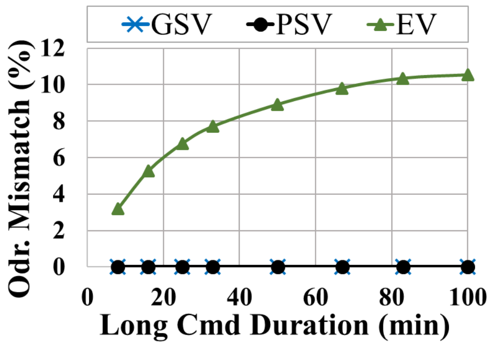


Figure 2.19: **Latency with varying device popularity α.**

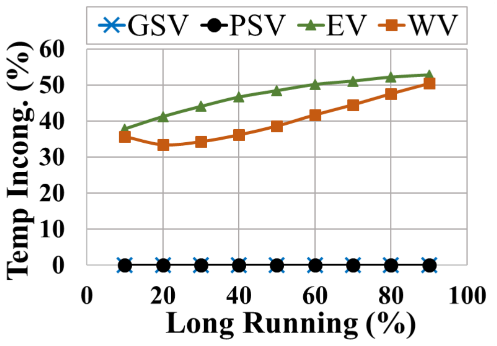
Impact of device popularity (α)**:** We use a Zipf distribution for the probability of devices being touched by routines. As one skews the distribution more in Fig. 2.19 (increasing α), we notice that EV's latency stays close to WV. More conflict makes PSV quickly become as slow as GSV.



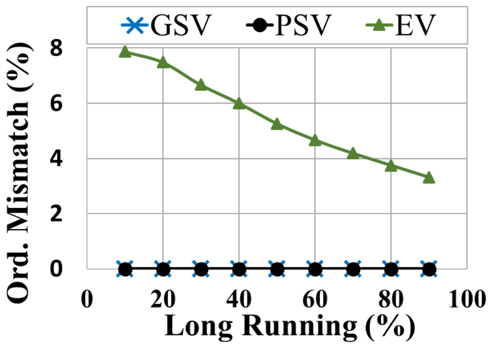
(a) Temporary Incongruence



(b) Order Mismatch



(c) Temporary Incongruence



(d) Order Mismatch

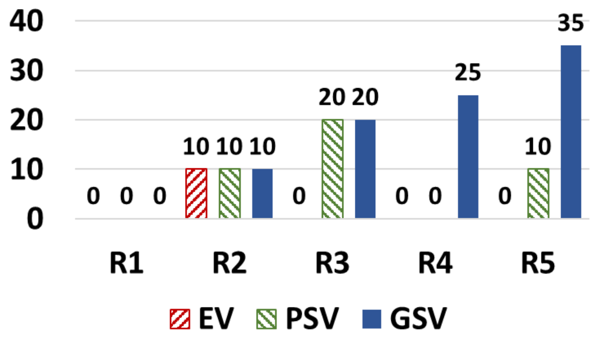
Figure 2.20: **Impact of: (a, b) Duration of long routine ($|\mathcal{L}|$), and (c, d) Fraction of routines that are long ($\mathcal{L}_{\%}$).**

Impact of long running routines: As the long running routine length $|\mathcal{L}|$ rises (Fig. 2.20a), temporary incongruences decrease since the run is now longer, routines are spread temporally, and less likely to conflict. If there are more long running routines ($\mathcal{L}_{\%}$), there is more conflict, and this increases temporary incongruence (Fig. 2.20c). We also measure the *order mismatch*, i.e., how much does the final serial equivalence order measure from the order in which routines were submitted (we use swap distance). Longer routines cause more order mismatches (Fig. 2.20b), but more long-running routines reduce the order mismatch (Fig. 2.20d) because post-leases dominate. Overall, the order mismatch numbers stay low, from 3%-10%.

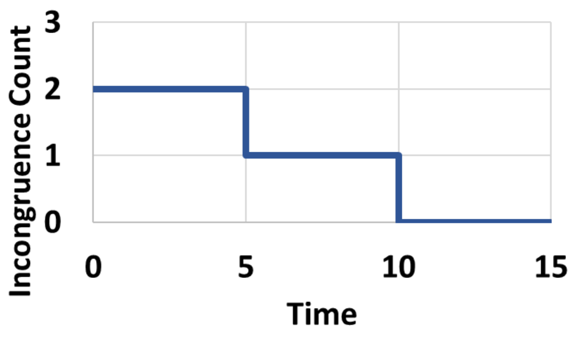
2.7.7 Real Smart-home deployment

To understand the impacts of different visibility models, we simultaneously run five different routines in a real smart-home deployment. We deploy SafeHome in a Raspberry Pi and use TP-Link smart-plugs (HS105 [69]) to mimic the real smart-devices. The following routines are inserted simultaneously in the system:
R_1: makeCoffee(Espresso), makePancake(Vanilla)
R_2: makeCoffee(Americano), makePancake(Strawberry)
R_3: makePancake(Regular)
R_4: startRoomba(Living room), startMopping(Living room)
R_5: startMopping(Kitchen)

The *makeCoffee(*)* and *makePancake(*)* commands respectively prepare different types of coffee and pancake (based on user's taste). *StartRoomba(*)* and *startMopping(*)* respectively start the vacuuming and mopping process for the assigned room. To better understand and analyze the scenarios, we assign 5 minutes for each command.
Routine execution strategies: Fig. 2.2 shows the lineage tables for different visibility models. The execution is strictly sequential for GSV, whereas for PSV, only the cluster of routines sharing common devices is executed sequentially. EV, due to its pre/post leases, can afford more parallelism. Besides, it is only EV that opportunistically searches for earliest empty slots and tries to schedule commands accordingly. Therefore, the actual execution order might vary from the routine submission order (e.g., in EV R3 and R5 executes before R1 and R4, respectively).
Wait times: Fig. 2.21a shows the wait time of each routine for different visibility models. Wait time is the time span between when the routine is first submitted to SafeHome framework and the moment when its execution starts. All routines in GSV face cumulatively increasing wait times (due to the strict serializability). In PSV, only the routines sharing



(a)



(b)

Figure 2.21: (a) Wait time in different visibility models. (b) The flow of incongruency in EV over time.

the common devices face the cumulatively increasing wait times. EV's early slot selection mechanisms (pre/post lease) parallelize the routines, which dramatically reduces the wait times.

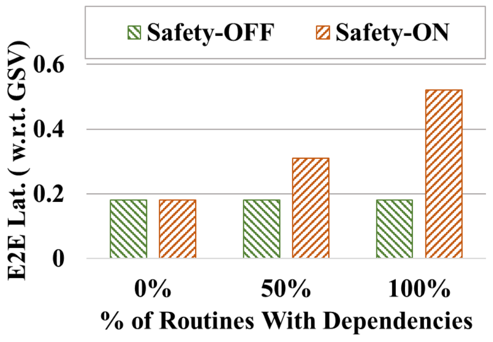
Temporary incongruence: EV's faster execution comes with the cost of temporary incongruence. When R_0 is executing in both GSV and PSV, none of its devices is accessed by other routines. This gives the user an impression of exclusive use of the devices assigned for R_0 (congruent execution). However, in EV, when R_0 is executing its first command (make coffee), it pre-leases the pancake machine to R_2 (since it is not in use immediately). Here multiple routines simultaneously access the common resources, which creates the incongruency. However, sucn incongruencies are temporary (Fig. 2.21b) and do not violate the correctness. The outcome becomes serializable at the end of EV's execution.

Both our simulation-based experiments and the real-deployment based experiments exhibit similar trends. Unlike GSV or PSV, EV parallelizes more routines which costs less wait time. EV suffers from temporary incongruence. However, EV's outcome becomes serializable eventually.

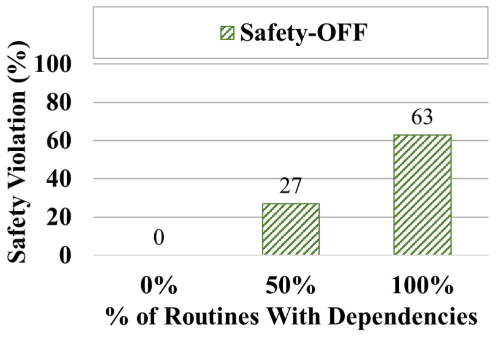
2.7.7.1 Safety Overhead in EV

We perform emulator based experiments to characterize the overhead of maintaining the device-state dependencies. The experiments use the default parameters from Table 2.3. Additionally, in this experiment we add a pair of dependent devices in routines and vary the percentage of such routines.

Fig. 2.22a shows that with the safety feature enabled, EV's end to end latency almost doubles as the percent of dependent routines increase from 50% to 100%. In this approach



(a)



(b)

Figure 2.22: **Measuring the safety overhead.**

the pre/post leases are rejected, which leaves EV with a relatively less optimized yet safe scheduler.

On the other hand, while the safety feature is disabled, the scheduler accepts any pre/post leases, which results in lower latency. The percent of dependent routines does not have any impact on this approach, therefore the latencies remain constant for all three scenarios. Besides, while the safety feature is disabled, as the routine dependencies increase, the safety violation increases too (Fig. 2.22b).

This experiment shows that even in the extreme rare case where 100% of the routines are dependent routines– TL's overall end to end latency is almost half than that of the GSV.

2.8 RELATED WORK

Support for Routines: Routines are supported by Alexa [83], Google Home [12], and others [84, 85, 86]. iRobot's Imprint [25, 87] supports long-running routines, coordinating between a vacuum [88] and a mop [89]. All these systems only support best-effort execution (akin to WV).

Safety and Reliability: SafeHome can be used orthogonally with transactuations [24], which provide a consistent soft-state. Transactuations maintains strict isolation by sequentially executing conflicting routines, making it somewhat akin to PSV. APEX [66] automatically deducts the pre-conditions of a user command and ensures its atomic execution. APEX uses two-phase-locking [90] to maintain Isolation. Such two-phase locking causes additional overhead. APEX's isolation scheme is similar to PSV used in SafeHome. APEX needs to acquire physical device-locks, whereas, in SafeHome, all locks are logical, maintained by the SafeHome orchestrator. This means, APEX requires to implement the two-phase-locking on the smart-device side. Therefore off-the-shelf smart-devices will not directly work with APEX. CityGuard [91] is a safety-aware watchdog architecture developed for Smart City that detects and resolves conflicts.

Abstractions: IFTTT [11] represents the home as a set of simple conditional statements, while HomeOS [60] provides a PC-like abstraction for the home where devices are analogous to peripherals in a PC. Beam [92] optimizes resource utilization by partitioning applications across devices. These and other abstractions for smart homes [93, 94, 95, 96, 97] do not address failures or concurrency.

Concurrency Control: Concurrency control is well-studied in databases [98]. Smart Home OSs like HomeOS, SIFT, and others [60, 61, 64, 65] explore different concurrency control schemes, however, none of these explore visibility models.

Task Graph: Task Graph scheduling [99, 100, 101, 102, 103, 104, 105] is a well-studied area that schedules dependent tasks (represented by DAGs) across available resources (CPU/GPU/VM etc.). Here the goal is to obtain an efficient execution strategy for the set of dependent tasks. It varies from the main purpose of SafeHome that aims to efficiently execute routines on a routine-specific set of devices, while also ensures a serializable end-state.

2.9 CONCLUSION

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

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

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

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

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

CHAPTER 3: SERVICE FABRIC: A DISTRIBUTED PLATFORM FOR BUILDING MICROSERVICES IN THE CLOUD

This chapter describes the Service Fabric (SF), Microsoft's distributed platform for building, running, and maintaining microservice applications in the cloud. SF has been running in production for 10+ years, powering many critical services at Microsoft. We outline key design philosophies in SF. We then adopt a bottom-up approach to describe low-level components in its architecture, focusing on modular use and support for strong semantics like fault-tolerance and consistency within each component of SF. This chapter also reveals the unique *ground-up* consistency approach used in SF. Later, we discuss lessons learned, and present experimental results from production data.

This Chapter is organized as follows: Sec. 3.1 introduces Service Fabric, Microsoft's platform to support microservice applications in cloud settings. Sec. 3.2 explains the concept of the microservice based architecture and briefly covers four real microservice based applications, Sec. 3.3 and Sec. 3.4 respectively give an overview and system design of SF's architecture, Sec. 3.5 presents both simulation and real-deployment based experimental results to evaluate SF's selective components. Sec. 3.6 analyze the state-of-art related works and finally Sec. 3.7 concludes the project.

Disclaimer: Microsoft Service Fabric's design started in the early 2000's. It is a culmination of over a decade and a half of design and development, involving 100+ engineers. Our collaboration with Microsoft Service Fabric team started in 2017, and is limited to understanding, analyzing and evaluating the system. As a part of it we study the Service Fabric's architecture, explore different design decisions, measure the performance of several core components and summarize our findings as the very first research paper on Microsoft Service Fabric [1].

3.1 INTRODUCTION

Cloud applications need to operate at scale across geographical regions, and offer fast content delivery as well as high resource utilization at low cost. The *monolithic* design approach for building such cloud services makes them hard to build, to update, and to scale. As a result modern cloud applications are increasingly being built using a *microservices* architecture. This philosophy involves building smaller and modular components (the microservices), connected via clean APIs. The components may be written in different languages, and as the business need evolves and grows, new components can be added and removed seamlessly, thus making application lifecycle management both agile and scalable.

The loose coupling inherent in a microservice-based cloud application also helps to isolate the effect of a failure to only individual components, and enables the developer to reason about fault-tolerance of each microservice. A monolithic cloud application may have disparate parts affected by a server failure or rack outage, often in unpredictable ways, making fault-tolerance analysis quite complex. Table 3.1 summarizes these and other advantages of microservices.

	Monolithic design	**Microservice-based design**
Application complexity	Complex	Modular
Fault-tolerance	Complex	Modular
Agile development	No	Yes
Communication between components	NA	RPCs
Easily scalable	No	Yes
Easy app lifecycle management	No	Yes
Cloud ready	No	Yes

Table 3.1: **Monolithic Vs. Microservice Applications.**

Service Fabric (henceforth denoted as SF) enables application lifecycle management of scalable and reliable applications composed of microservices running at very high density on a shared pool of machines, from development to deployment to management. A stronger consistency guarantee is required to orchestrate such a large number of loosely coupled modules. To ensure it, Service Fabric embraces a unique *ground-up* approach.

Today's SF system is a culmination of over a decade and a half of design and development. SF's design started in the early 2000's, and over the last decade (since 2007), many critical production systems inside Microsoft have been running atop SF. These include Azure SQL DB [3], Cosmos DB [4], Skype [5], Azure Event Hub [106], Intune [107], Azure IoT [108], Cortana [109] and others. Today, Azure SQL DB running on SF hosts 1.82 Million DBs containing 3.48 PB of data, and runs on over 100 K machines across multiple geo-distributed datacenters. Azure Cosmos DB runs on over 2 million cores and 100 K machines. The cloud telemetry engine on SF processes 3 Trillion events/week. Overall, SF runs 24×7 in multiple clusters (each with 100s to many 1000s of machines), totaling over 160 K machines with over 2.5 Million cores.

Driven by our production use cases, the architecture of SF follows five major design principles:

- **Modular and Layered Design** of its individual components, with clean APIs.

- **Self-* Properties** including self-healing and self-adjusting properties to enable automated failure recovery, scale out, and scale in. Self-sufficiency, meaning no external dependencies on external systems or storage.
- **Fully decentralized operation** avoids single points of contention and failure, and accommodates microservice applications from small groups to very large groups of VMs/containers.
- **Strong Consistency** both within and across components, to prevent cascades of inconsistency.
- **Support for Stateful Services** such as higher-level data-structures (e.g., dictionaries, queues) that are reliable, persistent, efficient, and transactional.

Service Fabric is the only microservice system that meets all the above principles. Existing systems provide varying levels of support for microservices, the most prominent being Nirmata [34], Akka [32], Bluemix [33], Kubernetes [46], Mesos [110], and AWS Lambda [111]. SF is more powerful: it is the only data-aware orchestration system today for stateful microservices. In particular, our need to support state and consistency in low-level architectural components drives us to solve hard distributed computing problems related to failure detection, failover, election, consistency, scalability, and manageability. Unlike these systems, SF has no external dependencies and is a standalone framework. Sec. 3.6 expands on further differences between SF and related systems.

Service Fabric was built over 16 years, by many (over 100 core) engineers. It is a vast system containing several interconnected and integrated subsystems. It is infeasible to compress this effort into one paper. Therefore, instead of a top-down architectural story, this paper performs a deep dive on selected critical subsystems of SF, illustrating via a bottom-up strategy how our principles drove the design of key low-level building blocks in SF.

The main contributions of this work are:

- We describe design goals, and SF components that: detect failures, route virtually among nodes, elect leaders, perform failover, balance load, and manage replicas.
- We touch on higher-level abstractions for stateful services (Reliable Collections).
- We discuss lessons learnt over 10+ years.
- We present experimental results from real datasets that we collected from SF production clusters.

3.2 MICROSERVICE APPROACH

The concepts underlying microservices have been around for many years, from object-oriented languages, to Service Oriented Architectures (SOA). Many companies (besides Microsoft), rely on a microservice-based approach. Netflix has used a fine-grained SOA [112] for a long time to withstand nearly two billion edge API requests per day [113].

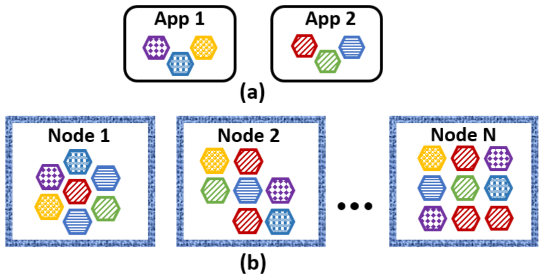


Figure 3.1: **A Microservice-based Application.** a) *Each colored/tiled hexagon type represents a microservice, and* b) *Its instances can be deployed flexibly across VMs.*

SF provides first-class support for full Application Lifecycle Management (ALM) of cloud applications, from development, deployment, daily management, to eventual decommissioning. It provides system services to deploy, upgrade, detect, and restart failed services; discover service location; manage state; and monitor health. SF clusters are today created in a variety of environments, in private and public clouds, and on Linux and Windows Server containers.

If such microservices were small in number, it may be possible to have a small team of developers managing them. In production environments, however, there are hundreds of thousands of such microservices running in an unpredictable cloud environment [114, 115, 116, 117, 118]. SF is an automated system that provides support for the complex task of managing these microservices.

Building cloud applications atop SF (via microservices) affords several advantages:

1. **Modular Design and Development:** By isolating the functionality and via clean APIs, services have well-defined inputs and outputs, which make unit testing, load testing, and integration testing easier.

2. **Agility:** Individual teams that own services can independently build, deploy, test, and manage them based on the team's expertise or what is most appropriate for the problem to be solved. This makes the development process more agile and lends itself to assigning each microservice to small nimble teams.

SF provides rolling upgrades, granular versioning, packaging, and deployment to achieve faster delivery cycles, and maintain up-time during upgrades. Build and deployment automation along with fault injection allows for continuous integration and deployment.

3. **Scalability:** A monolithic application can be scaled only by deploying the entire application logic on new nodes (VMs/containers). As Fig. 3.1 shows, in SF only individual microservices that need to scale can be added to new nodes, without impacting other services.

   This approach allows an application to scale as the number of users, devices and content grows, by scaling the cluster on demand. Incremental deployment is done in a controlled way: one at a time, or in groups, or all at once, depending on the deployment stage (integration testing, canary deployments, and production deployments).

4. **Resource Management:** SF manages multiple applications running on shared nodes, scaling themselves continuously, because the workloads change dynamically all the time. The components of SF that this paper fleshes out help keep nodes' load balanced, route messages efficiently, detect failures quickly and without confusion, and react to failures quickly and transparently.

5. **Support for State:** SF provides useful abstractions for stateful services, namely Reliable Collections, a data-structure that is distributed, fault-tolerant, and scalable.

3.2.1 Microservice Application Model in Service Fabric

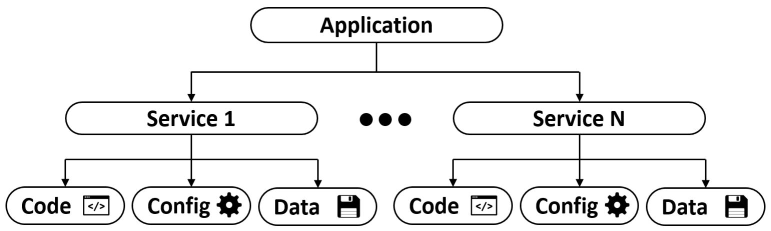


Figure 3.2: **Service Fabric Application Model.** *An application consists of N services, each of them with their own Code, Config. and Data.*

In Service Fabric, an application is a collection of constituent microservices (stateful or stateless), each of which performs a complete and standalone function and is composed of code, configuration and data. This is depicted in Fig. 3.2. The code consists of the executable binaries, the configurations consist of service settings that can be loaded at run time, and the data consists of arbitrary static data to be consumed by the microservice. A powerful feature of SF is that each component in the hierarchical application model can be versioned and upgraded independently.

3.2.2 Service Fabric and Its Goals

As mentioned earlier, Service Fabric (SF) provides first-class support for full Application Lifecycle Management (ALM) of microservice-based cloud applications, from development to deployment, daily management, and eventual decommissioning. The two most unique goals of SF are:

i) Support for Strong Consistency: A guiding principle is that SF's components must *each* offer strong consistency behaviors. Consistency means different things in different contexts: strong consistent failure detection in the membership module vs. ACID in Reliable Collections.

We considered two prevalent philosophies for building consistent applications: build them atop inconsistent components [43, 41, 42], or use consistent components from the ground up. The end to end principle [119] dictates that if the performance is worth the cost for a functionality then it can be built into the middle. Based on our use case studies we found that a majority of teams needing SF had strong consistency requirements, e.g., Azure SQL DB, Power BI etc., all rely on SF while executing transactions. If consistency were instead to only be built at the application layer, each distinct application will have to hire distributed systems developers, spend development resources, and take longer to reach production quality.

Supporting consistency at each layer: a) allows higher layer design to focus on their relevant notion of consistency (e.g., ACID at Reliable Collections layer), and b) allows both weakly consistent applications (key-value stores such as Azure Cosmos DB) and strongly consistent applications (DBs) to be built atop SF–this is easier than building consistent applications over an inconsistent substrate. With clear responsibilities in each component, we have found it easier to diagnose livesite issues (e.g., outages) by zeroing in on the component that is misbehaving, and isolating failures and root causes between platform and application layers.

ii) Support for Stateful Microservices: Besides the stateless microservices (e.g., protocol gateways, web proxies, etc.), SF supports stateful microservices that maintain a mutable, authoritative state beyond the service request and its response, e.g., for user accounts, databases, shopping carts etc. Two reasons to have stateful microservices along with stateless ones are: a) The ability to build high-throughput, low-latency, failure-tolerant online transaction processing (OLTP) services by keeping code and data close on the same machine, and b) To simplify the application design by removing the need for additional queues and caches. For instance, SF's stateful microservices are used by Skype to maintain important state such as address books, chat history, etc. In SF stateful services are implemented via Reliable Collections.

3.2.3 Use Cases: Real SF Applications

Since Service Fabric was made public in 2015 several external user organizations have built applications atop it. In order to illustrate how global-scale applications can be built using microservices, we briefly describe four of these use cases. Our use cases show: a) how real microservice applications can be built using SF; b) how the microservice approach was preferable to users than the monolithic approach; and c) how SF support for state and consistency (in particular Reliable Collections) are invaluable to developers. (This section can be skipped by the reader without loss in continuity.)

Tutorials are available to readers interested in learning how-to build microservice applications over Service Fabric–please see [120].

I. BMW is one of the largest luxury car companies in the world. Their in-vehicle app *BMW Connected* [121] is a personal mobility companion that learns a user's mobility patterns by combining machine-learned driver intents, real-time telemetry from devices, and up-to-date commute conditions such as traffic. This app relies on a cloud service that was built using SF and today runs on Microsoft Azure, supporting 6 million vehicles worldwide.

The SF application is called BMW's *Open Mobility Cloud (OMC)* [122, 123]. It needs to be continually updated with learned behaviors and from traffic commute update streams. OMC consists of several major subsystems. Among them, we will focus on the core component called the Context and Profile Subsystem (C&P). C&P consists of five key SF microservices:

1. **Context API Stateless Service:** Non-SF components communicate with the C&P via this service, e.g., mobile clients can create/change locations and trips.

2. **Driver Actor Stateful Service:** This per-driver stateful service tracks the driver's profile, and generates notifications such as trip start times. It receives data from five sources: sync messages from the Context API service, a stream of current locations of the driver (from Location Consumer service), learned destinations and predicted trips (from MySense machine learning service), deleted anonymous user IDs (from User Delete service), and trip time estimates (from ETA Queue service).

3. **Location Consumer Stateless Service:** Each mobile client sends a stream of geo-locations to the Microsoft Azure Event Hub, pulled by the Location Consumer service and fed to the appropriate driver actor.

4. **Commute Service:** The Commute service takes geo-location and trip start and end points, and then communicates with an external service to generate drive time.

5. **ETA Queue Stateful Service:** This decouples driver actors from the Commute server and allows asynchronous communication between the two services.

The use of SF makes BMW's C&P Subsystem highly-available, fault-tolerant, agile, and scalable. For instance, when the number of active vehicles increases, the Context API service and Driver actor services are scaled out in size. When the number of moving vehicles changes, the Location Consumer and ETA Queue stateful services can be scaled in size. The remaining services remain untouched. SF helps to optimize resource usage so that incurred dollar costs of using Azure are minimized.

II. Mesh Systems [124, 125] is an 11-year old company that provides IoT software and services for enterprise IoT customers. They started out with a monolithic application that was too complex, and were unable to accommodate the needs of their growing business. This previous system also underutilized their cluster.

Mesh System's SF application achieves high resource utilization, and scalability by leveraging Reliable Collections. One of their needs was to scale out the payload processing independent of notifications, and it was a good match with SF's ability to scale out individual microservices. Their SF application also leverages local state to improve performance, e.g., to minimize the load on Azure SQL DB, they implemented an SQL broker that periodically caches the most heavily-accessed metadata tables.

III. Quorum Business Solutions [126, 127] is a SCADA company that collects and manages data from field-operations platforms on tens of thousands of wells across North America. Their implementation on SF uses actors that reliably collect and process data because they are stateful, a stateless gateway service for auto-scalability, and a stateful batch aggregator service that monitors actors themselves. They implement interactions with third parties (SQL DB, Redis) via notification and retry microservices in SF.

IV. TalkTalkTV [128, 129] is one of the largest cable TV providers in United Kingdom. It delivers the latest TV and movie content to a million monthly users, via a variety of devices and smart TVs. Their SF application is used to encode movie streams before delivery to the customer, and uses stateful services, structured in a linear sequence: record encoding

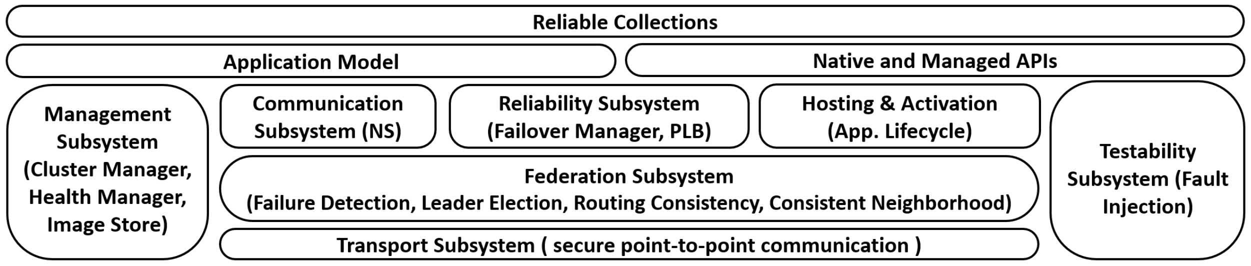

Reliable Collections
Application Model
Native and Managed APIs
Management Subsystem (Cluster Manager, Health Manager, Image Store)
Communication Subsystem (NS)
Reliability Subsystem (Failover Manager, PLB)
Hosting & Activation (App. Lifecycle)
Testability Subsystem (Fault Injection)
Federation Subsystem (Failure Detection, Leader Election, Routing Consistency, Consistent Neighborhood)
Transport Subsystem (secure point-to-point communication)

Figure 3.3: **Major Subsystems of Service Fabric.** *NS = Naming Service, PLB = Placement and Load Balancer.*

requests, initiate encoding processes, and track these processes. A stateless gateway interacts with clients.

3.3 SYSTEM OVERVIEW

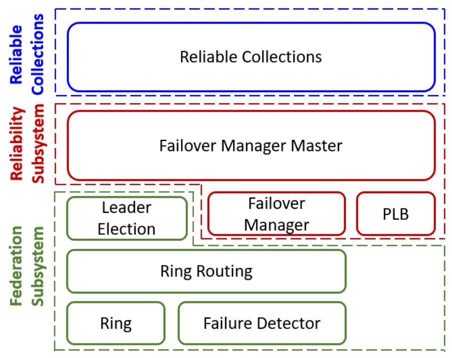


Figure 3.4: **Federation and Reliability Subsystems: Deep-Dive.**

Service Fabric (SF) is composed of multiple subsystems, relying on each other modularly via clean APIs and protocols. Fig. 3.3 depicts how they are stacked–upper subsystem layers leverage lower layers. Given space constraints, this chapter largely focuses on SF's most unique components, shown in Fig. 3.4. These lie in two subsystems: Federation and Reliability.

The *Federation Subsystem* (Sec. 3.4.1) forms the heart of SF. It solves critical distributed systems problems like failure detection, a consistent ring with routing, and leader election. The *Transport Subsystem* underneath provides secure node-to-node communication.

Built atop the Federation Subsystem is the *Reliability Subsystem* (Sec. 3.4.2), which provides replication and high availability. Its components are the Failover Manager (FM), Failover Manager Master (FMM), the Placement and Load Balancer (PLB), and replication protocols. This helps create distributed abstractions named Reliable Collections (Sec. 3.4.3).

Other SF subsystems not detailed in this chapter include the Management Subsystem which provides full application and cluster lifecycle management via the Cluster Manager, Health Manager, and Image Store. The Communication Subsystem allows reliable service discovery via the Naming Service. The Testability Subsystem contains a Fault Injection Service. Hosting and Activation Subsystems manage other parts of the application lifecycle.

3.4 SYSTEM DESIGN

This section covers the design details of the core modules of Service Fabric. Federation Subsystem (Sec. 3.4.1) describes the unique, consistent failure detector. Sec. 3.4.2 and 3.4.3 respectively describe the Failover Manage and Reliable Collections, that uses the lower layer's consistency guaranties to ensure their own form of consistency.

3.4.1 Federation Subsystem

This section describes SF's ring, failure detection, consistent routing, and leader election.

3.4.1.1 Basic SF-Ring

Nodes in SF are organized in a virtual ring, which we call *SF-Ring*. This consists of a virtual ring with 2^m points (e.g., $m = 128$ bits). Nodes and keys are mapped on to a point in the ring. A key is owned by the node *closest* to it, with ties won by the predecessor. Each node keeps track of multiple (a given number of) its immediate successor nodes and predecessor nodes in the ring–we call this the *neighborhood* set. Neighbors are used to run SF's membership and failure detection protocol, which we describe next.

Nodes also maintain long-distance routing partners. Sec. 3.4.1.5 will later outline these and consistent routing.

3.4.1.2 Consistent Membership and Failure Detection

Membership and failure detection in SF relies on two key design principles:

- **Strongly Consistent Membership:** All nodes responsible for monitoring a node X must agree on whether X is up or down. When used in the SF-Ring, this entails a *consistent neighborhood*, i.e., all successors/predecessors in the neighborhood of a node X agree on X's status.

- **Decoupling Failure Detection from Failure Decision:** Failure detection protocols can lead to conflicting detections. To mitigate this, we decouple the *decision* of which nodes are failed from the *detection* itself.

3.4.1.3 Lease-based Heartbeating

We first describe our heartbeating protocol in general terms, and then how it is used in SF-Ring.

Monitors and Leases: Heartbeating is fully decentralized. Each node X is monitored by a subset of other nodes, which we call its *monitors.* Node X periodically sends a lease renewal request (LR, heartbeat message with unique sequence number) to each of its monitors. When a monitor acknowledges (LR_{ack}), node X is said to obtain a lease, and the monitor guarantees not to detect X as failed for the leasing period. The leasing period, labeled T_m, is adjusted adaptively based on round trip time and some laxity, but a typical value is 30 s. To remain healthy, node X must obtain acks (leases) from *all* of its monitors. This defines strong consistency. If node X fails to renew *any* of its leases from its monitors, it considers removing itself from the group. If a monitor misses a heartbeat from X, it considers marking X as failed. In both these cases however, the final decision needs to be confirmed by the arbitrator group (described in Sec. 3.4.1.4).

Lease renewal is critical, but packet drops may cause it to fail. To mitigate this, if node X does not receive LR_{ack} within a timeout (based on RTT), it re-sends the lease message LR until it receives LR_{ack}. Resends are iterative.

Symmetric Monitoring in SF-Ring: The monitors of a node are its neighborhood (successors and predecessors in the ring). Neighborhood monitoring relationships are purely *symmetric.* When two nodes X and Y are monitoring each other, their lease protocols are run largely independently, with a few exceptions. First, if X fails to renew its own lease within the timeout, it denies any further lease requests from Y (since X will leave the group soon anyway). Second, if X detects Y as having failed, X stops sending lease renew requests to Y. Such cases have the potential to create inconsistencies, however our use of the arbitrator group (which we describe next) keeps the membership lists consistent.

3.4.1.4 Using the Arbitrator Group to Decouple Detection from Decision

Decoupling Failure Detection from Decision: Decentralized failure detection techniques carry many subtleties involving timeouts, indirection, pinging, etc. Protocols exist that give eventual liveness properties (e.g., [36, 130]), but in order to scale, they allow inconsistent membership lists. However, our need is to maintain a strongly consistent neighborhood in the ring, and also reach decisions quickly.

To accomplish these goals, we decouple *decisions* on failures from the act of detection. Failure detection is fully decentralized using Sec. 3.4.1.3's lease-based heartbeating. For decisions, we use a *lightweight* arbitrator. The arbitrator does not help in detecting failures (as this would increase load), but only in affirming or denying decisions.

Arbitrator: The arbitrator acts as a referee for failure detections, and for detection conflicts. For speed and fault-tolerance, the arbitrator is implemented as a decentralized group of nodes that operate independent of each other. When any node in the system detects a failure, before taking actions relevant to the failure, it needs to obtain confirmation from a majority (quorum) of nodes in the arbitrator group.

Failure reporting to/from an arbitrator node works as follows. Suppose a node X detects Y as having failed. X sends a fail(Y) message to the arbitrator. If the arbitrator already marked X as failed, the fail(Y) message is ignored, and X is again asked to leave the group. Otherwise, if this is the first failure report for Y, it is added to a *recently-failed list* at the arbitrator. An accept(fail(Y)) message is sent back to X within a timeout based on RTT (if this timeout elapses, X itself leaves the ring). The accept message also carries a timer value called T_o, so that X can wait for T_o time and then take actions w.r.t. Y (e.g., reclaim Y's portion of the ring).

When Y next attempts to renew its lease with X (this occurs within T_m time units after X detects it), X either denies it or does not respond. Y sends a fail(X) message to the arbitrator. Since Y is already present in the recently-failed list at the arbitrator, Y is asked to leave the group. (If this exchange fails, Y will leave anyway as it failed to renew its lease with X.) If on the other hand, Y's lease renewal failed because X was truly failed, then the arbitrator sends an accept(fail(X)) message to Y. We set: $T_o = T_m + laxity$ - (time since first detection). If this is the first detection, $T_o = T_m + laxity$. Here, *laxity* is typically 30 s, generously accounts for network latencies involved in arbitrator coordination, and independent of T_m. As all timeouts are large (tens of seconds), loose time synchronization suffices.

In SF-Ring: Inside SF-Ring, failure detections occur in the neighborhood, to maintain a consistent neighborhood. If node X suspects a neighbor (Y), it sends a fail(Y) to the arbitrator, but waits for T_o time after receiving the accept(.) message before reclaiming the portion of Y's ring. Any routing requests (Sec. 3.4.1.5) received meanwhile for Y will be queued, but processed only after the range has been inherited by Y's neighbors.

Arbitrator State: In the arbitrator group, each arbitrator keeps relatively small information, such as the recently-failed list containing recent failure reports and decisions. Entries in this list time out after T_m time units. When a new arbitrator joins the group, for the first T_m

seconds it rejects all failure requests (this is a conservative approach). After quick initialization it moves to normal operations as described earlier. This prevents: a) decisions by a new arbitrator conflicting with those by existing arbitrators, and b) spurious nodes, i.e., failed nodes continuing to persist in membership lists (a known issue in distributed membership protocols [130]). This ensures that a detected node leaves before being forgotten.

Conflict Resolution: The arbitrator group helps decide on both simple and complex conflicts. A common simple conflict is two nodes detecting each other as failed–the first received fail(.) message (or the first one to win quorum among arbitrators) wins in this case. An alternate variant of our arbitrator pings both such nodes and if they are healthy heals their membership lists and allows them to stay.

Network congestion or partitions may result in multiple nodes detecting each other as failed. In traditional DHTs like Chord [131], Pastry [132], this causes inconsistent membership lists. In NoSQL systems like Cassandra [133], it can lead to inconsistency in the ring. SF's arbitrator group essentially automates the conflict resolution procedure.

The decoupling of detection and decision helps the arbitrators catch and nip complex *cascading detections*. For instance, consider a node X that fails to renew its lease and thus voluntarily leaves. If another node Y immediately happens to send a lease request to X (before Y has been informed about X), Y will not receive an ack and will also leave–this process can cascade and result in many healthy nodes leaving. SF's arbitrators catch the first detection, and immediately make the neighborhood consistent, thus stopping the cascade early.

Vs. Related Work: SF's leases are comparable to heartbeat-style failure detection algorithms from the past (e.g., [130]). The novel idea in SF is to use lightweight arbitrator groups to ensure membership stays consistent (in the ring neighborhood). This allows the membership, and hence the ring, to scale to whole datacenters. Without the arbitrators, distributed membership will have inconsistencies (e.g., gossip, SWIM/Serf [36]), or one needs a heavyweight central group (e.g., Zookeeper [40], Chubby [47]) which has its own issues. Stronger consistent membership like virtual synchrony [38, 39] do not scale to datacenters.

3.4.1.5 Full SF-Ring and Consistent Routing

We describe the full SF-Ring, expanding on the basic design from Sec. 3.4.1.1. SF-Ring is a distributed hash table (DHT). It provides a seamless way of scaling from small groups to large groups. SF-Ring was developed internally [134, 135, 136, 137] in Microsoft, in the early 2000s, concurrent with the emergence of P2P DHTs like Pastry, Chord [132, 131], and others [138, 139, 140, 141]. We describe our original design, and inline evolutionary changes that occurred over time.

SF-Ring is unique in the following five ways (I-V):

I) Routing Table entries are bidirectional and symmetric: SF-Ring maintains Routing Partners (in Routing Tables) at exponentially increasing distances in the ring. As shown in Fig. 3.5, routing partners are maintained both clockwise and anticlockwise. That is, the i^{th} clockwise routing table entry is the node whose ID is closest to the key $(n + 2^i)mod(2^m)$, while the i^{th} anticlockwise routing table entry is the node with ID closest to $(n-2^i)mod(2^m)$.

Due to the bidirectionality, most routing partners are *symmetric*. This speeds up both spread of failure information and routing. P2P DHTs like Chord maintain exponentially far routing entries, but are unidirectional and largely not symmetric. Symmetric links lead to efficient transfer of data between nodes, fast spreading of failure information, and fast updating of routing tables after node churn [1].

II) Routing is bidirectional: When forwarding a message for a key, a node searches its routing table for the node whose ID is *closest* to the key, and forwards it. This is possible only because the routing tables are bidirectional and symmetric. This greedy routing is essentially a distributed version of binary search. This approach: i) allows the message to move both clockwise and anticlockwise, always taking the fastest path, and ii) avoid routing loops. In practice we noticed that once a message starts routing it tends to maintain its direction (clockwise or anticlockwise), until the last few hops, when directional changes may occur.

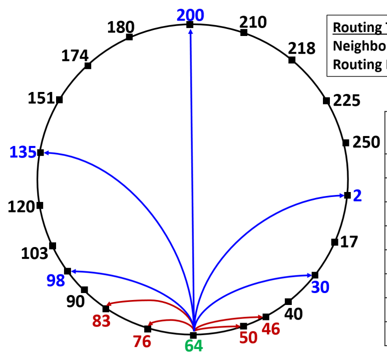
Compared to traditional DHTs like Chord which use clockwise routing, SF-Ring's bidirectional routing: i) routes messages faster, ii) provides more routing options when routing table entries are stale or empty, iii) spreads routing load more uniformly across nodes, and iv) due to the distributed binary search, avoids routing loops even under stale routing tables.

Changes to Routing over the Years: Once a building block is designed, its usage evolves over the years based on needs. SF-Ring's routing is no exception. Originally all messages were routed. Today, SF-Ring routing is used for: a) discovery routing when a node starts up, and b) routing to virtual addresses. After discovery when a source knows the destination's IP address, it communicates directly.

III) Routing Tables are eventually convergent: SF nodes use a chatter protocol to continuously exchange routing table information. Due to the symmetric nature of routing relationships, failure information propagates quickly leading to fast and eventual convergence of affected routing table entries.

When a node joins the ring, it goes through a transitional phase during which it initializes

[1]Symmetry may be violated in a small fraction of cases when another node is closer to $(n-2^i)mod(2^m)$, but our advantages still hold.



Routing Table at node 64:
Neighborhood = {83,76,50,46}
Routing Partner = {200,2,30,98,135,200}

Routing Partner Calculation

Clock.	Active Node	Anti-Clock.	Active Node
$64+2^0$	64	$64-2^0$	64
$64+2^1$	64	$64-2^1$	64
$64+2^2$	64	$64-2^2$	64
$64+2^3$	76	$64-2^3$	50
$64+2^4$	83	$64-2^4$	46
$64+2^5$	98	$64-2^5$	30
$64+2^6$	135	$64-2^6$	2
$64+2^7$	200	$64-2^7$	200

Figure 3.5: **Routing Table** *of node 64. The ring has* $2^{m=8}$ *points. Numbered dots represent active nodes.*

its routing tables, acquires tokens (described soon), but does not yet route messages. Once it finishes transitioning it starts routing messages.

In the chatter protocol, nodes periodically send liveness messages to their routing table partners. To efficiently use bandwidth, liveness messages also allow piggybacking of application messages that are about to be sent to the partner. Liveness messages contain information about the node, instance ID (distinguishes multiple rejoins by the same node), phase (e.g., transitioning or operational), weight (reputation based on uptime, etc.), and a freshness value (which decays with time, like in ad-hoc routing protocols [142, 143]).

The chatter protocol provides eventual consistency for the long distance neighbors (routing partners). A key result from the SF effort is that strongly consistent applications can be supported at scale by combining strong membership in the neighborhood (Sec. 3.4.1.2) with weakly consistent membership across the ring. Literature often equates strongly consistent membership with virtual synchrony [38], but this approach has scalability limits [39].

Changes to Partial Membership over the Years: SF microservices operate in a wide range of scales, from a few nodes to many 1000s of nodes. Microservices also need to scale out and scale in during their lifetime. To support this, today SF-Ring sets a (customizable) upper bound on the number of entries in the routing table. If the number of nodes is smaller than this bound, the routing tables (eventually but quickly) capture full information about all nodes; this makes routing fast and take $O(1)$ hops. If the number of nodes is higher, routing tables come into effect, creating an $O(log(N))$ lookup cost without blowing up memory.

This design decision allows SF-Ring to move seamlessly between partial membership and full membership. In comparison, NoSQL ring-based DHTs like Cassandra [133] and Dynamo [144] rely on full $O(N)$ membership. This makes them cumbersome at large scale and under churn–the overhead of maintaining correct membership lists outweighs benefits of 1 hop routing. In Cassandra, admins need to use *nodetool* [145] to manually verify that membership lists are correct. SF-Ring automates all membership management.

IV) Decoupled Mapping of Nodes and Keys: Nodes and objects (services) are mapped onto the ring in a way that is decoupled from the ring: i) nearby nodes on the ring are selected preferably from different failure domains (improving fault-tolerance), and ii) services are mapped in a near-optimal and load balanced way (Sec. 3.4.2.3), and not hashed.

V) Consistent Routing Tokens: Each SF node owns a *routing token*, a portion of the ring whose keys it is responsible for. The SF-Ring protocol ensures two *consistency* properties: *i) always safe*: there is no overlap among tokens owned by nodes, and *ii) eventually live:* every token range is eventually owned by at least one node. When the first SF node bootstraps, it owns the entire token space. Thereafter, tokens are created as follows: two immediate neighbors split the ring segment between them at exactly the half-way point.

Upon churn, a node join/leave protocol automatically transfers tokens among nodes. NoSQL systems like Cassandra [133] also use routing tokens, but may need manual involvement to ensure correctness (via nodetool). In SF-Ring, this checking is automatic and continuous.

When a node leaves, its successor and predecessor split the range between them halfway. If a node X's immediate successor Y fails, then X and its new successor Z will split the ring segment halfway between X and Z. In the common case this splitting incurs no communication between X and Z.

If all nodes satisfy token liveness and safety conditions, SF-Ring routing will eventually succeed. If the liveness condition is not yet true (e.g., no node owns a token containing destination ID), routing messages are queued.

Vs. Related Work: SF's consistent ring was invented internally around 2002, concurrent with the first DHTs like Chord [131] and Pastry [132]. While SF was being implemented, several other DHTs came out that used bidirectional routing, e.g., Kademlia [138]. While we could conceivably go back and try replacing SF-Ring with something like Kademlia, re-integration is hard and SF-Ring has been running successfully in production for a decade (if it ain't broke, don't fix it!).

3.4.1.6 Leader Election

SF-Ring's leader election protocol builds atop the combination of the ring, routing, and consistent neighborhood just described. For any key k in the SF-Ring, there is a unique leader: the node whose token range contains k (this is unique due to the safety and liveness of routing tokens). Any node can contact the leader by routing to key k. Leader election is thus implicit and entails no extra messages. In cases where a leader is needed for the entire ring we use $k = 0$ (e.g., FMM in Sec. 3.4.2.1).

3.4.2 Reliability Subsystem

The Reliability Subsystem is in charge of replication, load balancing, and high availability. Objects in SF are replicated at a primary node and multiple secondary nodes. The replication subsystem's *Replicator* component uses passive replication: clients communicate with the primary, which multicasts updates to secondaries. The Reliability Subsystem contains three major components: Failover Manager (FM), Naming, and Placement and Load Balancer (PLB).

3.4.2.1 The Failover Manager (FM)

This stateful SF service maintains a global view of all replica groups. The global view includes status of all nodes in the cluster, list of current applications and services, list of replicas and their placement, etc.

The FM manages creation of services, upgrades, etc. It works closely with daemons on each node called Reconfiguration Agents (RAs), which continually collect the node's available memory, CPU utilization, disk and network access behaviors, etc. The FM coordinates with the Placement and Load Balancer (PLB) (Sec. 3.4.2.3). The FM periodically receives load reports from the RAs running on each node, aggregates, and sends it to the PLB. Newly joined nodes explicitly inform the FM, and failures are detected via the mechanisms of Sec. 3.4.1.2 and relayed from the arbitrator to the FM as they occur. The FM's main actions are:

1. **Create a Replica:** When either: $\underline{\text{a)}}$ a replica is created for the first time, or $\underline{\text{b)}}$ a replica goes down and FM has to re-create it. In both cases FM consults with PLB which decides the placement for services/replicas, and FM initiates the placement.

2. **Move a Replica:** When an imbalance occurs, PLB calculates a replication migration plan, and FM executes it.

3. **Reconfiguration:** If a primary replica goes down, the FM selects a secondary replica and promotes it as primary. If the old primary comes up, it is marked as a secondary.

The failure of an entire FM (replica set), though rare, still needs fast recovery. This is handled by another stateless service called the **Failover Manager Master (FMM)**, which runs the same logic as the FM, except that it manages the FM instead of the microservices. If an FM fails the FMM restarts it quickly with cached state. If the FMM itself fails, it reconstructs its state from scratch by querying the SF-Ring. In SF-Ring, the FMM is elected consistently using the election protocol of Sec. 3.4.1.6, i.e., as the node whose token range contains the ID 0.

3.4.2.2 Naming and Resolution

Service Fabric's Naming Service maps service instance names to the endpoint addresses they listen on. All service instances in SF have unique names represented as URIs– a typical format is `SF:/MyApplication/MyService`. The name of the service does not change over its lifetime, only the endpoint address binding can change (e.g., if the service is migrated). Full names are DNS-style hierarchical names, e.g., `http://mycluster.eastus.cloudapp.microsoft.com:19008/MyApp/MyService?PartitionKey=3&PartitionKind=Int64Range`. This allows DNS to resolve the prefix, and SF's Naming Service to resolve the rest. The FM (Sec. 3.4.2.1) also caches name-target mappings for fast resolution and to make fast decisions upon failures.

3.4.2.3 Placement and Load Balancer (PLB)

The Placement and Load Balancer (PLB) is a stateful SF service in charge of placing replicas/instances (of microservices) at nodes and ensuring load balance. Unlike traditional DHTs, where object IDs are hashed to the ring, the PLB explicitly assigns each service's replicas (primary and secondaries) to nodes in SF-Ring. It takes into account: i) available resources at all nodes (e.g., memory, disk, CPU load, traffic, etc.), ii) conceptual resources (e.g., outstanding requests at a particular service), and iii) parameters of typical requests (e.g., request size, frequency, diurnal variation, etc.). The PLB's continuous role is to move sets of services from overly exhausted nodes to underutilized nodes. It also moves services away from a node that is about to be upgraded or is overloaded due to a long workload spike.

Large State Space: In practice the PLB needs to deal with a state space that is both huge (hundreds of different metrics and values, conflicting requirements, etc.), and occasionally quite constrained (e.g., placement of services only on certain nodes, fault-tolerance by avoiding replica colocation, etc.). A typical scenario involves tens of thousands of objects, replicated 3-ways, but spread over only a few hundred nodes. Worse, things change frequently: which resources are important, how many resources a particular workload is actually consuming, what the workload's constraints are, which nodes are failing and joining, etc., all change during the runtime of the service. This means that the decision taken currently might not be valid in future. Therefore, it is better to continuously make small improvements and re-evaluate them later. Quick and nimble decisions are preferable over algorithms that try to reach an optimal state but use up a lot of resources to explore the state space.

Simulated Annealing: In order to select a near-optimal placement of objects across nodes given the above constraints, the PLB uses simulated annealing [146]. We initially attempted to use LP/IP-based and genetic algorithms [147, 148, 149] but found they either took too long to converge or gave solution which were far from optimal. We picked simulated annealing as it bridged these worlds: it is both fast and close to optimal.

Simulated annealing calculates an energy for each state. PLB's energy function is user-definable but a common case is as the average standard deviation of all metrics in the cluster, with a lower score being more desirable. The simulated annealing algorithm sets a timer (default values later) and then explores the state space until either the timer expires or until convergence. Each step generates a random move, considers the energy of the new prospective state due to this move, and decides whether to jump. If the new state has lower energy the annealing process jumps with probability 1; otherwise if the new state has d more energy than the current and the current temperature is T, the jump happens with probability $e^{-\frac{d}{T}}$. This temperature T is high in initial steps (allowing jumps away from local minima) but falls linearly across iterations to allow convergence later.

The move chosen in each step is fine-grained. Examples include moving a secondary replica to another node, swapping primary and secondary replica, etc. SF only considers valid moves that satisfy constraints: i) under which the PLB operates, and ii) for fault-tolerance. For instance, the PLB cannot create new nodes, nor can it move a primary replica to colocate with a secondary replica of the same partition.

SF supports two modes of annealing: fast mode (10 s timer value), and a slow mode (120 s timer) that is more likely to converge to the optimal. During initial placement we run annealing for only 0.5 s.

When the annealing ends, the energy of the system's current state is recalculated (as it may have changed), and the new state is initiated only if it actually improves the energy. Moves are compacted using transitivity rules and are sent to the FM to execute.

3.4.3 Reliable Collections

Reliable Collections provide stateful services in SF. All the use cases described in Sec. 3.2.3 directly used Reliable Collections. Internally its biggest users are Microsoft Intune and Microsoft CRM Service..

SF's Reliable collections include Reliable Dictionary and Reliable Queue, available as classes in popular software programming frameworks. These data structures are:

- **Available and Fault-tolerant**: Via replication;
- **Persisted**: Via disk, to withstand server, rack, or datacenter outages;
- **Efficient**: Via asynchronous APIs that do not block threads on IO;
- **Transactional**: Via APIs with ACID semantics.

A key difference between storage systems built using SF APIs (e.g., Reliable Collections) and other highly-available systems (such as Azure Queue Storage [150], Azure Table Storage [151], and Redis [152]) is that the state is kept locally in the service instance while also being made highly available. Therefore, the most common operations i.e., reads, are local.

Writes are relayed from primary to secondaries via passive replication, and are considered complete when a quorum of secondaries acknowledge it. Further extension points allow an application to achieve weaker consistency by relaxing where the read can go, e.g., "always read from primary" to "read from secondary." Our users who build latency-sensitive applications find this particularly useful.

Applications can quickly failover from a failed node to a hot standby replica. Groups of applications can be migrated from one node to another during maintenance such as for patching or planned restarts.

Benefits of Reliance on Lower Layers: Reliable Collections leverage the components described previously in this paper. Replicas are organized in an SF-Ring (Sec. 3.4.1.5), failures are detected (Sec. 3.4.1.2), and a primary kept elected (Sec. 3.4.1.6). Periodically, as well as when replica changes occur (node joins, failures, leaves, etc.), FM+PLB (Sec. 3.4.2) keeps the replicas fault-tolerant and load-balanced.

SF is the only self-sufficient microservice system that can be used to build a transactional consistent database which is reliable, available, self-*, and upgradable. The developer only has to program with the Reliable Collections API; because lower layers assure consistency, she does not have to reason about those. Today there are 1.82 Million such transactional DBs over SF (100K machines).

3.5 EVALUATION

We evaluate the most critical aspects of Service Fabric: failure detection and membership, message delay, reconfiguration, and SF-Ring. Where available, we present results from production data (Sections 3.5.3, 3.5.4, 3.5.5). We use simulations in cases where we need to compare to alternative designs in a fair way (Sections 3.5.1, 3.5.6), or measure algorithmic overhead (Sections 3.5.1, 3.5.2).

We have made Service Fabric binaries available [120]. We are looking into sharing datasets, however we are limited by compliance reasons and proprietary issues. We are working on open-sourcing the code for SF.

3.5.1 Benefits of the Arbitrator

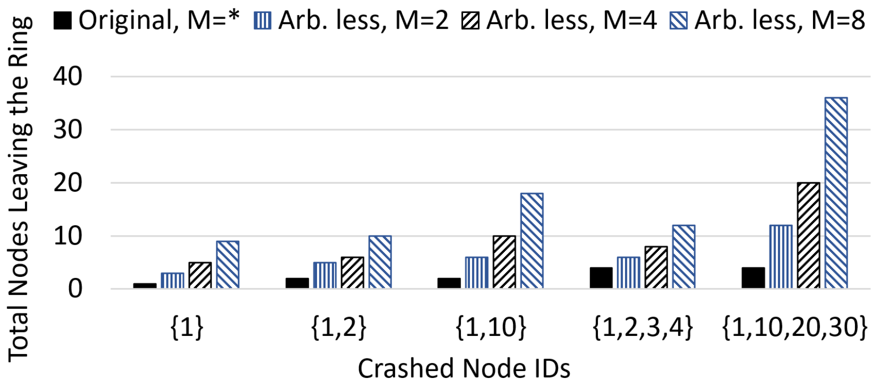


Figure 3.6: **Comparison: Arbitrator Vs. Arbitrator less scheme.** *Arbitrator handles cascading failures and reduces the number of nodes leaving the system. M = number of monitor per node.*

To show that SF's arbitrator mechanism (Sec. 3.4.1.4) efficiently helps maintain consistent membership, we compare it to an arbitrator-less mechanism we designed. In the latter approach, when a node fails to renew its lease, instead of contacting the arbitrators, it coordinates with its neighbors and then gracefully leaves the system. Neighbors communicate amongst each other to keep membership consistent.

Due to timeouts, both mechanisms may force good nodes to leave. Fig. 3.6 shows, for various failure scenarios, the total number of such false positives. We observe that SF's original arbitrator approach incurs far fewer false positives than the arbitrator-less scheme. In fact, the number of false positives under an arbitrator based scheme grows much slower (with number of failure) than under the arbitrator-less scheme. This is because of cascading failure detections (Section 3.4.1.4), while SF's arbitrator prevents such cascades.

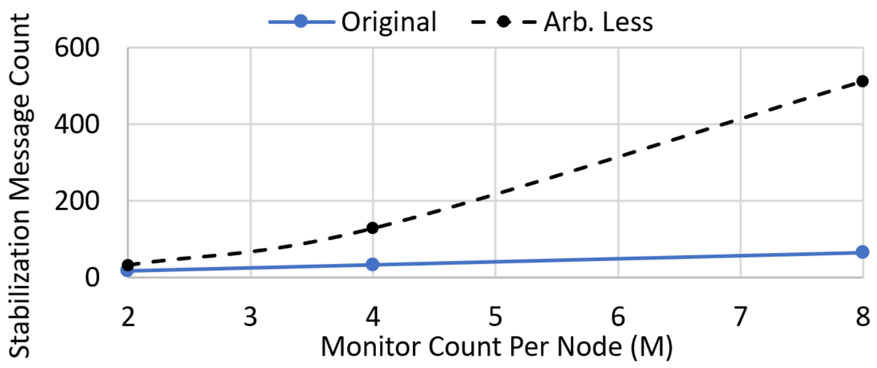


Figure 3.7: **Stabilization Message Count: Arbitrator Vs. Arbitrator less scheme.** *Crashed Node set {1, 10, 20, 30}. M = number of monitor per node.*

Fig. 3.7 shows how many messages are needed to stabilize the ring, after a failure. As we increase the number of monitors per node (M), SF's arbitrator's overhead grows slower than the arbitrator-less scheme. In fact, analytically, these overheads are linear and quadratic respectively. When using arbitrators, a failure causes the M monitors of a failed node to perform a request-reply to the arbitrator ($2M$ messages). In arbitrator-less approaches, a failure causes all M monitors to communicate with each other ($2M^2$ messages).

3.5.2 Failure Detector Overhead

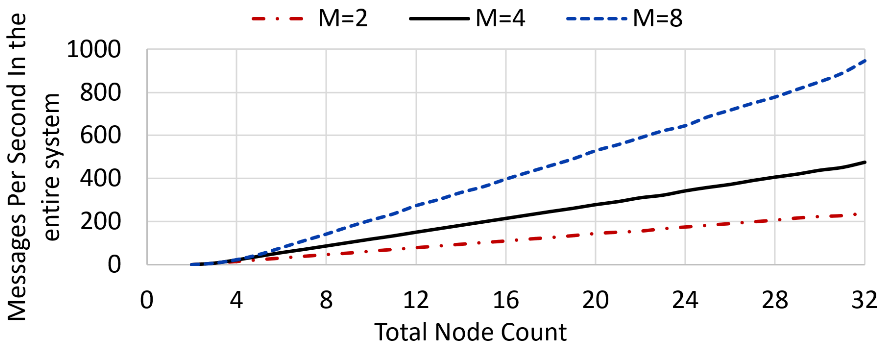


Figure 3.8: **Failure Detector (FD) message overhead.** *Cluster messages increase linearly with cluster size. M = number of monitors per node.*

Fig. 3.8 shows the *total cluster* overhead of the leasing mechanism (Sec. 3.4.1.3). For each M, cluster load scales linearly with the number of nodes (production SF uses $M = 4$ monitors per node). Hence SF's leasing mechanism incurs per-node overhead that is constant and scalable, independent of cluster size.

3.5.3 In Production: Arbitrator Behavior

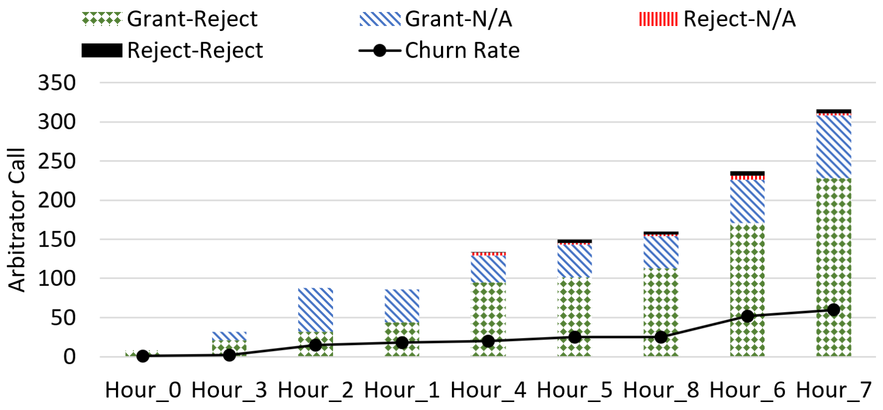


Figure 3.9: **Arbitrator Call count per hour (total 9 hours of traces).** *Hours are rearranged based on the churn rate.*

Fig. 3.9 evaluates the load on the arbitrator group. The data is from 9 hours of a 225+ machine production cluster. Each machine hosts an expected 4 SF instances. Below, we call each of these instances a "node". We sort trace hours in increasing order of churn, and the plot shows both event counts and hourly churn rate.

We explain the 4 event types. When a node A detects failure of node B and contacts the arbitrator, there are four possible outcomes: i) Grant-Reject: both nodes A and B send arbitration requests and only one is granted; ii) Reject-Reject: both nodes A and B send arbitration requests, and both of them are rejected; iii) Grant-N/A: only one node in a pair sends request and succeeds; iv) Reject-N/A: only one node in a pair sends request and is rejected.

First we observe that a majority of hours have medium churn with between 10-15 nodes churned per hour (Hours_2, 1, 4, 5, 8). Only 22% of hours (Hours_6, 7) have very high churn, and another 22% have low churn (Hours_0, 3). Second, the number of duplicate messages received at the arbitrator, and the wholesale rejections at arbitrator startup (first T_m time units: see Sec. 3.4.1.4), are together captured by the sum of Reject-Reject and Reject-N/A events (two topmost bar slivers). This is small at medium churn (Hours_4, 5, 8), and does not increase much at high churn (Hours_6, 7). Hence we conclude that: i) the arbitrator's effort is largely focused on resolving new detection conflicts rather than re-affirming past decisions to errant nodes; and ii) false positives due to arbitrator startup are small in number. Our data also indicates SF nodes leave quickly after they are asked to.

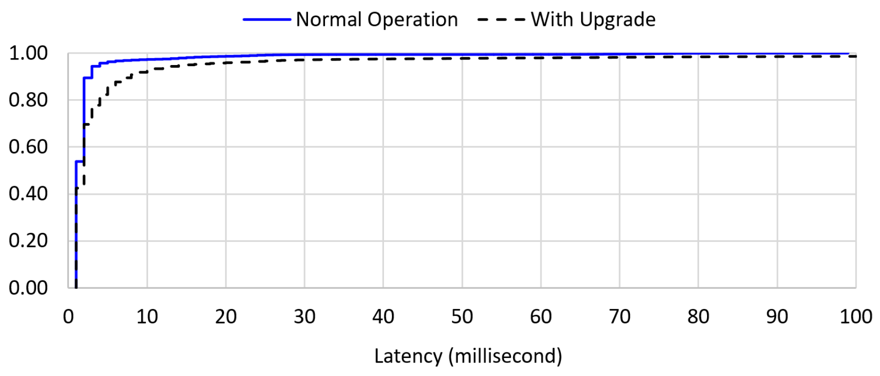


Figure 3.10: **CDF of Message Delay under Churn.** *Normal Operation has lower churn than with Upgrade.*

	1st Perc.	5th Perc.	50th Perc.	95th Perc.	99th Perc.
No Churn	1	1	1	4	24
Churn	1	1	2	14	175

Table 3.2: **Tail latency:** *Message Delay (millisecond).*

3.5.4 In Production: Message Delay Under Churn

Fig. 3.10 measures the total message delay (including routing latency) in a 24 hour trace of 205 VMs across 3 data-centers. Each VM is equipped with 24 cores, 168 GB RAM, 3×1.81 TB HDD and 4×445 GB SSD.

The plot shows the latency CDF for two scenarios: i) *Normal Operation*, prone to natural churn, e.g., due to failures; and ii) *With Upgrade*, when there is higher churn due to node upgrades, system upgrades, service upgrades, etc. Going from normal operation to upgrades, the 80^{th} percentile latency remains largely unaffected. Table 3.2 shows the median latency rises only two-fold, from 1 ms to 2 ms. $95th$ percentile latency rises a modest $3.5\times$. We conclude that SF deals with churn and upgrades in a low-overhead way.

3.5.5 In Production: Reconfiguration Time

Reconfigurations triggered by service failure, overloaded nodes, service upgrade, machine failure, system upgrades, etc., are handled by the FM and PLB (Sections 3.4.2.1, 3.4.2.3).

Failover	SwapPrimary	Other
1%	20%	79%

Table 3.3: **Different Reconfiguration Events.** *Over a 20-day trace.*

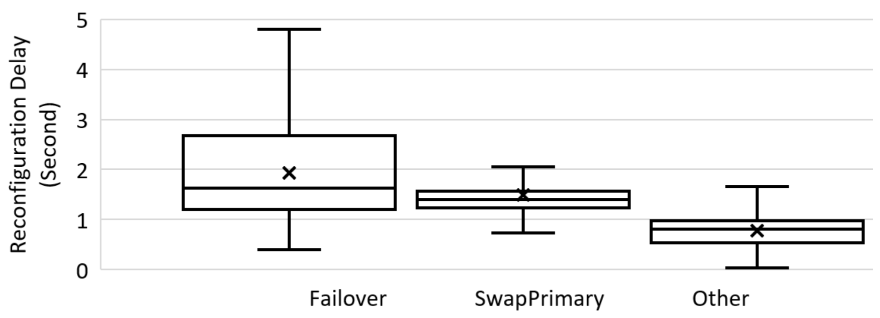


Figure 3.11: **Statistics of different Reconfiguration Delays in the 20 day trace.** *Candlestick plots show the* 1st *and* 99th *percentiles,* 1st, 2nd *and* 3rd *quartiles and the average (X's).*

We collected a 20 day trace with 3 Million events from the same production cluster as Sec. 3.5.4. Table 3.3 shows a breakdown by reconfiguration type. Only Failover and SwapPrimary events affect availability (total 21%). Fig. 3.11 shows that SF makes control decisions about these two types of events quickly. The average time to perform failover is 1.9 s, and 99th percentile is 4.8 s. While "Other" events constitute 79%, they do not affect data availability as they deal with per-replica reconfiguration, and are quite fast.

Fig. 3.12 depicts a timeline over 6 days of these 3 event types. Large spikes are due to preplanned upgrades of infrastructure, application, and SF. Otherwise, we observed no fixed or predictable patterns (e.g., periodic, diurnal). This indicates that modeling+prediction approaches would be excessive, and instead SF's reactive approach is preferable.

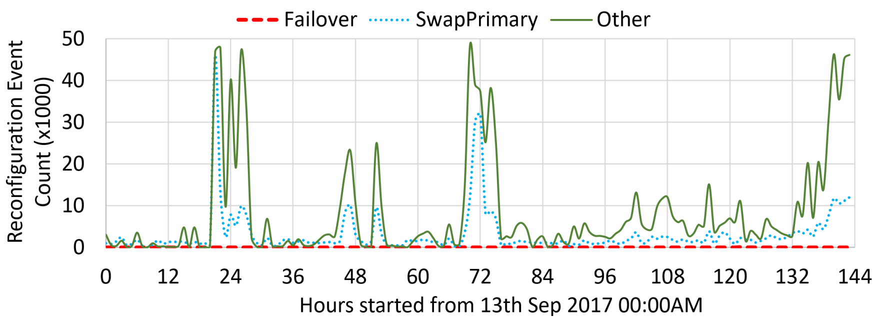


Figure 3.12: **Reconfiguration Event count per hour.** *Started from* 13th *September 2017 00:00AM.*

Fig. 3.13 shows the time to execute a reconfiguration. Across the week, we observe very stable reconfiguration times. Tail latency is within 2.2 s and the average latency hovers at around 1.0 s. This is the time to execute control actions for the reconfiguration, after simulated annealing and in parallel with data transfer (which itself is dependent on the size of the object). Overall, we conclude that SF reconfigures replicas very quickly and predictably.

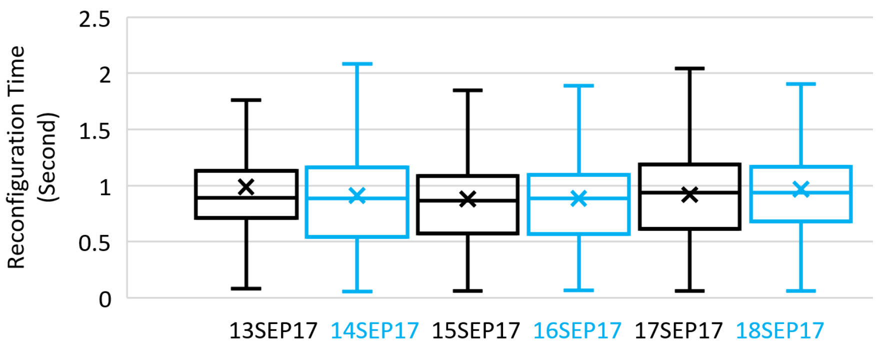


Figure 3.13: **Statistics of Reconfiguration Delay (at placement) across six days.** *Candlestick plots show the* 1st *and* 99th *percentiles,* 1st, 2nd *and* 3rd *quartiles and the average (X's).*

3.5.6 SF-Ring vs. Chord

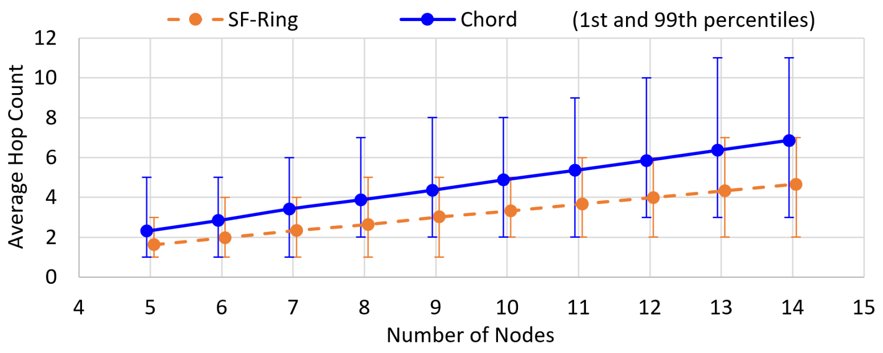


Figure 3.14: **SF-Ring vs. Chord: Hop Count as function of system size (log scale).** *Points perturbed slightly (±0.05 on X-axis) for clarity.*

We faithfully implemented a simulation of both SF-Ring and Chord [131] routing. Fig. 3.14 shows that SF-Ring messages transit 31% fewer hops in the ring than Chord. At the 1st percentile the savings is 20% and at the 99th percentile it is 34%. The slope of the SF-Ring and Chord lines are respectively 0.34 and 0.5. Therefore when the number of nodes doubles Chord requires 49.27% more hops than SF-Ring.

Fig. 3.15 compares the memory cost, calculated as the number of unique routing table entries, vs. cluster size (log scale). SF-Ring utilizes 117% higher memory than Chord. This is the cost to achieve faster routing latency. Yet, practically speaking SF-Ring's memory overhead is quite small–most containers/VMs today have many GBs of memory. In comparison, in an SF-Ring with 16K nodes, 99% of nodes store on average 33 routing table entries. Conceptually, with 100 B per entry, this comes out to only 3.3 KB of total memory (SF memory is higher in practice, but still small).

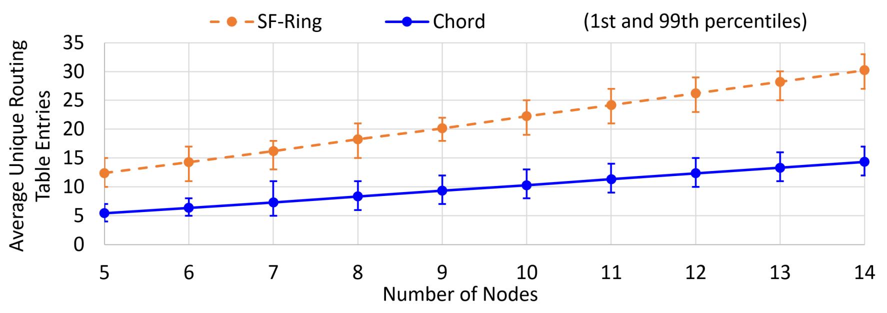


Figure 3.15: **SF-Ring vs. Chord: Memory.** *Unique Routing Table Entry Count as a function of system size (log scale).*

3.6 RELATED WORK

Microservice-like Frameworks: Nirmata [34] is a microservice platform built atop Netflix open-source components [153]: gateway service Zuul [154], registry service Eureka [155], management service Archaius [156], and REST client Ribbon [157]. Unlike Service Fabric, Nirmata does not have consistency and state built into the system. It also has external dependencies. Other microservices platforms include Pivotal Application Service [158] and Spring Cloud [159]. However, none of these support stateful services out of the box. Akka [32] is a platform that embraces actor-based programming to build microservices. These systems do not solve the hard problems related to state or consistency, and do not take as principled an approach to design as SF. AWS Lambda [111] and Azure functions [160] both provide event-driven, serverless computing platforms for running small pieces of short lived code. SF is differentiated because it is the only data-aware orchestration system today for stateful microservices.

SF is the only standalone microservice platform today. The systems just listed usually require an external/remote drive for state. Akka sits atop a JVM. Spanner [161] relies on Colossus, Paxos, and naming. In SF, beyond the OS/machine, there are no external dependencies at the distributed systems layer.

Container Orchestrators: Container services like Kubernetes [46], Azure Container Service [162], etc., allow code to run and be managed easily, but they are typically stateless. SF supports state, which entails further challenges related to failover, consistency, and manageability (our paper addressed these goals). Further, container systems do not provide prescriptive guidance on writing applications; SF provides full lifecycle management.

Strong Consistency in Storage Systems: It is clear that many users and applications prefer strong notions of consistency alongside high performance. Distributed storage systems have come full circle from relational databases to eventually consistent databases [58, 163, 164, 165, 166, 144, 167, 168, 169] to recently, high throughput transactional databases.

After eventually consistent databases, stronger models of consistency emerged (e.g., red-blue [170], causal+ [171], etc.). Many recent systems provide strong consistency and transaction support at high throughput: 1) systems layered atop unreliable replication, e.g., Yesquel [42], Callas [41], Tapir [43]; and 2) systems layered atop strong hardware abstractions, e.g., FaRM [172], RIFL [173], DrTM [174].

Cluster OSes: Prominent among cluster OSes that manage multi-tenancy via containers are: Apache YARN [175] which is used underneath Hadoop [176], Mesos [110] that provides dominant resource fairness, and Kubernetes [46].

Distributed Hash Tables and NoSQL: In the heyday of the P2P systems era, many DHTs were invented including: i) those that used routing tables (Chord [131], Pastry [132], Kademlia [138], Bamboo [177], etc.), and ii) those that used more memory for faster routing [141, 140]. P2P DHTs influenced the design of eventually consistent NoSQL storage systems including Dynamo [144], Riak [167], Cassandra [133], Voldemort [178], MongoDB [164], and many others.

3.7 CONCLUSION

This chapter presents Service Fabric (SF), a distributed platform at Microsoft running on the Azure public cloud. SF enables design and lifecycle management of microservices in the cloud. We have described several key components of SF, showing their modular design, self-* properties, decentralization, scalability, and especially the unique properties of the strong consistency, and stateful support from the *ground up*. Experimental results from real production traces reveal that Service Fabric: i) reconfigures quickly (within seconds); ii) efficiently uses an arbitrator to resolve failure detection conflicts, in spite of high churn; and iii) routes messages efficiently, quickly, and using small amounts of memory.

CHAPTER 4: A NEW FULLY-DISTRIBUTED ARBITRATION-BASED MEMBERSHIP PROTOCOL

Chap. 3 showed how Service Fabric embraces the unique centralized arbitrator-based membership protocol. This protocol claims to provide time bounds on how long membership lists can stay inconsistent—this property is critical in many distributed applications which need to take timely recovery actions. In this work, we: 1) present the first fully decentralized and stabilizing version of membership protocols in this class; 2) formally prove properties and claims about both our decentralized version and the original protocol, and 3) present experimental results from both a simulation and a real cluster implementation.

This chapter is organized as follows. Sec. 4.1 Introduces and motivates the necessity of a decentralized arbitrator based consistent membership protocol, Sec. 4.2 explains the background and system model, Sec. 4.3 proposes the distributed arbitrator based consistent failure detector, Sec. 4.4 performs theoretical analysis for both centralized and decentralized arbitrator based failure detector, Sec. 4.5 presents both simulation and cluster deployment result to measure and compare the performance of the new scheme, Sec. 4.6 analyzes the related works and finally Sec. 4.7 concludes the chapter.

4.1 INTRODUCTION

A group membership protocol, containing a failure detector, is a critical component of large-scale distributed systems and applications. Membership lists are used in datacenters for various purposes including for traffic routing [179, 180, 181], for multicast [182, 183], for replication [133], etc. These are used in distributed databases [184], publish-subscribe systems [185, 186], peer-to-peer systems [187], online gaming [188, 189], etc.

A membership service provides, at each node (process) in the distributed system, a view of a subset of the other nodes (processes) that are alive. We consider only fail-stop failures (when a process crashes it stops further actions; recovering processes rejoin with a new ID). The membership protocol automatically updates the membership list(s) upon node joins, voluntary departures, and especially upon fail-stop failures. The failure detector component of the membership protocol must be efficient in messages and detection time, not miss any failures (called *Completeness*) [190], make few mistakes in detection (called *Accuracy*), and scale with group size [36].

An additional critical requirement that we focus on is *consistency.* Membership protocols in use today sit at two opposite extremes of the consistency spectrum:

- **Weakly-consistent membership protocols** provide an eventual guarantee on membership lists, with some providing a (large) time bound on convergence. Nodes may see inconsistent views of the membership lists for very long periods of time, even under realistic conditions like zero clock drift. Examples include SWIM [36], ring-based heartbeating [191], gossip-style heartbeating [130, 192], etc. These are used in peer-to-peer systems [187] and key-value/NoSQL databases [164, 193], because these applications are themselves weakly-consistent.

- **Strongly-consistent membership protocols** ensure that membership lists are identical at all nodes. If the membership list delivery is totally ordered at alive nodes, this is called virtual synchrony (or view synchrony) [38, 183, 194]. At the same time, there are no timing guarantees on detection–alive nodes may see inconsistent views of the membership lists for indeterminately-long periods of time, even under realistic conditions like zero clock drift and reliable communication (but with unbounded latencies).

It is essential to design membership protocols which provide consistency that is a *timing guarantee* for how long two nodes' membership lists can stay mutually inconsistent w.r.t. a membership change. This is critical in many real-world applications such as banking, stock markets, air traffic control, vehicle routing, etc. [125, 126, 129]. In all these applications, consistent recovery actions need to be taken in a timely manner at multiple nodes, and concurrent incorrect actions by different alive nodes may cause significant application errors.

In Chap. 3, we unveil a new class of membership protocol which claim to provide timing guarantees on how long membership lists stay inconsistent. First proposed as part of the Microsoft's Service Fabric system in Eurosys 2018 [1], these membership protocols have provided sufficient consistency to build applications like Azure SQL DB, Azure Cosmos DB, Microsoft Skype, Azure Event Hub, Microsoft Intune, Azure IoT Suite, Microsoft Cortana etc.

At the same time, the consistency properties of the Service Fabric's failure detector were never formally proven (only hypothesized and sketched). In addition, the Service Fabric failure detector relies on a centralized component called the "arbitrator" to arbitrate conflicting failure detection decisions.

In this paper we make the following contributions:

- We fully decentralize the arbitrator in this new class of failure detectors (Sec. 4.3).

- We propose an efficient node join mechanism to accompany the decentralized arbitrator (Sec. 4.3.3).

- We present theoretical analysis for our new algorithm (Sec. 4.4). Some of this analysis also extends to the original algorithm used in the Service Fabric [1].
- We briefly present both simulation and cluster deployment results to measure and compare the performance of our new algorithms (Sec. 4.5).

4.2 BACKGROUND AND SYSTEM MODEL

We describe the system model and give a brief overview of Service Fabric's [1] *Centralized* Membership Protocol.

System Model: We assume that clock drifts are zero, i.e., clock rates are identical. Clocks can have skew. Messaging is reliable, timely, and ordered, e.g., via TCP. These are reasonable assumptions in datacenters today. We consider fail-stop failures only. There might be heterogeneity in the system, however all nodes are susceptible to failure.

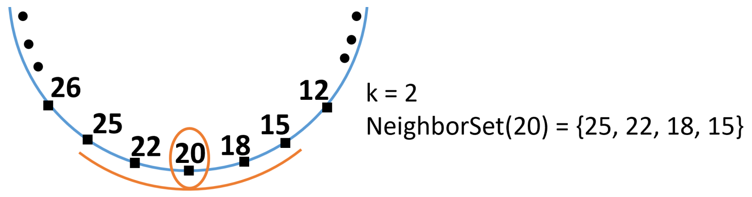


Figure 4.1: **Ring topology.** *Nodes are organized in a virtual ring. Each node maintains neighborhood set consists of k successors and k predecessors.*

Service Fabric's Centralized Membership Protocol, using Arbitrators: Service Fabric's new class of membership protocols [1] separate out failure *detection* (i.e., the act of finding a failure), from failure *decision* wherein detecting nodes start recovery actions for the failure. Failure detection is done via a fully distributed lease-based mechanism (akin to heartbeats). Failure decisions, on the other hand, are executed via a centralized group of nodes called arbitrators, which act as judges to arbitrate inconsistent detections. While centralized detection approaches like ZooKeeper [40] place the traffic of both detection and decision on a central group of nodes, the new approach distributes heavy detection traffic and arbitrators only handle the relatively-rare decision traffic and work.

In Service Fabric [1], nodes are organized in a virtual ring (Fig. 4.1) consisting of 2^m points. A node is mapped to a point on the ring, and so is a key (e.g., via hashing). A key is owned by its closest node, with ties won by the predecessor.

However, the arbitrator is still centralized (at a node or at a group of nodes). This means that if the arbitrator were to fail, or if a majority of an arbitrator group were to fail, then no decisions can be made. In such circumstances, all conflicts will result in mistaken detection+decisions and nodes being forced to leave the system.

Mechanisms Borrowed by Our Decentralized Arbitrator: Our fully-decentralized failure detector borrows two kinds of mechanisms from the original Service Fabric detector: *leasing mechanism* and *failure decision.* Later, Section 4.3 will build atop this to achieve the fully decentralized detector.

While the original paper [1] only cursorily sketched the central failure detector algorithm, here for completeness we present these important components in a formal manner. Table 4.1 presents all symbols used in this paper.

Symbol	**Uses**
T_a	Arbitration timeout interval
T_l	Leasing period
T_{arb}	$(2T_l + T_a)$; Once an arbitrator locally logs a node as failed, after this time units it can safely trim the log entry (Corollary 4.2)
L_{PQ}	Lease from node P to node Q
$L_{PQ(n)}$	n^{th} leasing session (building block of L_{PQ})
$LR_{PQ(n)}$	n^{th} leasing request sent from node P to node Q
$LKRQ_{CP}$	Lock Request from a new candidate node C to the existing node P
$ACK_{PQ(n)}$	Reply of $LR_{PQ(n)}$, sent from node Q to node P
N_P	Neighborhood set of node P
k	Neighbor count in clockwise/counter-clockwise direction in the ring. $\lvert N_P \rvert = 2k$
$\mathcal{A}_{PQ}$	Node P's view of the arbitrator group for the pair (P,Q)
$Arb(P \to Q)$	Arbitration request send from node P, suspecting node Q
$Propose(ver : P*, Q)$	Proposal message, send from node P to upgrade the arbitrator-group $\mathcal{A}_{PQ}$. $P*$ is the proposed arbitrator-group version-number.

Table 4.1: **Symbols used throughout this chapter.**

1. Lease-Based Monitoring: We describe the leasing scheme in detail, and an example is depicted in Fig. 4.2. Consider a lease L_{PQ} between a monitor node P and its monitored node Q. The protocol for maintaining and renewing the lease consists of consecutive, monotonically increasing, non-overlapping leasing sessions ($L_{PQ(*)}$), each lasting for a duration of T_l time units. Initially, at P, the status of both node Q and lease L_{PQ} are *Alive.* At the beginning of the n^{th} leasing session $L_{PQ(n)}$, node P sends a Lease Request $LR_{PQ(n)}$ to node Q and marks the lease session's status as *Pending.* If node P receives the ack $ACK_{PQ(n)}$ in a timely manner, the status of the leasing session is changed to *Established.*

At the end of the ongoing leasing session (T_l time units after $LR_{PQ(n)}$ was sent) node P checks the session's status and initiates the $(n+1)^{th}$ leasing session only if the current status is marked as *Established*. On the other hand, if the status still remains as *Pending* then this is a timeout and $ACK_{PQ(n)}$ was not received–then node P terminates the lease L_{PQ} and marks $STATUS(L_{PQ(n)}) = Timeout$. It also considers node Q as a *Suspected* node.

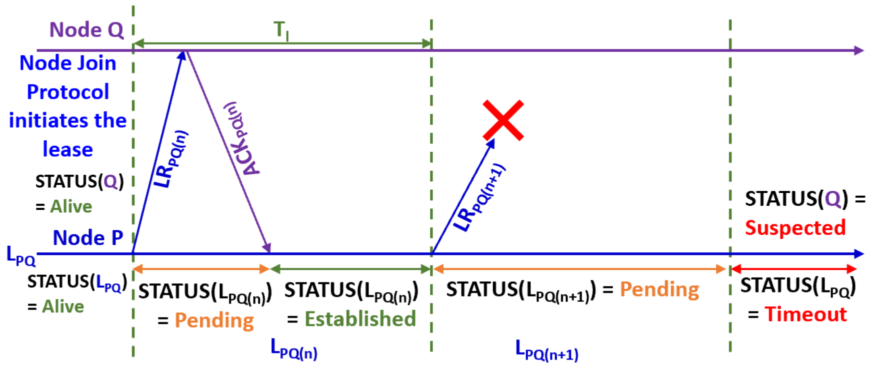


Figure 4.2: **Lease based monitoring.** *Monitor node P maintains lease L_{PQ} with its monitored node Q.*

Monitoring relationships are symmetric, stated formally as:

Rule 4.1 (Symmetric Monitoring (SM))**.** *Neighbor nodes P and Q independently establish leases to each other: L_{PQ} and L_{QP}. Without loss of generality, if $L_{PQ}(n)$ times out, then P ignores all subsequent $LR_{QP(*)}$ lease requests arising from Q.*

2a. Failure Detection: If P's lease request to Q times out, P detects Q as failed and marks node Q as *Suspected*.

However, the lack of further knowledge makes it impossible to draw an accurate conclusion about the suspected node's current status. This is one of the fundamental limitations of failure detection in asynchronous systems. Node Q could have been Suspected due to a plethora of reasons: i) The lease-request from node P was lost; ii) The ack from node Q was lost; iii) Slow or flaky network prevented the lease-request/ack from arriving in time; iv) Node Q actually died; v) Nodes P and Q are partitioned out. As such, it is possible that P and Q (and perhaps other mutual monitors) concurrently and contradictorily *suspect* each other.

In order to resolve such conflicts, when a node P detects Q as Suspected, it does not take failure recovery actions right away (e.g., reorganize the ring). Instead, P moves to a *Failure Decision* mode in order to confirm Q's failure.

2b. Failure Decision: Failure Decisions are done via a centralized group of nodes called as *arbitrator-nodes*. If a failure detection of Q by P is confirmed by the arbitrator-group,

Algorithm 4.1 Monitor-Arbitrator Protocol: Arbitrator Actions

```
 1: Input: Arbitrator receives arb. request $Arb(P \rightarrow Q)$
 2: if arbitrator has been up for less than $T_{arb}$ time units then
 3:     Recently-Failed ← (P ∪ Q ∪ Recently-Failed)
 4:     return reject
 5: else if node P ∈ Recently-Failed list then
 6:     return reject
 7: else if node Q ∈ Recently-Failed list then
 8:     return accept
 9: else                                        ▷ First Come First Serve approach
10:     Recently-Failed ← (Q ∪ Recently-Failed)
11:     return accept
12: end if
```

then this implies P has permission to recover from Q's failure: P can remove Q from its view of the ring, claim some of Q's keys, etc.

Rule 4.2 (Arbitration Request)**.** *If two nodes (P,Q) maintain Symmetric Monitoring and (without loss of generality), L_{PQ} times out, then Node P immediately sends an arbitration request $Arb(P \rightarrow Q)$ to the arbitrator group. If it receives no response from the arbitrators within T_a time units (T_a is a fixed parameter system-wide), P voluntarily leaves the system. Otherwise it obeys the arbitrator group's decision. Arbitrators follow Algo. 4.1.*

3. The Arbitrator-Group: The arbitrator-group acts as a referee and provides failure decisions. Each arbitrator node maintains small state but acts independently–there is zero sharing across arbitrator-nodes. Each arbitrator-node maintains a list, called *Recently-Failed*, which contains node IDs that it has recently declared as *dead*. The list contains only failures confirmed within a fixed recent time duration (Corollary 4.2).

Algo. 4.1 describes the actions taken by an arbitrator-node on receiving a *suspect* request from P about Q. If this arbitrator is new (e.g., a new joiner replacing a failed arbitrator), then it rejects all requests and marks both suspecter and suspected nodes as failed, and responds with a *reject* to P (Line 4). This boostrapping rule is needed to ensure zero-sharing across arbitrators, and avoid bad decisions by new arbitrators.

Otherwise, if the arbitrator is well-established, it checks if P was recently declared dead–then its request is rejected and P is again asked to leave the system (Line 6). If P is considered alive but Q was recently marked as dead, then P's request is accepted (Line 8). Finally (Line 9) if both P and Q were alive, then P's request is accepted–this means that if P and Q simultaneously suspected each other, the winner is the one among them whose request arrives *first* at the arbitrator-node.

4. Obeying the Arbitrator-Nodes' Decisions: Once node P detects a failure of Q, it sends arbitration requests individually to each arbitrator-node and awaits responses for T_a time units. A request to an arbitrator-node results is one of three outcomes: *accept*, *reject* or *timeout* (Algo. 4.1). After receiving all responses or after T_a time units–whichever occurs earlier–P marks Q as failed if and only if a majority of arbitrator-nodes (quorum in arbitrator-group) responded with an *accept* vote. Notice that arbitrator-nodes respond independent of each other and do not need to coordinate among each other. Consequently, after a wait of $(2T_l + T_a)$ time units since P sent arbitration requests (this wait time is calculated in Corollary 4.3), P can safely assume that the suspected node Q has left the system. Thereafter P can take actions to recover from Q's failure.

Otherwise, if a majority of P's arbitration requests result in a response that is in (*reject OR timeout*), then P voluntarily leaves the system. In this way, P sacrifices itself for the consistency of the system's failure decisions. We call such departures as *forced departures*, and later our experiments will measure them. While undesirable, forced departures are a crucial mechanism needed to maintain membership-consistency across the system.

4.3 DISTRIBUTED ARBITRATOR-BASED CONSISTENT FAILURE DETECTOR

The downsides of the consistent failure detector described in Chap. 3 arise from the use of the central arbitrator-group. During periods marked by a large number of node failures, arbitrator-nodes may become congested by a high volume of requests, causing timeouts at requesting nodes. If a majority of arbitrator-nodes are slow or faulty, all *suspecting* nodes will be forced to leave the group, causing massive forced departures of healthy nodes. Additionally, failed arbitrators need to be replaced manually in the original protocol (this is also true in Zookeeper via "rolling-restart")—our protocol allows automated arbitrator replacement, as they are chosen in a self-stabilizing way from alive nodes.

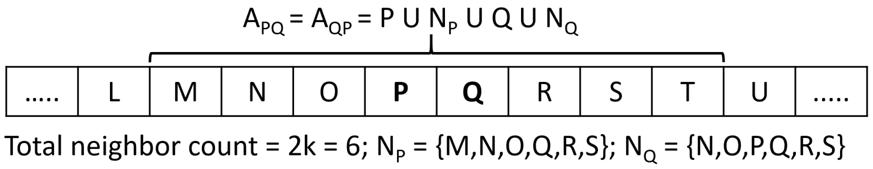
To address these issues, we decentralize the arbitrator-group itself. The key idea is to eschew having a fixed set of arbitrator-nodes. Instead, we allow each pair of monitoring nodes (P, Q) to select its own arbitrator-group. The challenge is to ensure that this is done in a way that retains correctness of the membership. This arbitration selection mechanism is our first contribution. Our second contribution is handling changes in the arbitrator-group itself–because of failures or departures, the arbitrator-group's membership cannot stay static, and needs to be changed over time. These two contributions are then combined with the leasing protocol (Fig. 4.2) and the Monitor-Arbitrator protocol from Algo. 4.1 to produce our overall membership protocol.

4.3.1 Decentralized Arbitrator Selection

Eliminating the centralized arbitrators would be straightforward had they been fully stateless. We observe first that the only state an arbitrator-node maintains is the Recently-Failed list (Algo. 4.1— The Arbitrator-Group).

We note that at a minimum, to maintain consistency, this Recently-Failed list needs to be checked only upon mutually-conflicting failures. That is, only under circumstances when P suspects Q, and Q suspects P, and thus both P and Q send arbitration requests. For all other requests (e.g., R suspects P after P has suspected Q), the Recently-Failed list minimizes the number of forced detections, but is not required for consistency.

Using this observation we eliminate the arbitrator-nodes as follows: we replace them with a subgroup of nodes chosen for pairwise arbitration. That is, we use a separate arbitrator-group for every pair of monitoring (neighbor) nodes P and Q. Later, our experiments will show that this does not cause an increase in forced departures.



$A_{PQ} = A_{QP} = P \cup N_P \cup Q \cup N_Q$
..... L M N O P Q R S T U .....
Total neighbor count = 2k = 6; N_P = {M,N,O,Q,R,S}; N_Q = {N,O,P,Q,R,S}

Figure 4.3: **Pairwise Arbitrator Group Formation Strategy.**

Fig. 4.3 depicts our pairwise arbitrator-group formation strategy via an example. The pairwise arbitrator set $\mathcal{A}_{PQ}$ maintained at P for the pair (P, Q) consists of P, Q, as well as their respective sets of neighbors (in the ring) N_P, N_Q. Formally:

$$\mathcal{A}_{PQ} = \mathcal{A}_{QP} = P \cup N_P \cup Q \cup N_Q \tag{4.1}$$

Both P and Q maintain this mutual arbitrator list, and if their mutual leases expire, they refer to $\mathcal{A}_{PQ}$ and $\mathcal{A}_{QP}$ respectively for the Failure Decision.

4.3.2 Dynamic Arbitrator-Groups

Maintaining Consistency Between $\mathcal{A}_{PQ}$ and $\mathcal{A}_{QP}$: Node P and Q must maintain a consistent view of $\mathcal{A}_{PQ}$ and $\mathcal{A}_{QP}$. Otherwise, in case of failure there is the risk that P and Q consult with two different set of arbitrator-nodes–these partially-overlapping/non-overlapping arbitrator-groups might independently *accept* both P and Q's arbitration requests, causing both P and Q to stay in the system but believing each other is failed.

At any time P's neighborhood N_P might change to $N_{P(ver:P1)}$ (because of node join/leave/failure, etc.). Therefore, to reflect the new neighborhood, $\mathcal{A}_{PQ}$ needs to be upgraded to a new version, denoted as $\mathcal{A}_{PQ(ver:P1)}$. However, node Q might not immediately be aware of node P's neighborhood change. Therefore, an immediate upgrade of $\mathcal{A}_{PQ}$ might cause inconsistency between $\mathcal{A}_{PQ}$ and $\mathcal{A}_{QP}$.

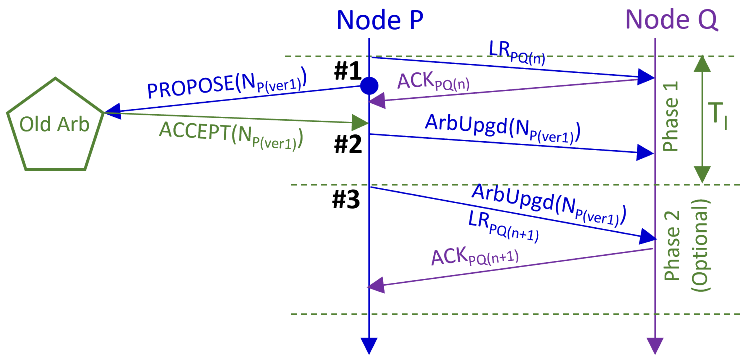


Figure 4.4: **Safe and Consistent Arbitrator Group Upgrade Strategy.** *For simplicity all lease requests initiated from node Q ($LR_{QP(*)}$) and their corresponding ACKs are omitted.*

Safe and Consistent Arbitrator-Group Upgrade Protocol: We use a novel approach that seamlessly upgrades the pair-wise arbitrator-groups and prevents inconsistency between them. Whenever P needs to upgrade the current arbitrator-group for Q, it follows a *two-phase* protocol that leverages the current arbitrator-group $\mathcal{A}_{PQ}$ in order to perform a safe and consistent upgrade. This is depicted via an example in Fig. 4.4, and we describe the phases below.

Phase 1: Before upgrading the arbitrator-group from $\mathcal{A}_{PQ}$ to $\mathcal{A}_{PQ(ver:P1)}$, P sends an *arbitrator-group upgrade* proposal $Propose(ver : P1, Q)$ to its current arbitrator-group $\mathcal{A}_{PQ}$ (Fig. 4.4:#1). The proposal contains the new *arbitrator-group version-number* proposed by P (in this case P1). These arbitrator-nodes record the proposal, and respond with acks. This results in one of three outcomes. These outcomes and P's subsequent actions are as follows:

1. **The majority times out:** Node P voluntarily leaves.
2. **The majority rejects:** Node P aborts the current arbitration change attempt and retries later.
3. **The majority accepts:** Node P upgrades the arbitrator-group and sends an explicit *arbitrator-group upgrade* confirmation message $ArbUpgrd(N_{P(ver:P1)})$ to Q (Fig. 4.4:#2).

This message contains the updated neighborhood $N_{P(ver:P1)}$ and the current *arbitrator-group version-number* $P1$. Node Q uses $N_{P(ver:P1)}$ to upgrade $\mathcal{A}_{QP}$ and attaches the new *arbitrator-group version-number* along with its future arbitration requests.

Phase 2: This phase piggybacks the *arbitrator-group upgrade* confirmation message with the next scheduled lease-request/ack and thus implicitly notifies Q about the arbitration-group change (Fig. 4.4:#3). This redundant phase ensures that even if the previous explicit *arbitrator-group upgrade* confirmation message was lost, the next lease-request/ack will convey it. In other words, *if P and Q continue to maintain successful leases into the future, then the arbitrator-group upgrade information will be propagated to Q within* $2T_l$ *time units after P upgrades its arbitrator-group (Theorem 4.1).*

Modified Arbitration Policy: Arbitrator-nodes perform normal request processing based on Algo. 4.1. In addition, arbitrator-group members need to deal with the arbitrator-group upgrade proposals as follows: when an arbitrator-node receives the *arbitrator-group upgrade* proposal from node P (i.e., $Propose(ver : P1, Q)$), it checks if it has seen any arbitration-group upgrade proposal from Q (i.e., $Propose(ver : Q*, P)$), or an arbitration request $Arb(Q \rightarrow P)$ from node Q since the last T_{arb} time units (Corollary 4.2). If any of these is true, the arbitrator-node *rejects* the request. Otherwise it locally stores the *current version (P1)* for $\mathcal{A}_{PQ}$ and replies an *accept* to P.

Handling the stale arbitrator-group in node Q: However, in a rare scenario Q might detect P as failed just before receiving the *arbitrator-group upgrade* message. In that case Q sends the *arbitration request* to the *old arbitrator-group* along with the *old arbitrator-group version-number*. Because at least majority of the *old arbitrator-group* has already accepted node P's *arbitrator-group change* request, they *detect* and *reject* Q's *stale* arbitration request. Therefore, in the case of such inconsistency, Q gracefully leaves the system.

Recently-Failed List: In the decentralized version, a node C that belongs to the arbitrator set for a pair of nodes (P, Q) still keeps a finite Recently-Failed list. This list consists of at most two entries: whether P was previously marked as dead (and when), and whether Q was previously marked as dead (and when). Unlike the Recently-Failed list in the centralized protocol (Algo. 4.1) which could be arbitrarily long, our Recently-Failed lists are limited in size to 2 entries. Further, since each node has $2k$ arbitrator sets and each such set has at-worst $4k$ members, by symmetry each node participates in at-most $2k \times 4k = 8k^2$ arbitrator sets.

4.3.3 Node Join Protocol under Decentralized Arbitrators

Because we have replaced the central arbitrator with a decentralized arbitrator, we need to design a new node join protocol to keep the required state consistent. This is is a four-phase protocol.

Phase 1 (Neighbor Discovery): The candidate joining node (say node C) sends a neighbor discovery request to that node Q which currently owns the key C. (The consistent ring routing algorithm described in [1] can be used for this).

If Q is currently serving another *node join* request, it *rejects* node C's request (C can retry). Otherwise Q replies with *accept*–this reply also piggybacks Q's current neighbor set N_Q.

In the case of *time out* or *rejection*, node C backs-off and retries later. Otherwise it moves forward by using the N_Q set to calculate its own potential neighborhood set N_C. This is feasible because N_C is always a subset of $(Q \cup N_Q)$ (Fig. 4.5).

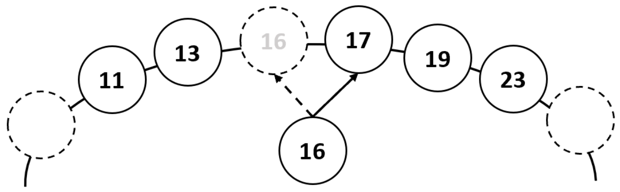


Figure 4.5: **Calculating the potential neighborhood.** *The ring-member node 17 currently holds the key 16. Neighbor count per direction,* $k = 2$*. The candidate node (node 16) gets the key holder's current neighborhood set* $N_{17} = \{11, 13, 19, 23\}$*. Node 16's potential neighbourhood set would be* $N_{16} = \{11, 13, 17, 19\}$*.*

Phase 2 (Lock Request): For correctness, our protocol needs to ensure that prospective neighbors (monitors) of the joining node are not involved in processing another node join. Hence joining node C next sends a lock request to all of its $2k$ potential neighbors, N_C (Fig. 4.6). A node receiving this lock request rejects the request only if it is actively processing another node join. Otherwise the lock is granted for the next $(3 \cdot T_l)$ time units (time to finish Phases 2-4). If C can acquire all the locks in time, it moves to Phase 3. Otherwise, it releases all the established locks, backs off and retries from Phase 1.

Phase 3 (Lease Invitation): Node C establishes independent leases L_{C*} and sends the first lease requests $LR_{C*(1)}$ to all members of N_C. It piggybacks its current potential neighborhood set N_C (this is used by C's neighbor P to create the mutual arbitrator list A_{PC} based on Equation 4.1).

Once a neighbor (node P) receives the first lease request $LR_{CP(1)}$ it prepares the acknowledgement $ACK_{CP(1)}$ and initiates the symmetrically-opposite lease L_{PC} for node C.

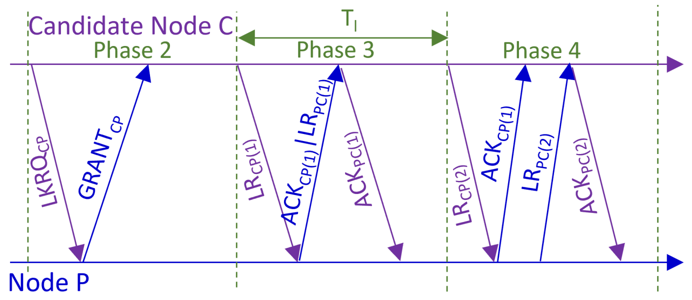


Figure 4.6: **Node join protocol (initiated from the candidate node C).** *For simplicity, Phase 1 is omitted and only one neighbor is shown.*

It piggybacks both the symmetrically-opposite first lease request $LR_{PC(1)}$ and its current neighbor set N_P with the ack message.

Node P also records the arbitrator set for its relationship with C: $\mathcal{A}_{PC} = P \cup N_P \cup C \cup N_C$, and marks the arbitrator-set as dormant. Dormant means that if the lease L_{PC} times out, instead of involving the arbitrator group $\mathcal{A}_{PC}$, node P will clean-up the lease and neither send further lease requests to C nor response to any of C's future lease-requests.

Upon receiving the incoming ack $ACK_{CP(1)}$, node C gets the new lease request from P, $LR_{PC(1)}$ and P's neighborhood set N_P. At this point node C continues the Symmetric Monitoring according to Rule 4.1. C sends back $ACK_{PC(1)}$ to P. Besides, C also prepares the arbitrator set $\mathcal{A}_{CP} = C \cup \{N_C\} \cup P \cup \{N_P\}$, and marks this as dormant.

If at least a single lease request with any of its neighbors times out, node C discards all the established leases L_{C*}, backs off and retries again starting from Phase 1. If a neighbor (say node P) has already established a lease L_{PC}, that lease will also automatically time out at P (since node C will not reply back eventually). At node P, the arbitrator-group $\mathcal{A}_{PC}$ has still been marked as dormant. Therefore, instead of involving the arbitrator-group, Node P merely deletes the timed-out lease L_{PC}, i.e., it does not attempt to renew this lease in the future. Otherwise, if C and each of its neighbors P successfully establish symmetric leases in this 1^{st} session $L_{C*(1)}$, then C moves to Phase 4.

Phase 4 (Wrap-up): Node C sends the 2^{nd} lease request $LR_{C*(2)}$ to all of its neighbors. Once $LR_{CP(2)}$ is received, the neighbor P can safely assume that C has successfully established the Symmetric Monitoring (Rule 4.1) with the members of N_C, hence safely marks the arbitrator group $\mathcal{A}_{PC}$ as active. Therefore, from now on whenever the lease L_{PC} times out, P will consult with the $\mathcal{A}_{PC}$ for failure decision.

Once node C establishes the 2^{nd} leasing session with all of its neighbors and receives all the corresponding acks ($ACK_{C*(2)}$), it considers itself as an active ring member and marks the corresponding arbitrator-groups ($\mathcal{A}_{C*}$) as active. At this point, the failure detection and decision algorithms (Algo. 4.1) can be used between P and C.

4.4 THEORETICAL ANALYSIS

We analyze correctness of: A) our decentralized (and thus the original centralized) arbitrator approach, B) our node join protocol, and C) overheads.

4.4.1 Decentralized Arbitrators

We first analyze our fully-decentralized failure detector from Sec. 4.3. Unless noted otherwise, our theorems below also apply to the original Service Fabric failure detector (centralized) from Sec. 4.2 and [1]. This is a side-contribution of our paper, because the original Service Fabric paper [1] (and its based-on white papers/patents [134, 136]) did not come associated with rigorous analysis—we thus prove their previously-held hypotheses. Our first theorem is specific to our new decentralized arbitrator.

Theorem 4.1 (Maximum Inconsistency Interval Between Arbitrator-groups $\mathcal{A}_{PQ}$ and $\mathcal{A}_{QP}$)**.** *Consider neighbors P and Q. Say P successfully upgrades the arbitrator-group $\mathcal{A}_{PQ}$ at absolute time T. If the lease L_{PQ} stays established for long enough into the future, then: Q will reflect the upgrade by time $T + 2T_l$.*

Proof. To upgrade the arbitrator-group $\mathcal{A}_{PQ}$, node P follows a two-step process (Fig. 4.4). Let upgrade time T occur in the m^{th} leasing session $L_{PQ(m)}$. P sends an explicit confirmation $ArbUpgrd(N_{P(ver*)})$ to node Q. However, this explicit message might get lost in the network. Yet, because $ArbUpgrd(N_{P(ver*)})$ will be piggybacked with the next subsequent request, if P and Q's mutual leases stay correct for long enough into the future, this next lease request for $L_{PQ(m+1)}$ will succeed and will communicate $ArbUpgrd(N_{P(ver*)})$ to Q. Hence Q will upgrade its view of the arbitrator-group $\mathcal{A}_{QP}$. Therefore, Q knows about the upgrade by time $T + 2T_l$.

The following results hold for both the new decentralized version and the centralized version of arbitrators.

Next we analyze failure detection times. For a given node Q, define its *failure detection time* (T_{FD}) as the time between Q's failure occurring and *all* of Q's monitors suspecting Q.

Theorem 4.2 (Failure Detection Time)**.** *When a node Q fails and at least one of its monitors is alive, then: T_{FD} is bounded from both below and above, as: $T_l \leq T_{FD} < 2T_l$.*

Proof. A pair of ring neighbors, node Q and node P maintains the Symmetric Monitoring (Rule 4.1). Let Q be the failing node and P be the alive monitor. Further, let Q fail during P's n^{th} lease period for Q, and the failure occurs β time units after the start of that lease period.

1. Node P initiates the n^{th} leasing session $L_{PQ(n)}$ and sends the leasing request $LR_{PQ(n)}$.

2. Node Q receives the lease request and replies back with $ACK_{PQ(n)}$. After sending this response, Q crashes.

3. Node P receives the $ACK_{PQ(n)}$ in time and marks the n^{th} leasing session as established. Once the current session completes (which takes T_l time units since the lease request), P initiates the $(n+1)^{th}$ leasing session.

4. However, as node Q has been crashed, node P's $(n+1)^{th}$ session with Q will timeout after a further T_l time units.

Starting from the point when node Q crashes, the time left at the n^{th} leasing session $L_{PQ(n)}$ is $(T_l - \beta)$. Node P detects the failure at the end of the $(n+1)^{th}$ leasing session. Therefore, the failure detection time $T_{FD} = (T_l - \beta) + T_l$. But since $\beta \in (0, T_l]$, hence we have $T_l \leq T_{FD} < 2T_l$.

Next we analyze how long it takes for two neighbors to mutually suspect each other.

Theorem 4.3 (Symmetric Lease Timeout)**.** *The Symmetric Monitoring (Rule 4.1) between a pair of nodes P and Q consists of two independent leases: L_{PQ} and L_{QP} respectively. Without loss of generality, assume L_{PQ} times out first. From that point onwards, if it takes T_{LT} time unit to time out L_{QP}, then: $T_l \leq T_{LT} < 2T_l$.*

Proof. The run consists of the following steps:

1. Node P detects the time out of L_{PQ} at absolute time T and starts rejecting any future lease requests from Q (Rule 4.1).

2. At Q, let $L_{QP(m)}$ be the ongoing leasing session that starts just before T and receives the corresponding $ACK_{QP(m)}$.

3. Starting from T, the time left to finish the ongoing m^{th} leasing session is T_b $(< T_l)$.

4. As the m^{th} session was a success, Q initiates the next session and sends the lease request $LR_{QP(m+1)}$ to P.

5. However, since node P is already rejecting all of node Q's future leasing requests, this $(m+1)^{th}$ leasing session at node Q (for P) will timeout after a further T_l time units and node Q will finally mark the lease L_{QP} as timed out.

Therefore, starting from the moment when P first detects the lease timeout of L_{PQ}, Q will also detect the timeout of L_{QP} no more than $(T_b + T_l)$ time units. As $T_b \in [0, T_l)$, hence the lease timeout interval T_{LT} is bounded by $T_l \leq T_{LT} < 2T_l$.

Corollary 4.1 (Mutual Arbitration Requests)**.** *Given two neighbors P and Q, if P suspects Q and sends an arbitration request, then: within another $T_{arbReqInt}$ time units, Q will also send an arbitration request suspecting P. $T_l \leq T_{arbReqInt} < 2T_l$.*

As soon as the lease L_{PQ} times out, P immediately sends an arbitration request suspecting Q (Rule 4.2). However, according to Theorem 4.3, L_{QP} also times out within T_{LT} time units and Q immediately sends an arbitration request suspecting P. The interval $(T_{arbReqInt})$ between the two arbitration requests coincides with T_{LT}. Hence $T_l \leq T_{arbReqInt} < 2T_l$.

We now analyze how long a suspected node can survive.

Theorem 4.4 (Maximum Lifespan of a Suspected Node)**.** *Let Q be suspected by a monitor P, and thus Q is forced to leave the system. Starting from the moment when node Q is first suspected by node P, the suspected node can stay in the system no later than $T_{maxLifespan} < (2T_l + T_a)$ time units.*

Proof. As soon as the lease L_{PQ} times out, P suspects Q and makes an arbitration request. Due to the First Come First Serve (FCFS) policy (Algo. 4.1), the arbitrator accepts that request. Q *may* also send an arbitration request no later than $2T_l$ (Corollary 4.1) time units after, and awaits the response. The arbitrator will reject the request due to FCFS. Two cases arise:

1. If node Q successfully receives the rejection it will immediately leave the system.

2. If the arbitration request times out (after T_a time units), node Q will leave the system (Rule 4.2).

Starting from the failure detection at node P, the suspected node (node Q) can stay in the system for at most $(T_{arbReqInt} + T_a)$ time units. Therefore, the maximum possible time that node Q can stay in the system is bounded from above as: $T_{maxLifespan} < (2T_l + T_a)$.

Corollary 4.2 (Arbitrator can safely remove any entry older than $T_{arb} = (2T_l + T_a)$ from the Recently-Failed list)**.** *A pair of ring neighbors, node P and Q is maintaining the Symmetric*

Monitoring (Rule 4.1). Without loss of generality, node P suspects Q first and sends an arbitration request.

Once node P suspects Q, starting from that time, node Q can stay in the system for at most $T_{maxLifespan} = 2T_l + T_a$ time units (Theorem 4.4). Therefore, it is guaranteed that if the arbitrator receives P's request at absolute time T, at $T + T_{maxLifespan}$ time units node Q must leave the system. Hence, the arbitrator can safely remove entries from the Recently-Failed list that are older than $T_{arb} = 2T_l + T_a$.

Corollary 4.3 (Safe time to Start Failure Recovery). *Given two neighbors P and Q, if P suspects Q (failure detection) and the arbitrator agrees with it (failure decision), then: node P should wait for $(2T_l + T_a)$ time-span to safely declare node Q as dead, and start any recovery actions.*

As Theorem 4.4 points, if node Q is suspected by node P at time T, counting from that time, node Q can stay in the system for at most $T_{maxLifespan} = (2T_l + T_a)$ time units.

Therefore, node P should wait for at least that time units before finally considering node Q as dead.

Finally, we can state that our protocol (as well as the centralized protocol of Sec. 4.2 and [1]) satisfy the time-based consistency that we outlined in Sec. 4.1:

Theorem 4.5 (Time-based Consistency). *The decentralized (and centralized) arbitrator-based failure detection protocol maintains time-based consistency, as defined in Sec. 4.1. Concretely, after a node Q fails, within another $(T_a + 4T_l)$ time units two conditions are true: i) Q leaves the system, and ii) all of Q's monitors know about Q's failure.*

Proof. When Q crashes, an alive monitor node P will detect it within T_{FD} time units where $T_l \leq T_{FD} < 2T_l$ (Theorem 4.2). However, node P has to wait for another $(2T_l + T_a)$ time units to safely mark node Q as dead (Theorem 4.4). Therefore, $(T_a + 4T_l)$ after node Q's failure, Q has left the system and all its monitors can start recovery actions.

4.4.2 Node Join Protocol Correctness

Next we analyze the Node Join protocol of Sec. 4.3.3.

Theorem 4.6 (Correctness of Single Node Join under Failures). *The Node Join Protocol for a given joining node maintains consistent membership lists in spite of failures.*

Proof. If any of the monitors of a joining node C fails at any point before the specific instant of time that C has successfully established the second leasing session with *all* of its monitors, then C will time out in one of the above phases, and gracefully leave the system. Subsequently, monitors will also release their locks after the $3T_l$ timespan. If on the other hand, C fails after it has finished Phase 4, our normal failure detection mechanism (Sec. 4.3 combined with Sec. 4.2) will detect C's failure.

Theorem 4.7 (Correctness under Multiple Node Joins)**.** *The Node Join Protocol Maintains consistent membership lists in spite of multiple nodes joining simultaneously.*

Proof. Because a joining node C first acquires locks on its potential monitors, no other joining nodes share the same monitors (neighbors) as C. Thus, our protocol allows simultaneous joining nodes if and only if their neighbor sets (monitor sets) are disjoint. Together with Theorem 4.6, this maintains consistency.

Theorem 4.8 (Time-bounded Node Join)**.** *When there are no failures or dropped messages, then: a new joining node, after Phase 1, finishes joining in another* $3 \cdot T_l$ *time units.*

Proof. Each of the remaining Phases 2-4 take T_l time units. Hence node joining finishes in $3 \cdot T_l$ time units.

4.4.3 Overhead and Downsides

Leasing Message Overhead: The leasing messages comprise of both lease-requests from any node P to Q ($LR_{PQ(*)}$) and the corresponding acks ($ACK_{PQ(*)}$). The number of leasing messages per second, per node, is calculated as:

$$2 \times (2k) \times \frac{1}{T_l} \tag{4.2}$$

This equation shows that the leasing message overhead per node scales with system size. Additionally, T_l represents a tradeoff between overhead and detection time: selecting a smaller T_l ensures faster failure detection, but also means frequent lease renewals and more stabilization messages.

Partitioning Behavior: Finally, in the interest of completeness of analysis, we observe that arbitrator-based approaches cannot avoid the well-known partitioning problem inherent to group membership systems. Both the centralized and decentralized arbitration schemes (Sec. 4.2, 4.3) are susceptible to collapse when the network is partitioned. In the centralized version, in the worst case, if none of the partitions has a quorum number of arbitrators, nodes detect failures of their neighbors in the other partitions, but are unable to obtain a quorum number of arbitrator responses, and therefore leave the system. These forced departures cascade, and eventually everyone leaves the system. The decentralized version also suffers from the same forced departure + cascading failure behavior.

4.5 EVALUATION

We implemented the decentralized arbitrator-based failure detector in both: 1) a simulator, and 2) a real Java implementation, which we run on the Emulab cluster [195].

4.5.1 Impact of Arbitrator Failure (Simulation)

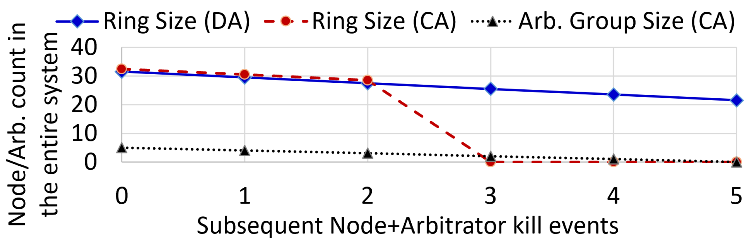


Figure 4.7: **Impact of Arbitrator Failure.** *X axis represents the arbitrator + regular node kill event. Y axis shows the total number of arbitrator/regular node in the system.*

Fig. 4.7 shows, via simulation, the impact of an arbitrator crash for two different systems: *CA* which uses the Centralized arbitrator from Service Fabric [1], and *DA* which is our scheme. The ring consists of 32 nodes, and the CA variant uses a set of 5 arbitrators. The DA uses 6 monitors per node.

The X-axis shows *crash events*–at each event, 2 nodes are killed: in CA, a member-node and an arbitrator-node; in DA, two random nodes. The Y-axis shows the active node count.

We observe that the original CA scheme is tolerant only up to 2 arbitrator failures. With 3 or more arbitrator failures, there are insufficient number of arbitrators left to make failure decisions, and as a result all nodes voluntarily leave the system, and the ring size (system size) drops to 0 quickly. In comparison, our DA scheme maintains a stable system size, which drops only by the number of crashed nodes.

In summary, we conclude that the distributed arbitrator based scheme is more resilient to arbitrator failure.

4.5.2 Failure Detection Accuracy (Emulab)

Next we measure whether failing nodes affect detection accuracy of otherwise healthy nodes. The main concern here is detection *cascades*, wherein failing or leaving nodes cause other nodes in the neighborhood to also voluntarily leave, due to the timeouts involved in leasing and/or arbitration. Furthermore, because our algorithm is decentralized, it is plausible to assume that such cascading failures may be exacerbated compared to the centralized version. This experiment implicitly measures the effect of this behavior.

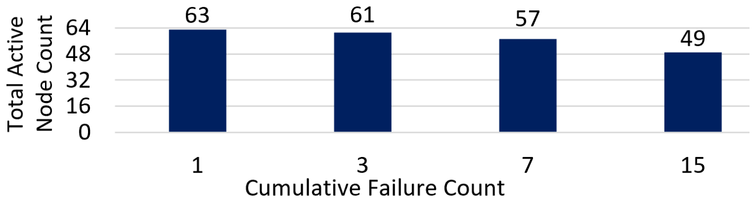


Figure 4.8: **Failure Detection Accuracy.** *X axis shows the number of nodes failed simultaneously. Y axis shows the total active node count in the system.*

We deploy a 64-member group in an EmuLab cluster with 5 d710 nodes [196]. Each node consists of one 2.4 GHz 64-bit Quad Core Xeon processor, 8MB L3 cache, 12 GB DDR2 Ram. The 64 nodes are in a ring, with 3 successors and 3 predecessors (thus a total of 6 monitors each).

In Fig. 4.8 we progressively kill a group of randomly-selected nodes. The x-axis shows these simultaneous failure events. Thus this plot is a timeline plot (without showing the time). The y-axis shows the number of alive nodes left after the detections+decisions have stabilized, after each failure event.

Fig. 4.8 shows that, at the very first event when only one node failed, the total alive node count becomes 63. This indicates that only the failed node leaves, and there are no additional or cascading departures. As the reader can observe, for each subsequent failure event, the marginal reduction in system size is identical to the number of failing nodes.

In summary, this experiment validates the desirable behavior that failures are detected accurately, and do not force further departures of nodes due to cascades.

4.5.3 Arbitrator Message Overhead (Simulation)

One of the goals of the decentralized scheme was to reduce load on the arbitrators. This is important as it keeps arbitrators fast and responding timely, and reduces risk of arbitration requestors timing out.

We run a simulation with a ring consists of total 1000 nodes. First we run the centralized scheme with $\alpha = 9$ arbitrator nodes. Neighbor count in each direction of the ring (clockwise and anticlockwise) is $k = 3$. $f = 100$ randomly selected nodes were failed sequentially. The interval between two consecutive failures was sufficient to bring the stability back into the system.

Each failure causes $2k$ arbitration requests at each of the α arbitrator nodes, creating an overhead of $(2 \cdot k \cdot f \cdot \alpha)$. This implies a total of 600 messages per arbitrator node.

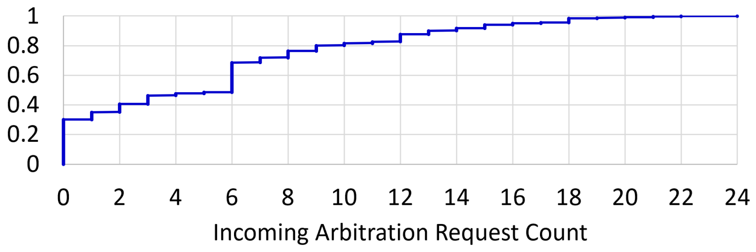


Figure 4.9: **Incoming Arbitration Requests: CDF.** *Our decentralized scheme distributes arbitration requests among* 1000 *nodes. The maximum number of arbitration requests received by a node is* 24*.*

In comparison, Fig. 4.9 shows that our decentralized scheme imposes a much lower overhead, and creates no bottlenecks. In this scheme, the arbitrator group size varies from $(2 + 2k)$ to $(1 + 3k)$ (depending on how far the monitoring nodes are from each other in the ring). The figure shows that 80% of the nodes receive fewer than 10 arbitration requests. The average is 5.4 arbitration requests per node, and the worst case is 24.

We conclude that compared to the centralized scheme, our decentralized approach reduces worst-case arbitration message overhead by at least two orders of magnitude.

4.5.4 Node Join Latency (Simulation)

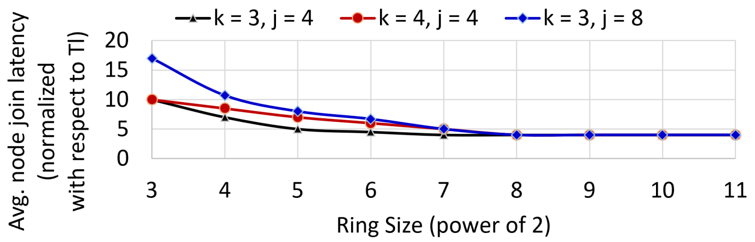


Figure 4.10: **Node Join Latency.** k = neighbor count in each direction. j = total number of nodes tried to join simultaneously. The node joining time increases if any of the k or j increases. As the ring grows, the node joining time decreases.

We use our simulation to measure the speed of our Node Join algorithm from Sec. 4.3.3. Fig. 4.10 shows the node join time (normalized with respect to T_l) for different ring sizes (x-axis is logarithmic in ring size) and different values of k, and j the total number of nodes attempting to join simultaneously. Nodes attempt to join at random positions in the ring. We observe that:

1. Increasing ring size reduces latency because it spreads out the joining load more. The minimum node join time $4T_l$ arises from the 4-phase node join procedure.
2. Latency increases with the number of monitors $(2k)$ (fixing ring size and j) because joining nodes need to coordinate with more ring neighbors (Phase 2 of the node join protocol described in Sec. 4.3.3).
3. Increasing the number of simultaneous joiners increases latency because of the locking involved in the joining process, which causes contention and some waiting.

In summary, we conclude that node join latencies scale very well and decrease with ring size, and are able to accommodate simultaneously joining nodes.

4.5.5 Failure Detection Time (Emulab)

Fig. 4.11 depicts the failure detection time (normalized with respect to T_l). We use the same EmuLab deployment described above, and vary T_l from 32 ms to 1.024 s.

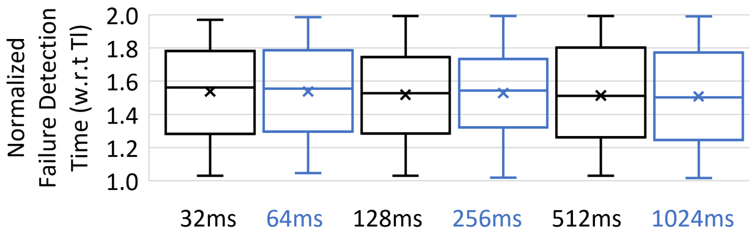


Figure 4.11: **Failure Detection Time vs. Leasing Interval T_l.** *Candlestick plots show the* 1^{st}, 2^{nd} *and* 3^{rd} *quartiles and the average (X's on plot). Y-axis normalized w.r.t.* T_l.

The failure detection time T_{FD} is bounded according to our proved result in Theorem 4.2. That is, $T_l \leq T_{FD} < 2T_l$. On average, it takes around $(1.5 \cdot T_l)$ time to detect a failure.

4.6 RELATED WORK

The key component of a membership protocol is the failure detector. The formal characterization of the properties of failure detectors was first offered by Chandra and Toueg [190] where they also showed that it is impossible for a failure detector algorithm to deterministically achieve both completeness and accuracy over an asynchronous unreliable network.

Chandra and Toueg's impossibility result [190] says that one cannot design a failure detector in an asynchronous network that both detects all failures (completeness) and makes no mistakes (accuracy). Subsequent failure detectors [197, 198, 199, 200, 201, 202, 130] choose to satisfy completeness because of the need for correct failure recovery, and they attempt to optimize accuracy (reduce false positives). Reliable failure detectors include Falcon [203] with sub-second detection times, but the paper states that this protocol does not scale.

Virtual Synchrony and similar approaches [38, 183, 194] offer totally-ordered and consistent membership view. However, members of this protocol family suffer from scalability limitations.

Among weakly consistent protocol are gossip-style heartbeating [130, 204] and SWIM [36]. SWIM uses random pinging for failure detection, and piggybacks failure notifications atop such pings and acks. Such probabilistic membership approaches are used in Cassandra [193], Akka [32], ScyllaDB [205], Serf [37], Redis Cluster [152], Orleans [206], Uber's Ringpop [207], Netflix's Dynomite [208], Amazon Dynamo [144], etc.

A widely used approach to achieve the consistent membership is to store the membership list in an auxiliary service such as Chubby [47], Etcd [209], ZooKeeper [40] etc. Offloading is attractive but increases the dependence on a small set of nodes. Under congestion or failure of a quorum of these special nodes, the membership service is completely unavailable. In comparison, in our system, even under an arbitrary number of failures, membership lists remain available.

Rapid [210] is an interesting protocol that can detect partitions (i.e., cuts). We believe Rapid can be orthogonally combined with our decentralized arbitrator-based failure detector.

4.7 CONCLUSION

In this chapter we present the design of a new fully-decentralized membership protocol that maintains strong time-based consistency of membership lists. Where past work relied on a central group of arbitrators to referee decisions and conflicts on failure detections, our approach fully decentralizes this arbitrator set. Via formal analysis, we proved important correctness and consistency properties of our scheme, and some of these results prove previously-held hypotheses about the centralized arbitrator scheme. Via both simulation and cluster deployment, we showed that our decentralized membership protocol: 1) minimizes forced departures of healthy nodes, 2) avoids failure cascades, 3) significantly reduces arbitration message overhead vs. centralized scheme, 4) incurs latency that decreases with system size, and 5) detects failures quickly.

CHAPTER 5: LESSONS LEARNT

This chapter shares the lesson learned throughout the projects.

5.1 SERVICE FABRIC AND THE DISTRIBUTED ARBITRATION SCHEME

We outline here lessons learnt by the Service Fabric (SF) team over the years. Our role in this has been to extract some of these lessons via discussions with the SF-team members.

5.1.1 Decisions SF-team Had to Revisit

Distributed systems are more than nodes and network: Applications are processes running on nodes and can fail in ways that do not always lead to total failure of the node. A common occurrence is something SF-team have come to term as "Grey Node Failure". In these cases, the OS continued to work without the presence of a fully functional OS disk. The absence of a functional disk renders the application unhealthy. However, the SF leasing mechanism continues to run at high priority without any page faults since it does not depend on disk. SF-team have since added an optional *disk heartbeat* in the leasing to work around this type of issue.

Application/Platform responsibilities need to be well isolated: Early on, SF's customer base was large internal teams who had systems expertise in building internet-scale services (e.g., Azure Cosmos DB). SF-team's initial application interaction model was designed for such teams who had close collaboration with the team. They always conformed to the requirements of the SF APIs. One simple example was when the platform needed to close a replica, SF would call `close` on the application code and wait for the replica close to complete. This allowed the replica to perform proper clean up. The same model does not necessarily work with a larger set of application developers who have bugs or take a long time to cleanly shutdown. SF-team have since made changes to SF where the `close` of the replica getting stuck does not cause availability loss and the system moves past it after a configurable close timeout.

Capacity planning is the application's responsibility (but developers need help): At the platform level SF-team cannot completely foresee application capacity requirements and have been frequently asked to investigate increased latency issues that arise when the application is driving the machine beyond its capacity like IO or memory. SF-team found this out the hard way when some of the core Microsoft Azure services kept having issues due to under-provisioning. Migrating to larger hardware was the only solution. SF-team have since instituted that all applications explicitly specify their capacity requirements like CPU, Memory, IO, etc. The system now ensures application containers run on sufficiently-provisioned machines and enforces developer-specified limits on these containers, so that developers gain the insight and accountability for the capacity limits they need to specify.

Different Subsystems Require Different Investments: The Federation Subsystem was the most intellectually challenging and took up the majority of time during the overall 15+ years of development (in these early days, the SF-team was small in size). It was very important to get this substrate right, as its correctness and consistency was critical in order to be able to build SF's remaining sub-systems above it. In comparison, the failover capabilities in the Reliability subsystem required far more human-hours as they had a larger number of moving parts, and were amenable to many more optimizations. The team was also larger by this later stage of the SF project.

5.1.2 Decisions That Stood The Test of Time

Monitored upgrades and clean rollbacks of platform upgrades allow faster releases and give customers confidence: Customers often avoid upgrading (to the latest SF release) because of fears around consequences of bad upgrades. SF handles this via an automatic roll-back mechanism if an upgrade starts causing health issues in the cluster. Rollbacks might be needed because of a bug in the application where upgrade was not being properly handled, or a bug in the platform, or a completely different environmental reason. SF-team have worked extremely hard to ensure auto-roll-back works smoothly, and it has given the customers immense confidence in upgrading quickly, as they know that bad upgrades will be rolled back automatically. This also allows SF-team to quickly iterate and ship faster. Cases where rollbacks have gotten stuck are relatively rare.

Changes to the system should be staged: SF upgrades (for code or configuration) have always been staged. Some customers tried to work around SF staged deployments by performing a silent configuration deployment via external configuration stores. While this gave the perception of faster upgrades, almost all these cases eventually caused outages due to fat fingering. Today most SF customers understand the value of orchestrated upgrades and adhere to them.

Health reporting leads to better lifecycle management and easier debugging: Applications can tap into SF's Health Store to ensure that application+platform upgrades are proceeding without availability loss. The Health Store also allows for easier debugging of the cluster due to metrics it collects. This service also increases customer confidence.

"Invisible" external dependencies need care: External dependencies like DNS, Kerberos [211], Certificate Revocation Lists, need to be clearly identified. Application developers need to be cognizant of how failures of these dependencies affect the application. SF-team had an incident where a customer was hosting a large SF cluster using machines with Fully-Qualified Domain Names (FQDNs), which meant that the SF cluster needed a DNS server for FQDN resolution. The customer did the right thing in ensuring that the DNS service is highly available by replicating it. However, DNS replication is only eventually consistent. This meant that occasionally the same FQDN was resolving to two different machines (IPs), leading to availability outages in the system. To resolve this issue, the customer switched back to directly using IP addresses instead of FQDNs.

5.2 LESSONS LEARNT: SAFEHOME

Any device with a basic set of APIs should be compatible with SafeHome: Most of the smart-devices expose only a limited and basic set of APIs (e.g., turn on, turn off, set brightness to x%, etc.). We might modify the device firmware and implement/expose more APIs to facilitate seamless integration with SafeHome. For example, implementing a two-phase locking [90] requires modifications to the smart-device. However, there are a myriad number of smart-device vendors; each comes with their proprietary designs/protocols. In SafeHome, any special API dependency requires modifying the firmware of each of these devices– it is not a scalable and smart choice. Systems for smart homes need to be designed in a way so that the basic set of APIs is sufficient to support any smart-device.

Smart homes need a local fallback technique and edge is the answer: Unlike geo-distributed cloud-centric systems (e.g., Service Fabric [1]), smart homes are significantly smaller, but a direct human-facing system. Here, any delay/disruption might directly and immediately impact the end-user. Existing smart home orchestrators, e.g., Google Home, Alexa, Siri, solely rely on the remote cloud that often causes service disruption [212, 26, 27, 28]. Therefore, to ensure quick response and uninterrupted service, systems for smart-homes should be deployed at the nearest proximity. We deploy SafeHome at the local edge to avoid strict cloud dependencies.

Traditional transactional management systems are not sufficient: Routines used in smart homes are akin to transactions in Database Management Systems (DBMS). One of the main goals of SafeHome is ensuring serialized routine execution. While looking for a solution, we explored the serialization techniques used in DBMS systems. We found a few factors that prevented us from directly borrowing the existing serialization techniques used in DBMS:

i) Direct human-facing nature: DBMS typically creates a snapshot of the database, uses it to simulate concurrent transactions, and validates the transactions at the end. It is a widely used approach that enhances parallelism. However, SafeHome's direct human-facing nature prevents us from adopting such techniques. For example, simulating the long-running routine {`Water Sprinkler: turn ON for 15 minutes`} for 15 minutes before its actual execution is not a reasonable choice. As a way-around, SafeHome adopts the Pessimistic Concurrency Control [213], where the device locks are acquired beforehand.

ii) Immediately visible outcomes: in a smart home, each intermediate operation is immediately visible to the user. Any unnecessary device-changes/flickering might make the user uncomfortable. Therefore, reducing the unnecessary device state-changes is one of the main goals of SafeHome. Traditional DBMS based approaches do not suffer such state-changes/flickering since they simulate concurrent transactions on snapshot first and commit only the final results.

A pure system-centric solution alone is not enough to deal with a direct human-facing system: The current version of SafeHome is build based on pure system design and intentionally avoids the human in the loop. Such avoidance of humans in a direct-human facing system often fails to meet human-expectations: for example, SafeHome handles concurrency by re-scheduling routines, which might change the routines' final execution order from their initial submission order. Think of three routines R1, R2, and R3 submitted concurrently, and SafeHome serializes them as R2, R3, R1. User-A might be OK with this ordering, whereas User-B might prefer R1 before R3. Even worse, the same user's preference might change over time (e.g., in morning R1 before R3, in evening R3 before R2). A pure system-centric solution alone is not sufficient to resolve such problems, and it is necessary to incorporate human in the loop [51].

Existing micro-service orchestration schemes (e.g., Service Fabric) are not reusable in smart homes:

Both cloud-centric large scale deployments and relatively smaller smart-home deployments are distributed systems. It might sound intuitive to re-use some of the well-established modules (e.g., the orchestrator). However, a few fundamental differences between these two systems limit us from re-using components.

i) It is not feasible to replicate the physical smart-device: Cloud-centric systems are designed to deal with intangible objects, where replication is used for fault-tolerance. Failure of one object can be fixed immediately by recovering it from one of its replicas. Cyber-physical systems, e.g., smart homes lack such luxury. It might be possible to replicate the log of the last executed command, but it is not feasible to replicate the smart-device along with its last executed state. Therefore, once a device fails, there is no immediate recovery. To deal with such scenarios, SafeHome classifies commands as "must" and "best-effort". SafeHome rolls back the entire routine only if any of its "must" command fails.

ii) Different failure-recovery characteristics: After a smart device recovers from a failure, its current desired state may vary. For example, after a power failure, when the power comes back: 1) a refrigerator should be in `ON` state, and 2) a hair-dryer should be in `OFF` state. Whereas, in case of a recovered microservice, it always retains its last state. Therefore, SafeHome orchestrators must have a *context-aware* failure recovery scheme.

iii) Rolling back is not always an option in smart home: Traditional DBMS systems rollback by setting the relevant objects' value to their last committed state. This scheme might be applicable for a few smart-devices: e.g., rollback TV to its previous committed state `(ON/OFF)`. However, such rolling back is not feasible for a large number of smart devices. For example, it is not feasible to undo the impact of a water-sprinkler. In such cases, smart home orchestrators should roll back the device state (e.g., turn `OFF` Water Sprinkler) and notify the user about the possible consequences (e.g., the garden is wet).

Goto-Safe States and Dilemma: For a device that fails while a routine is executing a command on it, the edge may not know what state the device restarts in. We address this by having devices restarted in a pre-determined "goto-state". Goto-states are convenient but could cause "Goto dilemmas". If a garage door opener's goto-state is OPEN, burglars may be let in; if it is CLOSED, it might close on a car underneath it. Both are safety violations. How to handle these is an open question. Note that such dilemmas also occur in other cyber-physical environments like self-driving cars [214].

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

In this thesis, *we presented new internal orchestrators for maintaining consistency in different distributed systems*: i) distributed systems running on local edge devices e.g., Smart Home and ii) large scale distributed systems running on geo-distributed data-centers e.g. Microsoft Service Fabric.

A safe and reliable smart home requires a carefully designed routine management system, concurrency control schemes and inbuilt safety mechanism. We presented SafeHome (under submission [20]), a internal orchestrator designed to maintain congruent smart home state. This is the first implementation of relaxed visibility models for smart homes running concurrent routines. It is also the first system that reasons about failures alongside concurrent routines. SafeHome incorporates a new *Lineage Table* data-structure that maximizes concurrency by applying a unique and safe *lock-leasing* technique among the conflicting routines.

Our work on Microsoft Service Fabric (SF) [1] revealed how SF, a distributed microservice framework that is designed to run on geo-distributed data centers, ensures consistency across the entire stack. To maintain consistency, it uses an internal orchestrator that relies on a consistent failure detector. SF is widely used to run different Microsoft Azure services. SF's lower-most layer contains a novel consistent failure detector that provides a consistent view of the failed node. The consistency is systematically propagated across the whole system in a ground-up manner where each layer forms its own notion of consistency by leveraging its lower-most layer's consistency guarantees. The experimental results from both simulations and real-production traces revealed that SF: i) reconfigures quickly after a failure, ii) efficiently uses the arbitrator group to resolve failure detection conflicts and iii) routes messages efficiently, quickly and using small amounts of memory.

In our next work [21], we decentralized Service Fabric's internal orchestrator that maintains strong time-based consistency of membership lists. Via formal analysis, we proved important correctness and consistency properties of our scheme, and some of these results proved previously-held hypotheses about the centralized arbitrator scheme. Via both simulation and cluster deployment, we showed that our decentralized membership protocol: 1) minimizes forced departures of healthy nodes, 2) avoids failure cascades, 3) significantly reduces arbitration message overhead, 4) incurs latency that decreases with system size, and 5) detects failures quickly.

6.2 FUTURE WORK

We suggest several directions for future work related to this thesis.

6.2.1 Exploring the Optimistic Concurrency Control (OCC) for Smart Home Scenario

Currently, SafeHome uses a pessimistic concurrency control to handle concurrent routines. The lock-based Pessimistic concurrency control is inherently slower than its Optimistic counterpart. However, routines' *immediately visible* execution strategy forces us to adhere to Pessimistic concurrency control.

An interesting future direction could be to search for an abstraction that directly applies OCC over the smart home environment. Such an abstraction will bridge the gap between current smart homes and traditional database systems. It would also open the opportunity of directly incorporating different concurrency and isolation schemes used in today's database systems and apply those into the smart home arena.

6.2.2 Decentralizing the SafeHome Orchestrator

The current version of SafeHome orchestrator is designed to run on a single edge device (e.g., Raspberry pi), which might become a single point of failure. Therefore, it is worth to explore ways to decentralize it.

The current Centralized orchestrator (akin to the centralized arbitrator used in Service Fabric (Chap. 3)) efficiently manages the distributed smart-devices. Distributing the centralized core will make it fault-tolerant; however, it will introduce additional challenges. For example, to ensure consistent execution of a routine, multiple orchestrator replicas need to communicate and sync among themselves. This will certainly cost additional latency.

6.2.3 Adding user-friendly features

Our proposed SafeHome is a vast project, and this thesis covers only the tip of the iceberg. The main goal of this thesis is to ensure a flexible yet serializable scheduler for handling concurrent routines. Various other user-friendly features can be added to SafeHome:

Interrupt, pauses, priorities: A complete OS for smart homes should support these features. A higher priority routine might interrupt a lower priority routine and immediately start the high-priority task. This is similar to process preemption [215]. However, in such cases, the human-facing nature of SafeHome might cause additional challenges. Our current design does not support rescheduling routines. EV only tries to schedule a new routine before existing scheduled-routines, given that there are sufficient valid slots in the lineage table.

Device permission and Security: In a smart home, users should not have equal access to all smart-devices. For example, only parents may have access to the garage door, while children should not. Therefore, each device needs to maintain a per-user access rule. This is not a new field [60]. However, incorporating such concepts into the SafeHome framework might bring up new challenges. Current SafeHome does not deal with security. However, one future research goal might be to infuse existing security solutions [216] in SafeHome.

Enhanced Safety Feature: Current SafeHome offers basic safety features. However, there are scopes to enhance it. While safety properties can be specified in a myriad number of ways [65, 66], we find that a large majority of clauses can be specified using the following grammar.

```
A:- if A then A else A
A:- DeviceID.StateID ==<>!= foo
A:- ALL(A), ANY(A), !A, ATLEAST(k)(A), ATMOST(k)(A), A AND A, A OR A
```

A future SafeHome might support such grammars where a user may define a safety property either because it is critical to human safety, e.g.,
`if (stove==ON) then (fire-alarm==HEALTHY)`,
or for user convenience, e.g.,
`if (GarageDoor.State==OPEN) then (GarageLight.State==ON).`

6.2.4 Exploring the "Human" Facing Side of SafeHome:

Unlike conventional distributed systems, smart home is a human-facing system where the Human-Computer Interaction (HCI) plays a vital role. Our thesis explores only the "system" side of it. For example, in the case of a routine level conflict, our system silently aborts the appropriate routine. This approach might seem logical with respect to "machines" (e.g., roomba [88], other smart-devices, etc.). However, such action might not always be the best fit for human-users. An interesting future direction might be to explore the "Human" facing side of such systems [51].

Designing and defining routines is another topic that needs minute attention. As smart homes grow in complexity, designing routines to control the home also becomes an increasingly complicated and error-prone activity. Humans are prone to error; therefore, such human-facing systems should have the capability to detect errors early.

A wrong *sequence* of commands/routines, *poorly designed* routines, or *ill-maintained* routines (especially when adding or removing devices, which is typical in a smart home) can put the smart home in a precarious state.

Current safety and routine checking mechanisms presented in this thesis are proof of concepts. It is worth investing time in designing a declarative language that allows writing the safety properties conveniently. Besides, it needs a mechanism to check for safety property conflicts. Detecting such conflicts is not new: for example, for network verification, several systems such as NICE [217], Anteater [218], VeriFlow [219] and others [220, 221, 222] detect violating rules and configurations. Similarly, in the IoT space systems such as APEX[66] and [63, 64, 65] enable user-specified conditions and dependencies and verify them. Despite the similarities in these approaches and SafeHome's safety properties, SafeHome is solving a more generic problem (including failures, conflicts, and safety violations) where these approaches can be applied to SafeHome's safety checker engine.

REFERENCES

[1] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi, M. Mohsin, R. Kong, A. Ahuja, O. Platon, A. Wun, M. Snider, C. Daniel, D. Mastrian, Y. Li, A. Rao, V. Kidambi, R. Wang, A. Ram, S. Shivaprakash, R. Nair, A. Warwick, B. S. Narasimman, M. Lin, J. Chen, A. B. Mhatre, P. Subbarayalu, M. Coskun, and I. Gupta, "Service fabric: A distributed platform for building microservices in the cloud," in Proceedings of the Thirteenth EuroSys Conference, ser. EuroSys '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190546 pp. 33:1–33:15.

[2] "What are microservices?" https://microservices.io/, last accessed February 2018.

[3] "Azure SQL DB," https://azure.microsoft.com/en-us/services/sql-database/, last accessed October 2018.

[4] "Azure Cosmos DB," https://azure.microsoft.com/en-us/services/cosmos-db/, last accessed October 2018.

[5] "Skype for Business," https://www.skype.com/en/business/skype-for-business/, last accessed October 2018.

[6] "Smart home market worth $151.4 billion by 2024," https://www.marketsandmarkets.com/PressReleases/global-smart-homes-market.asp, last accessed December 2019.

[7] "Smart home," https://www.statista.com/outlook/279/109/smart-home/united-states, last accessed December 2019.

[8] "Mapping the Smart-Home Market," https://www.bcg.com/publications/2018/mapping-smart-home-market.aspx, last accessed December 2019.

[9] "EE: Average UK Smart Home will have 50 connected devices by 2023," https://www.totaltele.com/500103/EE-Average-UK-Smart-Home-will-have-50-connected-devices-by-2023, last accessed December 2019.

[10] "WeMo App," https://play.google.com/store/apps/details?id=com.belkin.wemoandroid&hl=en_US, last accessed February 2018.

[11] "IFTTT," https://ifttt.com/, last accessed December 2019.

[12] "Google Home," https://store.google.com/us/product/google_home, last accessed December 2019.

[13] "Amazon Alexa App," https://apps.apple.com/us/app/amazon-alexa/id944011620, last accessed February 2018.

[14] "Philips Hue App," https://play.google.com/store/apps/details?id=com.philips.lighting.hue2&hl=en, last accessed February 2018.

[15] "Nest App," https://play.google.com/store/apps/details?id=com.nest.android&hl=en, last accessed February 2018.

[16] "Apple HomeKit App," https://www.apple.com/uk/ios/home/, last accessed February 2018.

[17] "Samsung SmartThings App," last accessed February 2018.

[18] "Split-brain (computing)," https://en.m.wikipedia.org/wiki/Split-brain_(computing), last accessed December 2019.

[19] "Microsoft Service Fabric - Reliable Collection," https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-reliable-collections, last accessed February 2018.

[20] S. B. Ahsan, R. Yang, S. A. Noghabi, and I. Gupta, "Home, safehome: Ensuring a safe and reliable home using the edge," in 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19). Renton, WA: USENIX Association, July 2019. [Online]. Available: https://www.usenix.org/conference/hotedge19/presentation/ahsan

[21] S. B. Ahsan and I. Gupta, "A new fully-distributed arbitration-based membership protocol," in 2020 IEEE Conference on Computer Communications, INFOCOM 2020. IEEE, 2020.

[22] "Stop shouting at your smart home so much and set up multi-step routines," https://www.popsci.com/smart-home-routines-apple-google-amazon, last accessed December 2019.

[23] "Amazon Alexa + SmartThings routines and scenes," https://support.smartthings.com/hc/en-us/articles/210204906-Amazon-Alexa-SmartThings-Routines-and-Scenes, last accessed December 2019.

[24] A. S., T. L., M. S. A., and C. A. S., "Transactuations: Where transactions meet the physical world," in 2019 USENIX Annual Technical Conference (USENIX ATC 19). Renton, WA: USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/sengupta pp. 91–106.

[25] "What is imprint$^{\text{TM}}$ link technology?" https://homesupport.irobot.com/app/answers/detail/a_id/21088/~/what-is-imprint%E2%84%A2-link-technology%3F, last accessed December 2019.

[26] "Internet of Shit," https://twitter.com/internetofshit, last accessed December 2019.

[27] "The top 5 problems with smart home tech and how to troubleshoot them," https://www.nachi.org/problems-smart-home-tech.htm, last accessed December 2019.

[28] "Google's smart home ecosystem is a complete mess," https://www.cnet.com/news/googles-smart-home-ecosystem-is-a-complete-mess/, last accessed December 2019.

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

[30] S. B. Ahsan, R. Yang, S. A. Noghabi, and I. Gupta, "Home, SafeHome: Ensuring a safe and reliable home using the edge," in 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19). Renton, WA: USENIX Association, July 2019. [Online]. Available: https://www.usenix.org/conference/hotedge19/presentation/ahsan

[31] "TP Link KASA HS100, HS220, KL130," https://www.kasasmart.com/, last accessed December 2019.

[32] "Akka," http://akka.io/, last accessed October 2018.

[33] "Bluemix," https://www.ibm.com/cloud-computing/bluemix, last accessed February 2018.

[34] "Nirmata," http://www.nirmata.com/, last accessed February 2018.

[35] R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, ser. Middleware '98. London, UK, UK: Springer-Verlag, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=1659232.1659238 pp. 55–70.

[36] A. Das, I. Gupta, and A. Motivala, "SWIM: scalable weakly-consistent infection-style process group membership protocol," in Proceedings International Conference on Dependable Systems and Networks, ser. DSN '02, 2002, pp. 303–312.

[37] "Serf: A decentralized solution for service discovery and orchestration," https://www.serf.io/, last accessed October 2018.

[38] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in Proceedings of the 11th ACM Symposium on Operating Systems Principles, ser. SOSP '87. New York, NY, USA: ACM, 1987. [Online]. Available: http://doi.acm.org/10.1145/41457.37515 pp. 123–138.

[39] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," ACM Transactions on Computer Systems (TOCS), vol. 17, no. 2, pp. 41–88, May 1999. [Online]. Available: http://doi.acm.org/10.1145/312203.312207

[40] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, ser. USENIX-ATC'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855851 pp. 11–11.

[41] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang, "High-performance ACID via modular concurrency control," in Proceedings of the 25th Symposium on Operating Systems Principles, ser. SOSP '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815430 pp. 279–294.

[42] M. K. Aguilera, J. B. Leners, and M. Walfish, "Yesquel: Scalable SQL storage for web applications," in Proceedings of the 25th Symposium on Operating Systems Principles, ser. SOSP '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815413 pp. 245–262.

[43] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," in Proceedings of the 25th Symposium on Operating Systems Principles, ser. SOSP '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815404 pp. 263–278.

[44] "Hbase," https://hbase.apache.org/, last accessed October 2018.

[45] "Kafka," https://kafka.apache.org/, last accessed October 2018.

[46] "Kubernetes," https://kubernetes.io/, last accessed October 2018.

[47] M. Burrows, "The Chubby Lock Service for Loosely-coupled Distributed Systems," in Proceedings of the 7th Symposium on Operating Systems Design and Implementation, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298487 pp. 335–350.

[48] J. Koon, "Smart sensor applications in manufacturing (Enterprise IoT Insights)," https://enterpriseiotinsights.com/20180827/channels/fundamentals/iotsensors-smart-sensor-applications-manufacturing, last accessed April 2020.

[49] "Future of smart hospitals (ASME)," https://aabme.asme.org/posts/future-of-smart-hospitals, last accessed April 2020.

[50] M. Campbell-Kelly, "Historical reflections: The rise, fall, and resurrection of software as a service," Commun. ACM, vol. 52, no. 5, pp. 28–30, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1506409.1506419

[51] S. Davidoff, M. K. Lee, C. Yiu, J. Zimmerman, and A. K. Dey, "Principles of smart home control," in Proceedings of the 8th International Conference on Ubiquitous Computing, ser. UbiComp'06. Berlin, Heidelberg: Springer-Verlag, 2006. [Online]. Available: https://doi.org/10.1007/11853565_2 p. 19–34.

[52] "TP Link KASA HS105, HS110, HS200," https://www.kasasmart.com/, last accessed December 2019.

[53] "Fenestra: Make your windows smart," http://www.smartfenestra.com/home, last accessed April 2020.

[54] "Velux: Smart home, smart skylights," https://whyskylights.com/, last accessed April 2020.

[55] "Smartcan: Take control of your chores," https://www.rezzicompany.com/, last accessed April 2020.

[56] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," ACM Transactions on Computer Systems, vol. 10, no. 1, pp. 3–25, Feb. 1992. [Online]. Available: http://doi.acm.org/10.1145/146941.146942

[57] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in Proceedings of the 15th ACM Symposium on Operating Systems Principles, ser. SOSP '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/224056.224070 pp. 172–182.

[58] V. Vogels, "Eventually consistent," Communications of the ACM, vol. 52, no. 1, pp. 40–44, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1435417.1435432

[59] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," IEEE Computer, vol. 29, no. 12, pp. 66–76, Dec. 1996. [Online]. Available: http://dx.doi.org/10.1109/2.546611

[60] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl, "An operating system for the home," in Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228332 pp. 25–25.

[61] D. Retkowitz and S. Kulle, "Dependency management in smart homes," in IFIP International Conference on Distributed Applications and Interoperable Systems. Springer, 2009, pp. 143–156.

[62] M. S. Ardekani, R. P. Singh, N. Agrawal, D. B. Terry, and R. O. Suminto, "Rivulet: A fault-tolerant platform for Smart-home Applications," in Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, ser. Middleware '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3135974.3135988 pp. 41–54.

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

[64] S. Munir and J. A. Stankovic, "Depsys: Dependency aware integration of cyber-physical systems for Smart Homes," in ICCPS '14: ACM/IEEE 5th International Conference on Cyber-Physical Systems (with CPS Week 2014), ser. ICCPS '14. Washington, DC, USA: IEEE Computer Society, 2014. [Online]. Available: http://dx.doi.org/10.1109/ICCPS.2014.6843717 pp. 127–138.

[65] C. M. Liang, B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu, "SIFT: Building an internet of safe things," in Proceedings of the 14th International Conference on Information Processing in Sensor Networks, ser. IPSN '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2737095.2737115 pp. 298–309.

[66] Q. Zhou and F. Ye, "Apex: Automatic precondition execution with isolation and atomicity in internet-of-things," in Proceedings of the International Conference on Internet of Things Design and Implementation, ser. IoTDI '19. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3302505.3310066 pp. 25–36.

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

[68] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," ACM Trans. Database Syst., vol. 30, no. 1, p. 122–173, Mar. 2005. [Online]. Available: https://doi.org/10.1145/1061318.1061322

[69] "TP Link HS105," https://www.tp-link.com/us/download/HS105.html, last accessed December 2019.

[70] R. Ladin, B. Liskov, and L. Shrira, "A technique for constructing highly available services," Algorithmica, vol. 3, no. 3, pp. 393–420, Apr. 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=42468.42469

[71] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing, ser. PODC '88. New York, NY, USA: ACM, 1988. [Online]. Available: http://doi.acm.org/10.1145/62546.62549 pp. 8–17.

[72] "Is Xfinity having an outage right now?" https://outage.report/us/xfinity, last accessed December 2019.

[73] "Comcast outage," https://webdownstatus.com/outages/
comcast, last accessed December 2019.

[74] "Amazon Alexa outage," https://downdetector.com/status/amazon-alexa/news/235561-problems-at-alexa, last accessed December 2019.

[75] "Google Home outage," https://downdetector.com/status/google-home, last accessed December 2019.

[76] "SmartThings outage," https://downdetector.com/status/smartthings/news/224625-problems-at-smartthings, last accessed December 2019.

[77] "TP Link device driver," https://github.com/intrbiz/hs110, last accessed December 2019.

[78] "Wemo," https://www.wemo.com/, last accessed December 2019.

[79] "TP-link KASA android app," https://www.tp-link.com/us/kasa-smart/kasa.html, last accessed December 2019.

[80] "SmartThings Smart Apps," https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps, last accessed December 2019.

[81] "IoTBench test-suite," https://github.com/IoTBench/IoTBench-test-suite/tree/master/openHAB, last accessed December 2019.

[82] "Raspberry Pi 3 Model B+," https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/, last accessed December 2019.

[83] "Amazon Alexa," https://developer.amazon.com/alexa, last accessed December 2019.

[84] "Scheduled routines not reliable," https://support.google.com/assistant/thread/366154?hl=en, last accessed December 2019.

[85] "Routines not working," https://support.google.com/assistant/thread/3444653?hl=en, last accessed December 2019.

[86] "Alexa routines show promise and limitations," https://www.timeatlas.com/create-alexa-routines/, last accessed December 2019.

[87] "Imprint^{TM} link technology: Getting started," https://homesupport.irobot.com/app/answers/detail/a_id/21090/ /imprint%E2%84%A2-link-technology%3A-getting-started, last accessed December 2019.

[88] "Roomba," https://www.irobot.com/roomba, last accessed December 2019.

[89] "Braava," https://www.irobot.com/braava, last accessed December 2019.

[90] "Two-phase locking," https://en.wikipedia.org/wiki/Two-phase_locking, last accessed April 2020.

[91] M. Ma, S. M. Preum, and J. A. Stankovic, "CityGuard: A watchdog for safety-aware conflict detection in Smart Cities," in Proceedings of the 2nd International Conference on Internet-of-Things Design and Implementation, ser. IoTDI '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3054977.3054989 pp. 259–270.

[92] C. S., R. P. S., A. P., A. K., and R. M., "Beam: Ending monolithic applications for connected devices," in 2016 USENIX Annual Technical Conference (USENIX ATC 16). Denver, CO: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/shen pp. 143–157.

[93] "OpenHAB," https://www.openhab.org/, last accessed December 2019.

[94] "Zapier," https://zapier.com/, last accessed December 2019.

[95] "Microsoft Flow," https://flow.microsoft.com, last accessed December 2019.

[96] "Automate.io," https://automate.io/, last accessed December 2019.

[97] "Workflow," https://workflow.is/, last accessed December 2019.

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

[99] L. Canon, L. Marchal, B. Simon, and F. Vivien, "Online scheduling of task graphs on heterogeneous platforms," IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 3, pp. 721–732, 2020.

[100] D. Barthou and E. Jeannot, "Spaghetti: Scheduling/placement approach for task-graphs on heterogeneous architecture," in Euro-Par 2014 Parallel Processing, F. Silva, I. Dutra, and V. Santos Costa, Eds. Cham: Springer International Publishing, 2014, pp. 174–185.

[101] G. Lee, "Resource allocation and scheduling in heterogeneous cloud environments," Ph.D. dissertation, University of California at Berkeley, USA, 2012.

[102] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram, "Generic algorithms for scheduling applications on hybrid multi-core machines," in Euro-Par 2017: Parallel Processing, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Cham: Springer International Publishing, 2017, pp. 220–231.

[103] A. Arunarani, D. Manjula, and V. Sugumaran, "Task scheduling techniques in cloud computing: A literature survey," Future Generation Computer Systems, vol. 91, 09 2018.

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

[105] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," SIAM J. Comput., vol. 18, no. 2, p. 244–257, Apr. 1989. [Online]. Available: https://doi.org/10.1137/0218016

[106] "Azure Event Hubs," https://azure.microsoft.com/en-us/services/event-hubs/, last accessed October 2018.

[107] "Microsoft Intune," https://www.microsoft.com/en-us/cloud-platform/microsoft-intune, last accessed October 2018.

[108] "Azure IoT," https://azure.microsoft.com/en-us/suites/iot-suite/, last accessed October 2018.

[109] "Microsoft cortana," https://www.microsoft.com/en-us/mobile/experiences/cortana/, last accessed October 2018.

[110] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, ser. NSDI '11. Berkeley, CA, USA: USENIX Association, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488 pp. 295–308.

[111] "AWS Lambda," https://aws.amazon.com/lambda/, last accessed February 2018.

[112] A. Wang and S. Tonse, "Announcing Ribbon: Tying the Netflix mid-tier services together," http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html, last accessed February 2018.

[113] S. Tonse, "Scalable microservices at Netflix. challenges and tools of the trade," https://www.infoq.com/presentations/netflix-ipc, last accessed February 2018.

[114] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures using microservices: An experience report," Computing Research Repository, vol. abs/1507.08217, 2015. [Online]. Available: http://arxiv.org/abs/1507.08217

[115] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," IEEE Software, vol. 33, no. 3, pp. 42–52, 2016.

[116] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," Computing Research Repository, vol. abs/1606.04036, 2016. [Online]. Available: http://arxiv.org/abs/1606.04036

[117] C. Esposito, A. Castiglione, and K. K. R. Choo, "Challenges in delivering software in the cloud as microservices," IEEE Cloud Computing, vol. 3, no. 5, pp. 10–14, Sept 2016.

[118] B. Wheeler, "Should your apps be cloud-native?" https://devops.com/apps-cloud-native/, last accessed February 2018.

[119] J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-end arguments in system design,” ACM Transactions on Computer Systems, vol. 2, no. 4, pp. 277–288, Nov. 1984. [Online]. Available: http://doi.acm.org/10.1145/357401.357402

[120] “Microsoft Service Fabric,” https://azure.microsoft.com/en-us/services/service-fabric/, last accessed February 2018.

[121] “BMW Connected App,” http://www.bmwblog.com/2016/10/06/new-bmw-connected-app-now-available-ios-android/, last accessed February 2018.

[122] “BMW Open Mobility Cloud,” http://www.bmwblog.com/tag/open-mobility-cloud/, last accessed February 2018.

[123] “Service Fabric Customer Profile: BMW Technology Corporation,” https://blogs.msdn.microsoft.com/azureservicefabric/2016/08/24/service-fabric-customer-profile-bmw-technology-corporation/, last accessed February 2018.

[124] “Service Fabric Customer Profile: Mesh Systems,” https://blogs.msdn.microsoft.com/azureservicefabric/2016/06/20/service-fabric-customer-profile-mesh-systems/, last accessed February 2018.

[125] “Mesh Systems,” http://www.mesh-systems.com/, last accessed February 2018.

[126] “Quorum Business Solutions,” https://www.qbsol.com/, last accessed February 2018.

[127] “Service Fabric Customer Profile: Quorum Business Solutions,” https://blogs.msdn.microsoft.com/azureservicefabric/2016/11/15/service-fabric-customer-profile-quorum-business-solutions/, last accessed February 2018.

[128] “Service Fabric Customer Profile: TalkTalk TV,” https://blogs.msdn.microsoft.com/azureservicefabric/2016/03/15/service-fabric-customer-profile-talktalk-tv/, last accessed February 2018.

[129] “Talk Talk TV,” http://www.talktalk.co.uk/, last accessed February 2018.

[130] R. van Renesse, Y. Minsky, and M. Hayden, “A gossip-style failure detection service,” in Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, ser. Middleware ’98. London, UK, UK: Springer-Verlag, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=1659232.1659238 pp. 55–70.

[131] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ser. SIGCOMM ’01. New York, NY, USA: ACM, 2001. [Online]. Available: http://doi.acm.org/10.1145/383059.383071 pp. 149–160.

[132] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, ser. Middleware '01. London, UK, UK: Springer-Verlag, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=646591.697650 pp. 329–350.

[133] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system." Operating Systems Review, vol. 44, no. 2, pp. 35–40, 2010.

[134] R. Hasha, L. Xun, G. Kakivaya, and D. Malkhi, "Allocating and reclaiming resources within a rendezvous federation," https://patents.google.com/patent/US20080031246 A1, 2008, uS Patent 11,752,198.

[135] R. L. Hasha, L. Xun, G. K. R. Kakivaya, and D. Malkhi, "Maintaining consistency within a federation infrastructure," https://patents.google.com/patent/US20080288659 A1, 2008, uS Patent 11,936,589.

[136] G. Kakivaya, R. Hasha, L. Xun, and D. Malkhi, "Maintaining routing consistency within a rendezvous federation," https://patents.google.com/patent/US20080005624 A1, 2008, uS Patent 11,549,332.

[137] G. K. R. Kakivaya and L. Xun, "Neighborhood maintenance in the federation," https://patents.google.com/patent/US20090213757 A1, 2009, uS Patent 12,038,363.

[138] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric," in Revised Papers from the First International Workshop on Peer-to-Peer Systems, ser. IPTPS '01. London, UK, UK: Springer-Verlag, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=646334.687801 pp. 53–65.

[139] V. Ramasubramanian and E. G. Sirer, "Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays," in Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1, ser. NSDI '04. Berkeley, CA, USA: USENIX Association, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251175.1251183 pp. 8–8.

[140] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse, "Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead," in Proceedings of the 2nd International Workshop on Peer-to-Peer Systems, 2003.

[141] A. Gupta, B. Liskov, and R. Rodrigues, "One hop lookups for peer-to-peer overlays," in Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9, ser. HOTOS'03. Berkeley, CA, USA: USENIX Association, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251054.1251056 pp. 2–2.

[142] C. E. Perkins and E. M. Royer, "Ad-hoc on-demand distance vector (AODV) routing," in In Proceedings of the 2nd IEEE Workshop On Mobile Computing Systems and Applications, 1997, pp. 90–100.

[143] D. B. Johnson and D. A. Maltz, "Dynamic source routing in ad hoc wireless networks," in Mobile Computing. Kluwer Academic Publishers, 1996, pp. 153–181.

[144] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles, ser. SOSP '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281 pp. 205–220.

[145] "Adding nodes to an existing cluster," https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_add_node_to_cluster_t.html, last accessed February 2018.

[146] A. Khachaturyan, S. Semenovsovskaya, and B. Vainshtein, "The thermodynamic approach to the structure analysis of crystals," Acta Crystallographica Section A, vol. 37, no. 5, pp. 742–754, Sep 1981. [Online]. Available: https://doi.org/10.1107/S0567739481001630

[147] Y. Ge and G. Wei, "GA-Based Task Scheduler for the Cloud Computing Systems," in Proceedings of International Conference on Web Information Systems and Mining, vol. 2, Oct 2010, pp. 181–186.

[148] J. Carretero and F. Xhafa, "Genetic algorithm based schedulers for Grid computing systems," in International Journal of Innovative Computing, Information, and Control ICIC 3, vol. 5, 01 2007, pp. 1053–1071.

[149] G. ning Gan, T. lei Huang, and S. Gao, "Genetic simulated annealing algorithm for task scheduling based on cloud computing environment," in 2010 International Conference on Intelligent Computing and Integrated Systems, Oct 2010, pp. 60–63.

[150] "Azure Queue Storage," https://azure.microsoft.com/en-us/services/storage/queues/, last accessed February 2018.

[151] "Azure Table Storage," https://azure.microsoft.com/en-us/services/storage/tables/, last accessed February 2018.

[152] "Redis," https://redis.io/, last accessed February 2018.

[153] "Netflix Open Source Software Center," https://netflix.github.io/, last accessed February 2018.

[154] "Zuul," https://github.com/Netflix/zuul, last accessed February 2018.

[155] "Eureka," https://github.com/Netflix/eureka, last accessed February 2018.

[156] "Archaius," https://github.com/Netflix/archaius, last accessed February 2018.

[157] "Ribbon," https://github.com/Netflix/ribbon, last accessed February 2018.

[158] "Pivotal Application," https://pivotal.io/platform/pivotal-application-service, last accessed February 2018.

[159] "Spring Cloud," http://projects.spring.io/spring-cloud/, last accessed February 2018.

[160] "Azure Functions," https://azure.microsoft.com/en-us/services/functions/, last accessed February 2018.

[161] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally-distributed Database," in Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387905 pp. 251–264.

[162] "Azure Container Service," https://azure.microsoft.com/en-us/services/container-service/, last accessed February 2018.

[163] "MariaDB," https://mariadb.org/, last accessed February 2018.

[164] "MongoDB," https://www.mongodb.org/, last accessed February 2018.

[165] "CouchDB," http://couchdb.apache.org/, last accessed February 2018.

[166] "Amazon SimpleDB," https://aws.amazon.com/simpledb/, last accessed February 2018.

[167] "Riak," http://basho.com/products/, last accessed February 2018.

[168] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring, "DeeDS : Towards a distributed and active real-time database system," ACM SIGMOD Record, vol. 25, no. 1, pp. 38–51, Mar. 1996. [Online]. Available: http://doi.acm.org/10.1145/381854.381881

[169] B. Carstoiu and D. Carstoiu, "High performance eventually consistent distributed database Zatara," in Proceedings of the 6th International Conference on Networked Computing, May 2010, pp. 1–6.

[170] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, ser. OSDI '12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387906 pp. 265–278.

[171] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in Proceedings of the 23rd ACM Symposium on Operating Systems Principles, ser. SOSP '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043593 pp. 401–416.

[172] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: Distributed transactions with consistency, availability, and performance," in Proceedings of the 25th Symposium on Operating Systems Principles, ser. SOSP '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815425 pp. 54–70.

[173] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout, "Implementing linearizability at large scale and low latency," in Proceedings of the 25th Symposium on Operating Systems Principles, ser. SOSP '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815416 pp. 71–86.

[174] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in Proceedings of the 25th Symposium on Operating Systems Principles, ser. SOSP '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815419 pp. 87–104.

[175] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in Proceedings of the 4th Annual Symposium on Cloud Computing, ser. SOCC '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633 pp. 5:1–5:16.

[176] "Hadoop," http://hadoop.apache.org/, last accessed February 2018.

[177] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," in Proceedings of the Annual Conference on USENIX Annual Technical Conference, ser. ATEC '04. Berkeley, CA, USA: USENIX Association, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247415.1247425 pp. 10–10.

[178] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in Proceedings of the 10th USENIX Conference on File and Storage Technologies, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208479 pp. 18–18.

[179] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron, "Virtual ring routing: Network routing inspired by dhts," SIGCOMM Comput. Commun. Rev., vol. 36, no. 4, pp. 351–362, Aug. 2006. [Online]. Available: http://doi.acm.org/10.1145/1151659.1159954

[180] A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," in Proc. of the 1st International Conference on Embedded Networked Sensor Systems, ser. SenSys '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/958491.958494 pp. 14–27.

[181] S. Marti, T. J. Giuli, K. Lai, and M. Baker, "Mitigating routing misbehavior in mobile ad hoc networks," in Proc. of the 6th Annual International Conference on Mobile Computing and Networking, ser. MobiCom '00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/345910.345955 pp. 255–265.

[182] K. P. Birman, "The process group approach to reliable distributed computing," Commun. ACM, vol. 36, no. 12, pp. 37–53, Dec. 1993. [Online]. Available: http://doi.acm.org/10.1145/163298.163303

[183] I. Gupta, K. Birman, and van Renesse R., "Fighting fire with fire: Using randomized gossip to combat stochastic scalability limits," Quality and Reliability Engineering Int. Journal, vol. 18, no. 3, pp. 165–184, 2002.

[184] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in Proc. of the Sixteenth ACM Symposium on Operating Systems Principles, 1997. [Online]. Available: http://doi.acm.org/10.1145/268998.266711 pp. 288–301.

[185] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman, "An efficient multicast protocol for content-based publish-subscribe systems," in Proc. 19th IEEE International Conference on Distributed Computing Systems, June 1999, pp. 262–272.

[186] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," SIGMOD Rec., vol. 30, no. 2, pp. 115–126, May 2001. [Online]. Available: http://doi.acm.org/10.1145/376284.375677

[187] R. Rodrigues and P. Druschel, "Peer-to-peer systems," Commun. ACM, vol. 53, no. 10, pp. 72–82, Oct. 2010. [Online]. Available: http://doi.acm.org/10.1145/1831407.1831427

[188] T. Hampel, T. Bopp, and R. Hinn, "A peer-to-peer architecture for massive multiplayer online games," in Proc. of 5th ACM SIGCOMM Workshop on Network and System Support for Games, 2006. [Online]. Available: http://doi.acm.org/10.1145/1230040.1230058

[189] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-peer support for massively multiplayer games," in IEEE INFOCOM, vol. 1, March 2004, p. 107.

[190] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," J. ACM, vol. 43, no. 2, pp. 225–267, Mar. 1996. [Online]. Available: http://doi.acm.org/10.1145/226643.226647

[191] "What is the IBM SP2?" https://www.cac.cornell.edu/slantz/what_is_sp2.html.

[192] A. J. Ganesh, A. Kermarrec, and L. Massoulie, "Peer-to-peer membership management for gossip-based protocols," IEEE Transactions on Computers, vol. 52, no. 2, pp. 139–149, Feb 2003.

[193] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The ϕ accrual failure detector," in Proc. of the 23rd IEEE International Symposium on Reliable Distributed Systems. Washington, DC, USA: IEEE Computer Society, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1032662.1034350 pp. 66–78.

[194] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in Proc. of the Third Symposium on Operating Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=296806.296824 pp. 173–186.

[195] "Emulab," https://www.emulab.net/.

[196] "Emulab: D710," https://wiki.emulab.net/wiki/d710.

[197] M. K. Aguilera, W. Chen, and S. Toueg, "Heartbeat: A timeout-free failure detector for quiescent reliable communication," in Distributed Algorithms, 11th International Workshop, WDAG '97, Saarbrücken, Germany, September 24-26, 1997, Proceedings, 1997. [Online]. Available: https://doi.org/10.1007/BFb0030680 pp. 126–140.

[198] C. Almeida and P. Verissimo, "Timing failure detection and real-time group communication in quasi-synchronous systems," in Proceedings of the Eighth Euromicro Workshop on Real-Time Systems, June 1996, pp. 230–235.

[199] Bollo, L. Narzul, Raynal, and Tronel, "Probabilistic analysis of a group failure detection protocol," in 1999 Proceedings. Fourth International Workshop on Object-Oriented Real-Time Dependable Systems, Jan 1999, pp. 156–162.

[200] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," IEEE Transactions on Computers, vol. 51, no. 1, pp. 13–32, Jan 2002.

[201] S. A. Fakhouri, G. Goldszmidt, and I. Gupta, "Gulfstream - a system for dynamic topology management in multi-domain server farms," in Proceedings 42nd IEEE Symposium on Foundations of Computer Science, Oct 2001, pp. 55–62.

[202] C. Fetzer and F. Cristian, "Fail-awareness in timed asynchronous systems," in Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, ser. PODC '96. New York, NY, USA: ACM, 1996. [Online]. Available: http://doi.acm.org/10.1145/248052.248119 pp. 314–321.

[203] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish, "Detecting failures in distributed systems with the falcon spy network," in Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp. 279–294.

[204] R. Van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," ACM Trans. Comput. Syst., vol. 21, no. 2, pp. 164–206, May 2003. [Online]. Available: http://doi.acm.org/10.1145/762483.762485

[205] "ScyllaDB," https://www.scylladb.com/.

[206] A. Newell, G. Kliot, I. Menache, A. Gopalan, S. Akiyama, and M. Silberstein, "Optimizing distributed actor systems for dynamic interactive services," in Proc. of the Eleventh European Conference on Computer Systems, 2016. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901343 pp. 38:1–38:15.

[207] "UBER Ringpop. ringpup-184: Leader election on top of a weakly consistent system," https://github.com/uber/ringpop-go/issues/184.

[208] "Netflix Dynomite," https://github.com/Netflix/dynomite.

[209] "Etcd," https://coreos.com/etcd/.

[210] L. Suresh, D. Malkhi, P. Gopalan, I. P. Carreiro, and Z. Lokhandwala, "Stable and consistent membership at scale with rapid," in 2018 USENIX Annual Technical Conference, July 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/suresh pp. 387–400.

[211] "Kerberos," https://web.mit.edu/kerberos/, last accessed February 2018.

[212] "Nest outages prove that the smart home needs a local fallback," https://www.androidpolice.com/2020/03/28/nest-outages-prove-that-the-smart-home-needs-a-local-fallback/, last accessed April 2020.

[213] "Concurrency control," https://en.wikipedia.org/wiki/Concurrency_control, last accessed April 2020.

[214] "Understanding the fatal Tesla accident on Autopilot and the NHTSA probe," https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe/, last accessed February 2019.

[215] G. Berry, "Preemption in concurrent systems," in Foundations of Software Technology and Theoretical Computer Science, R. K. Shyamasundar, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 72–93.

[216] N. Komninos, E. Philippou, and A. Pitsillides, "Survey in smart grid and smart home security: Issues, challenges and countermeasures," IEEE Communications Surveys Tutorials, vol. 16, no. 4, pp. 1933–1954, 2014.

[217] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), 2012, pp. 127–140.

[218] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," in ACM SIGCOMM Computer Communication Review. ACM, 2011, pp. 290–301.

[219] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), 2013, pp. 15–27.

[220] S. Son, S. Shin, V. Yegneswaran, P. A. Porras, and G. Gu, "Model checking invariant security properties in openflow." in ICC, 2013, pp. 1974–1979.

[221] H. Hu, G.-J. Ahn, W. Han, and Z. Zhao, "Towards a reliable *SDN* firewall," in Presented as part of the Open Networking Summit 2014 ({ONS} 2014), 2014.

[222] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "Flowguard: building robust firewalls for software-defined networks," in Proceedings of the third workshop on Hot topics in software defined networking. ACM, 2014, pp. 97–102.