UCSB Technical Report #2024-01

University of California
Santa Barbara

Replicated Versioned Data for the Internet of Things (IoT)

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Nazmus Saquib

Committee in charge:

Professor Chandra Krintz, Co-Chair
Professor Rich Wolski, Co-Chair
Professor Amr El Abbadi

July 2023

The Dissertation of Nazmus Saquib is approved.

Professor Amr El Abbadi

Professor Rich Wolski, Committee Co-Chair

Professor Chandra Krintz, Committee Co-Chair

July 2023

Replicated Versioned Data for the Internet of Things (IoT)

Acknowledgements

My profound appreciation goes out to my mentors Chandra Krintz and Rich Wolski. Not only have their guidance and mentorship made me a better academic, but they have also positively impacted other aspects of my life apart from my graduate program. I am grateful for the time and effort they have invested in me. The freedom they gave me when it came to conducting research is something I will always treasure. More than anything, I will fondly remember the compassion they showed toward me during challenging times.

I thank Amr El Abbadi for being a part of my committee. His valuable insights and feedback gave my work a polished and refined appearance. I thoroughly enjoyed our multiple discussions that stemmed from his interest in my work. I greatly appreciate the time he took to review my work.

I am grateful to my labmates, both past and present. It is not an exaggeration to say that they often served as friends, brothers, and sisters to me.

Finally, I thank my parents for all they have done for me.

Curriculum Vitæ
Nazmus Saquib

Education

2023	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2017	M.S. in Computer Science, Bangladesh University of Engineering and Technology.
2015	B.S. in Computer Science, Bangladesh University of Engineering and Technology.

Publications

GreenCoin: A Renewable Energy-Aware Cryptocurrency

N. Saquib, S. Kapoor, C. Krintz, R. Wolski, and M. Mock

IEEE International Conference on Cloud Engineering (IC2E), 2023.

Replicated Versioned Data Structures for Wide-Area Distributed Systems

N. Saquib, C. Krintz, and R. Wolski

IEEE Transactions on Parallel and Distributed Systems (TPDS), 2022.

Log-Based CRDT for Edge Applications

N. Saquib, C. Krintz, and R. Wolski

IEEE International Conference on Cloud Engineering (IC2E), 2022.

Ordering Operations for Generic Replicated Data Types Using Version Trees

N. Saquib, C. Krintz, and R. Wolski

Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC), 2022.

A Resource-Efficient Smart Contract for Privacy Preserving Smart Home Systems

N. Saquib, F. Bakir, C. Krintz, and R. Wolski

IEEE Smart City Innovation (SCI), 2021.

Pedals: Persisting versioned data structures

N. Saquib, C. Krintz, and R. Wolski

IEEE International Conference on Cloud Engineering (IC2E), 2021.

Please Note: Text and figures from these papers are used and appear in this dissertation.

Abstract

Replicated Versioned Data for the Internet of Things (IoT)

by

Nazmus Saquib

Modern Internet of Things (IoT) deployments are geo-distributed and comprise various devices and architectures. The ever-increasing pervasiveness of IoT has led to the popularity of multi-tier (cloud/edge/sensor) architectures. However, the data interaction among the devices of the different tiers is fraught with various and often related challenges. For starters, these devices are vastly heterogeneous. On one end, we have cloud devices with high computing and storage capacities; on the other, we have resource-constrained microcontrollers and sensors. Due to the nature of the deployment of end devices, i.e., microcontrollers and sensors, stable network connectivity is often inaccessible to them. Unreliable power sources add complexity to the already challenging environment of end devices. Hence, we need novel approaches to create robust, fault-tolerant IoT deployments by negotiating these challenges.

To that end, we employ a three-pronged approach to make distributed IoT systems reliable and fault-tolerant. First, we propose leveraging versioned data structures to track the evolution and efficient querying/modification of data. As versioned data structures retain the past states, we can record the data lineage, which we can later use for system debugging and root cause analysis. Second, we propose a novel replication method that reduces coordination overhead. As IoT environments are failure-prone, protocols requiring more coordination can result in multiple stop-start scenarios, thus wasting energy in environments already in short supply. Finally, we explore an energy-efficient, energy-conscious way of making data tamper-proof. Tamper-proofing gives us confidence in the veracity of data and provides a way to detect faulty or malicious devices in the system.

We empirically evaluate our methodology through various real-world IoT applications. Our results show that data versioning can lead to efficient temporal queries and historical data retention that is useful for debugging IoT applications and deployments. Moreover, experimental results on replicating versioned data using our proposed protocol demonstrate better throughput and overall better latency than existing protocols with similar consistency guarantees. Finally, our proof-of-concept system for tamper-proof data management shows it is feasible to create a renewable-energy-aware system that ensures data trustworthiness in an energy efficient way. Hence, we conclude that we can leverage data versioning and replication to create the next generation of distributed, reliable, fault-tolerant IoT systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past decades, hardware design and fabrication technology advancements have made low-cost, low-power, and low-energy computing devices feasible. These advancements, in turn, have propelled the ubiquity of the Internet of Things (IoT). In a nutshell, IoT is a rapidly growing and continuously evolving set of technologies where *things*, i.e., devices, communicate with each other over the *Internet*. IoT makes it possible to glean insights from our surroundings through sensors and perform actions through actuators to respond to the collected information. Consequently, IoT applications have been seamlessly integrated into our daily lives, ranging from compact wearables like smartwatches to larger industry-standard devices that monitor and regulate supply chain management systems.

As this technology permeates the world around us, we are experiencing an explosion of data due to advancements in sensing, data storage, analytics, and artificial intelligence. This massive amount of data, coupled with advancements in data analytics and artificial intelligence, has allowed us to gather actionable insights. The potential of such insights have in turn resulted in a wide range of IoT applications that are increasingly data driven.

Data-driven IoT makes it possible to collect, mine, analyze, and actuate using information from physical objects across geographical locations. IoT applications amalgamate clouds, edge devices, and sensors as ensemble deployments implementing automation, decision support, and control for objects, devices, and systems in the environment. As a result, IoT deployments are geo-distributed and compose a vast diversity of devices, architectures, resource constraints, and communication systems with highly variable performance, failure, and availability characteristics.

Although IoT has tremendous potential for positively impacting society, it also has its unique challenges which center around its heterogeneity. Modern IoT deployments are multi-tiered and comprise resource-constrained devices at the end tier, moderately capable devices at the edge tier, and devices with high computing power and ample space at the cloud tier. This results in a vast range of devices with different scales and capabilities. Moreover, the mode of communication these devices use can vary among cloud/edge/end and even within the same tier. There is no unifying set of programming abstractions designed to span resource scales of heterogeneous devices — from resource-restricted IoT end devices (e.g., microcontrollers) to the extensive resource pools available from public clouds.

In part because of this heterogeneity, IoT deployments are inherently failure-prone. IoT deployments place end devices in environments lacking stable network connectivity or a reliable power source. Hence, many multi-tier applications must tolerate frequent communication disruptions, outages, and failures when operating across tiers. In the case of a crash, a device must be able to retain the stored data and resume operation upon successful restart. Moreover, if a change in some system component causes the failure, tracking the evolution of the impacted data, i.e., the data lineage, can facilitate debugging. One way to efficiently capture data lineage is through data versioning. In particular, versioned data structures provide a semantically uniform interface across interacting devices. In the case of communication disruption between any two devices, there must be a way to resume communication and make progress once the network is up. However, frequent communication drops can lead to unnecessary overhead if machines must retransmit messages often, thus making the system energy inefficient. Hence, we require new protocols that minimize message transmission.

A prime example of a case where communication overhead can have a significant adverse performance impact is replication. Distributed systems such as IoT often replicate data for low latency and high availability. The replicas maintain shared state among themselves according to different system and data models. For example, strong consistency requires a machine to coordinate with other devices to execute an operation. Coordination increases latency, and the execution may fail due to a network partition, preventing the necessary communication between replicas. Moreover, dropped messages may lead to frequent start-stop of iterations within a consensus protocol, making the deployment energy inefficient. Weaker consistency models, such as eventual consistency, enable a replica to execute an operation locally and asynchronously propagate it to others. This results in lower latency and communication overhead but causes a temporary divergence in replica views. The devices must eventually reconcile this temporary divergence. Fortunately, many IoT applications can sacrifice strong consistency in favor of high availability and partition tolerance. The adoption of a weaker consistency model, in turn, can result in an energy-efficient deployment due to lower communication overhead.

As the world becomes data-driven, applications must also ensure that the data they use is trustworthy if it is to be useful for responsible decision-making. Falsified or fabricated data can result in undesirable consequences. A popular method to ensure the trustworthiness of data is to use a blockchain. Blockchains are decentralized, eventually consistent, tamper-proof ledgers. Traditionally, blockchain protocols involve solving mathematically complex puzzles that are resource-intensive and energy inefficient. As a result, they are only suitable for the cloud tier containing devices with high compute and storage capabilities and uninterrupted access to grid power. Although less resource-intensive alternatives exist for some protocols, there remains an opportunity to optimize them further to take advantage of renewable energy sources when available.

Given the current limitations, opportunities, and requirements of multi-tier IoT deployments, we pose the following Thesis question:

Can we leverage data versioning and replication to make distributed IoT systems reliable, fault-tolerant, and trustworthy?

To address the Thesis question, we explore the following topics in the domain of multi-tier IoT deployments:

- An approach to extending data services with a well-defined versioning scheme that allows us to capture the evolution of data and facilitates efficient storage and query,
- A replication protocol that minimizes coordination overhead while being able to retain versioning information, and
- An energy-aware blockchain system for providing tamper-proof IoT data.

We initiate our exploration with versioned data structures. Versioned data structures can efficiently store and query their past states while exposing a simple interface where end users can interact. Hence, if application data is versioned, its evolution, i.e., lineage, is inherently preserved rather than capturing this lineage being an afterthought. Versioning facilitates monitoring changes in data and, in turn, system debugging. Although past research on versioned data structures exists, these works were constrained to in-memory and single-machine data structures. However, IoT deployments are distributed and failure-prone. Hence, to employ versioned data structures for multi-tier IoT deployments, we must extend it with persistent storage and distributed programming.

To this end, we propose PEDaLS, our approach to combine storage persistence and data structure versioning. PEDaLS is a language and runtime extension set that provides new abstractions for program data structures to simplify and expedite software development for event-triggered and failure-prone settings. It stores program data structures transparently in non-volatile storage to survive unexpected program termination, system failure, and power outages. PEDaLS uses append-only logs as the underlying storage abstraction to facilitate failure recovery and data structure versioning. We use append-only logs for storage as it provides immutability, which, in turn, offers robustness [1] and helps in debugging systems [2], both of which are highly desirable features in a distributed environment. Using append-only logs as the backing store introduces new challenges, as any modification of a node in a data structure must be recorded using an append rather than an in-place update. PEDaLS addresses these challenges while maintaining the same time and space complexity of the original single-machine, in-memory versioning algorithm and facilitating distributed storage persistence. Moreover, using append-only logs enables the integration of PEDaLS with several existing logging systems [3, 4, 5, 6, 7, 8].

While data versioning facilitates system debugging and crash recovery, replication adds availability and robustness to the system. Replication makes program data structures durable, highly available, and concurrently accessible despite resource failures. Although PEDaLS provides storage persistence and distributed programming, it cannot independently replicate the underlying data structures. However, multi-tier deployments pose unique challenges, which make many existing replication protocols inapplicable. First, these deployments include many heterogeneous devices, including resource-constrained systems such as microcontrollers and single-board computers. Thus, protocols requiring significant memory or complex computation are unsuitable for them. Second, these devices are commonly battery-powered and connect via unstable networks. Protocols requiring frequent coordination among replicas can experience repeated failure and resumption of the replication procedure in such settings. Due to these reasons, protocols that require a quorum through multiple messages among the replicas are precluded for many IoT devices. Although protocols exist that forego frequent coordination at the cost of stronger consistency semantics, they often come with conditions that limit the applicability of the system. Hence, we must introduce new replication protocols that can address these challenges and thus are better suited for multi-tier IoT deployments.

To employ a replication protocol well suited for multi-tier IoT deployments, we next introduce Log-Structured CRDTs (LSCRDTs) – a new form of *C*onflict-Free *R*eplicated *D*ata *T*ypes [9, 10, 11] that is better suited for use in multi-tier cloud deployments than its antecedents. CRDTs enable program data types to be distributed, shared (concurrently modified), replicated, and made consistent (via clever update merging). Specifically, CRDT employs Strong Eventual Consistency (SEC) [9], which guarantees that two replicas reach the same state if they receive the same set of updates (possibly in different orders). These features facilitate robustness, high availability, and coordination avoidance [1], which makes them suitable for failure-prone settings. However, many existing CRDT designs impose limitations that make them difficult to use in multi-tier cloud deployments. In particular, previous CRDT formulations do not support *operation reversal*, i.e., reverting a data type to an earlier state.[1] They do not readily support non-commutative operations, and for operation-based CRDTs, they cannot tolerate out-of-order operation delivery.

LSCRDT uses distributed logs to overcome these limitations. As logs are append-only, our approach can use them to further reduce the coordination required to merge inconsistent replicas. LSCRDT guarantees that replicas execute operations in the same order, irrespective of data type, thus removing the need for commutability. LSCRDTs also avoid the complex networking required to guarantee exactly-once, causal delivery of operations needed by operation-based CRDTs. Finally, LSCRDT provides programmatic access to data structure versions, which helps with debugging, history-based program-

[1]Note that operation reversal is distinct from the *undo* operation [12], which may have side effects.

ming [13], and data replay and repair [14]. We extend PEDaLS with LSCRDT to provide replication of versioned program data structures.

Data versioning and replication are desirable features in multi-tier IoT deployment from a system perspective. In this data-driven era of IoT, end-users are placing increasing importance on the security and trustworthiness of data, which is the third crucial feature to consider. Recent years have seen a rise in the popularity of blockchain, which can act as a decentralized, tamper-resistant database. Blockchains possess inherent resistance to data tampering, due to which it is practically impossible to modify data retroactively once they record a transaction. However, most blockchain systems are notoriously resource intensive. Many traditional blockchain deployments consume massive amounts of energy for their operations, adversely impacting our environment. Although there has been a recent drive towards increasing the utilization of renewable energy, existing blockchain systems need a way to ensure the participating nodes are using them.

To incorporate security and trustworthiness provided by blockchain into multi-tier IoT deployments in an energy-efficient and energy-conscious way, we propose GreenCoin. GreenCoin is a cryptocurrency that marks each crypto coin with an indelible measure of the renewable energy that a blockchain node used to create it. GreenCoin relies on a novel Proof-of-Stake (PoS) protocol called Green Proof-of-Stake (GPoS). As PoS is significantly less power intensive than its famous counterpart Proof-of-Work (PoW), we choose to base our protocol on PoS. GPoS attaches a “green score” denoting the fraction of available energy from renewable sources to each crypto coin. It employs an incentive system that favors, i.e., attaches a greater value to greener coins. Further, GreenCoin and GPoS permit the formulation of “green” smart contracts that specify a minimum greenness score that must be presented to execute the contract. Note that we define GPoS as a PoS protocol as we believe future protocols will likely take this form, but the fundamental tenets of GreenCoin apply to PoW protocols as well.

To summarize, we address the thesis question in a three-pronged fashion. First, we introduce a set of language and runtime extensions called PEDaLS to efficiently version, persist, and query program data structures for multi-tier IoT deployments. Second, we formulate a novel replication protocol based on CRDT, called LSCRDT , that is well-suited for failure-prone environments without complex requirements from the underlying network or application constraints. Finally, we provide data security, reliability, and trustworthiness through a novel, renewable energy-aware cryptocurrency called Green-Coin.

We structure the remaining of this dissertation as follows. Chapter 2 presents existing technologies and research relevant to our work. In Chapter 3, we put forth PEDaLS, our approach to making program data structures storage persistent along with retaining versioning information. Chapter 4 explores LSCRDT, a replication protocol suitable for multi-tier IoT deployments. At the same time, we explain how LSCRDT and PEDaLS can provide both versioning and replication of data. Chapter 5 describes how we can add trust to our data and its operations through a renewable energy-aware cryptocurrency system called GreenCoin. Finally, we conclude our learning outcomes and present possible future research directions in Chapter 6.

Chapter 2

Background

The IoT paradigm is becoming increasingly ubiquitous and often employs a multi-tier (cloud/edge/device) architecture. Many modern IoT deployments require efficient computation and storage solely on the cloud tier and seamless integration of computational/storage concepts on the edge and the device tier. Edge servers must communicate with high-end cloud servers while overseeing IoT devices at the same time. Unfortunately, the technologies that guarantee failure resilience and robustness on the cloud do not translate directly to the edge/device layer due to resource constraints and unstable network connectivity. Thus, new programming support and data abstractions are needed to manage this complexity and automatically enhance the robustness and efficiency of multi-scale IoT deployments.

The next generation of IoT can benefit from *versioned data*, as it facilitates debugging and can make applications experiencing frequent temporal queries efficient. Although *replicated data* is not a new concept, data replication in multi-tier IoT deployments requires ways to counter its inherent challenges of heterogeneity, unstable network, and unreliable power sources. Finally, in this age of data explosion, end users often desire *tamper-proof data* to ensure data integrity and trustworthiness. This chapter provides background on these three seemingly disparate but related topics that are key to making future IoT deployments reliable and fault-tolerant.

2.1 Versioned Data

Versioned data structures preserve their past states. Modifying such a data structure creates a new version while keeping the past versions accessible. Versioned data structures are also known as persistent data structures in the literature, as coined by Driscoll, Sarnak, Sleator, and Tarjan [15]. In this context, persistence does not imply storage persistence (i.e., in a system context, the ability to persist data during a power-off state); rather, it denotes that the data structures retain the previous versions (i.e., states). The formulation of Driscoll et al. is for single-machine, in-memory program data structures.

Persistent data structures come in multiple forms. Data structures for which all versions can be accessed, but only the latest/newest can be modified, are called *partially persistent*. Those for which all versions can be accessed and modified are called *fully persistent*. A *confluently persistent* [16] data structure can merge different versions into one. Data structures without versioning support are called ephemeral. We focus on partially persistent data structures in our work because version histories are immutable and data structures are append-only – properties that are desirable in highly concurrent and failure-prone distributed settings. We refer to partially persistent data structures simply as PDSs throughout this dissertation.

PDS update operations result in a new version of the structure. PDSs achieve algorithmic efficiency (in both space and time) by (i) embedding the versions within the original data structure and sharing unmodified sections of the data structure among versions, (ii) by implementing this embedding using *mutable*, pointer-based memory structures behind the scenes, and (iii) by optimizing accesses and updates to versions [15, 17, 18, 16, 19, 20].

PDSs enable a wide range of programming support, including debugging [21], history programming [22, 23], undo and replay [24, 25], lock avoidance [26], referential transparency, and functional programming [27, 28, 29, 30]. Although most works on PDSs are related to theoretical computer science, PDSs also have applications in distributed systems. For example, sensors in an IoT environment can often malfunction and generate erroneous values. If we have a PDS deployed in the system, we can query the past versions of it to determine at what point we started receiving faulty values and perform further analysis (e.g., what trend in data it resulted, how it affected the other components of the system that depend on this data, etc.). Systems that experience a high volume of temporal queries, such as the location of items of interest (e.g., delivery tracking, livestock tracking, etc.) throughout the day, can also benefit from PDSs.

A technology related to data versioning is git [31]. Git is a version control system typically used for text documents with some support for binary documents. GitHub [32] is an online platform for hosting git repositories. A primary difference between persistent data structures and git is that the former is used for program data, whereas the latter is used mostly for source codes or files. Git provides support for exploring the difference between two versions, working on different versions (similar to fully persistent data structures), and merging different versions (similar to confluently persistent data structures), among others.

2.2 Replicated Data

Distributed systems replicate data to ensure the availability and robustness of the system. However, multi-tier deployments pose unique challenges which make many existing replication protocols inapplicable. First, these deployments include a vast range of heterogeneous devices including resource-constrained systems such as microcontrollers and single-board computers. Thus, protocols which require significant memory or complex computation are not suitable for them. Second, these devices are commonly battery-powered and are connected via unstable networks.

Protocols that require frequent coordination among replicas can experience repeated failure and restart of the replication procedure in such settings. Due to these reasons, protocols such as Paxos [33] and Raft [34] that provide strong consistency and require a quorum through multiple messages among the replicas are precluded for many IoT devices. Multiple works in the literature suggest resorting to a weaker consistency model in favor of availability [35, 36, 37, 38, 39]. Fortunately, many IoT use cases do not require strong consistency semantics and instead can tolerate weaker consistency models (with lower coordination requirements) such as eventual consistency [40, 41, 42]. Thus we can trade off strong consistency for high availability (which many of these use cases do require) in the face of network partitions [43].

Motivated by the various limitations of Paxos, researchers have proposed multiple variations over the years. In Paxos, the leader performs a disproportionately large amount of communication compared to followers. PigPaxos [44] attempts to reduce the load of the leader by distributing some communication responsibilities to follower nodes, called relays. However, for write-heavy workloads (e.g. for sensor-driven IoT applications), the communication load is inherently distributed, limiting PigPaxos' advantage. Moreover, PigPaxos only redistributes the communication workload among nodes; it does not necessarily reduce the total amount of required communication (versus Paxos). Additionally, in failure-prone, heterogeneous environments such as IoT, replicas running Paxos-based protocols often must give up on current progress and start fresh if a quorum is not met. A quorum might not be met for many reasons including multiple contending writers, device/power failure, network latency, etc. Hence the overhead of Paxos-based algorithms is compounded in such scenarios. On the other hand, eventual consistency-based protocols can progress even in failure-prone environments in presence of multiple writers, as each replica is able to execute its operation immediately and incorporate the rest of the operations when the other devices become reachable.

DPaxos [45] proposes a dynamic allocation of quorums to avoid unnecessary wide-area communication. The communication reduction of DPaxos is heavily dependent on the premise that leader election is infrequent. However, if a leader-based protocol is used for write-heavy workloads, the leader election phase will be more frequent. Moreover, in the case of multiple writes, the dynamic allocation of quorums can be prolonged multiple times; effectively being computationally more complex than Paxos without providing any additional advantage. Hence we employ strong eventual consistency for the write-heavy edge deployments to reduce overall communication overhead.

2.3 Tamper-Proof Data

With technological advancements such as big data, data analytics, and artificial intelligence, data has become a pivotal component of many applications. However, the continued growth of data has also given rise to many questions, one of which is data trustworthiness and reliability.

A blockchain is a distributed, peer-to-peer, immutable digital ledger consisting of blocks. Due to blockchain's inherent tamper-resistance, it has become a popular mode of ensuring data has not been tampered with. Each block contains one or more transactions along with a hash of these transactions. These blocks are mined (i.e., added to the chain) by solving cryptographically complex problems requiring significant computational power. High-end devices that are used to solve these problems are called miners. There are generally multiple miners in a blockchain forming a peer-to-peer network.

The ordering of blocks in a blockchain is achieved using a consensus algorithm. Each block contains a hash of the previous one, meaning the entire chain can be traversed and validated from any block up to the first one (called genesis). The cryptographic algorithms are designed in such a way that for an adversarial entity to tamper with a block and still get validated would require an impractical amount of computational power. A blockchain provides multiple features that are desirable to ensure privacy and security:

- *Decentralization*: As a blockchain is a peer-to-peer system, users need not rely on an untrusted third party to store their data.
- *Immutability*: A blockchain can ensure the integrity of transactions through its immutable ledger. Transactions performed between two parties using the blockchain cannot be disputed.
- *Transparency*: A transaction mined in one node is propagated to multiple nodes in the blockchain. Moreover, these transactions are validated by the receiving nodes. This adds to the confidence of the involved parties regarding the correctness of the transactions.

While versioned data structures provide immutability from a programming construct and blockchains provide cryptographic guarantees to ensure the same, append-only logs can offer a different layer of immutability from a storage perspective. Due to a decline in storage costs, append-only logs are used widely in distributed and cloud computing systems to facilitate immutability, robustness, and scalability. Examples of log-based systems include cloud object stores [4, 6], event systems [46], distributed databases and file systems [47, 48, 49, 50, 51], log-based transaction systems [52, 53, 54], and popular messaging and streaming services [55, 7, 8].

Immutability facilitates robustness and coordination avoidance [1, 56] as well as high availability (through eventual consistency) for cloud storage, gossip protocols, collaborative editing, and revision control, among others [57, 58, 5]. While data versioning provides immutability from a software level, append-only logs provide immutability from a storage level. Entries in a log are ordered, and most log storage systems offer a form of *sequence number* (e.g., Kafka [59] provides offsets, and Facebook LogDevice [60] provides log sequence numbers) that reflects the log order. Log storage systems typically provide a simple API for creating a log, appending to a log, and retrieving entries from specific sequence numbers in a log. This generic API facilitates communication among heterogeneous devices.

While blockchain has cryptographic backing to ensure the integrity of each added block, append-only semantics of logs are only guaranteed through their exposed interfaces. That is, if a malicious user were to change data previously appended to a log through some means, there is no feasible way to detect this anomaly. On the other hand, any such attempt in a blockchain would reveal this modification through cryptographic algorithms. The downside is that, in addition, blockchain is resource-intensive. Blockchain is inherently decentralized, with algorithms that add to its integrity through this decentralization.

Chapter 3

PEDaLS: Persisting Versioned Data Structures

The Internet of Things (IoT), Big Data, and artificial intelligence are coalescing to transform the world around us, making it possible to collect, mine, analyze, and actuate using information from physical objects across geographical locations. To do so, IoT applications amalgamate clouds, edge devices, and sensors as ensemble deployments implementing automation, decision support, and control for objects, devices, and systems in the environment. As a result, IoT deployments are geo-distributed and compose a vast diversity of devices, architectures, resource constraints, and communication systems with highly variable performance, failure, and availability characteristics. This massive influx of data, coupled with the heterogeneity of the devices, demands efficient ways to store, communicate, and analyze data. We start our endeavor to make distributed IoT systems reliable and fault-tolerant by introducing data versioning. Versioned data structures can efficiently store the evolution of data, make the application data reliable and fault-tolerant, and provide a standard interface for the heterogeneous devices to communicate with each other.

Many popular and important cloud-based services for durable data management and data-driven computation (e.g. AWS S3 [5], HDFS [48], CORFU [52], Kafka [59], AWS Lambda [61], etc.) appear at first to provide the scale and fault-resiliency required for IoT. However, in practice, these services turn out to be ill-suited for IoT deployments because they assume "resource-rich" (e.g. machines with large memories and high clock speeds), cloud-based, and comparatively homogeneous infrastructure connected via high quality and large-capacity networks.

In this chapter, we consider how to evolve and extend data services for IoT applications that target tiered edge-and-cloud IoT deployments. Our goal is to tame the heterogeneity of such deployments via new programming abstractions in the form of sophisticated program data structures that simplify and expedite software development for event-triggered and failure-prone settings. Our approach, called *PEDaLS*, combines storage persistence and data structure versioning. First, we transparently store program data structures in non-volatile storage so that they survive unexpected program termination, system failure, and power outage. Second, we use an append-only log as the underlying storage abstraction to facilitate failure recovery as well as data structure versioning and immutability. Doing so enables integration with several existing distributed logging systems [3, 4, 5, 6, 7, 8].

In this chapter, we focus on persisting versions of linked program data structures (e.g. linked lists and trees). To do so efficiently, we base our design on algorithmically efficient in-memory, mutable algorithm implementations from Driscoll, Sarnak, Sleator, and Tarjan [15], which embed versions (i.e. nodes and edges) within the original data structure. Note that while the above work proposes a versioning scheme for single-machine, in-memory data structures, our work formulates a versioning scheme for storage in persistent data structures which can span multiple machines in a distributed setting. We use append-only logs for storage as it provides immutability, which in turn provides robustness [1] and helps in debugging systems [2], both of which are highly desirable features in a distributed environment. The use of append-only logs as the backing store introduces new challenges, as any modification of a node in a data structure must be recorded using an append rather than in-place modification. PEDaLS addresses these challenges while maintaining the same time and space complexity of the original work and facilitating distributed storage persistence.

In addition to this new distributed formulation of persistent versioning, we describe a working "real world" implementation of PEDaLS for cloud and IoT settings. We prototype our approach using an open-source, distributed runtime system [3] that supports distributed logs as a first-class storage abstraction. We use this prototype to evaluate empirically the various overheads associated with versioning and persistence for operation workloads on linked lists and binary search trees, and investigate the effect of failures on the logging process. Our results show that PEDaLS maintains the algorithmic complexities of in-memory versioning, and enhances the robustness of distributed data structures with low overhead.

3.1 Related Work

We first provide background, related work, and context for the contributions we describe in the sections that follow. Specifically, we overview mutable, in-memory versioned data structures (referred to as partially persistent data structures (PDSs) in the literature), non-volatile program object storage, and append-only storage advances that enhance the robustness of distributed systems.

3.1.1 Partially Persistent Data Structures (PDSs)

PDSs track version histories for in-memory, program data structures such that versions can be accessed programmatically [15]. Data structures for which all versions can be accessed, but only the latest/newest can be modified, are called *partially persistent*. Those for which all versions can be accessed and modified are called *fully persistent*. Data structures without versioning support are called *ephemeral*. We focus on partially persistent data structures in this work because version histories are immutable and data structures are append-only – properties that are desirable in highly concurrent and failure-prone, distributed settings. We refer to partially persistent data structures simply as PDSs throughout this dissertation.

PDS update operations result in a new version of the structure. PDSs achieve algorithmic efficiency (in both space and time) by (i) embedding the versions within the original data structure and sharing unmodified sections of the data structure among versions, (ii) by implementing this embedding using *mutable*, pointer-based memory structures behind the scenes, and (iii) by optimizing accesses and updates to versions [15, 17, 18, 16, 19, 20]. PDSs enable a wide range of programming support including debugging [21], history programming [22, 23], undo and replay [24, 25], lock avoidance [26], referential transparency and functional programming [27, 28, 29, 30]. Although most works on PDSs are related to the field of theoretical computer science, PDSs have applications in distributed systems as well. For example, sensors in an IoT environment can often malfunction and generate erroneous values. If we have a PDS deployed in the system, we can query the past versions of it to determine at what point we started receiving erroneous values and perform further analysis (e.g. what trend in data it resulted, how it affected the other components of the system that depend on this data, etc.). Systems that experience a high volume of temporal queries, such as the location of items of interest (e.g. delivery tracking, livestock tracking, etc.) throughout the day can also benefit from PDSs.

PEDaLS brings PDS support to programs in a new way, using efficient algorithms that are backed by append-only disk storage exported via logs. For IoT, these efficiencies are necessary to enable battery-powered, resource-restricted platforms (e.g. IoT devices) to use as little power as possible. Often, power consumption is proportional to computation duration and storage access frequency. Append-only semantics are also attractive in this context to mitigate component failures, particularly in the form of communication network partitions.

3.1.2 Object Storage

The term *persistence* is also used in computer science to describe the long-term, non-volatile storage of data (e.g. in files or databases on disk), i.e. enabling data to exceed the lifetime of any particular program activation. Related work has investigated (i) tools and language support that automate the process of persisting program (in-memory) data structures and objects to disk (and more recently to non-volatile memory), and (ii) ways of unifying the treatment of transient and persistent objects in programs to simplify programming [62, 63, 51, 64, 65, 66, 67, 68, 69, 70, 71, 72]. These advances are referred to as persistent storage, persistent object storage, persistent object systems, orthogonal persistence, and persistent programming in the literature. PEDaLS pursues both automation and unification of persistent object storage, but is unique in that it persists *versioned data structures* (i.e. PDSs) to local or remote append-only disk storage. Doing so enables both program data structures and their versions to survive program termination and be accessed by distributed clients.

3.1.3 Append-Only Storage Systems

Append-only storage is employed in distributed and cloud computing systems to facilitate immutability, robustness, and scalability, as storage costs have plummeted [1]. It is used by cloud object stores [4, 6], event systems [46], distributed databases and file systems [47, 48, 49, 50, 51], log-based transaction systems [52, 53, 54], and popular messaging and streaming services [55, 7, 8].

Immutability facilitates robustness and coordination avoidance [1, 56] as well as high availability (through eventual consistency) for cloud storage, gossip protocols, collaborative editing, and revision control, among others [57, 58, 5]. In particular, versioning in distributed systems allows applications to make progress from the last available consistent state [73]. While our evaluation implementation uses the versioning features of a storage system specifically designed for IoT [3], PEDaLS can use any append-only storage system that exports ordered, version information (e.g. sequence numbers) as its backing store for program-level, versioned data structures.

3.2 PDS Node-Copy Method

Driscoll et. al. introduced a *node-copy method* to version linked data structures (with constant in-degree) by embedding versions efficiently within the structure itself [15]. PEDaLS extends this method using append-only logs. We first overview the original approach and then describe our advances.

Using the node-copy method, a versioned linked data structure consists of nodes and edges, and each node contains a constant value and 1+ edges (i.e. pointer fields). The method adds a fixed number of *extra pointer fields* beyond those required by the original structure. For example, in the case of a binary search tree (BST), a node contains fields for its value and left and right pointer. Additional pointer fields represent versions – i.e.

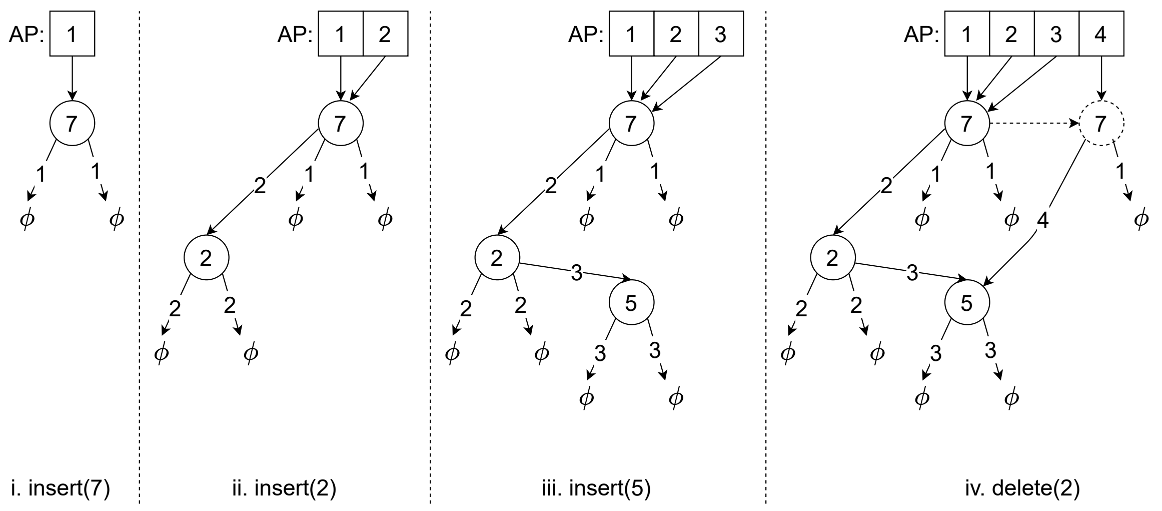


Figure 3.1: Partially persistent binary search tree using the node-copy method with one extra pointer. Circles denote node with information field within. Arrows with labels denote pointers with version stamps. Dashed arrows/circles denote that a node has been copied. AP is the access pointer list. ϕ denotes the null node.

updates to the left or right pointer of the node.

Once all of the extra fields have been used to accommodate update operations, the method makes a copy of the node with only the most recent pointer fields – creating a new set of extra pointer fields for use in future updates. Moreover, the predecessor of the node stores a pointer to this new copy. This way, the method avoids walking through chains of copies of the same node to locate a particular version. Note that if the predecessor runs out of extra pointer fields while pointing to a new copy of a node, the predecessor is copied as well. In the worst case, this copying operation and chaining continue to the root node. The method also maintains a list of root nodes indexed by version stamps called the *access pointer* (AP) list. The AP facilitates constant time lookup of the root node for any version.

The number of extra pointers used by the method is a tunable parameter. If the number of extra pointers is small, the time to scan them is short but the number of copies generated may increase, resulting in a higher time and space overhead. If the number of extra pointers is large, it takes more time to scan all the pointers but fewer copies will be needed. We explore this time-space tradeoff for PEDaLS in Section 3.4.

To illustrate how the node-copy method works for linked data structures, consider the binary search tree as shown in Figure 3.1. We assume that the number of extra pointers is one. We start with the empty tree and insert 7. Both the left and right pointer of the node containing 7 points to null. Assuming version stamps start at one and monotonically increase thereafter, we stamp these pointers with version 1. We also update the AP list (assuming indexing starts at 1) to point to this newly created node.

Next, we insert 2 in the same way. Because 2 is less than 7, we install a new left pointer in the BST using the extra pointer in the node containing 7 and stamp it with the current version (2). The type of the extra pointer (i.e. left or right – in this case, left) is recorded (not shown in the figure for brevity). As the root node does not change, index 2 of AP list points to the same node as AP index 1.

Next, we insert 5 which follows similar insertion steps. Finally, we delete 2. To do this, the node containing 7 must point to the node containing 5 (using a left pointer) and the null node (using a right pointer). However, the node containing 7 has run out of extra pointers and thus must be copied. The original left pointer of this new copy is set to point to the node containing 5 and is stamped with version 4. The original right pointer need not be updated and thus still points to the null node. The extra pointer of this new copy remains unused and is available for future updates. Note that as the node originally containing 7 has been copied, index 4 (i.e. the current version stamp) of the AP list points to the copied node rather than to the original node.

Access operations (i.e. find/search) for a particular version stamp vs traverse pointers with the greatest version stamp less than or equal to vs at each node. Instead of specifying only a value ($find(val)$), a PDS find operation can also include a version stamp ($find(val, vs)$). As an example, consider the operation $find(2, 3)$ – find 2 in version stamp 3, after the execution of all the operations in Figure 3.1. That is, the current BST is represented by the last column of Figure 3.1.

The access operation starts from index 3 in the AP list, which points to the node containing 7. As 2 is less than 7, we find the left pointer with the largest version stamp that is less than or equal to 3. In this case, there are two left pointers – one with a version stamp 1 and the other with a version stamp 2. As both are less than the target version stamp 3, we follow the larger one. This leads us to the node containing 2, i.e., 2 is present in version stamp 3. Note that if we search for 2 in version stamp 4 instead, we eventually end up with the null node, indicating that 2 is not present in version stamp 4.

The amortized time complexity for insert and delete using the node-copy method is constant per operation step, where an operation step is defined as the traversal from one node to another [15]. The worst-case time complexity for access using this method is also constant per operation step. Moreover, the worst-case space complexity for insert and delete using the node-copy method is constant per operation step.

3.3 PEDaLS

PEDaLS is a set of language and runtime extensions that

- Transparently store immutable and versioned linked data structures in distributed, non-volatile, log-based storage;
- Expose data structure versions to developers for use in dependency tracking and program analysis [74, 75, 76], history-aware programming [22, 13], and repair and replay [14, 2, 77, 78, 79] in distributed settings; and
- Enables portability across heterogeneous deployments by requiring only a limited "generic" functionality for generating and accessing storage-persistent logging sys-

tems (e.g. [4, 5, 6, 59, 60]) in a distributed setting.

As a result, PEDaLS data structures are able to support versioning and immutability end-to-end as distributed application and systems properties, which are desirable in highly concurrent and failure-prone settings [1, 80].

To enable this, we develop a methodology for realizing PDSs using generic, distributed log structures to facilitate integration with existing systems. A PEDaLS log must

- support append-only updates with ordered entries,
- be network addressable so that they can be co-located or remote relative to the process accessing them, and
- have some mechanism for controlling log length (e.g., size or log entry/element lifetime for automatic garbage collection).

In addition, log elements can be of any type and must be accessible via a comparable index (e.g. a sequence number).

The API functions that PEDaLS expects the storage system to support (or compose to support) are:

- `createLog(log_name)`: create a log with the name `log_name`. Upon completion, this call returns a value that indicates whether or not the log was successfully created.
- `put(log_name,elem)`: append the element `elem` to the log named `log_name`, assigning it the next available sequence number, and return the sequence number to the caller (or an error value if the operation fails).
- `get(log_name,seq_no)`: return the element at sequence number `seq_no` in the log (or an error value if the operation fails or `seq_no` does not exist).

- `getLatestSeqNo(log_name)`: return the latest sequence number of the log named `log_name` (or an error value if the operation fails or the log is empty).

Example systems that support these persistent storage functionalities directly include Kafka [59], Facebook LogDevice [60], and CSPOT [3] among others. Most cloud object stores also support versioning (e.g. Amazon S3 [4] and Google Cloud Storage [6]) and can be integrated into PEDaLS with some additional bookkeeping (e.g. combining version IDs with their timestamp to maintain order). To map PDSs to these distributed storage systems, PEDaLS must overcome multiple challenges that we describe in the subsection that follows.

Figure 3.2 provides a high-level overview of our approach, which models the PDS node-copy method described previously, using persistent logs. Each node and original field has a version stamp (vs) that denotes the version at which the node was created. Each node also has a constant number of extra fields (1 is used/shown in the figure), which hold version stamps that track the version at which original field updates occur in each node. As in the original method, we assume that information (value) fields are constant and that pointer fields (e.g. the next pointer of a linked list) can change across versions. End-users interact with the data structure using library calls as shown in the top left corner of Figure 3.2. Any modification to a data structure node is translated to low-level API functions of the underlying log storage, possibly affecting multiple geographically distributed logs. PEDaLS hides this translation from the end-user.

3.3.1 Challenges Using Logs to Implement Node-Copy

To map node-copy to logs, we must preserve the original time and space complexity of the original, in-memory, algorithm. Although the AP list of the node-copy method (cf. Section 3.2) can be modeled as a separate log for efficiency, doing so imposes undue

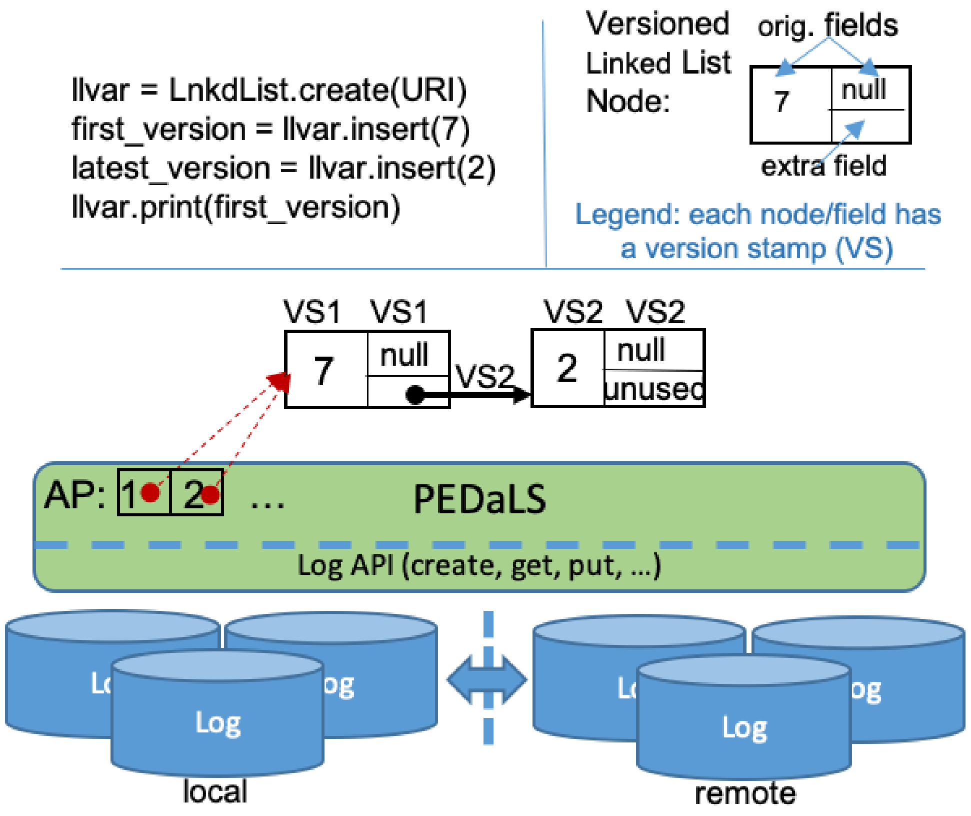

llvar = LnkdList.create(URI)
first_version = llvar.insert(7)
latest_version = llvar.insert(2)
llvar.print(first_version)
Versioned Linked List Node:
orig. fields
7
null
extra field
Legend: each node/field has a version stamp (VS)
VS1 VS1
7
null
VS2
VS2 VS2
2
null
unused
AP: 1 2 ...
PEDaLS
Log API (create, get, put, ...)
Log
local
Log
remote

Figure 3.2: High-level architecture of PEDaLS that uses node-copy to version linked data structures. The top left shows user code using PEDaLS library operations. Each node in this list (e.g. top right) has an integer value and next pointer as original (developer-defined) fields. PEDaLS embeds versions/modifications within a single data structure using "extra" fields in each node. Also, each node and field has a version stamp representing their creation "time". The PEDaLS AP represents the access pointer and indexes the root node of each version for fast access. PEDaLS persists the structure and its versions using 1+ non-volatile, append-only, distributed logs.

complexities. First, it is not clear how to represent both the information field and pointer fields of a node using logs. Moreover, logs are append-only – pointer manipulations must be expressed as appends (e.g. we cannot have an entry representing a pointer to a null node and later update that entry to point to a different node).

This leads to a challenge that is even more intricate – if we represent an updated node link by appending to a single log repeatedly, we potentially require a full log scan to find an arbitrary link – defeating our goal of maintaining the original time complexity of node-copy. To avoid this, we use multiple logs to represent nodes and their connections. This, however, leads to a new challenge – although an append to a single log is atomic, appends to multiple logs are not. Moreover, logs can be distributed across a network, so a network failure could potentially leave an underlying data structure semantically inconsistent.

To summarize, there are four primary challenges in implementing versioning via node-copy using logs:

- *C1*: Logs are append-only and thus we cannot perform any updates in place (as we do for in-memory structures as described above – specifically, creating links on the fly).

- *C2*: A scan of a log with an arbitrary number of entries will violate the amortized and the worst case time complexity of the operations guaranteed by the node-copy method.

- *C3*: Updates to a single log are atomic, however, PEDaLS must also guarantee that multi-log updates are also atomic if used to manage versioning.

- *C4*: Because the persistent backing store can be local to the function or on a host across a network, we must consider the impact of failures in our algorithms and

Table 3.1: Logs used by PEDaLS .

Log	**Field**	**Description**
DataLog	vs	version stamp during node creation
	val	information field of the node
	link	name of link log for the node
LinkLog	vs	version stamp during pointer creation
	dseq	DataLog seq. no. where the information field of the node being pointed to is stored
	lseq	LinkLog seq. no. of the node being pointed to where the first pointer among the contiguous pointers of the required copy is stored
	rem	number of extra pointers remaining after the insertion of the current pointer
	type	type of pointer, e.g., left/right for binary search tree
APLog	vs	version stamp of the data structure
	dseq	DataLog seq. no. where the root node's information field is stored
	lseq	LinkLog seq. no. of the root node where the first pointer of the required copy is stored

analyses.

We next describe a design that allows us to efficiently implement node-copy using logs while addressing these challenges.

3.3.2 Implementing Node-Copy Method using Logs

Our log mapping design, which avoids log scans (addressing *C2*), derives from two primary observations. First, data structure updates modify node pointers and these updates can be interleaved. We thus use a separate log per node to avoid scanning entries from unrelated updates. Second, when we copy a node (when it runs out of extra pointers), the information (e.g. value) does not change. We thus use a shared log (across nodes) to hold node information. This combination allows versioned data structure updates to occur independently, while maintaining the efficiency of find/search operations, avoiding copy overhead, and conserving space.

Specifically, PEDaLS represents a node in a linked data structure by a pair of log sequence numbers: one for the shared information log – the *DataLog*, and another for the node-specific pointer log – a *LinkLog*. When a pointer is added to a node, we append an entry to the *LinkLog* of the node (addressing *C1*). The second sequence number is used to distinguish node copies.

The efficiency of the in-memory node-copy method lies in the fact that every predecessor node points to the required copy of the successor node. To traverse a copy of a node we need only scan a *fixed* number of pointers. That is, we scan ($p = o + e$) pointers, where o is the number of original pointers and e is the number of extra pointers. Therefore, to achieve similar time complexity, we must restrict (i.e. fix) the number of entries in the LinkLog that we need to scan in order to traverse a node. This is relatively straightforward to do: because copies of a node are not interleaved (i.e. a node is copied only when the previous copy becomes full), we can use contiguous log entries of a LinkLog to represent a particular copy.

Initially, it appears that we can use contiguous p LinkLog entries to store a copy of a node and denote a copy using the first sequence number among these entries. That is, for the n-th copy of a node (considering the original node to be the "first" copy), the p entries starting from the sequence number ($(n-1)*p+1$) store that copy. This is indeed the case – in absence of network failures.

However, failures alter the situation. We consider two types of failures. (i) *Type 1*: an append to a log fails. (ii) *Type 2*: an append to a log succeeds, but the acknowledgment (which returns the sequence number where the entry was appended) is lost. In both cases PEDaLS retries. However, note that Type 2 failures violate the boundary conditions discussed above. Copies of a node do not strictly end at multiples of p anymore. This implies that we must embed the information regarding where a copy ends within an entry of the LinkLog.

To account for failures, we record the number of extra pointers left after the insertion of that entry in a dedicated field in the LinkLog. This way, once this field reads 0 while scanning entries of a copy in the LinkLog, we know we have reached the end of the current copy.

In general, we embed sufficient information in an entry of a log so that append to that log becomes idempotent (this addresses *C4*). Note that in presence of failures the number of entries we need to scan is bounded by the number of failures f ($p = o + e + f$).

Next, the node-copy method uses an AP list for constant time access to the root node of a particular version of the data structure. Similarly, PEDaLS maintains an *APLog* to store version root nodes for the data structure. PEDaLS writes the APLog last (the order of write is DataLog>LinkLog(s)>APLog). Therefore, an append to the APLog denotes the successful completion of a version.

That is, APLog acts as a checkpoint denoting the complete versions currently present in the data structure. This design choice addresses *C3*: if there are rogue entries in LinkLogs/DataLog with version stamps vs that are greater than the latest version stamp recorded in the APLog (this can be identified after a system crash or network failure), we know that the last operation did not complete and can either trim these entries or retry the operation (we log requested operations before we start execution for the latter).

Table 3.1 summarizes the different types of logs used by PEDaLS along with a description of the fields stored in each of their entries. Note that *type* field in LinkLog is used for the sake of generalization; data structures that have only one type of entry (e.g. singly linked list) ignore this field.

Most log storage systems have some form of built-in retention policy which prevents logs from growing without bounds. For example, Kafka [59] provides retention policies based both on time (messages older than a configured time are deleted) and on space (once a log reaches a configured space limit messages are deleted from the end). CSPOT [3] provides rollover where once a log reaches a specified number of entries, newer entries start overwriting the older ones. Therefore, to ensure all the required versions are preserved in their entirety, an end-user has to specify the number of versions K he/she wants to retain. PEDaLS then allocates enough log space for each type of log based on the value of K. Currently, PEDaLS refuses update operations once it reaches K versions. This sort of policy where service is refused based on the unavailability of space is not uncommon (e.g. Redis [81]). Note that PEDaLS continues to service read operations even after it reaches K versions.

3.3.3 Node-Copy using Logs: Step by Step Example

Figure 3.3 shows the contents of the different logs used by PEDaLS across multiple operations on a PEDaLS binary search tree (BST). As in Figure 3.1, we start with the empty BST and assume the number of extra pointers is 1. For the simplicity of exposition, we name the LinkLog of a node as *link_val*, where *val* is the node value (information).

To *insert*(7), we append the entry *(1,7,link_7)* to the DataLog which returns the sequence number 1. This sequence number will be used later to record the root in the APLog. We append two entries in *link_*7, one for the BST left pointer and one for the right pointer. Note that for both of these entries, the *rem* field is 1, denoting the number of extra pointers after the insertion. As both of these pointers point to the null node, we use an invalid sequence number (0) for *dseq* and *lseq*. The first append to LinkLog returns the sequence number 1. Therefore, we conclude the insertion of 7 by recording the tuple *(1,1,1)* in the APLog.

Operation *insert*(2) follows a similar approach with two added steps. First, we find the current version of the data structure. To do this, we read the tail of the APLog. This reveals that the latest version is 1 (this is the first field of the last entry in the APLog

DataLog

seq	vs	val	link
1	1	7	link_7

LinkLog: link_7

seq	vs	dseq	lseq	rem	type
1	1	0	0	1	L
2	1	0	0	1	R

APLog

seq	vs	dseq	lseq
1	1	1	1

i. insert(7)

DataLog

seq	vs	val	link
1	1	7	link_7
2	2	2	link_2

LinkLog: link_7

seq	vs	dseq	lseq	rem	type
1	1	0	0	1	L
2	1	0	0	1	R
3	2	2	1	0	L

APLog

seq	vs	dseq	lseq
1	1	1	1
2	2	1	1

LinkLog: link_2

seq	vs	dseq	lseq	rem	type
1	2	0	0	1	L
2	2	0	0	1	R

ii. insert(2)

DataLog

seq	vs	val	link
1	1	7	link_7
2	2	2	link_2
3	3	5	link_5

LinkLog: link_7

seq	vs	dseq	lseq	rem	type
1	1	0	0	1	L
2	1	0	0	1	R
3	2	2	1	0	L

APLog

seq	vs	dseq	lseq
1	1	1	1
2	2	1	1
3	3	1	1

LinkLog: link_2

seq	vs	dseq	lseq	rem	type
1	2	0	0	1	L
2	2	0	0	1	R
3	3	3	1	0	R

LinkLog: link_5

seq	vs	dseq	lseq	rem	type
1	3	0	0	1	L
2	3	0	0	1	R

iii. insert(5)

DataLog

seq	vs	val	link
1	1	7	link_7
2	2	2	link_2
3	3	5	link_5

LinkLog: link_7

seq	vs	dseq	lseq	rem	type
1	1	0	0	1	L
2	1	0	0	1	R
3	2	2	1	0	L
4	1	0	0	1	R
5	4	3	1	1	L

APLog

seq	vs	dseq	lseq
1	1	1	1
2	2	1	1
3	3	1	1
4	4	1	4

LinkLog: link_2

seq	vs	dseq	lseq	rem	type
1	2	0	0	1	L
2	2	0	0	1	R
3	3	3	1	0	R

LinkLog: link_5

seq	vs	dseq	lseq	rem	type
1	3	0	0	1	L
2	3	0	0	1	R

iv. delete(2)

Figure 3.3: A versioned binary search tree (BST) using node-copy with one extra pointer implemented on logs (this mirrors the in-memory tree in Figure 3.1). 0 is assumed to be an invalid sequence number and hence is used to denote null nodes. DataLog sequence numbers are color-coded to represent the links. LinkLog sequence numbers are color-coded only if the entry denotes the start of a root node.

in the top left corner of Figure 3.3), so the current working version is 2. Second, we add a pointer from the node containing 7 to the node containing 2. As the data for the node containing 2 was inserted at sequence number 2 of the DataLog and the first pointer of the node was inserted at sequence number 1 of its LinkLog, *dseq* and *lseq* values of this pointer are 2 and 1 respectively.

After recording this pointer, we decrement the number of extra pointers (recorded as 0 in the *rem* field). The value to be decremented comes from the tail of $link\_7$, which is 1 at this point. Note that the APLog entry for version 2 is identical to that of version 1, as the root node does not change. Execution of $insert(5)$ follows similar steps.

Execution of $delete(2)$ involves some additional steps, as no more extra pointers are left in the node containing 7 but we need to add 5 to the left of 7. Therefore, we make a copy of the node containing 7. Since the right pointer does not change, we copy over only the latest right pointer. This is done in sequence number 4 of $link\_7$ (bottom right corner of Figure 3.3). Next, we install the new left pointer with $dseq$ and $lseq$ values set to 3 (5 was inserted in sequence number 3 of DataLog) and 1 (node containing 2 pointed to 5 using $lseq$ value of 1), respectively. As the root node is copied in this case, we record the node in the APLog by appending $(4, 1, 4)$.

Note that for update operations, the crux of the algorithm is in node connectivity. We present the $AddNode$ routine for BST in Algorithm 1 (the routine for linked list is similar and simpler as there is only one original link and hence no link must be copied during node copy). The routine adds a child node ($cNode$) to the desired parent node ($pNode$). As successive predecessors may run out of extra pointers to accommodate this addition (cf. Section 3.2), the full path from the root of the tree to the desired parent node is supplied to the routine. We assume the node representation in the algorithm is a structure containing $dseq$, $lseq$, and the $link$ fields (cf. Table 3.1). The last entry in a log L is represented by $tail(L)$, the entry at sequence number i in log L is represented by

$L[i]$, and a field f in an entry e of a log is $e.f$. Note that the versions of a data structure are strictly ordered and a new version is obtained by modifying the previous version. Therefore, we need to know the previous version in its entirety before we can generate the next version of a data structure. Thus, PEDaLS does not allow concurrent updates.

Access operations follow a similar pattern to that of node-copy. As an example, we consider the operation $find(2, 3)$ – find 2 in version stamp 3, after executing the above operations (i.e. BST has the representation shown in the bottom right corner of Figure 3.3). We start by first locating the root node for version 3 from the APLog. In this case, the root node for version 3 is recorded in the entry at sequence number 3 in the APLog. From this entry, we know that the root node's data is stored in sequence number 1 ($dseq = 1$) of the DataLog.

The entry at sequence number 1 of DataLog provides us with the name of the LinkLog. As $lseq = 1$ in the APLog entry, we start scanning $link\_7$ from sequence number 1. Scanning the top three entries is enough to fully traverse the current copy of the node (as the third entry has $rem = 0$, denoting the end of the current copy). This reveals two left pointers, one with version stamp 1 and the other with version stamp 2. As $1 < 2$, we follow the latter one, i.e., the link at sequence number 3 of $link\_7$. This leads to the node whose information is stored in the entry at sequence number 2 of the DataLog. Reading this entry reveals the value stored here is indeed 2, completing the access operation.

3.4 Evaluation

In this section, we empirically evaluate the performance of PEDaLS. We implement PEDaLS over CSPOT [3], an open-source, distributed runtime system that runs on edge, cloud, and sensor systems, and uses memory-mapped files for its log abstraction. We use this implementation of PEDaLS to run microbenchmarks in Section 3.4.2. To demon-

Algorithm 1 Node Copy: AddNode (BST)

Require: childNode $cNode$; stack of nodes leading from root of BST to parent of $cNode$, S; number of links per node $linksPerNode$; working version stamp vs

Ensure: $cNode$ is added to its parent

```
 1: while $S \neq \phi$ do
 2:     $pNode \leftarrow S.pop()$
 3:     $lastLink \leftarrow tail(pNode.link)$
 4:     $newLink \leftarrow \{\}$
 5:     $newLink.vs \leftarrow vs$
 6:     $newLink.dseq \leftarrow cNode.dseq$
 7:     $newLink.lseq \leftarrow cNode.lseq$
 8:     $childType \leftarrow getType(pNode.dseq, cNode.dseq)$   ▷ getType returns type of link i.e. left
    or right
 9:     if $lastLink.rem > 0$ then                                ▷ node not full
10:         $newLink.rem \leftarrow lastLink.rem - 1$
11:         $pNode.link.append(newLink)$
12:         break
13:     else
14:         $leftLink \leftarrow \{\}$
15:         $rightLink \leftarrow \{\}$
16:         $i = pNode.leq$
17:         $iteratorLink \leftarrow pNode.link[i]$
18:         while $iteratorLink \neq \phi$ & $iteratorLink.rem \geq 0$ do
19:             if $iteratorLink.type = L$ then                   ▷ iterator is a left pointer
20:                 $leftLink \leftarrow iteratorLink$
21:             else
22:                 $rightLink \leftarrow iteratorLink$
23:             end if
24:             $i \leftarrow i + 1$
25:             $iteratorLink \leftarrow pNode.link[i]$
26:         end while
27:         if $childType = L$ then                                 ▷ copy right child
28:             $linkLogSeq \leftarrow pNode.link.append(rightLink)$
29:         else
30:             $linkLogSeq \leftarrow pNode.link.append(leftLink)$
31:         end if
32:         $newLink.rem \leftarrow linksPerNode - 2$
33:         $pNode.link.append(newLink)$
34:     end if
35:     $cNode.dseq \leftarrow pNode.dseq$
36:     $cNode.lseq \leftarrow linkLogSeq$
37:     $cNode.link \leftarrow pNode.link$
38: end while
```

strate PEDaLS is agnostic of the underlying log storage system, we implement it using only memory-mapped files as well without the extra features provided by CSPOT. We use this implementation of PEDaLS for end-to-end applications in Section 3.4.5. We evaluate linked lists and binary search trees (BST) as representative linked data structures since both are used by developers as building blocks for more complex structures (e.g. stacks, queues, ordered collections, etc.). We present an example application where linked lists are used as queues in Section 3.4.5. Unless otherwise specified, we perform our experiments using a virtual machine instance in a private cloud running Eucalyptus [82]. The instance has two 2GHz CPUs and 2GB of memory.

3.4.1 Experimental Methodology

To evaluate PEDaLS, we have devised a set of update (insert/delete) workloads for linked lists and BSTs. We execute insert for linked lists at the end of the list. We present average workload time, which includes scan/find time (for both linked list and BST). We construct 100 different workloads (combinations of insert and delete operations), each with 1000 operations.

Our workload generator uses a uniform probability distribution to select operations. For insertion, the generator randomly chooses an integer between 1 to 100 with uniform distribution. If the integer is already present, it selects the next integer not already present in the data structure. For deletion, the generator randomly chooses an integer already present in the data structure with uniform distribution. This way the generator guarantees all operations will execute to completion. [1] Unless otherwise specified, our results present the average across 100 workloads.

In addition to microbenchmarks, we empirically evaluate the overhead associated

[1]Our workloads are available (for reproducibility purposes) as part of our open-source release of PEDaLS at https://github.com/MAYHEM-Lab/PEDaLS.

with introducing versioning to in-memory ephemeral data structures and subsequent integration of in-memory logs to represent versioning. Note that only this experiment is conducted completely in memory.

Next, we present a sample application of an image server using PEDaLS. The application (presented in Section 3.4.4) implements a simple clone of Amazon Simple Storage Service (S3) for storing and serving images using PEDaLS.

Moreover, we describe a number of end-to-end applications that leverages the efficiency of PEDaLS in answering complex temporal queries in Section 3.4.5 including comparison with popular data storage systems. To demonstrate that PEDaLS is agnostic of the underlying log storage as long as it provides functionalities to create, append to, and read from logs, we use memory-mapped files as the logs for this set of experiments.

Finally, we present a probabilistic study on bounding log sizes to ensure a user provided minimum number of versions can be fully recorded by PEDaLS in Section 3.4.6.

3.4.2 Microbenchmarks

To the best of our knowledge, no other system provides general-purpose versioning and storage persistence of program data structures. Moreover, we want to explore the cost of providing versioning and storage support to systems relying on in-memory ephemeral data structures. Thus, we compare PEDaLS data structures against in-memory ephemeral and in-memory persistent data structures (denoted as simply *ephemeral* and *persistent* in the results). We use memory-mapped files as a backing store for the in-memory data structures.

To evaluate the trade-off in space and time using extra pointers, we consider 1, 5, and 10 extra pointers for PDSs (both in-memory and PEDaLS). We refer to the PEDaLS implementation using n number of extra pointers as PEDaLS-n. Similarly, we refer to the in-memory persistent implementation using n number of extra pointers as persistent-n. Unless otherwise specified, we co-locate the logs and workload for this study.

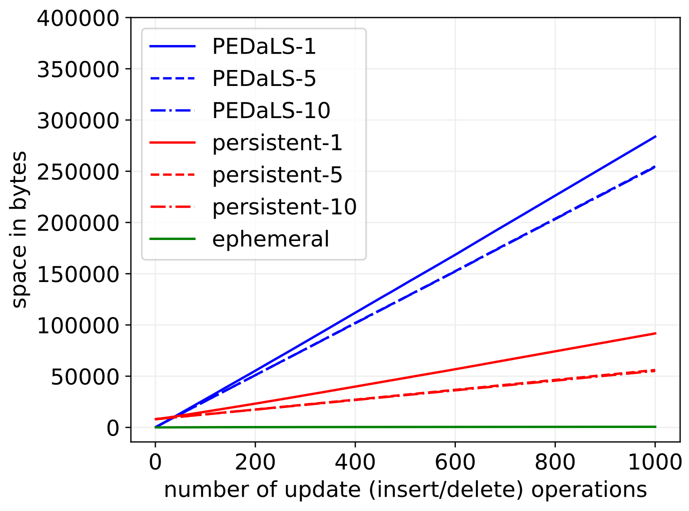
Space Analysis

We first evaluate PEDaLS space usage. Figures 3.4a and 3.4b show the average space in bytes used by linked list and BST respectively to execute 1 to 1000 operations. The results show that PEDaLS space requirements are linear with respect to the number of operations for versioned data structures (persistent-n and PEDaLS-n). The space requirements for ephemeral data structures are linear in the number of nodes present at any instant of time (due to scaling it appears to be constant in Figure 3.4).

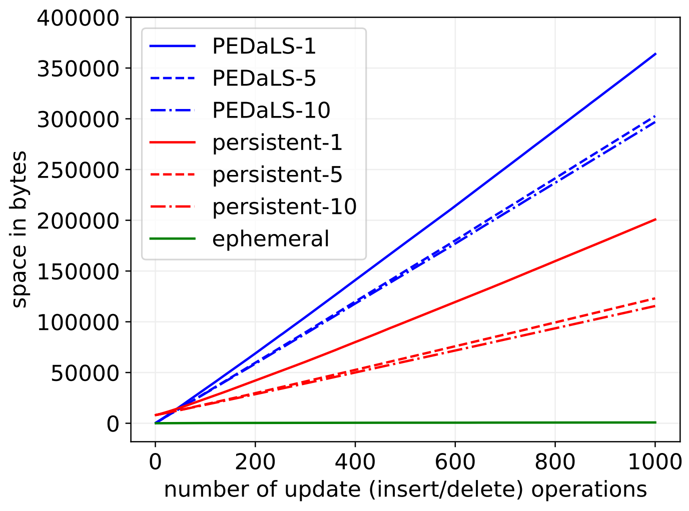
We expect that when the number of extra pointers is small, the node-copy method will copy more nodes (and consume more space). This is evident in the results. The average slope of the lines for PEDaLS-1 BST, PEDaLS-5 BST, and PEDaLS-10 BST are respectively 352, 300, and 296 (Figure 3.4b). This implies that, on average, each update operation in PEDaLS-1 BST requires 352 bytes, whereas each update operation in PEDaLS-10 BST requires 296 bytes.

Note that although the difference in average slope between PEDaLS-10 BST and PEDaLS-1 BST is more than 50, this difference reduces to 4 when considering PEDaLS-5 BST and PEDaLS-10 BST. That is, we only reduce space consumption via extra pointers up to a point. For linked list PEDaLS-10 saves roughly one byte of space per operation as compared to PEDaLS-5. When compared to persistent-n BSTs, PEDaLS-n BSTs require $1.50x$, $1.75x$, and $1.80x$ more space for $n = 1, 5$, and 10 respectively. That is, the space overhead to map the in-memory node-copy method to logs is quite low.

Similar observations for linked list from Figure 3.4a reveal space overhead is as low as $2.00x$. Ephemeral data structures can free the corresponding memory once a node is deleted. Moreover, they do not have to perform bookkeeping related to maintaining



(a) Linked list.



(b) BST.

Figure 3.4: Average space usage.

versioning information. Therefore, we expect their space requirement to be lower. Unsurprisingly, the space overhead to maintain PEDaLS-10 BST as opposed to ephemeral BST is $178x$. The same overhead is $202x$ for linked list.

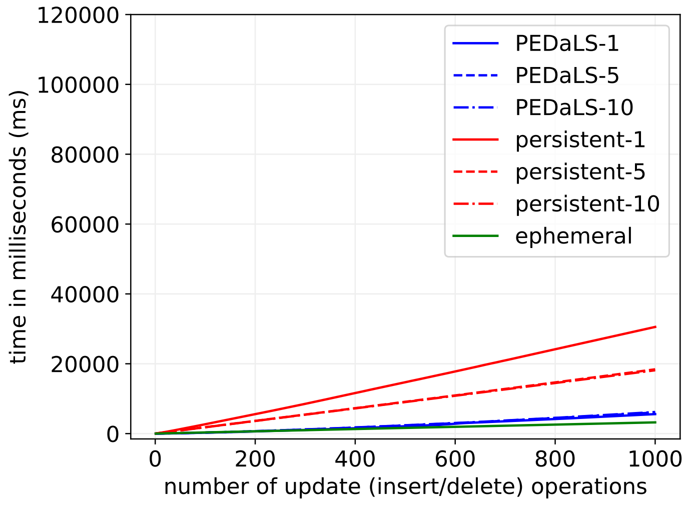
Time Analysis

We next consider the additional time needed for versioning and disk persistence. Figure 3.5 shows the average time taken by the different data structures to execute a number of operations ranging from 1 to 1000. The time requirements (shown on the y-axis) are linear with respect to the number of operations (shown on the x-axis).

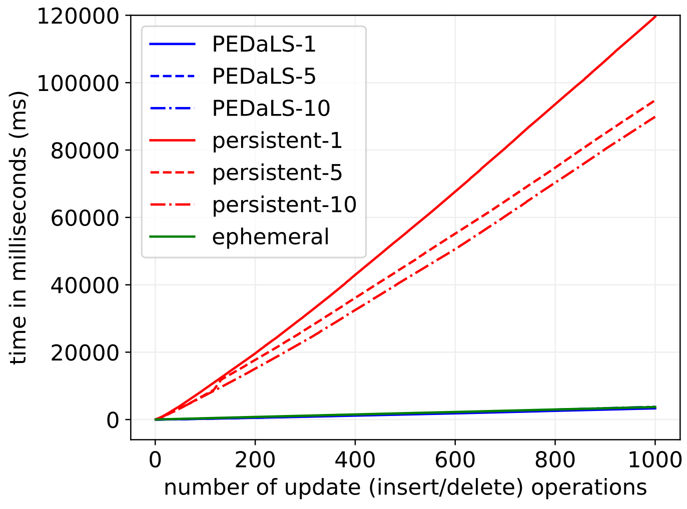
For ephemeral linked list, the average time taken to execute an update operation is 3.30 milliseconds. This value is 4.11 milliseconds, 4.47 milliseconds, and 4.51 milliseconds for PEDaLS-1 linked list, PEDaLS-5 linked list, and PEDaLS-10 linked list respectively. That is, even the slowest PEDaLS-n linked list implementation introduces only $1.35x$ overhead. The persistent-n implementations are the slowest, requiring 28.85 milliseconds, 18.28 milliseconds, and 18.18 milliseconds for $n = 1, 5$, and 10 respectively.

These results are surprising. We expect the performance order would be ephemeral >persistent-n >PEDaLS-n. However, in the case of linked list the experiments show PEDaLS-n has a better performance than persistent-n. Moreover, the small overhead in PEDaLS-n when compared to ephemeral, shows that it is possible to use a log-based approach to implement storage-persistent PDS linked list *without a significant performance penalty*, relative to the standard, mutable, and un-versioned pointer-based implementations.

These experiments indicate that the performance of the memory allocator plays a key role in the performance of versioned persistence. The underlying memory allocator that CSPOT (runtime system over which PEDaLS is currently implemented) uses is trivial: it simply appends to a fixed-size, pre-allocated, circular log buffer that is mapped to a



(a) Linked list.



(b) BST.

Figure 3.5: Average execution time.

Linux file. Further, there is no deallocation – the log "wraps" to automatically garbage collect log entries [3]. For the ephemeral and persistent implementations, the memory allocator uses a first-fit dynamic allocation algorithm with eager coalescing of adjacent free blocks on deallocation. Thus, the implementations that use a dynamic allocator cause it to "chase" an internal free list of blocks from time to time during allocation and coalescing.

Additionally, both allocators flush a modified memory region to the backing store (i.e. a Linux file) to prevent corruption in case of a system crash. The dynamic allocator versions (ephemeral and persistent) use the Linux `msync()` system call to write back modified mapped memory. For CentOS 7, this call causes either one or two (due to alignment) 8 kilobyte pages to be flushed to the backing file. Moreover, it must flush the modified memory to the backing store (i.e. a Linux file) each time the data changes in an allocated buffer or in the internal memory allocated data structures.

A cursory examination of the CSPOT source code [2] shows that it unmaps each storage log after each append operation, causing dirty pages to be flushed back. Thus it is likely that ephemeral performance is dominated by backing-store synchronization traffic. As a result, the additional computational overhead associated with versioning using the node-copy method is negligible.

Figure 3.5b shows the average time taken by the different implementations of BST to perform the workloads. The time requirements are again linear with respect to the number of operations. Ephemeral BST requires 3.82 milliseconds to execute an operation. PEDaLS-n BSTs require slightly lower – 2.88 milliseconds, 3.04 milliseconds, and 3.24 milliseconds for $n = 1, 5$, and 10 respectively.

The additional complexities inherent in a PDS implementation of BST put additional performance pressure on the dynamic memory allocator. This is reflected in the per-

[2]https://github.com/MAYHEM-Lab/cspot

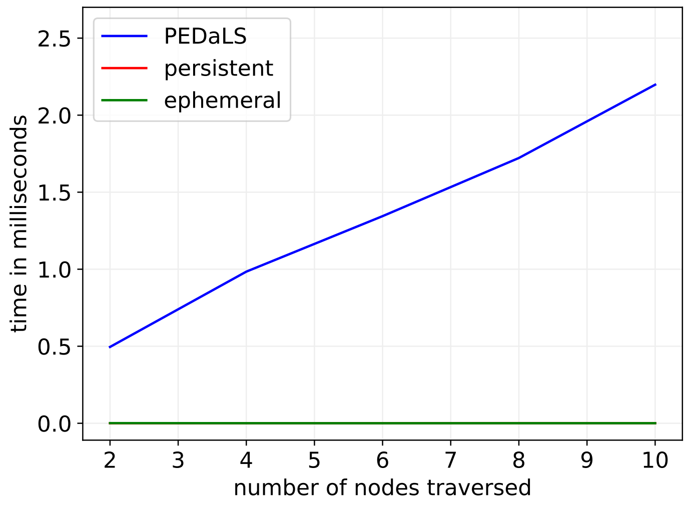
operation times for persistent-n, which are 106.71 milliseconds, 88.48 milliseconds, and 81.26 milliseconds for $n = 1, 5$, and 10 respectively. That is, persistent-n can have an overhead of as much as $28x$ when compared to ephemeral.

Search Performance

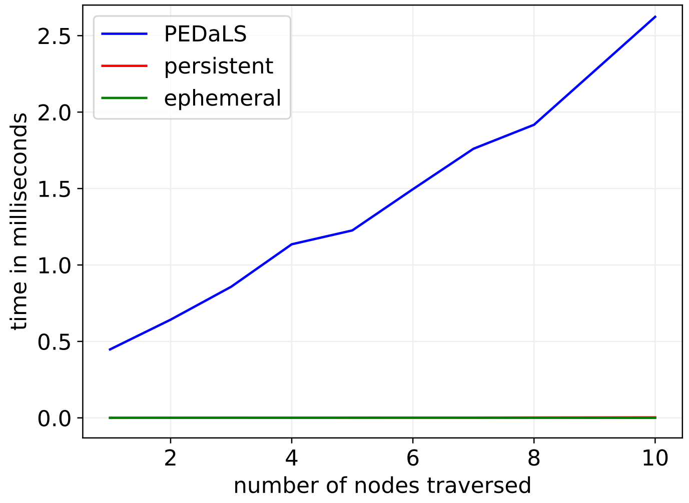
Note that update operations (insert/delete) require traversals of existing nodes (e.g. to find leaf node to which a new node is inserted). We next break out the time to perform access (find/search) operations alone, i.e. data structure traversal. We consider different access operations for the two data structures: (i) finding the last node in linked list (Figure 3.6a) and (ii) finding the maximum value in BST (Figure 3.6b). Figure 3.6 shows traversal time as a function of the number of nodes.

Note that although many workloads resulted in having > 10 nodes in linked list, We do not show the complete results for the sake of visual comparability (e.g. no workload resulted in a depth of > 10 BST nodes). Because we search from the latest version, we simply follow the last link (i.e. at most 2 links for BST) of each node. We find that varying the number of extra pointers for this experiment has no significant performance difference.

Figure 3.6 shows that the access time is linear in the number of nodes for PEDaLS. On average, PEDaLS requires 0.23 milliseconds and PEDaLS BST requires 0.29 milliseconds. Both the persistent and ephemeral data structures are three orders of magnitude faster than PEDaLS for access, with no significant difference between each other. This again emphasizes the importance of the performance of the memory allocator. For updates, PEDaLS is faster than persistent for both linked list and BST and is on par with ephemeral.



(a) Linked list.



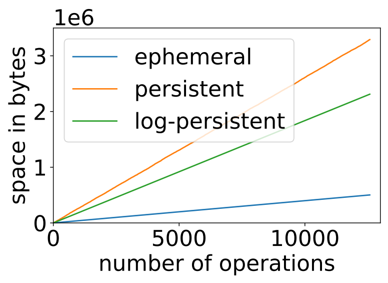
(b) BST.

Figure 3.6: Access (node traversal) time vs number of nodes traversed.

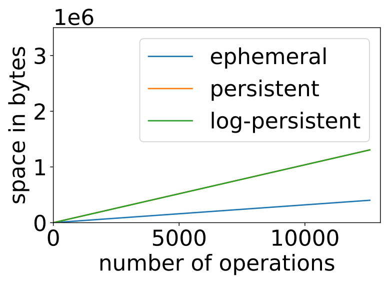
3.4.3 Versioning Overhead

In this experiment, we investigate the overhead associated with making in-memory ephemeral data structures versioned, i.e. in-memory persistent. We also investigate the overhead associated with implementing versioned data structures using in-memory logs (i.e., same as PEDaLS but without storage persistence), which we term as *log-persistent*. Note that in this experiment we are concerned exclusively with the overhead in these two cases: (i) making data structures versioned (i.e. persistent as described in [15]) and (ii) making data structures persistent using logs as described in Section 3.3.2. Hence we keep the storage (in this case, main memory) the same for all three types of data structures: ephemeral, persistent, and log-persistent. Moreover, we design the experiment to retain how a storage persistent log implementation would perform – by making an extra copy from a data structure node to the in-memory log. While the previous set of experiments in Section 3.4.2 use memory-mapped files for storage of all types of data structures, the experiments in this section store all types of data structures solely in memory. Our results show that log-persistent algorithms preserve the same time and space complexity as persistent algorithms.

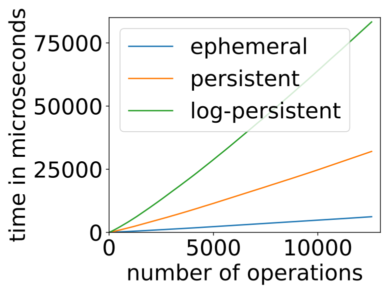
We choose linked list and binary search tree (bst) as representative linked data structures for this experiment since both are used by developers as building blocks for more complex structures, e.g., stacks, queues, ordered collections, etc. Note that the original work on persistent data structures [15] provides space/time complexity guarantees only for linked data structures having constant in-degree. Hence, although a similar mechanism can be employed for other data structures, a similar space/time complexity is not guaranteed. We use CityPulse [83] temperature dataset containing 12579 data points collected from the city of Aarhus in Denmark between February-June 2014 for this experiment. In case of bst, we use the UNIX timestamp at the time of the collection of



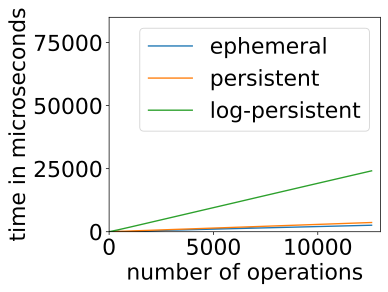
(a) BST space.



(b) Linked list space.



(c) BST time.



(d) Linked list time.

Figure 3.7: Average space and time requirement for bst and linked list. Both the space and the time requirements are linear with respect to the number of operations for the two data structures.

data point as the key and the temperature as the value, whereas linked list stores both as the value.

We present the space requirements for storing these data points in Figure 3.7a and Figure 3.7b. Note that this space is for the storage of the data structure, i.e. any auxiliary storage used (and subsequently freed) for intermediate computation is not included.

As evident from the figures, the space complexity is linear in number of operations for all three types of data structures – and differ only in the value of the constant. This signifies that log-persistent data structures are able to maintain a similar space complexity as persistent data structures. As expected, log-persistent and persistent data structures require more space than ephemeral data structures due to their retainment of information regarding past versions. In case of bst, persistent and log-persistent data structures require 6.55x and 4.60x space respectively of that of ephemeral data structure. In case of linked list, this overhead is 3.25x for both persistent and log-persistent data structures.

Interestingly, log-persistent bst requires less space than persistent bst. This is because copying a node in persistent data structures involves copying both the data field and pointer fields [15]. On the other hand, copying a node in log-persistent data structures (as in PEDaLS) involves copying only the pointer fields. In the case of linked list, both the persistent and log-persistent versions require the same amount of space. This is because in this experiment we are only inserting values in our linked list at the head, which never triggers copying of a node, thus keeping the storage space the same for both persistent and log-persistent data structures.

We present the time requirements of storing the temperature data in Figure 3.7c and Figure 3.7d. We see that the time complexity is also linear in the number of operations, signifying that log-persistent data structures are able to maintain a similar time complexity as their in-memory counterparts (persistent data structures). Log-persistent and

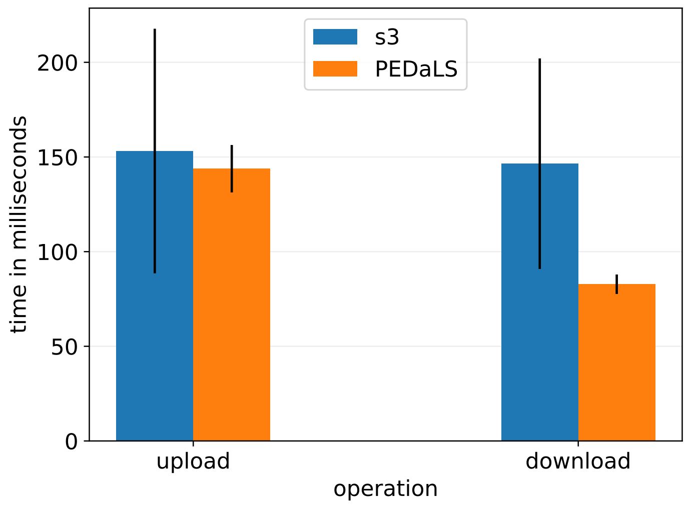


Figure 3.8: Average upload and download time per image for the Amazon S3 and PEDaLS image servers.

persistent data structures require more time than ephemeral data structures due to their relatively complex traversal rules. In the case of bst, persistent and log-persistent data structures require 5.17x and 13.44x times that of ephemeral data structure, respectively. In the case of linked list, these overheads are 1.42x and 9.38x for persistent and log-persistent data structures, respectively. The greater time required by log-persistent data structures is expected, as it involves additional non-trivial steps in its operations, such as complex methods to find boundaries among copies of nodes instead of having direct pointers to those copies.

3.4.4 Sample Application: Image Server

We evaluate the use of PEDaLS for a sample application commonly found in IoT settings, e.g. [84]. The program implements an image server, which sensors and/or users can use to upload and download images for analysis.

We compare two different implementations of this image server. For the first implementation, we use an Amazon S3 bucket located in the us-west-2 region as the server.

The client process is located in a private cloud in UCSB and interacts with the bucket using the boto3 library (the instance has the same specifications as the ones used so far – 2GHz CPU, 2GB RAM). For the second implementation, we use a t3.small (2GHz CPU, 2GB RAM) EC2 instance, also located in the us-west-2 region. We employ a PEDaLS -1 BST in this instance that acts as an image indexer. The average RTT between the client instance and the EC2 instance that we observe is approximately 30 milliseconds. Note that our goal is not to outperform the S3 image server, rather explore the utility of PEDaLS in an IoT setting.

For this experiment, we first upload 500 images (256 KB each) to the server, followed by the retrieval of the images from the server. Surprisingly, PEDaLS based server outperforms s3 based server in both upload time and download time. Figure 3.8 shows the average upload and download times per image for the two servers. The average upload time is 153 milliseconds for the S3 based server, whereas this value is 144 milliseconds for PEDaLS. That is, the latter is 1.1x faster. The average download time is 146 milliseconds for S3 based server, whereas this value is 83 milliseconds for PEDaLS based server. In this case, PEDaLS is 1.8x faster.

3.4.5 End-to-End Applications: Comparison with Popular Storage Systems

In this section, we evaluate the performance of PEDaLS for three different applications: (i) banking queue monitoring, (ii) room occupancy detection, and (iii) livestock tracking. We compare the performance of PEDaLS with a relational database (PostgreSQL) and a NoSQL database (MongoDB) for each of the above-mentioned applications. We summarize these results in Table 3.2 and provide a visual representation of the results in Figure 3.9.

Table 3.2: The mean update and query execution time for different end-to-end applications. The standard deviation is shown in parentheses. The best (fastest) execution time for each type of operation for each application is presented in bold. PEDaLS outperforms the other two storage systems in regard to query execution time for all applications.

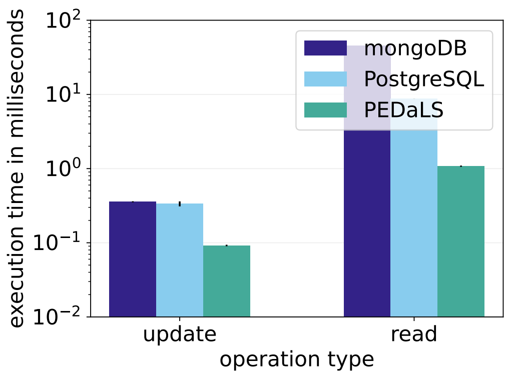
	Banking Queue		**Room Occupancy**		**Livestock Tracking**	
	update in ms	query in ms	update in ms	query in ms	update in ms	query in ms
PostgreSQL	0.335 (0.026)	8.734 (0.045)	**0.354 (0.039)**	0.460 (0.003)	**0.342 (0.022)**	5.186 (0.141)
MongoDB	0.357 (0.006)	45.184 (0.295)	0.359 (0.007)	2.255 (0.019)	0.355 (0.007)	11.687 (0.222)
PEDaLS	**0.092 (0.002)**	**1.078 (0.022)**	0.468 (0.017)	**0.124 (0.003)**	0.553 (0.014)	**2.333 (0.083)**

Our results show that PEDaLS outperforms the other two storage systems when it comes to answering complex temporal queries for all applications. Although PEDaLS is outperformed by the other two storage systems for update operations in two applications, the speedup provided by PEDaLS in executing temporal queries far outweighs the slowdown in updates. Moreover, PEDaLS provides programmatic access to versioned data, while other systems do not. In essence, PEDaLS makes a tradeoff between update performance and version accessibility in the favor of the latter.

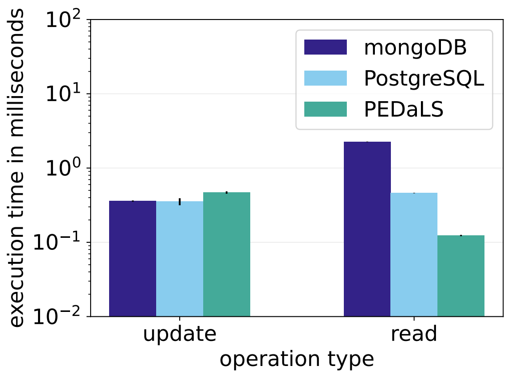
Banking Queue Monitoring

In mathematics, queuing theory involves the analysis of several related events such as arriving at a queue, the wait time at a queue, and departure from the queue. The models created through such analysis can be used to make decisions regarding increasing servers, optimizing queue length, and approximating heavy and light traffic [85]. One practical example of the application of these mathematical concepts is to determine the efficiency of a system as indicated by the average wait time of a request/person in a queue. Data related to physical queues can also help in providing location information for individuals (e.g. was a person in the queue at a certain time) and describing individual behavior (e.g. when does a person generally show up for a service).

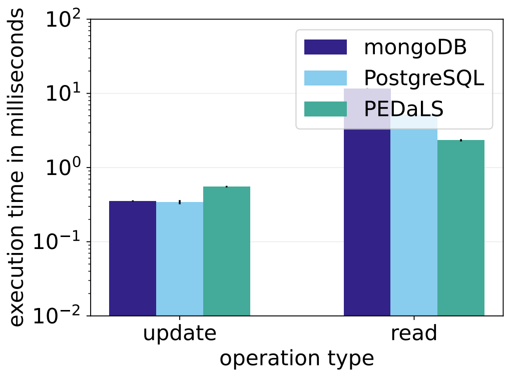
In [85], authors provide a queue dataset for three banks in Ogun State of Nigeria



(a) Banking queue.



(b) Room occupancy.



(c) Livestock tracking.

Figure 3.9: The mean update and query execution time for different end-to-end applications as presented in Table 3.2. As the execution time varies in order of magnitude among the different applications, we present the execution time in log scale for a better and uniform visualization.

```sql
SELECT userid
FROM bankingqueue
WHERE ts <= X
AND op = 1
AND userid NOT IN (
SELECT userid
FROM bankingqueue
WHERE ts <= X
AND op = 0)
```

```
db.bankingqueue.find({
'ts': {'$lte': X},
'op': 1,
'userid': {
'$nin': db.bankingqueue.distinct(
'userid',
{'ts': {'$lte': X}, 'op': 0}
)}},
{ 'userid': 1, '_id': 0 })
```

Figure 3.10: SQL (left) and NoSQL (right) queries to find the set of users in the queue at time X. Here *ts* stands for timestamp and *op* stands for enqueue/dequeue operation. Note that *distinct* is not necessary for the NoSQL query to get the correct result as a user is issued a unique ID during each entrance to the queue. However, it is faster than the alternative which involves converting the subquery result into an array and subsequently mapping it to a function that picks out each user ID in the array from the JSON object enclosing it. The *distinct* function extracts the value from the JSON object automatically and presents an array readily usable with the not in (*nin*) operator.

collected over four weeks for each bank. The dataset contains the wall-clock time when a user enters the bank and the time in minutes for the user to reach the front of the queue. The dataset assigns a monotonically increasing integer as a user ID to every user entering the bank each day (starting with 1) but does not contain the information whether a user ID x of one day corresponds to the same person having user ID y on some other day. Hence, for the purpose of this experiment, we assume every user is assigned a new ID upon each entry. We select the data for one bank and augment it by calculating the timestamp of departure by adding the wait time in queue to the timestamp of arrival. Thus each data point contains three information: a user id, a timestamp, and an operation (enqueue or dequeue). This information is recorded in a storage system every time a user enters and leaves the queue. The final dataset contains 35534 data points.

We assume we want to answer the query Q_1: "which users are present in the queue at time X"? This is an example of a temporal query that can be efficiently answered using a versioned queue. Although the storage of the data is simple for both of the databases under consideration (i.e., a single insertion for both PostgreSQL and MongoDB) and PEDaLS queue (i.e., a single enqueue or dequeue operation with the timestamp as the version and user ID as the value), answering Q_1 requires complex queries on part of the databases as shown in Figure 3.10. On the other hand, PEDaLS only requires a versioned traversal of the queue to answer Q_1.

Our results show that the average update times for PostgreSQL, MongoDB, and PEDaLS are 0.335ms, 0.357ms, and 0.092ms respectively. That is, PEDaLS is 3.64x as fast as PostgreSQL and 3.88x as fast as MongoDB in regard to updates for the application under consideration. Our results further reveal that the average query time for PostgreSQL, MongoDB, and PEDaLS are 8.734ms, 45.184ms, and 1.078ms respectively. That is, PEDaLS is 8.10x as fast as PostgreSQL and 41.91x as fast as MongoDB in regard to queries for the application under consideration.

Several works in the literature suggest that NoSQL databases can outperform SQL databases in the case of a high volume of unstructured data [86, 87]. SQL databases need complex design and multiple writes at different tables for unstructured data, making the updates slower. On a similar note, NoSQL databases can store denormalized data resulting in many cases in a simple single query to retrieve relevant information. However, the data under consideration is structured and the query, although complex, does not involve reading from multiple tables. Hence we do not observe the advantages generally associated with NoSQL databases over SQL databases for the application under consideration. This is supported by the similar update time for both PostgreSQL and MongoDB. However, the query time for MongoDB is higher than PostgreSQL. This is expected, as the MongoDB *aggregation pipeline* [88] (i.e., the pipeline responsible for processing documents in stages such as matching, grouping, projection, etc.) is still relatively immature and thus is not as well performant as the query engine of PostgreSQL. In fact, the aggregation pipeline is not as expressive as SQL queries either, which is ap-parent from the complex query of Figure 3.10. PEDaLS on the other hand is the fastest for both update and query. During update, PEDaLS must either add a node to the end of the queue or delete a node from the head of the queue. As the queue records both the head and the tail, both of the above operations can be performed efficiently. Moreover, none of these operations result in copying of a node, as a node can have at most one child which can be deleted at most once. As a result, the update performance of PEDaLS is better than both PostgreSQL and MongoDB. To answer Q_1, PEDaLS must perform versioned traversal of the queue, which is inherently fast due to the node-copy method. In contrast, both PostgreSQL and MongoDB has to perform complex queries (cf. Figure 3.10). Hence PEDaLS outperforms the others in regard to read as well.

Room Occupancy Detection

Indoor positioning systems are used to locate, track, and identify individuals/objects in indoor settings where technologies like GPS cannot perform with desired precision. A common application of such a system is to detect the occupancy of a room at different times of the day. Apart from eliciting information pertinent to security (e.g., was user x present in room y during some event z), it can also provide historical data to aid in scheduling of events depending on the occupancy of different rooms throughout different times of the day.

SmartBench [89] is a benchmark focusing on queries resulting from (near) real-time applications and longer-term analysis of IoT data. For this experiment, we use the data generation tool of SmartBench to generate room occupancy dataset based on seed data collected from a real system. The generated dataset contains periodic location data including the room in which a user is. However, the data does not contain the explicit timestamp at which a user leaves a room. Hence, we augment the data with information that denote a user has left the old room by inserting a new record with a timestamp

```
SELECT room
FROM roomoccupancy
WHERE ts <= X AND
userid = uid
ORDER BY ts
DESC LIMIT 1
```

```
db.roomoccupancy.find(
{'ts': { '$lte': X }, 'userid': uid },
{ 'room': 1, '_id': 0 }
).sort('ts', -1).limit(1)
```

Figure 3.11: SQL (left) and NoSQL (right) queries to find the room at which user with ID *uid* is at time X. Here ts stands for timestamp.

that falls between the timestamp when the user was last seen in the old room and the timestamp when the user was first seen in the new room. We assign an invalid room number for such records, which essentially denotes the user is on its way from the old room to the new room. The preprocessed dataset contains 4002 timestamped data points with information regarding the location (i.e., room numbers) of users at different times of the day.

The storage of this data for databases involves the insertion of a single row containing the timestamp, user ID, and the location of the user at that timestamp. For PEDaLS, we maintain a bst with user IDs as the keys and room numbers as the values. We treat the timestamp as the version stamp for the bst. A record in the dataset containing an invalid room number corresponds to a deletion from the bst and an insertion into the bst otherwise. We assume we want to answer the query Q_2: "what is the location of user with ID *uid* at time X"? PEDaLS bst performs a simple versioned search with X as the timestamp and *uid* as the key to answer this query. However, PostgreSQL and MongoDB require complex queries as shown in Figure 3.11.

Our results show that the average query time for PostgreSQL, MongoDB, and PEDaLS are 0.460ms, 2.255ms, and 0.124ms respectively. That is, PEDaLS is 3.71x as fast as PostgreSQL and 18.19x as fast as MongoDB in regard to queries for the application under consideration. As in the banking queue application of Section 3.4.5, MongoDB does not demonstrate any advantage over PostgreSQL for structured data of the application under consideration. PEDaLS must perform a versioned search of a key in the bst to answer Q_2. This is relatively simpler than the complex queries required by PostgreSQL and MongoDB as presented in Figure 3.11. Hence PEDaLS outperforms both PostgreSQL and MongoDB in regard to read for this application as well. Our results also indicate that both PostgreSQL (0.354ms) and MongoDB (0.359ms) perform better than PEDaLS (0.468ms) for update operations (approximately 1.32x times). This is expected, as the databases need only to insert a single entry to record an update, whereas PEDaLS has to search the tree for the appropriate position of a new node to be inserted or an old node to be deleted. In contrast, PEDaLS queue for Section 3.4.5 did not require an extra read per update operation to find the existence and/or position of a data structure node. This explains the superior performance of PEDaLS update in the banking queue monitoring application.

Livestock Tracking

Livestock tracking is a popular application of IoT in smart farms to study animal behavior and animal-ecosystem interaction [90, 91]. In a livestock tracking system, each animal is fitted with a tracking device (e.g., a GPS collar) that records the location information at a pre-defined regular interval. This information can then be used to analyze different behavior of the cattle, such as grazing patterns. As the tracking and data processing devices are typically battery-powered in these deployments, we must minimize the number of times a location is recorded/communicated over the network; without losing valuable information.

In [90], the authors provide a dataset containing daytime (approx. 8 hours per day) grazing locations of cattle collected from 6 Alpine farms in the summer of 2011. 2 to 4 cows from each of these farms (in total 15) were equipped with a GPS collar and a logger box. Although data were collected at 20 seconds interval in this work, an analysis of the

Table 3.3: Average distance between the locations at the start and the end of an interval for an individual cow.

interval (seconds)	**distance (meters)**
20	1.62
60	4.31
100	6.81
140	9.22
180	11.56

dataset shows that even a larger interval of 3 minutes shows an average movement (i.e. distance between the locations at the beginning and at the end of the interval) of 12 meters for a cow (cf. Table 3.3). Depending on the use case, this might be an acceptable distance.

We consider a scenario where a new patch of land B is opened up for grazing beside an old patch of land A. We want to know which are the cows that prefer the new patch of land B, along with at what time of the day. Specifically, we want to answer the query Q_3: "which cows are present in B at time X"? This is an example of a temporal query that can be efficiently answered using a versioned bst. As the dataset in [90] contains data for at most 4 cows from any one farm, we use synthetic data generated using random walk for 100 cows. The one-dimensional random walk starts from 0. Whenever the random variable has a value >= 0, we assume the location is in land B, otherwise land A. As in [90], we assume each cow is equipped with a GPS collar.

From Table 3.3, we see that the point-to-point distance covered by a cow on average is approximately only 12 meters for a 3 minute interval. Hence, for energy-efficiency we assume the GPS collar collects location information at 3 minutes (180 seconds) interval instead of 20 seconds over the duration of 8 hours ($8 * 60/3 = 160$ data points for each cow). We further assume the timestamped location data is sent to a nearby edge server for storage in two methods. In the first method, as common to many IoT deployments, the server stores each location data into a database upon reception. Each data point

```
SELECT lt1.cattle_id
FROM livestocktracking AS lt1
JOIN
(SELECT cattle_id, MAX(ts) as
maxts
FROM livestocktracking
WHERE ts <= X
GROUP BY cattle_id) AS lt2
ON lt1.cattle_id = lt2.cattle_id
AND lt1.ts = lt2.maxts
WHERE lt1.pos >= 0
```

```
db.livestocktracking.aggregate([
{'$match': {'ts': {'$lte': X}}},
{'$sort': {'ts': -1}},
{'$group':{
'_id': '$cattle_id',
'maxTS': {'$max': '$ts'},
'currentPos': {'$first': '$pos'}
}},
{'$match': {'currentPos': {'$gte': 0}}},
{'$project': { '_id': 1}}
])
```

Figure 3.12: SQL (left) and NoSQL (right) queries to find the set of cows present in plot B at time X. Here ts and pos stand for timestamp and position respectively.

contains a cattle id and its position at the time the data was collected. Note that in this kind of setup, the collection of data is fast as it involves only a single insertion to a database table. However, retrieving answers to complex queries such as Q_3 can be time-consuming. We present the SQL and NoSQL queries to answer Q_3 in Figure 3.12.

In the second method, the server maintains a PEDaLS bst to store the IDs of cows that are in land B using timestamps as the version stamps. Although this requires some preprocessing during insertion to determine whether a cow is already in land B, this makes query Q_3 faster compared to the former method; as we can now perform a tree traversal at version X in the bst to retrieve the full set of cows present in land B. This also removes the onus on part of the developer to write complex queries, as the relevant information is readily available through a versioned access operation (in this case, tree traversal) in PEDaLS bst.

As described above, there are 160 data points for each cow, i.e., 160 unique timestamps. We perform $160 * 100$ insertions and 160 queries (one for each unique timestamp) in one iteration of the experiment. The average update times for PEDaLS and PostgreSQL are 0.553ms and 0.342ms respectively, whereas the average query times are 2.333ms and 5.186ms respectively. That is, although for insertion PostgreSQL is 1.62x as fast as PEDaLS, for query PEDaLS is 2.22x as fast as PostgreSQL. From a power consumption perspective, as long as the number of updates to the number of queries ratio is less than $(5.186 - 2.333)/(0.553 - 0.342) = 13.52$, PEDaLS is more efficient than PostgreSQL. The higher time requirement for PEDaLS update is expected, as every update operation involves searching for a node in the bst as well. On the other hand, the databases only have to insert a single row for an update. However, due to the efficient versioning scheme of PEDaLS bst, answering query Q_3 simply amounts to traversing a particular version of the bst. In contrast, the databases have to perform complex queries as presented in Figure 3.12. Hence PEDaLS outperforms both PostgreSQL and MongoDB in regard to read. Although the update time (0.355ms) for MongoDB is comparable to that of PostgreSQL, much like Section 3.4.5 and Section 3.4.5, MongoDB requires significantly more time for reads (11.687ms). This is again due to the inefficient aggregation pipeline of MongoDB.

3.4.6 Bounding Log Size

One challenge with PEDaLS is that the underlying append-only, persistent backing storage abstraction may have a size limit due to host resource constraints or storage service design. For example, CSPOT implements logs of fixed size as a circular buffer [3]. Although this approach provides fast and automatic garbage collection of log entries, if a PEDaLS log "wraps" it may lose entries that it requires to maintain a particular version in the version history. A PEDaLS user specifies the maximum number of data structure versions (i.e. the version history) she wishes to maintain. PEDaLS ensures that at least this length of version history is available.

To do so, PEDaLS provides a probabilistic guarantee that a program never accesses a version for which only partial information is available due to log wrap. It uses the probability that it must maintain a given history length to compute the log sizes that are necessary to ensure that this guarantee is met. In the case where this size is attempted to be exceeded due to an update operation performing a log append, PEDaLS refuses further update operations. This practice of refusing update requests in absence of adequate space is not uncommon (e.g. Redis [81]). However, read requests can still be serviced.

Our failure model assumes that storage failures occur only during log-append operations and these failures take two forms. Either the data is not written at all, or it is written but the sequence number associated with the write is not returned. In both cases, PEDaLS assumes that the failure can occur silently and thus must be remediated by a timeout and a retry. As stated in Section 3.3.2, we term the first type of failure as a *Type 1* failure and the second a *Type 2*.

As node copy is most frequent when the number of extra pointers e is one, we consider the required log size for the different logs when $e = 1$. Then for any case where $e > 1$, the required log sizes will not be greater than the ones calculated for $e = 1$. We begin by determining the log sizes assuming a fault-free system and then consider a distributed system with failures. Let K be the maximum number of versions in the version history specified by the PEDaLS user, K_i the number of inserts, and K_d the number of deletions ($K = K_i + K_d$). The APLog which tracks all operations (each one creating a version) thus requires K entries while the DataLog only grows as a result of insert operations and thus it must minimally contain K_i entries. However, worst case, all of the operations are inserts so the DataLog must contain K entries.

We consider the required size of the LinkLog separately for linked list and BST. In the case of linked list insert or delete, PEDaLS requires one new link. As linked list has only one type of pointer, even if PEDaLS must copy a node, it only introduces one new link. Therefore, we require the log size for the LinkLog for linked list to be at most K. Even in the worst-case scenario when we are inserting to and deleting from the end of the same node K times, a LinkLog of size K will not experience rollover and wrap.

In the case of BST insert, PEDaLS adds one new link to a node. However, this might result in node copying. Unlike linked list, for BST, PEDaLS must copy both pointers in a node during node copy. That is, if we are changing the left (or right) pointer, PEDaLS must copy over the old right (or left) pointer apart from introducing the new left (or right) pointer. Thus every second modification of a node forces it to make one extra entry into the LinkLog. Aside from the fact that the first version of a node requires two entries in the LinkLog, the number of links in the LinkLog is at most 1.67 times that of the number of versions K when all operations are inserts. In practice, this factor is much lower. As an example, over 100 iterations of 12579 insertions (different order for each iteration) for the CityPulse temperature dataset used in Section 3.4.3 revealed the maximum links required by any LinkLog is only 26, i.e. 0.002 times of $K(12579)$.

However, in the case of BST delete, PEDaLS might have to introduce at most two entries to the same node when the deleted node has two children. If node copying is necessary (the additional node pointer is occupied), the total number of new links to be introduced is three. Therefore, assuming the worst-case scenario where every operation is a deletion and where the target node has both of its children present, we can say that the number of entries at each LinkLog will not exceed $3K$. Thus PEDaLS can preserve K versions of BST when the LinkLog is set to $3K$ in the absence of Type 2 log failures.

Now that we have set a baseline for the size of the history of each log in absence of failures, we next consider how this changes with the introduction of our failure modes. We are specifically concerned about Type 2 failures where an entry is appended but the resulting log sequence number is lost (since a Type 1 failure does not grow any log).

Let us assume that a log append fails with probability q and that the probability of any such failure is independent of any other. Given that we need to insert i entries into a log ($i = K$ for APLog, $i = K$ for DataLog, $i = K$ for linked list LinkLog, and $i = 3K$ for BST LinkLog) and we must tolerate at most F failures (i.e. F additional entries are needed in each log), the total number appends that we can make is $i + F$. As an append either fails or succeeds, we can model the probability of failure of f failures in at most $i + F$ appends as a binomial distribution having a probability mass function λ parameterized by the failures f:

$$\lambda(f) = \binom{i+F}{f} q^f (1-q)^{i+F-f} \tag{3.1}$$

Given that we set the log size to $i + F$, rollover happens only when $f > F$. Thus, the probability of rollover is 1.0 minus the probability that $f \leq F$ expressed as:

$$P_{rollover} = 1 - \sum_{f=0}^{f=F} \lambda(f) \tag{3.2}$$

Given probability q that any append fails and a history version size i, we can programmatically find the number of additional entries F (in turn the size of a log) required to make the probability of rollover arbitrarily low. As a worst-case example, for an append failure probability $q = 0.5$ and a version history of $K = 1000$ versions, the APLog and DataLog for BST would need to contain 2223 entries each and the LinkLog would require 6379 entries to make the rollover probability for any of these logs < 0.0000001. That is, with a failure probability of 0.5 the APLog, DataLog, and LinkLog are approximately 2.2 times as large as they would be without failures.

This overhead decreases with failure probability. For example, when the append failure probability is $q = 0.1$ this overhead reduces to a factor of 1.2 compared to the failure-free case, and when $q = 0.001$ the overhead factor is 1.009 (9 extra entries in APLog and DataLog and 14 extra entries in the LinkLog). As a practical matter, the failure probability q for any given append is almost certainly well below 0.001 (e.g. we observed no failures in tens of millions of distributed logging events over the several

Table 3.4: Required Log size to keep the probability of roll over below 0.000001. i denotes the required number of successful appends, F denotes the maximum allowable number of unsuccessful appends, and q denotes the probability of failure of an append. The overhead is calculated as $\frac{i+F}{i}$.

i	q	$i+F$	overhead
1000	0.1	1168	1.168
	0.2	1339	1.339
	0.3	1553	1.553
	0.4	1834	1.834
	0.5	2223	2.223
2000	0.1	2301	1.151
	0.2	2624	1.312
	0.3	3030	1.515
	0.4	3566	1.783
	0.5	4311	2.155
3000	0.1	3429	1.143
	0.2	3901	1.300
	0.3	4496	1.499
	0.4	5283	1.761
	0.5	6379	2.126

months [92] was in preparation) making the additional space required to tolerate failures in a real-world setting negligible. Table 3.4 shows the log size required to keep the probability of rollover below 0.000001 for different values of q.

3.5 Summary

Both partially persistent data structures (PDSs) and append-only storage systems provide immutability and history-based programming – albeit at different "levels" (program versus systems). These features are useful at both levels in distributed, large-scale, and failure-prone contexts such as those for heterogeneous and pervasive Internet of Things (IoT) deployments. In this chapter, we investigate how to combine the two so that high-level, linked program data structure operations with versioning support, automatically and transparently map to append-only persistent storage – enabling, for the first time, survivability and programmatic access by distributed clients to both the data structures and their version histories.

To enable this, we present a new approach for efficiently supporting versioned, linked data structures in programs by leveraging algorithmic advances from partially persistent data structures. We use these methods to design a mapping and library implementation of version-aware data structure operations that are backed by append-only storage. We implement this approach using an append-only storage abstraction from a portable, open-source event system for IoT. We use this system to evaluate the algorithmic complexities and performance overhead for operation workloads for linked list and binary search tree (BST) structures as well as end-to-end using multiple different applications. Our results show that we are able to achieve the algorithmic complexities of the original PDSs and low overhead for storage-persistent versioning.

Chapter 4

LSCRDT: Log-Based CRDT for Multi-Tier IoT

The present cloud computing paradigm is becoming increasingly ubiquitous and often employs a multi-tier (cloud/edge/device) architecture. Many modern conventional cloud deployments not only require efficient computation and storage solely on the cloud tier, but also seamless integration of computational/storage concepts on edge and device tier. A prime example of these are edge applications, where the edge servers must communicate with high-end cloud servers while overseeing Internet of Things (IoT) devices at the same time. Unfortunately, the same technologies used to guarantee failure resilience and robustness on the cloud do not translate directly to the edge/device layer due to resource-constraints and unstable network connectivity. Thus, new programming support and data abstractions are needed to manage this complexity and automatically enhance the robustness and efficiency of multi-scale cloud deployments.

A common approach to providing fault tolerance and robustness to distributed systems is introducing data replication. When we replicate data to multiple nodes, we can access the data from other nodes even when some fail. Replication also facilitates data availability by allowing us to retrieve data from the nearest or most updated functioning node. However, traditional replication protocols are often unsuitable for IoT deployments due to unstable network connectivity, unreliable power source, and resource-constrained devices. Hence we must devise and employ new protocols more suited for IoT applications. The previous chapter discussed how data versioning can make IoT applications reliable and fault-tolerant. In this chapter, we introduce data replication for IoT applications and explain how versioning and replication can work together to make distributed IoT systems reliable and fault-tolerant. We propose a replication protocol that inherently records versioning information during replication.

In this chapter, we introduce Log-Structured CRDTs (LSCRDTs) – a new form of *C*onflict-Free *R*eplicated *D*ata *T*ypes [9, 10, 11] that is better suited for use in multi-tier cloud deployments than its antecedents. In general, CRDTs enable program data types to be distributed, shared (concurrently modified), replicated, and made consistent (via clever update merging). Specifically, CRDT replicas employ Strong Eventual Consistency (SEC) [9], which guarantees that two replicas reach the same state if they receive the same set of updates (possibly in different orders). These features facilitate robustness, high availability, and coordination avoidance [1], which makes them suitable for failure prone settings.

However, many existing CRDT designs impose limitations that make them difficult to use in multi-tier cloud deployments. In particular, previous CRDT formulations do not support *operation reversal* – the process of reverting a data type to a previous state for operations that are not executed.[1] They do not easily support non-commutative operations, and (for operation-based CRDTs) they cannot tolerate out of order operation delivery.

LSCRDT uses distributed logs to overcome these limitations. As logs are append-

[1]Note that operation reversal is distinct from the *undo* operation [12], which may have side effects.

only, our approach is able to use them to further reduce the coordination required to merge inconsistent replicas. LSCRDT guarantees that replicas execute operations in the same order, irrespective of data type thus removing the need for commutability. LSCRDTs also avoid the complex networking required to guarantee exactly-once, causal delivery of operations required by operation-based CRDTs. Finally, LSCRDT provides programmatic access to data structure versions, which is useful for debugging, history-based programming [13] and data replay and repair [14].

In this chapter, we describe the design and implementation of LSCRDT and provide a comparative study of LSCRDT and δ-CRDT [93] – a popular category of CRDT that combines the advantages of a wide range of CRDT implementations. We evaluate latency and throughput using three, extensively studied, CRDT data types – register, counter, and set [94, 95, 96, 97, 98]. Our results show that, apart from providing the aforementioned properties, LSCRDT outperforms δ-CRDT in terms of write latency. Moreover, for update-heavy workloads (typical of sensor-driven edge applications), LSCRDT exhibits up to $1.8x$ higher throughput than δ-CRDT.

4.1 Related Work

Conflict-Free Replicated Data Types (CRDTs) are abstract data types that provide a principled approach for the asynchronous reconciliation of divergent data resulting from concurrent updates [9]. CRDTs provide strong eventual consistency (SEC), which guarantees that whenever two replicas receive the same set of updates, they reach the same state. Broadly, there are two types of CRDTs: *state-based* and *operation-based* (or op-based) [9]. In state-based CRDTs, an operation is executed on the local replica state. A replica periodically propagates its state to other replicas to achieve consistency. A disadvantage of this approach is the communication overhead associated with shipping the full state, which can be large. In op-based CRDTs, an operation is executed on the local replica and the operation is asynchronously propagated to other replicas. Although operation-based CRDTs do not communicate state, they require exactly-once causal broadcast. Delta State CRDTs (δ-CRDTs) [93] combine the advantages of state-based and op-based CRDTs. Like state-based, δ-CRDTs can tolerate unreliable networks and, in particular, do not require exactly-once causal broadcast as a communication network property. However, like the op-based approach, they do not require that the full replica state be communicated, but rather, they communicate only state changes or "deltas".

Most CRDT studies are evaluated using the foundational data types, register, set, and counter [94, 95, 96, 97, 98, 99, 100]. CRDTs for JSON data [101] allow programmers to create custom data types by nesting maps and lists into new CRDTs. Although this approach adds a new dimension to the foundational data types, it is still insufficiently generic (e.g., we can not create a self-balancing tree with JSON). Other works discuss replicated trees capable of executing arbitrary concurrent *move* operations [102], where a log of move operations is maintained, possibly involving rollback and replay. As we describe in Section 4.5, LSCRDT uses a similar approach. Strong eventually consistent replicated objects (SECROs) [103] are general-purpose data types that implements SEC without placing any restriction on operations, i.e., operations can be non-commutative. SECROs find a conflict-free ordering of concurrent operations depending on application-specific information. However, to ensure that the application-specific conditions are not violated, SECROs at times must give up on a computation path and backtrack to a previous state, thus imposing significant computational load.

4.2 Motivation

In this section, we motivate the integration of logs and general purpose, operation-based CRDTs for use at the edge. Many edge applications do not require strong consistency (e.g., smart locks [40], energy management [41], temperature prediction [42], etc.) and can choose a weaker consistency model in favor of increased availability, lower coordination overhead, and better energy efficiency. Due to their ability to provide these advantages, CRDTs are suitable for many edge deployments. Although CRDTs are not new, they have been employed mostly for collaborative editing [104, 105, 106, 107, 108]. Edge deployments impose new environmental characteristics that past research does not fully address. Specifically, edge applications often coordinate among devices deployed in an environment with poor and unreliable network connectivity. They consist of heterogeneous, resource-constrained devices (with regards to computation, space, and power). Moreover, data-driven applications also demand data resilience, availability, and the ability to audit data lineage. Finally, the ubiquity of edge applications makes a generalized abstraction approach to storing and manipulating data desirable [109].

Past works attempt to address different aspects of these challenges, but none provide a holistic approach within a single system. For example, δ-CRDT can tolerate unreliable networking but does not provide a general approach to merge/join deltas (e.g., joining deltas for counters is different than that for sets). Moreover, in many cases, δ-CRDTs must resort to using a variant of the data type rather than the original one (e.g., using a 2P set instead of a conventional set).

LSCRDT can both tolerate unreliable networking and manage arbitrary data type without the need for data type-specific join algorithm. In LSCRDT, each replica logs the executed operation along with a unique *version stamp* (allowing detection of duplicates) in causal order. Each replica periodically reads from other replicas' logs in log order (e.g.

similar to what is done in Kafka [59] logs: each replica reads monotonically increasing offsets). LSCRDT maintains causal order by ensuring log order is maintained during reads.

Logs also provide immutability, which can be used to aid data lineage tracking and versioning [92], data availability and robustness, and application debugging. For example, efficient versioning is critical for temporal queries (typical in stream-based edge applications), and for operation reversal and rollback, and replay of applications and analyses, in the face of errors and malfunctioning sensors and services (which are common occurrences in large scale multi-tier IoT deployments) [14]. Moreover, logs aid interoperability of heterogeneous devices via the use of a simple interface.

Finally, our LSCRDT design is operation based to facilitate generality (i.e. to be agnostic to the data structure type and operation implementation). Many CRDTs require *type-specific* merge/join algorithms [10]. LSCRDT instead forms a consistent order of operations, irrespective of the underlying data type. As we will see in Section 4.5, agreeing upon a consistent order of operation is not type-specific in LSCRDT, making it more generic and extensible than existing approaches.

4.3 System Model and Overview

We consider a distributed system of N replicas. Each replica is assigned a node ID from a set S. We represent a replica as $X_s, s \in S$. The underlying network is asynchronous and unreliable; messages may be dropped, duplicated, or reordered. The network may partition and eventually recover. Each replica has local durable storage. We assume replicas may face non-byzantine failures; a replica may crash but will have access to the information recorded in durable storage upon recovery.

Over time replicas may diverge from each other due to update requests from clients (i.e., processes that can mutate or query a data type by sending requests to any replica). To reconcile this divergence each replica periodically performs a round of *merge steps* with the other replicas. A merge step is always between a pair of replicas. Therefore, in a round, there are at most $N - 1$ merge steps. In a merge step, one replica (known as the *reader*) reads entries in the log of operation from another replica (known as the *source*). The goal of the merge step is for the reader to identify and incorporate operations "unknown" to it (i.e. not previously executed at the reader) that the source has already executed. The reader ensures that the causality relationship among the operations is retained while creating this merged list of operations and subsequently executing them. We present the details of the LSCRDT merge step in Section 4.5. Note that during a merge step, a reader may have to rollback some operations and re-execute them along with new operations. As long as the replicas execute operations in the same log order, replicas will converge irrespective of operation commutativity, similar to the replicated state machine concept used in Raft [34].

Although our approach requires log rollback to maintain the order of operations, it does not require any locking mechanism among replicas to agree upon a unique order of operation execution. This order can be determined from the timestamped operations of the source as described in Section 4.5. Any partial order formed during merge steps maintains the causal order observed within the total order. Rollbacks provide two capabilities to LSCRDTs: (i) implementing arbitrary non-commutative data types and (ii) maintaining a global version history consistent among all replicas. Note that as an optimization (not explored in this work) if the underlying data type is commutative and a global version history is not needed LSCRDT can store state deltas (like δ-CRDTs) and, thus, remove rollbacks (and their performance impact) altogether.

Various algorithms have been proposed to maintain order in list or sequence CRDTs such as Logoot [104], LSEQ [105], RGA [106], Treedoc [110], and WOOT [111]. Our method to maintain order among logs of operations is an adaptation of [101], which is based on RGA [106]. This approach provides us a uniform way to create a replicated data type irrespective of the operations supported by it; once we establish a common order of operations among all the replicas our system will converge. Although RGA-based approach has been used in other data types such as JSON [101], the use of logs introduces a new performance challenge – avoiding log scans. We explain how we can avoid full log scans in Section 4.5.

4.4 Data Type Using Logs

We next explain how LSCRDTs are stored using logs. We overview the replication process in Section 4.5. We choose three data types widely studied in CRDT literature for this exposition: registers, counters, and sets [94, 95, 96, 97, 98]. Our approach is agnostic of the underlying log storage system. We assume entries in a log can be addressed by monotonically increasing *sequence numbers* (e.g. *offset*s in the case of Kafka [59], *LSN*s in the case of Facebook LogDevice [60], and *sequence number*s in the case of CSPOT [3]). We further assume the log storage system exposes functionalities (i) to create logs with a given name, (ii) to write to a specified log and get the sequence number corresponding to the write on success, (iii) to read from a specified log at a given sequence number, (iv) to retrieve the latest sequence number of a specified log, and (v) to trim the log up to a specified sequence number i.e. all entries with greater sequence numbers are removed. As long as these criteria are met, we can use any log storage system.

LSCRDTs tag each operation performed on a data type with a *version stamp* (Lamport timestamp [112]), which is a concatenation of a counter and a node ID drawn from S. We represent the counter and node ID of a version stamp vs as $vs.counter$ and $vs.nodeID$, respectively. We say version stamp vs_a is less than version stamp vs_b ($vs_a < vs_b$) if (i) the counter of vs_a is less than that of vs_b, or (ii) both the counters are same but node ID of vs_a is less than that of vs_b. When replica X_s executes a new operation in response to a client request, it tags it with version stamp vs ($vs.nodeID = s$), which is greater than all other version stamps it has observed so far (operations that *happened before*). Thus if operation op_a happens before op_b, $vs_a < vs_b$ where vs_a and vs_b are the version stamps of operations op_a and op_b, respectively.

Version stamps of concurrent operations can be ordered arbitrarily but deterministically. Throughout the rest of the chapter, we use version stamps to refer both to the version stamp itself and to the operation it tags. The intended use will be clear from the context. We say vs is an operation of X_s (alternatively, X_s is the originator of vs) if $vs.nodeID = s$.

4.4.1 Register

The register data type maintains a single value (e.g. an integer, an object, etc.). It supports two operations, *assign* to set a value and *retrieve* to get a value. We introduce *OpLog* to store all the update operations, in this case, assigns. There is one OpLog per replica. We represent the OpLog of the replica X_s as $OpLog(X_s)$. As all the update operations are recorded in the OpLog, a data type can be reconstructed up to a certain version if required. Replicas can read each other's OpLogs to create a merged list of all the update operations. As the retrieve operations do not update the register, we do not have to record those. Each entry in an OpLog is the tuple *(vs,op,val)*. *vs* is the version stamp of the operation, *op* is the type of update operation (in case of register there is only one, i.e., assign), and *val* is the operand of *op*.

To execute an assign operation a replica writes the appropriate entry to its OpLog. For example, suppose a request to assign the value 5 to a register comes to X_A. We further assume the greatest counter value among all the version stamps X_A has seen so far is 2. Then X_A writes the entry $(3A, assign, 5)$ to its OpLog to execute the operation. To respond to a retrieve request, X_A simply returns the *val* field of the last entry in its OpLog. If some previous version stamp vs_i is supplied as an argument of value, the replica can search the OpLog for the entry with the *vs* field equal to vs_i.

To make this search efficient, LSCRDT maintains an in-memory map from version stamps to sequence numbers in OpLog. This approach has been used in other log-based systems as well, such as Riak Bitcask [113]. Note that this is an optimization and not necessary for the correctness of the system. This in-memory map is populated at startup and can be reconstructed at any time. Any update to the register is first appended to the log and then a map entry is created.

4.4.2 Counter

The counter data type supports increment (*inc*), decrement (*dec*), and *retrieve* operations. Like register, a counter also has an OpLog. However, it maintains one additional field per entry for the cumulative sum to avoid log scans while computing the counter value corresponding to a version. For example, assuming the initial value of a counter is 0, if replica A first increments the counter by 5, next decrements the counter by 2, and finally increments it by 1, the entries of the OpLog will be $(1A, inc, 5, 5)$, $(2A, dec, 2, 3)$, and $(3A, inc, 1, 4)$. To find the latest value of the counter, the replica can now return the value in the last field of the last entry in the OpLog. Similar to register, an in-memory map can expedite the response to a retrieve request corresponding to an earlier version.

4.4.3 Set

Our set data type supports *add* and *remove* update operations and *in* and *all* read operations. Note that although CRDTs resort to using some variant of sets such as two-phase set (2P-set [9]), grow-only set (G-set [10]) etc; LSCRDT set works like a conventional set due to its capability to find a consistent ordering of non-commutative operations. The structure of OpLog of set is similar to that of register. However, set is different from register and counter in that each version is a collection of elements rather than a single value. Note that to reconstruct a set up to the latest version, we must scan the OpLog from the top. To avoid such expensive operations, we cache the elements in the set after every *cp_interval* number of operations using a second log. To make the search for the latest version fast, we also keep a copy of the latest operations that have not yet been checkpointed in memory. This in-memory list of operations is purged every time we checkpoint our progress. Therefore, we have at most $cp\_interval - 1$ operations cached in memory at a time (which users can set). Setting the *cp_interval* to a low value makes queries faster but uses more space. Doing so may also decrease write throughput due to more frequent checkpointing.

Note that to query a previous version of the set, we might need to access OpLog. For example, suppose *cp_interval* is set to 100, we have already executed 400 operations and we want to query the 257th version of the set. In this case, we first need to read the entry that was checkpointed after the 200th operation and then read the 57 following operations from the OpLog to reconstruct the desired version of the set. A version of a data type might change due to updates from other replicas (cf. Section 4.5). In that case, the checkpoint that contains that version and the following checkpoints must be overwritten.

4.5 Merge Step

Many data types have operations that do not commute (e.g., add and remove of the same element in a set). To achieve a consistent state for *replicated* data types, we must impose a total order on the execution of operations [103]. An alternative to imposing a total order for arbitrary non-commutative data types is to switch between stronger and weaker form of consistency [114, 115, 116, 117]. However, as discussed in Section 4.3, a total order also helps us maintain a consistent version history among all replicas. In our work, we model the history of operations (OpLog) as a list CRDT and use an adaptation of the method used in [101] which is based on RGA [106] for maintaining order in the list i.e. the OpLog, as explained next.

When a replica X_A executes an operation as a direct request from a client, it appends the operation at the end of $OpLog(X_A)$. Apart from direct client requests, replicas also execute operations that are unknown to them from other replicas' OpLogs. Assume vs_{new} is an operation in $OpLog(X_B)$ that X_A has not yet executed. We denote the operation immediately preceding vs_{new} in $OpLog(X_B)$ as vs_{pred}. As the intention is to maintain a consistent order of operations, X_A tries to place vs_{new} in its own OpLog after vs_{pred} as well. Therefore, to incorporate the unknown operation vs_{new}, X_A first locates vs_{pred} in $OpLog(X_A)$. Let us denote the operation in $OpLog(X_A)$ immediately succeeding vs_{pred} as vs_{succ}. That is, vs_{new} and vs_{succ} are concurrent operations. Now X_A inserts vs_{new} in $OpLog(X_A)$ immediately after vs_{pred} if $vs_{new} > vs_{succ}$. Otherwise, X_A skips over all contiguous version stamps that are greater than vs_{new} and then places vs_{new}. Of course, it might happen that vs_{pred} is not present in X_A to begin with. In that case vs_{pred} must be inserted first. This implies that X_A should start reading $OpLog(X_B)$ from the earliest sequence number that contains an operation unknown to it. We express this whole procedure of inserting operation vs_{new} after vs_{pred} as $insert(vs_{new}, vs_{pred})$.

$OpLog(X_A)$	
seq	vs
1	1A
2	2A

$OpLog(X_B)$	
seq	vs
1	1A
2	2B

initial state

$OpLog(X_A)$	
seq	vs
1	1A
2	2B
3	2A

$OpLog(X_B)$	
seq	vs
1	1A
2	2B

after X_A syncs with X_B

$OpLog(X_A)$	
seq	vs
1	1A
2	2B
3	2A

$OpLog(X_B)$	
seq	vs
1	1A
2	2B
3	2A

after X_B syncs with X_A

(a) X_A merges with X_B then X_B merges with X_A.

$OpLog(X_A)$	
seq	vs
1	1A
2	2A

$OpLog(X_B)$	
seq	vs
1	1A
2	2B

initial state

$OpLog(X_A)$	
seq	vs
1	1A
2	2A

$OpLog(X_B)$	
seq	vs
1	1A
2	2B
3	2A

after X_B syncs with X_A

$OpLog(X_A)$	
seq	vs
1	1A
2	2B
3	2A

$OpLog(X_B)$	
seq	vs
1	1A
2	2B
3	2A

after X_A syncs with X_B

(b) X_B merges with X_A then X_A merges with X_B.

Figure 4.1: Change in OpLogs as replicas merge with each other. We notice the two different sequence of merge steps results in the same consistent state at the end.

To illustrate how *insert* works, we refer to the OpLogs in Figure 4.1 (only the sequence numbers and version stamps are shown for brevity). We consider two replicas in our system, X_A and X_B. Let us assume X_A executed operation $1A$ that X_B became aware of during the latter's merge step. At this point, both X_A and X_B executed one operation independently but concurrently, operation $2A$ and $2B$ respectively. Now we consider two different scenarios. (i) Figure 4.1a. X_A (reader) merges with X_B (source). For now, we assume readers start comparing the two OpLogs from the beginning (we show in Section 4.5.2 how full log scans can be avoided). Both OpLogs have $1A$ as the first entry, so no action is needed. However, X_B has $2B$ in the second entry whereas X_A has $2A$. This is equivalent to the insert operation $insert(2B, 1A)$ i.e. insert $2B$ after $1A$ in $OpLog(X_A)$ (as $2B$ comes after $1A$ in $OpLog(X_B)$). We note how the insertion operation is implicitly embedded in the log order. As X_A currently has $2A$ after $1A$ and $2B > 2A$, it can place $2B$ after $1A$. "Placing $2B$ after $1A$" is a multi-step process: X_A trims its OpLog up to sequence number 1, appends the entry containing $2B$, and finally re-appends the entry containing $2A$. Additionally, it trims/(re-)appends to any logs used by the underlying data type. When X_B (reader) merges with X_A (source) after this, X_B can simply append $2A$ after $2B$ in its OpLog. (ii) Figure 4.1b. X_B (reader) merges with X_A (source). Starting comparison from the top of the OpLogs as before reveals different entries in the second entry: $OpLog(X_A)$ has $2A$ as the second entry whereas $OpLog(X_B)$ has $2B$. This translates to the operation $insert(2A, 1A)$ to be executed in OpLog of X_B. As the version stamp after $1A$ at X_B is $2B$ and $2A < 2B$, $2A$ is placed after $2B$. Merging the other way follows the steps similar to the previous scenario. We see that in both scenarios we end up with the same final state in both the replicas.

Note that we could have forgone this relatively complex ordering following [101] and instead chosen a strict ascending order of version stamp counters, breaking ties through lexicographical order of node IDs. However, this approach would have resulted in interleaving sequences of operations made by different replicas concurrently. Our current choice of the method in [101] on the other hand makes sure concurrent sequence of operations executed by different replicas are not interleaved. Also note that to break tie between concurrent operations we choose the greater version stamp to take precedence over the smaller one (e.g. $2B$ appears before $2A$) to maintain similarity with existing work [101]. In practice, we could have chosen the reverse.

Proof of convergence of *insert* **:** To understand how *insert* converges, we note a few points. First, the order of an operation in an OpLog either remains unchanged or is pushed down, but never pulled up. This is due to how *insert* works: it either appends a new operation at the end, in which case there is no change in the order of operations; or it inserts an operation in between existing operations, effectively pushing all the operations that follow down by one. This also implies that the relative order of two operations in a log once set remains the same. Second, *insert* breaks tie between two concurrent operations arbitrarily but deterministically (the two concurrent operations are the new operation from the source and the current successor of the intended predecessor in the reader). To see this, let us consider two concurrent operations vs_a and vs_b. Without the loss of generality, let us assume $vs_a > vs_b$. Now if vs_a has already been executed (i.e. vs_a is the successor of the intended predecessor in reader), vs_b skips over vs_a. Therefore vs_a comes before vs_b. On the other hand, if vs_b has already been executed (i.e. vs_b is the successor of the intended predecessor in reader), vs_a can be placed in its place, effectively pushing down vs_b. Therefore, vs_a comes before vs_b in this case as well. Finally, a reader always reads the operations unknown to it from the source in monotonically increasing sequence numbers. This means even when a reader has not observed all the operations executed by the different replicas in the system, its OpLog contains a partial order of the total order formed by all the operations (due to the first and the second points). Hence once the replicas observe all the operations, the system achieves consistency.

In a merge step between a reader X_i and a source X_j, the reader performs two tasks: (i) *Conflict detection*: The reader detects whether it is in conflict with the source, i.e. whether the source has operations that the reader does not know of. Note that we are concerned with unidirectional conflict, i.e. if the reader has operations that are unknown to the source no extra steps are taken (this is resolved during some other merge step when the current source becomes a reader). (ii) *Conflict resolution*: In case of conflict, the reader resolves this conflict, possibly by reordering the operations which require rollback and replay of some operations. The conflict detection stage finds the sequence numbers of the two OpLogs from where the comparison should be started ($reader_{start}$ and $source_{start}$ for OpLog of the reader and the source respectively) to guarantee that the reader encounters all the operations it has not seen that have been already executed by the source. These two sequence numbers are used by the conflict resolution stage to incorporate all the unknown operations in the reader's OpLog.

We next introduce a new log that helps us to avoid full log scan (Section 4.5.1). We show how the conflict detection stage uses this log to detect the presence and point of conflict (Section 4.5.2). Section 4.5.3 then describes how the conflict resolution stage takes this information and uses *insert* operations to execute a list of ordered operations.

4.5.1 KnowledgeLogs

OpLogs grow over time and the merge steps become costly if we must scan from the top. To avoid a full scan of the OpLog of the source by the reader, a replica maintains a map of the last observed version stamp from each replica to a sequence number in its OpLog using one KnowledgeLog for each replica. Each entry in a KnowledgeLog contains the tuple *(vs, op_seq)*. *vs* denotes the version stamp of the operation. *op_seq* denotes the sequence number of OpLog where the operation with *vs* was first appended. More precisely, each entry of KnowledgeLog K_i^j on host X_i contains tuples that map each version stamp *vs* whose node ID is j to a sequence number in $OpLog(X_i)$. Although the position of a version stamp might change due to later merge steps, note that a version stamp can only be pushed down in order but never pulled up due to the way *insert* works. Thus, the sequence numbers stored in KnowledgeLogs provide us a starting point to search for a version stamp. The version stamp might be at that sequence number, or at a later one, but never at an earlier one. For improved performance, we can also opt to cache a fixed number of entries from the end of KnowledgeLog in memory. As all the information needed to maintain a KnowledgeLog are present in the OpLog, KnowledgeLogs can be reconstructed after a system crash.

We refer to Figure 4.2 as an example of interactions among OpLogs and KnowledgeLogs. Operation $1A$ is inserted in $OpLog(X_A)$ at sequence number 1. To record the mapping from version stamp $1A$ to sequence number 1, X_A appends $(1A, 1)$ to K_A^A. Similarly, X_A appends $(2A, 2)$ to K_A^A to record that the operation with version stamp $2A$

$OpLog(X_A)$	
seq	vs
1	1A

K_A^A		
seq	vs	op_seq
1	1A	1

K_A^B		
seq	vs	op_seq

X_A executes $1A$

$OpLog(X_A)$	
seq	vs
1	1A
2	2A

K_A^A		
seq	vs	op_seq
1	1A	1
2	2A	2

K_A^B		
seq	vs	op_seq

X_A executes $2A$

$OpLog(X_A)$	
seq	vs
1	1A
2	2B
3	2A

K_A^A		
seq	vs	op_seq
1	1A	1
2	2A	2

K_A^B		
seq	vs	op_seq
1	2B	2

X_A merges with X_B

Figure 4.2: Mapping from version stamps to sequence number of OpLog in KnowledgeLogs.

was inserted in $OpLog(X_A)$ at sequence number 2. A merge step with X_B results in the operation with version stamp $2A$ to be pushed down in order i.e., at sequence number 3. As we have already recorded $2A$ in K_A^A and we can reach $2A$ in $OpLog(X_A)$ even if we start scanning from the recorded *op_seq* value (in this case 2), we can keep it unchanged. We only append the entry $(2B, 2)$ to K_A^B. Now if X_A (reader) performs a merge step with an arbitrary replica X_s (source) and wants to know whether X_s has any operation originating from replica X_B that the reader does not know of, it can simply compare the tails of K_A^B and K_s^B. If the last entry of K_A^B contains a version stamp that is less than that of the version stamp contained in the last entry of K_s^B, then X_s has operation originating from X_B that X_A does not know of (as two version stamps with same node ID follow happens-before relationship and version stamps are written to the KnowledgeLog in increasing order). This process is explained in detail in the next section.

4.5.2 Conflict Detection

In the conflict detection stage during a merge step between reader X_i and source X_j, the reader X_i compares the last entries of K_i^m and K_j^m, $\forall m \in S$. We represent the last entry of a log L by $tail(L)$ and a field f in entry e by $e.f$. If $tail(K_i^m).vs < tail(K_j^m).vs$, this means X_j (source) has executed operations that X_i has not. This holds as the operations in a KnowledgeLog have the same node ID and are executed in increasing order of their counter. The counter captures the happens-before relationship between two version stamps with the same node ID. We say X_i lags behind X_j with respect to X_m when $tail(K_i^m).vs < tail(K_j^m).vs$. X_i might lag behind X_j with respect to more than one replica. Let us represent the set of all replicas with respect to which X_i lags behind X_j as X_{lag}.

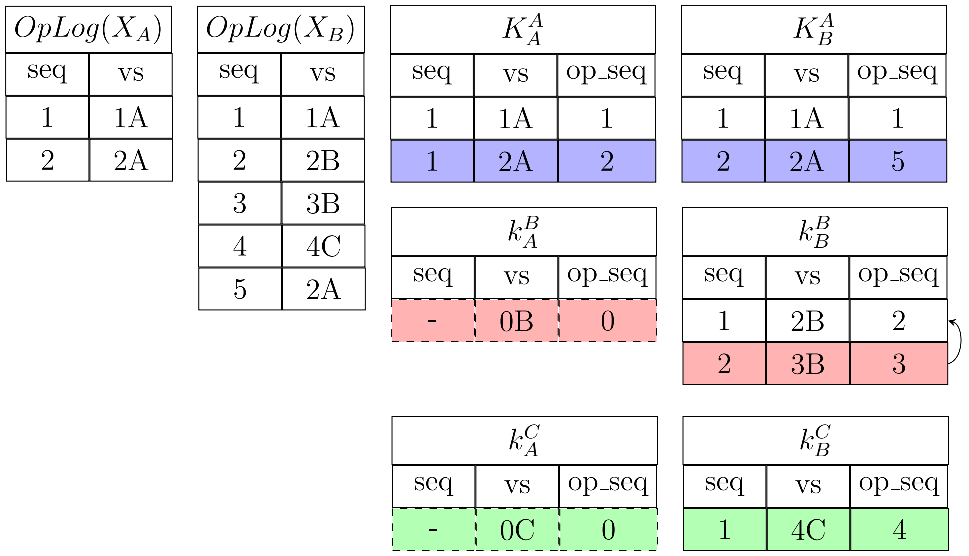
We represent the set of node IDs of the replicas in X_{lag} as S_{lag}. We find the replica X_p in X_{lag} such that $tail(K_j^p).op\_seq < tail(K_j^l).op\_seq, \forall l \in S_{lag} \wedge l \neq p$. That is, X_p is the replica whose operation is at the earliest point of conflict between X_i and X_j. However, X_i might not know about operations of X_p that have version stamps less than $tail(K_j^p).vs$. To ensure X_i can detect all unknown operations, it scans backward from the tail of K_j^p until it finds the entry e such that the entry before it has a version stamp equal to $tail(K_i^p).vs$. Then $e.op\_seq$ is the sequence number from which the reader starts scanning the source's OpLog (i.e. $source_{start} = e.op\_seq$). In other words, $e.vs$ is the earliest operation in $OpLog(X_B)$ that X_A has not yet executed.

Proof that a reader can identify all operations unknown to it during conflict detection: Reader X_i will identify all the operations unknown to it if there are no operations in $OpLog(X_j)$ before the sequence number $source_{start}$ that X_i does not know of (as X_i starts reading $OpLog(X_j)$ from the sequence number $source_{start}$). We can prove this by contradiction. Let us assume there is indeed a version stamp vs in $OpLog(X_j)$

at sequence number $source^{'}_{start}$ such that $source^{'}_{start} < source_{start}$ and $vs.nodeID = q$. That would mean one of the following: (i) $p \neq q$. That is, the tail of K^q_j contains an entry with *op_seq* value that is smaller than all other *op_seq* values of knowledge logs with respect to which X_i lags X_j. However, this is not possible, as we are taking the minimum of *op_seq* values of the tails of the relevant knowledge logs to find p. (ii) $p = q$. In that case, there must be an entry in K^p_j with vs greater than $tail(K^p_i).vs$ and *op_seq* less than $source_{start}$. However, this is not possible either, as we scan back to make sure we find the earliest version stamp in K^p_j that X_i has not seen. Hence we arrive at a contradiction and the reader must be able to identify all operations unknown to it during conflict detection.

Let the version stamp of the sequence number $source_{start} - 1$ in $OpLog(X_j)$ be vs_{prev}. To incorporate $e.vs$, X_i executes $insert(e.vs, vs_{prev})$ in $OpLog(X_i)$. To do this, X_i first finds the sequence number of $e.vs$ in $OpLog(X_i)$ – the value of $reader_{start}$ is this sequence number plus one. Note that all operations in $OpLog(X_j)$ from sequence number 1 to $source_{start} - 1$ must be present in $OpLog(X_i)$, otherwise there is some operation between these two sequence numbers in $OpLog(X_B)$ that X_A has not seen, and the value of $source_{start}$ found by the previous steps would have been different. Therefore, $reader_{start}$ must be greater than or equal to $source_{start}$. To find the value of $reader_{start}$, X_i starts scanning $OpLog(X_A)$ from the sequence number $source_{start} - 1$. It stops scanning if the currently scanned entry's version stamp is equal to vs_{prev}. The required value of $reader_{start}$ is the sequence number where we stop scanning plus one.

To illustrate the conflict detection stage, we consider the scenario in Figure 4.3. Let us assume there are three replicas in our system, X_A, X_B, and X_C. The OpLog of X_A has $1A$ and $2A$, whereas the OpLog of X_B has $1A$, $2B$, $3B$, $4C$, and $2A$. One possible sequence of actions that might lead to this state: X_A executed operation $1A$. X_B merged with X_A, and then executed two operations $2B$ and $3B$. X_C (not shown in the figure)



$OpLog(X_A)$	
seq	vs
1	1A
2	2A

$OpLog(X_B)$	
seq	vs
1	1A
2	2B
3	3B
4	4C
5	2A

K_A^A		
seq	vs	op_seq
1	1A	1
1	2A	2

K_B^A		
seq	vs	op_seq
1	1A	1
2	2A	5

k_A^B		
seq	vs	op_seq
-	0B	0

k_B^B		
seq	vs	op_seq
1	2B	2
2	3B	3

k_A^C		
seq	vs	op_seq
-	0C	0

k_B^C		
seq	vs	op_seq
1	4C	4

Figure 4.3: OpLogs and KLogs of replicas X_A and X_B in a system with three replicas. Dashed entries represent placeholders used during computation when a knowledge log is empty. During conflict detection, the reader X_A compares the same colored entries with each other to find the earliest possible point of conflict. The arrow from the second entry to the first entry of K_B^B, represents X_A's backward scan to find the earliest version stamp with node ID B that it does not know of.

merged with X_B and executed $4C$. X_B merged with X_C. X_A executed operation $2A$. Finally, X_B merged with X_A again. Now let us consider X_A performs a merge step with X_B. Comparing the tails of K_A^m and K_B^m, $m \in \{A, B, C\}$, we see that X_A lags behind X_B with respect to X_B and X_C, i.e., $X_{lag} = \{X_B, X_C\}$ (we assume the absence of entry in a KnowledgeLog to be equivalent to having a placeholder entry with a version stamp with minimum possible invalid counter value, in this case, 0). As the *op_seq* value of $tail(K_B^B)$ (i.e. 3) is smaller than that of $tail(K_B^C)$ (i.e. 5), $X_p = X_B$. However, X_A is not yet certain $tail(K_B^B).vs$ is the earliest unknown version stamp. X_A scans K_B^B backwards to find the earliest unknown version stamp, which in this case is $2B$. The corresponding *op_seq* value is 2, therefore $source_{start} = 2$. The entry immediately preceding $2B$ in $OpLog(X_B)$ has the version stamp $1A$. X_A reads the entry at sequence number $source\_start - 1 = 1$ in $OpLog(X_A)$ and finds that the entry contains $1A$. Therefore *reader_start* is equal to

$1 + 1 = 2$ as well. The conflict detection algorithm is presented in Algorithm 2.

4.5.3 Conflict Resolution

Conflict resolution is triggered when a conflict is detected, to find and execute a merged order of operations between the reader and the source. When there are one or more conflicts between the reader and the source, it rolls back the OpLog of the reader to the earliest point where the reader does not lag behind the source with respect to the version stamps before it and then replays the operations at the reader (adjusting the OpLog of the reader) to reflect the merged order. At the start of conflict resolution, X_i knows both $source_{start}$ and $reader_{start}$, i.e., the sequence number of $OpLog(X_i)$ and the sequence number of $OpLog(X_j)$ at which X_i should start comparing the two OpLogs. X_i creates an ordered list, R_i, of the operations in $OrdLog(X_i)$ starting from the sequence number $reader_{start}$ up to its latest sequence number. X_i creates a second ordered list, R_j, of the operations in $OrdLog(X_j)$ starting from the sequence number $source_{start}$ up to its latest sequence number.

To incorporate the operations unknown to itself, X_i first includes those operations from R_j to R_i by invoking *insert* procedures: for each entry e in R_j, X_i first finds the entry e_{pred} in R_i which contains the version stamp immediately preceding e in R_j. If the version stamp of the entry following e_{pred} in R_i is smaller than $e.vs$, X_i inserts e immediately after e_{pred} (provided e is not already present there). Otherwise, it skips over all contiguous entries where the version stamp is greater than $e.vs$, and then inserts e (provided that e is not already present there). Once X_i has all the operations in R_i, it rolls back, i.e., prunes, $OpLog(X_i)$ starting from $reader_{start}$ and then replays all operations in R_i at $OpLog(X_i)$. The conflict resolution algorithm is presented in Algorithm 3.

Algorithm 2 Conflict Detection

```
Require: reader replica $X_i$, source replica $X_j$, and set of node IDs $S$
Ensure: earliest point of conflicts $source_{start}$ and $reader_{start}$
 1: $X_{lag} \leftarrow \phi$
 2: for $m \in S$ do
 3:     if $tail(K_i^m).vs < tail(K_j^m).vs$ then
 4:         $X_{lag} \leftarrow X_{lag} \cup \{X_m\}$
 5:     end if
 6: end for
 7: if $X_{lag} = \phi$ then
 8:     return
 9: end if
10: $S_{lag} \leftarrow \phi$
11: for $X_m \in X_{lag}$ do
12:     $S_{lag} \leftarrow S_{lag} \cup \{m\}$
13: end for
14: $p \leftarrow \underset{m}{\operatorname{argmin}}(tail(X_j^m).op\_seq), m \in S_{lag}$
15: $idx \leftarrow latest\_seq(X_j^p)$
16: while $idx > 0$ do
17:     if $tail(K_i^p).vs < K_j^p[idx].vs$ then
18:         $source_{start} \leftarrow K_j^p[idx].op\_seq$
19:         $idx \leftarrow idx - 1$
20:     else
21:         break
22:     end if
23: end while
24: $vs_{prev} \leftarrow OpLog(X_j)[source_{start} - 1].vs$
25: $idx \leftarrow source_{start} - 1$
26: while $idx \leq latest\_seq(OpLog(X_i))$ do
27:     if $OpLog(X_i)[idx].vs = vs_{prev}$ then
28:         $reader_{start} \leftarrow idx + 1$
29:         break
30:     else
31:         $idx \leftarrow idx + 1$
32:     end if
33: end while
34: ResolveConflict($reader_{start}, source_{start}$)
```

4.6 Handling Bounded Log Sizes

Logically, logs are append-only storages to which we can continuously append. Practically, we are bounded by the physical storage of our devices. Therefore, we can not record an unbounded number of versions of our data types. In this section, we describe how we can safely retain the last K versions of our data type by removing old entries from OpLog. We make two underlying assumptions: (i) our physical storage has the capacity to store more than K versions and (ii) the replicas merge among themselves at a rate such that there is at least one version common at the top of the OpLogs among all replicas before any replica runs out of space.

The major challenge in keeping a history of at least the last K versions is that this set of last K versions is constantly changing due to updates from different replicas. So instead, we identify versions that we know for certain are not in the set of last K versions.

Consider an arbitrary version vs. Referring back to how our *insert* operation of Section 4.5 works, we know vs in the OpLog of a reader can be pushed down in order due to a merge step involving an unknown operation that the reader has not seen before. This essentially means that even if vs was not in the set of last K versions, it may become so. However, if all replicas in our system have observed all the same operations from the start up to vs, we know the positions of those versions will not change.

This means that a replica X_i can safely remove n operations from the top of its OpLog while preserving at least the last K versions if: (i) all the other replicas in the system have those n operations in the top n entries of their respective OpLogs and (ii) the replica has already seen at least K more new operations after those n operations. Note that X_i need not check whether the other replicas have received K more new operations to trim its own OpLog. It just has to make sure there is a set of operations from the top that all the replicas have executed in the same order. This trimming operation can be triggered after a user-defined number ($> K$) of versions have been recorded. However, if this number is near K, trimming will be frequent and may adversely affect system performance.

A replica X_i need not scan the top of each replica's OpLog to find the entries that can be trimmed. Instead, it consults the tails of KnowledgeLogs. For each node ID $s \in S$, X_i reads the tails of the N KnowledgeLogs $K^s_u(u \in S)$ and identifies the smallest version stamp. We denote this set of N versions as C. Then C contains one version vs for each originator X_s such that vs is the greatest version with node ID s that all the replicas have seen.

Next, X_i locates the earliest sequence number in its log where such a version is present. Let this sequence number be *seq*. Note that any version in an entry preceding the entry at *seq* in $OpLog(X_i)$ must be present in all the other replicas. Therefore, X_i can trim all the entries up to *seq* provided that there are at least K versions following it. If not, it can backtrack the required number of entries to meet this condition and then proceed with trimming.

There are some caveats to this approach that we must overcome. *First*, although the earliest n versions trimmed in this approach are not part of the latest K versions, we may still need the last version of the earliest n versions as an anchor point for future *insert* operations. This can happen if there is at least one replica such that it had exactly n versions at the time X_i trimmed its OpLog. Hence, we must record the last version in the set of trimmed versions every time we perform this operation. We can safely overwrite this record every time we trim the log. *Second*, the mappings in KnowledgeLogs will be off by n whenever n operations are trimmed. To counter this, we introduce *Virtual Sequence Numbers (VSN)*. VSN of an entry in a log represents the sequence number the entry would have if the log was never trimmed. Therefore, we must track an *offset* (initially zero) that is updated every time we trim an OpLog from the top (incremented by the number of entries trimmed). Then to get the VSN of an entry, we can simply add its sequence number to the offset. KnowledgeLogs now store VSNs instead of sequence numbers. While trimming the OpLog, we can trim the corresponding KnowledgeLog entries as well. Just like OpLogs, we track the last entry removed from a KnowledgeLog to notify the readers of the trimmed operations. *Third*, if a new replica joins the cluster with an initially empty state, it can force a change in the order of operations. This can be overcome in two ways: we can copy the initial state of the new replica from an arbitrary existing replica, or we can assign an ID to the new replica that is smaller than all the existing replicas. A smaller ID will force the new replica to put its entries at the end of all the existing versions and thus, not alter the current order.

4.7 Evaluation

In this section, we empirically evaluate the performance of LSCRDT. As LSCRDTs combine the advantages of both op-based and state-based CRDTs much like δ-CRDTs [93], we compare the performance of LSCRDTs with that of δ-CRDTs. We implement both LSCRDT and δ-CRDT in C and use memory-mapped files for persistent storage, including logs. Note that our goal is not to outperform δ-CRDTs, but to explore the feasibility of LSCRDT while providing all the advantages associated with using logs.

We conduct our experiments using the foundational data types register, counter, and set widely studied in CRDT literature. We use the last-writer-wins (LWW) variant of register and positive-negative (PN) variant of counter for both δ-CRDT and LSCRDT. We use two-phase (2P) variant of set for δ-CRDT (an element cannot be added again once removed) whereas LSCRDT set works in the conventional way. We follow the δ-CRDT implementation of [118] for the data types.

In our results we focus on: (i) how much extra time is introduced for a single operation

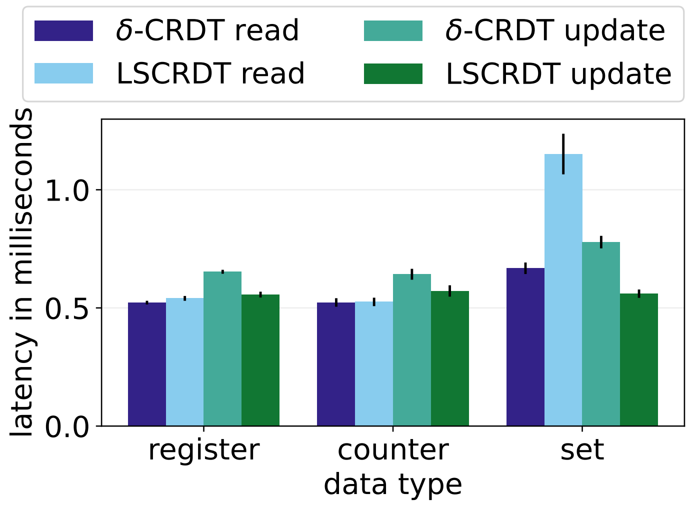
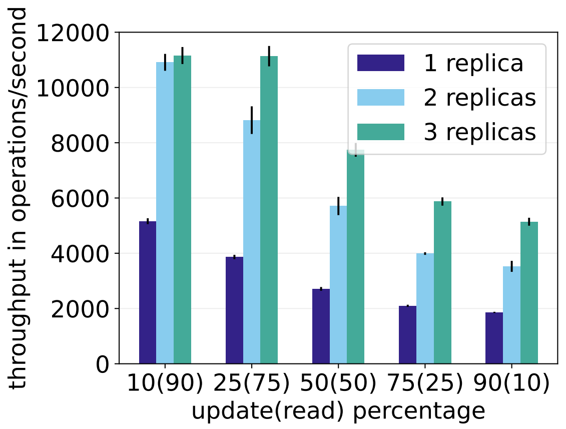


Figure 4.4: Comparison of latency between LSCRDTs and δ-CRDTs on operations executed at a single replica.

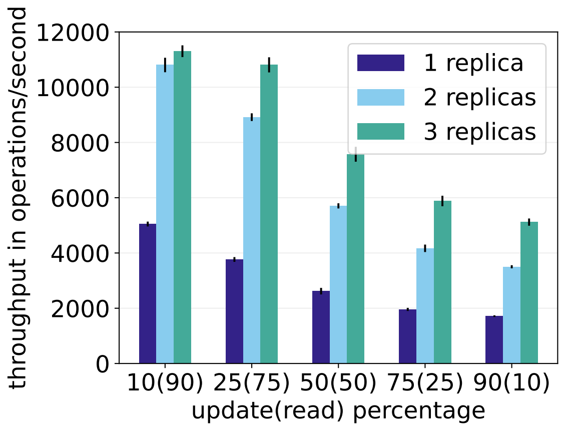
to execute due to the introduction of logs to store the data types, (ii) what is the effect of logs on scalability, and (iii) how time-consuming is versioned read compared to reading the latest value. We run each experiment ten times and show the average result. We reset each data type to its initial state before each run.

Moreover, to demonstrate the versatility of the replication protocol used in LSCRDT, we use it to replicate PEDaLS and present the results in Section 4.7.4. In addition, we explore the effect of Knowledge logs on the time to detect conflict in Section 4.7.5.

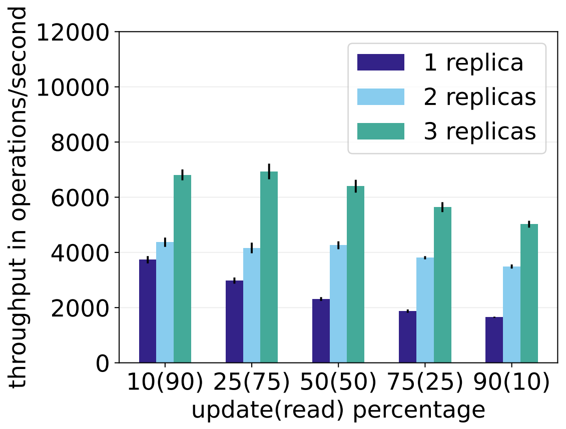
We perform the replication experiments with a cluster of three replicas and one client. Many cloud-based storage systems such as Amazon S3 use three nodes for replication. Although edge/device tier do not have the same level of reliability as cloud servers, it is not uncommon for edge applications to have a replication factor of three as well [119, 3, 40]. All of the machines are running the CentOS 7 Linux operating system each with two dedicated 2GHz vCPUs and 2GB of memory. The machines communicate among themselves using 1Gb/second Ethernet. The average latency among the machines is observed to be 0.45ms. In the case of LSCRDTs, each replica executes a merge step



(a) Register.

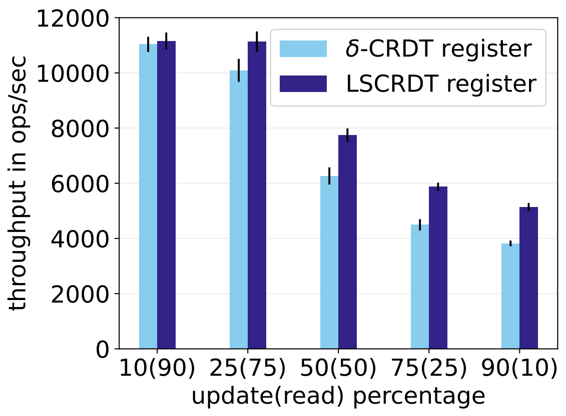


(b) Counter.

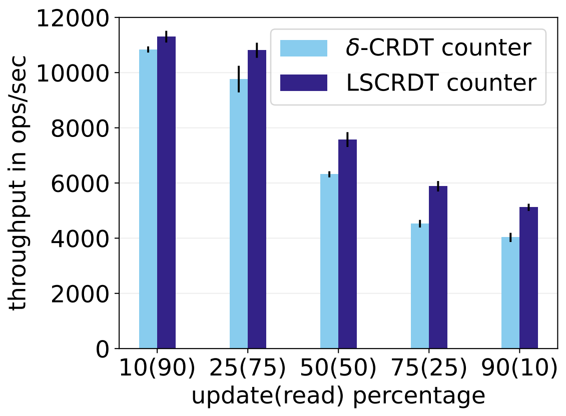


(c) Set.

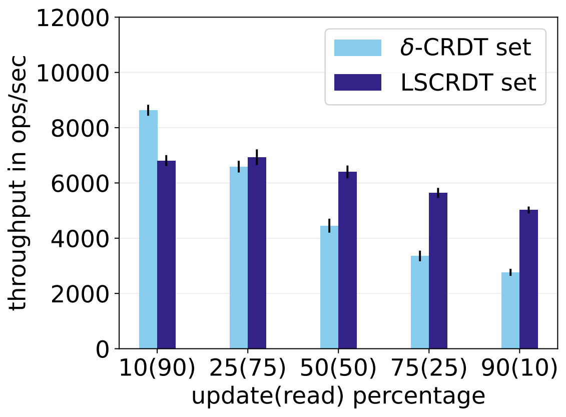
Figure 4.5: Scalability of LSCRDTs.



(a) Register.



(b) Counter.



(c) Set.

Figure 4.6: Comparison of throughput between LSCRDTs and δ-CRDTs using three replicas.

with another replica in round-robin fashion at one second intervals. In case if δ-CRDTs, deltas are queued for propagation immediately after local execution.

4.7.1 Single Node Latency

To evaluate latency, we send 10000 randomly generated operations (half reads, half updates) sequentially from the client to a replica and measure the average latency. The arguments of the update operations are randomly generated integers between 1 and 1000. Read operations read the latest versions of respective data types in the case of LSCRDT.

Figure 4.4 shows the read and update latencies for the three data types. In case of all three data types, LSCRDT shows a lower update latency than δ-CRDT counterparts. Specifically, LSCRDT is $1.17x$, $1.12x$, and $1.39x$ faster than δ-CRDT for register, counter, and set respectively. There is no significant difference in read latencies between LSCRDT and δ-CRDT for register and counter. However, δ-CRDT set provides faster read than LSCRDT set. Specifically, the read latency of δ-CRDT set is 0.668ms whereas that of LSCRDT set is 1.151ms. That is, δ-CRDT reads are 1.72 times faster than LSCRDT for set. This difference in read latency is expected. Recall that although LSCRDT set works in the conventional way, the variant of δ-CRDT set we are using (i.e., 2P-set) does not allow an element to be added again once it has been removed. That is, the query in 2P-set can be performed faster by first looking at the list of elements marked as removed. On the other hand, LSCRDT set has to first read the last checkpointed state from a log and take into account all the operations that have been performed since that checkpoint. Although the read latency of LSCRDT set is higher than that of δ-CRDT set, we will see in Section 4.7.2 that the former has higher throughput in the presence of high volume of updates. As devices at the edge are generally write-heavy, LSCRDT is thus a better option than δ-CRDT for edge applications.

4.7.2 Scalability

To evaluate the scalability of LSCRDT, we randomly generate workloads of 10000 operations. As updates are more expensive than reads in general, we vary the percentage of update (read) operations among 10(90), 25(75), 50(50), 75(25), and 90(10) to observe the impact of workloads with different update/read composition on scalability. We also vary the number of replicas the client sends requests to among 1, 2, and 3. All 3 replicas are live and perform merge (LSCRDT) or join (δ-CRDT) even if the client sends requests to fewer than 3 replicas. A client evenly distributes operations across replicas using round-robin without delay.

Figure 4.5 shows the throughput of the system in operations per second. For LSCRDT registers, throughput nearly doubles as the number of replicas is increased from one to two for most workloads. This increase is around 2.8 times when the number of replicas is increased from 1 to 3 for all workloads except that with 10% updates. The increase in throughput with the increase in the number of replicas indicates that LSCRDT registers are scalable, although this increase is not strictly linear. This is due to the processing required for background merge steps. Figure 4.5b reveals a similar increase in throughput for counter as the number of replicas increases. We can observe a similar trend in increase in throughput for set for workloads with a higher percentage of update from Figure 4.5c. The increase in throughput is not as pronounced for workloads with a lower percentage of update due to the higher read latency of LSCRDT set.

Figure 4.6 compares LSCRDT throughput with that of δ-CRDT for the data types using three replicas. As the percentage of update is increased, LSCRDT data types start showing higher throughputs than δ-CRDT counterparts. Specifically, LSCRDT register and counter show 1.1x, 1.2x, 1.3x, and 1.3x higher throughput than δ-CRDT counterparts for workloads with 25, 50, 75, and 90 percentage of update respectively.

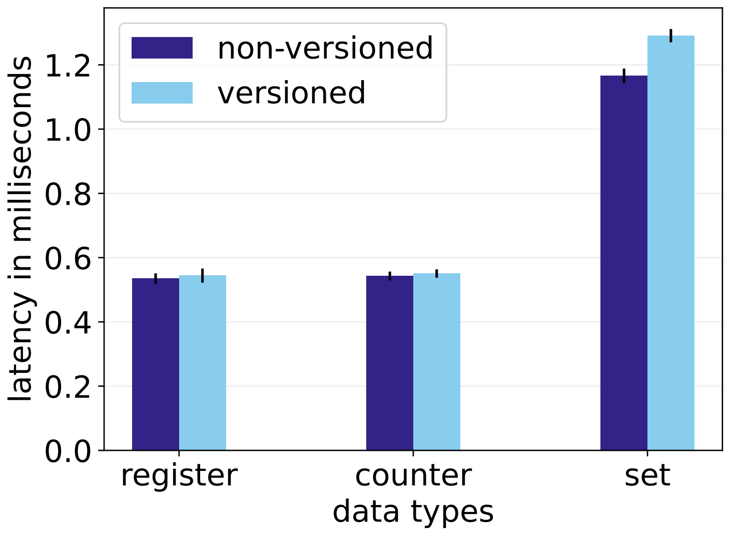


Figure 4.7: Comparison of average latency in milliseconds between 5000 non-versioned reads (i.e. reads to the latest version after each update) and 5000 versioned reads (one read per version after all updates).

In the case of LSCRDT set, the increase in throughput is $1.1x$, $1.4x$, $1.7x$, and $1.8x$ than its δ-CRDT counterpart for workloads with 25, 50, 75, and 90 percentage of update respectively. Note that LSCRDT set has a higher read latency but a lower write latency than δ-CRDT set. The lower write latency along with the efficient lock-free merge step of LSCRDT results in a higher throughput in LSCRDT set for workloads with a high volume of updates. As edge applications are typically write-heavy, LSCRDT presents itself as the better option.

4.7.3 Versioned Reads

Maintaining version history is a unique feature of LSCRDTs not available in CRDTs. To understand the overhead associated with querying earlier versions, we send 5000 update requests to a data type. We send two sets of 5000 read requests: (i) we send one read request after each update request for non-versioned read, and (ii) we send 5000

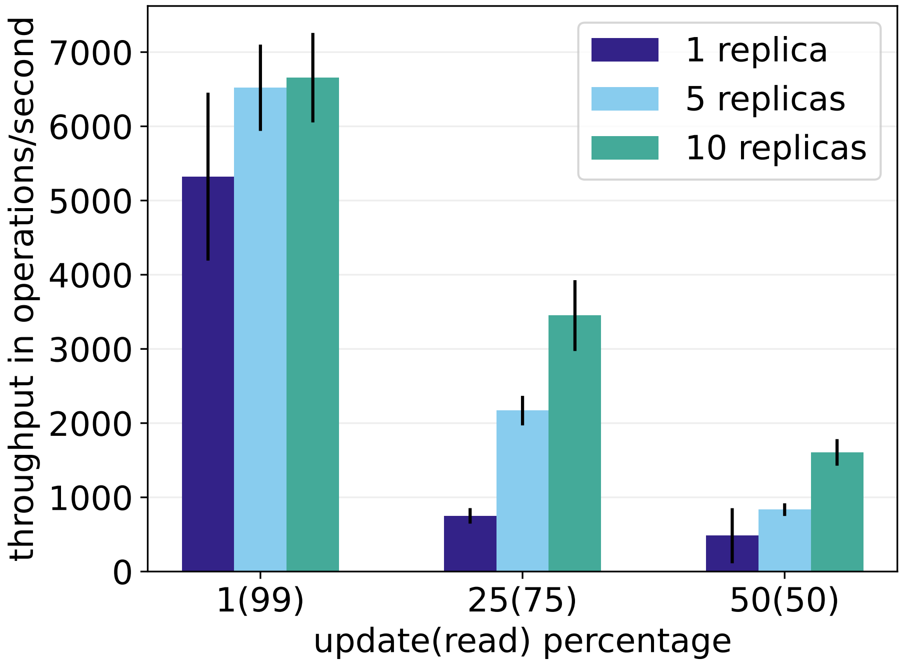


Figure 4.8: Scalability of PEDaLS using LSCRDT's replication protocol.

read requests after all the update requests for versioned read (one for each version). We show the average latency of the non-versioned and versioned reads in Figure 4.7. The latencies for versioned and non-versioned reads for both counters and registers are similar. However, set presents us with a different scenario. The latest operations are cached in memory and can be used for non-versioned read, whereas versioned reads might require reading the OpLog as described in Section 4.4.3. Due to this extra read from the OpLog, versioned reads (1.29ms) experience higher latency than non-versioned (1.17ms) reads on average.

4.7.4 PEDaLS Replication and Scalability

To evaluate the scalability of PEDaLS when replicated using LSCRDT's protocol, we use a subset of the CityPulse temperature dataset used in Section 3.4.3. As updates are more expensive than reads in general, we augment the subset with random reads to capture the performance of PEDaLS under diverse workloads consisting of different percentages of update and read operations. We consider update percentages (read % is in parenthesis) of 1(99), 25(75), and 50(50) to observe the impact of workloads with different update/read compositions on scalability over 10000 operations (update+read). We also vary the number of replicas the client sends requests to among 1, 5, and 10. The round trip latency among the instances (replicas+client) as determined through the *ping* utility varies between 0.53ms to 0.93ms asymmetrically on average. The network bandwidth among the instances is approximately 1Gbits/second. To simulate the effect of horizontal scaling where clients located in different regions access different replicas, a client process evenly distributes operations across replicas using round-robin without delay.

Figure 4.8 shows the throughput of the system in operations per second for PEDaLS bst. As evident from the figure, throughput decreases with the increase in write percentage. This is expected, as update operations involve appends to multiple logs in addition to searching for a value. Moreover, we observe that although throughput of PEDaLS increases with an increase in the number of instances, this increase is not linear. This is due to the processing required for background merge steps.

4.7.5 Effect of Knowledge Logs in Conflict Detection

To reduce the time required for conflict detection, LSCRDT uses Knowledge logs. In this section, we explore the time benefit provided by Knowledge logs by isolating the time to detect conflict between two replicas. Specifically, we run two different sets of experiments. In one set of experiments, as has the case been so far, LSCRDT uses Knowledge logs to optimize conflict detection. In the other set of experiments, LSCRDT

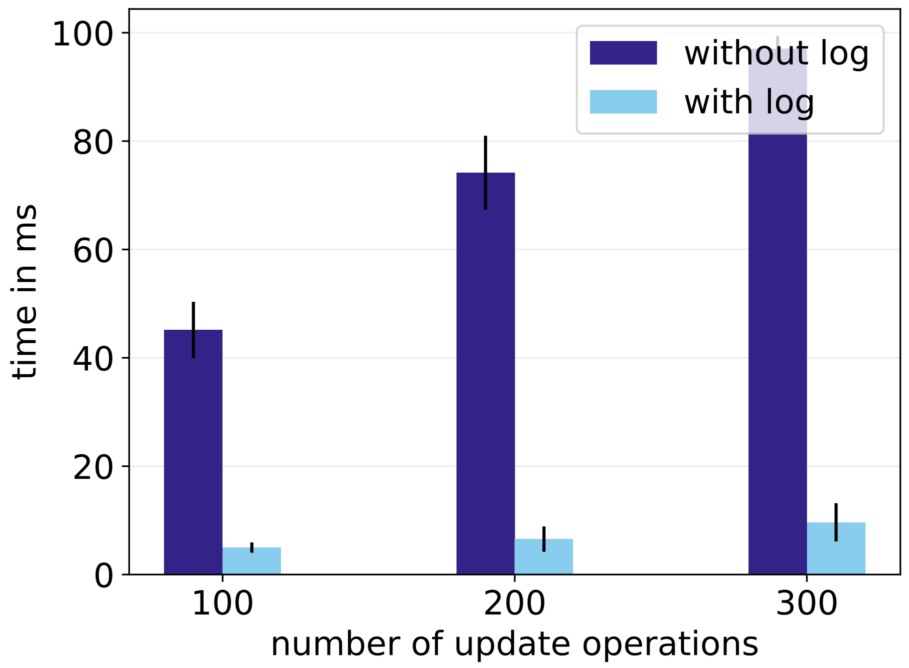


Figure 4.9: Conflict detection with and without Knowledge logs.

performs naive conflict detection, i.e., reads the OpLogs of the reader and the source from the top until a point of mismatch is found or one of the logs is exhausted.

For this experiment, we vary the number of update operations among 100, 200, and 300. We send half of the operations to one replica X_A and the other half to another replica X_B, both chosen from the pool of replicas used in Section 4.7.4. As the execution time of conflict detection can depend on the role (source or reader) of a replica, we calculate the time for conflict detection in both ways and take the average. The number of operations already present in a replica can affect the time for conflict detection as well. Hence, we consider four cases: (i) both X_A and X_B each have half of the operations, (ii) X_A has half of the operations and X_B has all of the operations (due to a merge step), (iii) X_A has all of the operations and X_B has half of the operations, and finally (iv) both X_A and X_B have all of the operations.

We show the total time taken for conflict detection for different workloads in Figure 4.9. Our results show that conflict detection with Knowledge log can be 11.34x as fast as conflict detection without Knowledge log for 300 update operations. In general, irrespective of whether Knowledge log is used or not, the time for conflict detection increases with the number of update operations. This is expected, as more update operations result in more entries in both OpLog and Knowledge log, which in turn leads to more log scans.

4.8 Summary

In this work, we introduce LSCRDTs to integrate the benefits of distributed causal logging into operation-based CRDTs. By doing so, LSCRDT can provide a robust and uniform way to reverse operations for arbitrary data types. In addition, LSCRDT overcomes the restrictions of commutative data types, exactly-once causal delivery, operation idempotence, and data type-specific join operations (a side effect of state-based CRDTs). Finally, LSCRDT is the first CRDT system to track version histories of data structures and provide programmatic access to them. Our results show that LSCRDT can result in up to 1.8x higher throughput than δ-CRDT, making it suitable for update-heavy edge workloads.

Algorithm 3 Conflict Resolution

Require: reader replica X_i, source replica X_j, $source_{start}$ and $reader_{start}$ value obtained from Conflict Detection stage

Ensure: X_i is not lagging behind X_j

```
 1: procedure ResolveConflict($source_{start}, reader_{start}$)
 2:     $R_i \leftarrow \{OpLog(X_i)[reader_{start}], \dots, tail(OpLog(X_i))\}$
 3:     $R_j \leftarrow \{OpLog(X_j)[source_{start}], \dots, tail(OpLog(X_j))\}$
 4:     for all $e \in R_j$ do
 5:         $e_{pred} \leftarrow$ the entry before $e$ in $R_j$        ▷ fixed dummy value assumed for first element
 6:         $insert(e.vs, e_{pred}.vs)$ in $R_i$
 7:     end for
 8:     Prune $OpLog(X_i)$ starting from sequence number $reader_{start}$
 9:     for all $e \in R_i$ do
10:         Replay/Execute $e.op$ and append $e$ to $OpLog(X_i)$
11:         $q \leftarrow$ sequence number of $e$ in $OpLog(X_i)$
12:         $k \leftarrow e.vs.nodeID$
13:         if $e.vs > tail(K_i^k).vs$ then
14:             append $(e.vs, q)$ to $K_i^k$
15:         end if
16:     end for
17: end procedure
```

Chapter 5

GreenCoin: A Renewable Energy-Aware Cryptocurrency

The previous two chapters demonstrated how data versioning and replication can make distributed IoT systems reliable and fault-tolerant. Data trustworthiness is paramount in this era of data-driven IoT applications. While replicated and versioned data are significant for a well-functioning system, tamper-proof data is desirable from the end user's perspective. The popularity of blockchain systems has resulted in their wide adoption for making application data tamper-proof. However, most blockchain protocols are resource intensive and can adversely affect the environment due to the massive energy required. Moreover, the existing blockchain protocols do not differentiate between renewable and non-renewable energy. Given that the world is moving towards renewable energy due to its availability, affordability, and overall favorable impact on the climate, the next generation of blockchain protocols must make this distinction and preferably become more energy-efficient. In this chapter, we propose a novel renewable energy-aware blockchain protocol, GPoS, and describe how we can use this to drive an energy-efficient cryptocurrency, GreenCoin.

Cryptocurrencies powered by blockchain have the potential to revolutionize society along several dimensions, from commerce to political and corporate governance, to social trust. These innovations and potential benefits carry environmental costs largely (but not exclusively) accruing to the energy required by distributed cryptocurrency implementations. At a high level, a "crypto coin" (an unforgeable value-carrying token) represents some amount of "work" associated with its production.

In the original formulations of cryptocurrencies (cf. [120, 121]) "miners" solved computationally intensive or memory intensive [122] puzzles, the solutions they announced to all participants, which could easily verify the solutions. These "proof-of-work" (PoW) protocols created a consensus among all participants about the temporal order in which puzzles were solved that could only be subverted by an adversary controlling a substantial fraction [123] of all of the participants. By attaching a transaction to each puzzle solution, then, the overall system could achieve consensus on transaction order in a way that is difficult (or expensive) to subvert. In particular, it was possible to determine the "earliest" transaction that achieved consensus thereby preventing one puzzle from being used to represent multiple transactions. That is, the solution to a puzzle (termed the "minting" of a crypto coin), could only be spent once.

PoW protocols are currently in use by many cryptocurrencies, e.g., Dogecoin [124], Monero [125], Litecoin [122], and most notably Bitcoin [120]. Colored coin [126], a cryptocurrency based on Bitcoin, proposes tracking the origin of a given bitcoin and color a set of coins to distinguish it from the rest. However, they all require the computers solving the puzzles (i.e., the "miners") to expend much electrical energy on each puzzle solution. Furthermore, puzzle solutions are deliberately made rare to prevent inflation. Thus as solver technology improves (often due to greater energy expenditure), energy consumption by miners increases.

To address the environmental concerns associated with PoW protocols, particularly concerning the use of fossil fuels to generate the electricity they require, cryptocurrencies have been turning to new “Proof-of-Stake” (PoS) protocols that are far more energy efficient. PoS protocols, unlike their PoW counterparts, rely on economic penalties to incentivize truth-telling concerning transaction order and uniqueness. With these new protocols, “miners” announce transactions without solving a puzzle associated with each. Instead, each miner is “staked” to a hoard of crypto cash (that carries value) which can be confiscated should a miner make a false announcement where veracity is determined by consensus.

By eliminating the need to solve energy-consuming puzzles, PoS protocols are far more energy efficient than corresponding PoW protocols. As a result, many current cryptocurrencies (c.f. [127, 128, 129]) have begun to use them, most notably Ethereum [130].

Regardless of whether a cryptocurrency uses a PoW protocol or a PoS protocol, however, they require a distributed collection of computers to achieve consensus on a global transaction order in a way that can be recorded (again by all participants) that is tamper-proof. These computers must consume electrical power to implement the protocol functionality.

In this chapter, we describe GreenCoin – a cryptocurrency that marks each crypto coin with an indelible measure of the renewable energy that was used to create it. GreenCoin relies on a new PoS protocol (GPoS) to attach a “green score” (that maps to renewable energy available at the time of the transaction proposal) to each crypto coin in a way that cannot be removed. Further, GreenCoin and GPoS permit the formulation of “green” smart contracts that specify a minimum greenness score that must be presented to execute the contract. Note that we have defined GPoS as a PoS protocol as we believe future protocols are likely to take this form, but the fundamental tenets of GreenCoin apply to PoW protocols as well.

Implementing GreenCoin and GPoS requires that we

- develop a way to score each coin with a greenness score and to compose/decompose scores from multiple coins when they are used to represent transactions,
- develop an incentive system that favors (i.e., attaches greater value to) greener coins, and
- develop a system for securely determining the amount of renewable energy used when minting a coin.

In this chapter, we describe research results addressing these three challenges. We describe a methodology for attaching a green score to each coin, each wallet (containing multiple coins), and for transacting green scores. We also describe the use of Trusted Execution Environments (TEEs) and attestation [131] to determine the location of each machine proposing a transaction. This location information then indexes fuel mix data by location either from a publicly available database such as the US Energy Information Administration (EIA) database [132] or from energy utility records that must be published in the ledger. In this work, we assume that fuel mix data by geographic region is available in real-time (or near real-time) from a trusted data source. We validate these results using a prototype "permissioned" implementation of GreenCoin designed to support Internet of Things (IoT) deployments comprising participants whose identities are not anonymized. Our results indicate that GreenCoin feasibly captures and publishes into the ledger the relative renewable energy usage associated with each blockchain transaction and also enables smart contracts that are contingent upon sufficient use of renewable energy.

5.1 Related Work

In this section, we provide a literature review on the basic building blocks of GreenCoin. GreenCoin requires (i) an energy-efficient consensus protocol for blockchain, (ii) a trusted and secure way to find the location of a blockchain node, and (iii) a method to incorporate a trusted source of fuel mix data for use by the system. Hence, we first compare different protocols used in current cryptocurrency systems and explain our choice of PoS. Next, we discuss the current state-of-the-art in location verification and justify our choice of using trusted execution environments. Finally, we present sources of fuel mix data, review methods to incorporate third-party data into blockchain, and introduce our chosen system to do so, called Depot [133]. Depot is an open-source data lake that is designed to allow community data contribution with contributor-defined access control policies and shared hosting costs. Depot was developed as part of the RiPiT project (cf. [134, 135]) to host carbon-emissions data from public sources and community-contributing researchers.

5.1.1 Consensus Protocols

Most cryptocurrencies use blockchain as the core underlying technology [136]. When it comes to blockchain, there are multiple consensus protocols, such as proof of work (PoW), proof of stake (PoS), proof of authority (PoA), proof of retrievability (PoR), proof of elapsed time (PoET), etc. [137]. However, two of the most widely used protocols are PoW and PoS [138].

In PoW, any node that wants to participate in mining, i.e., block generation must solve a computationally complex problem to ensure the validity of the newly mined block [139]. Although finding the solution to this problem is challenging, verifying the solution is easy. The main criticism against PoW is its high energy consumption required to solve these crypto puzzles. The Bitcoin network, which uses PoW, was estimated to consume 2.55 gigawatts of electricity in 2018 [140]. This consumption was projected to reach 7.67 gigawatts, making it comparable to the energy consumption of entire countries such as Ireland (3.1 gigawatts) and Austria (8.2 gigawatts) [140]. As of July 2021, Bitcoin's carbon footprint showed 64.18 megatons of CO_2 emission, close to the emissions by Greece and Oman [141]. This high CO_2 footprint has led to a spike in research interest in energy-efficient protocols.

In contrast, PoS does not involve computationally intensive crypto puzzles and hence it is comparatively energy-efficient. In PoS, a blockchain node can opt to stake, i.e., set aside a portion of its coins as collateral. Instead of mining power as in PoW, the probability of a node being selected as the proposer, i.e., the entity permitted to create the next block (and hence earn the associated reward), is proportional to the stake. Just as a node can receive a reward for honest behavior, it can be penalized for malicious behavior. This penalty is executed in the form of *slashing*, i.e., taking away a portion of the stake. Due to the energy efficiency of PoS over PoW, we base our protocol, Green PoS (GPoS), on PoS.

5.1.2 Location Verification

Location-aware Internet applications commonly use IP addresses to determine the location of a connected device [142]. There are two primary methods of IP geolocation: (i) IP geolocation databases and (ii) active network measurements [143].

IP geolocation databases provide a mapping between an IP address and *(lat, long)* coordinates [144]. These databases can be either proprietary [145] or public [146, 147]. Although the exact methods of constructing these databases are not always divulged to the public, they are often based on a combination of *whois* services, autonomous system (AS) numbers, DNS LOC records, etc. [143, 144]. While these databases can achieve country-level accuracy, discrepancies among databases are prevalent regarding city-level accuracy [148]. The accuracy of these databases starts diminishing with increasing granularity. As the availability of renewable energy can vary even within the same city, IP geolocation databases are inadequate for supporting GreenCoins.

Measurement-based geolocation algorithms heavily depend on a set of geographically distributed *landmarks* with known locations [143, 142]. These landmarks measure different network properties between them and the target IP, such as the delay and path taken by traffic [149, 150]. However, the availability of such landmarks can be sparse [142], resulting in lower geolocation accuracy. Moreover, studies suggest that an adversarial target can falsify measurements without detection, thus advertising itself to be at a different location than it truly is [143]. Hence, measurement-based geolocation algorithms are also insufficient for our application.

A relatively accurate approach to determining the location of a machine is to equip it with a GPS module [151]. However, having a tamper-proof GPS module does not prevent a machine from falsifying its location. In addition, we must ensure that both the process that retrieves data from the GPS device and the communication channel between the process and the GPS device are tamper-proof. A prime candidate for such a system is a trusted execution environment (TEE). Studies suggest that using TEEs coupled with GPS devices can provide a secure and tamper-proof way of location verification (cf. [152, 153, 154]).

Trusted Execution Environments: A *trusted execution environment* is an isolated processing environment in which applications can be executed while precluding malicious interventions of the host OS [155]. Examples of popular technologies used to provide a TEE are Arm TrustZone [156] and Intel SGX [157]. A TEE provides isolation for programs from the rest of the device, called the rich execution environment (REE). REEs typically include an operating system, e.g., Linux, and user space applications.

The TEE uses a combination of software and hardware-based security mechanisms to ensure that applications running inside the TEE remain secure even when the REE is compromised [158]. Figure 5.1a shows a high-level diagram of the interaction among REE, TEE, and peripherals within a single device. Desired security features of a TEE include isolated execution, secure storage, remote attestation, secure provisioning, and trusted path [159]. Many TEEs provide most of these features, while some can be built upon the existing functionalities provided by the TEE. For example, Intel SGX provides built-in remote attestation, which verifies three things: (i) the identity of an application, (ii) whether it is intact, i.e., that it has not been tampered with, and (iii) that it is running securely inside the TEE [160]. Although Intel SGX does not provide a trusted path between peripherals and itself out-of-the-box, multiple studies show it is possible to establish secure paths between I/O devices and Intel SGX [161, 162].

Security Concerns of TEEs and Countermeasures: Despite the promising security features of TEEs, multiple studies have demonstrated that TEEs are susceptible to some attacks [163, 164]. Many of these are side-channel attacks that compromise the secure keys used for attestation, rendering the software running in these TEEs untrustworthy [158]. Cache-based side-channel attacks such as prime+probe [165] and flush+reload [166] observe the timing differences of different measurements to identify whether data is retrieved from the cache or the main memory, thereby allowing the attacker to learn about the memory access patterns of the victim. Transient execution-based side-channel attacks exploit branch misprediction leading to discarded instructions after a pipeline flush [167]. SgxPectre [168] and Foreshadow [169] are examples of transient execution-based side-channel attacks capable of extracting secret keys from the victim TEE. One study suggests using a secure co-processor such as Google's Titan M [170] accessible by TEE but separate from the main processor can circumvent side-channel attacks [158]. The study assumes a secure communication channel between the TEE and the co-processor, which makes the setup free of side-channel attacks. The co-processor is entrusted with performing sensitive cryptographic operations and storing cryptographic secrets.

5.1.3 Energy Mix Data

The total energy consumed in a given geographical region can be broken down by primary energy sources. This mapping between what fraction of consumed energy is generated by which source is known as fuel mix or energy mix [171]. Throughout this chapter, we refer to the renewable energy portion of the energy mix as the *energy mix score*. Thus the energy mix score can be between 0.0 (no fraction of energy was obtained from renewable sources) to 1.0 (all consumed energy was obtained from renewable sources). This is essentially a greenness score of a region at a given time. The higher this score is, the more green the region is, i.e., the proportionally more the region has energy generated by renewable sources.

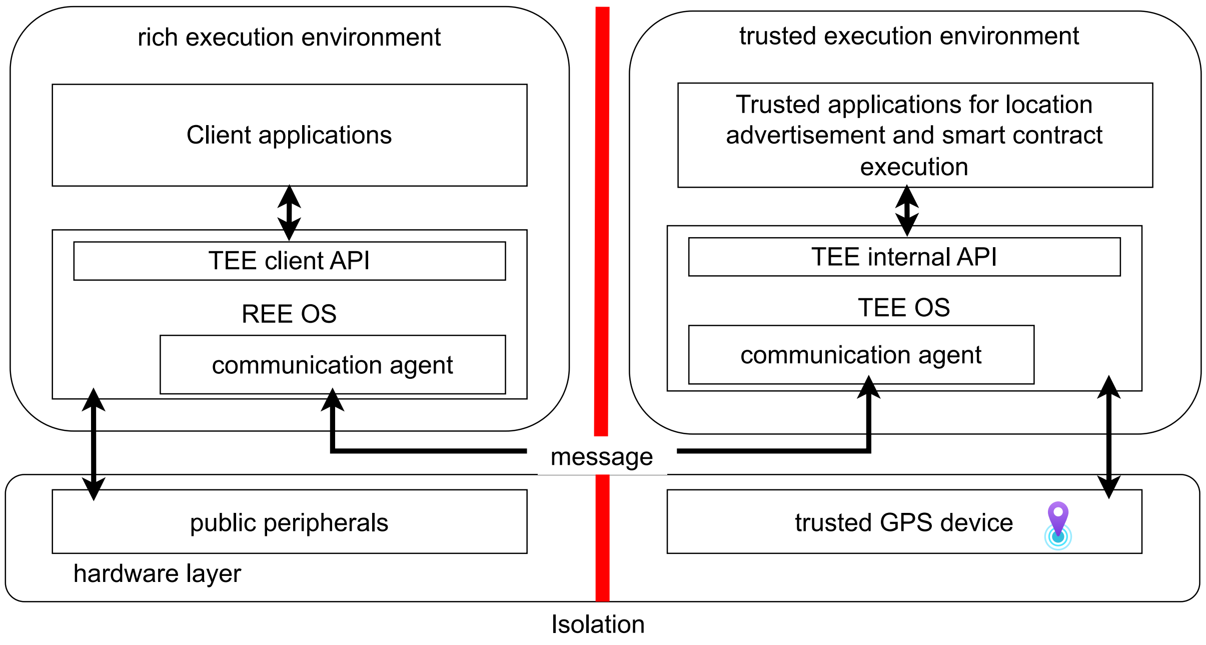
Independent System Operations (ISOs) maintain renewable energy data for different regions, e.g., California ISO [172] maintains renewable energy data for California. We can query these data sources to get an estimate of renewable energy that was available at a specific time and region. Hence, when minting a coin in GreenCoin the proposer can query these sources and assign a score to the minted coin. GreenCoin uses an existing system called Depot that pools multiple ISO data sources and exposes an API to query the sources. Note that traditionally, third-party data is incorporated into blockchain networks using oracles [173]. Due to the absence of a competing system providing data similar to Depot, Depot itself acts as an oracle. In Section 5.2, we describe how Depot and a blockchain node securely communicate with each other.

5.2 GreenCoin System Architecture

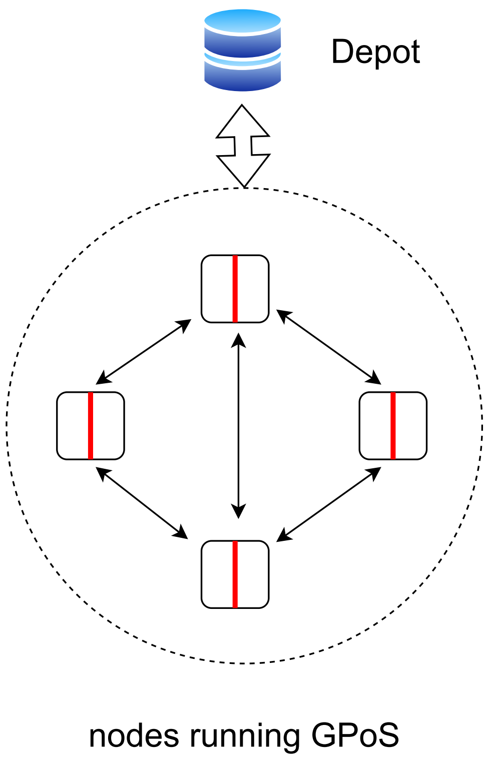
In GreenCoin, the location of a blockchain node, i.e., a machine running the blockchain protocol, plays a significant role. The system favors accounts maintained by nodes with a higher fuel mix score when it comes to endowing rewards, e.g., for block creation, and privileges, e.g., executing smart contracts. Hence, there is a strong coupling between a blockchain node and a blockchain account. Although a blockchain node can maintain multiple different accounts, for the simplicity of exposition, we assume there is a one-to-one mapping between a blockchain node and a blockchain account. Throughout this chapter, we use the terms node and account interchangeably.

Figure 5.1 shows the high-level system architecture for GreenCoin using Green PoS (GPoS). As described in Section 5.1.2, each node runs a TEE. Each node is also equipped with a tamper-proof GPS device. This GPS device is connected to the TEE through a secure I/O channel. Blockchain nodes contact Depot to retrieve their fuel mix score. As TEEs can be resource constrained, we opt to run only functions which need an extra layer of security and trust on TEEs rather than the full blockchain system. There are two cases when we execute operations inside a TEE: for (i) block proposal, and (ii) smart contract execution.

At the time of block creation, a node must determine the fuel mix score for its location. The location data must come from the TEE to ensure the node cannot advertise a false location. Therefore, to retrieve the fuel mix score, the client application first requests the trusted application running inside TEE to read the location data. In response, the trusted application reads the location from the tamper-proof GPS device through the secure I/O provided by the TEE, signs the location reading, and sends it back to the client application. The client application itself cannot tamper with this location data as the TEE signs it with a tamper-proof hardware key. Once it receives the location



(a) Details of a single node running a TEE.



(b) Interaction between nodes and Depot.

Figure 5.1: High level system architecture of GreenCoin.

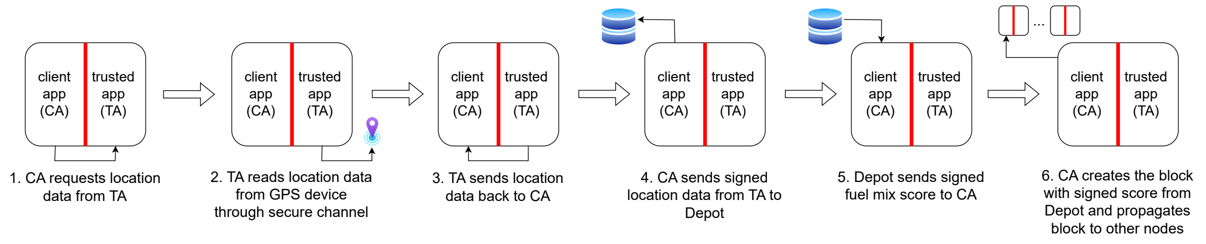

client app (CA)
trusted app (TA)
1. CA requests location data from TA
2. TA reads location data from GPS device through secure channel
3. TA sends location data back to CA
4. CA sends signed location data from TA to Depot
5. Depot sends signed fuel mix score to CA
6. CA creates the block with signed score from Depot and propagates block to other nodes

Figure 5.2: Steps involved in retrieving fuel mix score for block creation.

data, it sends a request encapsulating the location data to Depot to retrieve the fuel mix score. Depot can check the trustworthiness of the trusted application through remote attestation. Once Depot verifies the program run by TEE, it decrypts the location data and sends the signed fuel mix score to the client application. Upon receipt of the fuel mix score, the client application finalizes the block by adding the signed fuel mix score to the block and propagating it to other nodes. The other nodes do not need to execute any extra verification steps, as they can simply decrypt the signed fuel mix score. Figure 5.2 delineates the steps involved in retrieving the fuel mix score as a part of the block creation process.

Smart contracts in GreenCoin are also executed inside TEEs. GreenCoin imposes single-node execution of a smart contract (cf. Section 5.3.6), where the smart contract is executed only by the block proposer, which proposes the block containing the transaction to execute that contract. Hence other nodes must have a mechanism to verify that the proposer executed the intended contract along with the appropriate input and output. GreenCoin uses TEEs to implement this trust mechanism. Whenever a proposed block contains a transaction to execute a contract, the proposer executes the contract in the TEE and propagates supplementary information along with the proposed block to other nodes. The supplementary data includes a verifiable report of the code executed by the TEE, along with its input and output.

In this work, our primary focus is on the underlying blockchain protocol and not the inner workings of off-the-shelf TEEs. Hence, we assume the nodes are running TEEs and implement a proof of concept distributed system executing GPoS using Python. We empirically evaluate GPoS-driven GreenCoin using multiple cloud instances and present our findings in Section 5.4.

5.3 GPoS Implementation

This section describes the details of the GPoS building blocks. We first present how coins are represented and how their greenness score is determined. Then, we define the greenness score of a collection of coins, i.e., the wallet or balance of an account. Next, we describe how stakes in GPoS differ from other PoS protocols. Following that, we present how honest proposers are incentivized through block reward and malicious proposers are disincentivized through slashing. Then, we explain the different aspects of a smart contract. Finally, we wrap up this section with an overview of the cryptoeconomics of GPoS-powered GreenCoin.

5.3.1 Coins

Each coin in GreenCoin is tagged with a greenness score, which signifies the percentage of renewable energy available during the time and at the site of coin generation. Hence, in contrast to other cryptocurrencies, each coin in our system is represented by two values – one to represent the greenness score of the coin and the other to represent the amount of coin. Throughout this chapter, we use the tuple $(score, amount)$ to represent a coin. For example, $(0.5, 10.0)$ represents a coin of amount 10.0 having a score of 0.5. The amount of a coin is divisible, but not the score. For example, the owner of the above coin can decide to transfer half the amount of the coin to another account. In this case, the second account receives the coin $(0.5, 5.0)$ and the first account is left with $(0.5, 5.0)$. However, the first account cannot send $(0.25, 10.0)$ to the second account.

5.3.2 Wallet Score

Algorithm 4 getWalletScore

```
Require: list of coins in $(score, amount)$ tuple format, $C$
Ensure: wallet score
 1: $sumDot \leftarrow 0; sumAmount \leftarrow 0; N \leftarrow \text{LENGTH}(C)$
 2: for $k \leftarrow 1$ to $N$ do
 3:     $c \leftarrow C[k]$
 4:     $sumDot \leftarrow sumDot + c.score \times c.amount$
 5:     $sumAmount \leftarrow sumAmount + c.amount$
 6: end for
 7: if $log(sumAmount) \leq 0$ or $log(sumDot) \leq 0$ then
 8:     return 0
 9: else
10:     return $log(sumDot)/log(sumAmount)$
11: end if
```

Over time, an account may consist of an array of coins with different greenness scores. We term the collection of coins owned by an account as its *account balance* or *wallet* and represent it as a list of coins $[(score_1, amount_1), \ldots, (score_n, amount_n)]$. As our goal is to favor accounts having greener coins, we need a metric to evaluate the overall greenness score of an account, which we term the *wallet score*. We define *wallet score* as the ratio of the logarithm of the dot product between the scores and amounts to the logarithm of the sum of amounts. Algorithm 4 shows the wallet score calculation. Note that our choice of wallet score calculation over a normalized dot product allows us to assign different scores for wallets having coins with the same score but different amounts. For example, two different wallets $[(0.5, 10.0)]$ and $[(0.5, 100.0)]$ have a score of 0.50 according to the normalized dot product. However, our approach differentiates these two cases and assigns the scores 0.70 and 0.84 to the two wallets, respectively. Our method, as evident from line 7 of Algorithm 4, uses only non-negative wallet scores setting the value to 0 when either logarithm is negative.

5.3.3 Stake

In PoS protocols, an account can "stake" or set aside a portion of its balance as collateral. The higher the stake compared to other accounts, the greater the probability of the account becoming a proposer. Malicious behavior by the account results in a portion of the stake being lost, i.e., slashed (cf. Section 5.3.5). As our goal is to favor accounts with higher greenness scores, stake in GPoS is not dependent only on the amount of coins owned by an account but on the overall wallet score of the account. However, an account can not selectively stake its balances, as this can give a false representation of the greenness of an account. For example, the wallet score of an account with balance $[(0.1, 1000), (1.0, 10)]$ is 0.68. If the account had been able to stake a portion of its balance, it could have chosen to stake $[(1.0, 10)]$ and advertise itself as having a greenness score of 1.0. Hence, GPoS does not allow an account to explicitly stake balances, instead, the full wallet acts as the stake. Consequently, slashing impacts the full wallet of the account rather than a portion of it.

5.3.4 Block Reward

A proposer generates a coin c as a reward when a block is successfully proposed with approval from at least 51% of the nodes. The score of c is the fuel mix data received from Depot. This data represents the fraction of energy that is renewable at the time of generation of the coin and the location of the node on which the account is running. The amount of c is the average score of the transactions within the proposed block. Just as a coin has a score, so does a transaction. The score of a transaction

Algorithm 5 endowBlockReward

```
Require: block with embedded signed fuel mix score from Depot, $b$
    minimum reward amount for empty transactions, $m$
Ensure: none, ensures appropriate state change
 1: $amount \leftarrow 0; N \leftarrow$ LENGTH$(b.transactions)$
 2: for $k \leftarrow 1$ to $N$ do
 3:     $tx \leftarrow b.transactions[k]$
 4:     $amount \leftarrow amount + tx.score$
 5: end for
 6: $amount \leftarrow amount/N$
 7: if $amount = 0$ then
 8:     $amount \leftarrow m$
 9: end if
10: UPDATEBALANCE$(b.proposer, b.depotScore, amount)$    ▷ adds
    the coin $(b.depotScore, amount)$ to the block proposer's balance, followed by updating its
    wallet score.
```

depends on its type as presented in Table 5.1. Making the amount of c dependent on the scores of the transactions incentivizes the proposers to include transactions with higher scores in blocks, which in turn encourages all the nodes to maintain a high greenness score throughout the GreenCoin deployment. If the proposed block contains an empty transaction list, a predefined default value is used as the amount of c. Algorithm 5 presents the calculation of block reward.

5.3.5 Slashing

In PoS algorithms, slashing is used to discourage malicious behavior by taking away a portion of the stake of the slashee, i.e., the node being slashed. At the same time, by rewarding a portion of the stake to the slasher, i.e., the block proposer whose block included the slashing transaction, nodes are encouraged to punish malicious behavior actively. In GPoS, slashing performs a similar functionality. However, slashing is relatively complex in GPoS due to the coin being not only a single value but an amount tagged by a score. GPoS slashes a faulty proposer so that its wallet score falls below half its pre-slashed

Algorithm 6 slashAccount

```
Require: public key of slasher, $p_{slasher}$
    public key of slashee, $p_{slashee}$
    list of coins in $(score, amount)$ tuple format owned by slashee, $C_{slashee}$
    minimum amount of coin at or below which the amount is set to 0, $m$
    factor by which amount of coin is reduced at each slashing step, $f$, $(0.0 \leq f < 1.0)$
Ensure: none, ensures appropriate state change
 1: SORT($C_{slashee}$)                                  ▷ sorts $C_{slashee}$ in descending order of score
 2: $S_c \leftarrow$ GETWALLETSCORE($C_{slashee}$)                          ▷ current wallet score
 3: $S_t \leftarrow S_c \times 0.5$                                         ▷ target wallet score
 4: $N \leftarrow$LENGTH($C_{slashee}$)
 5: $C_{initial} \leftarrow$COPY($C_{slashee}$)                          ▷ creates a copy of $C_{slashee}$
 6: $C_{diff} \leftarrow []$                                ▷ list of slashed coins is populated here
 7: for $k \leftarrow 1$ to $N$ do
 8:     while $S_c > S_t$ and $C_{slashee}[k].amount > 0$ do
 9:         if $C_{slashee}[k].amount < m$ then
10:             $C_{slashee}[k].amount \leftarrow 0$
11:         else
12:             $C_{slashee}[k].amount \leftarrow C_{slashee}[k].amount \times f$
13:         end if
14:         $S_c \leftarrow$ GETWALLETSCORE($C_{slashee}$)
15:     end while
16:     $amount' \leftarrow C_{initial}[k].amount - C_{slashee}[k].amount$
17:     APPEND($C_{diff}, (C_{slashee}[k].score, amount')$)       ▷ appends a coin to the list $C_{diff}$
18:     if $S_c \leq S_t$ then
19:         break
20:     end if
21: end for
22: ASSIGNBALANCES($p_{slashee}, C_{slashee}$)    ▷ sets the balance of slashee to $C_{slashee}$, followed by
    updating its wallet score
23: ADDBALANCES($p_{slasher}, C_{diff}$) ▷ adds the coins in $C_{diff}$ to the balance of slasher, followed
    by updating its wallet score
```

Table 5.1: Different types of transactions and their scores.

type	**description**	**score**
TRANSFER	transfers a coin c from one account to another	score of c
SLASH	slashes an account	predefined default
PEN_SLOW	penalizes slow node	predefined default
CREATE_SC	creates a smart contract	product of *score* and *amount* of gas[1]
EXEC_SC	executes a smart contract	product of *score* and *amount* of gas[2]

[1,2] cf. Section 5.3.6

value. Potentially numerous combinations of coins can be taken away to achieve this condition. However, GPoS starts slashing by taking away the coins with higher scores first. At each iteration of slashing, GPoS takes away a portion of the amount of the highest-scored coin and calculates the new wallet score. It stops if the new wallet score is half of the initial score. Otherwise, it continues slashing a portion of the highest-scored coin. If the amount of this coin falls to zero or a predefined minimum value, this coin is completely removed from the account, and the process continues from the next highest coin. Once the set of coins that must be deducted from the malicious node's wallet is determined, those coins are added to the wallet of the slasher as a reward. Algorithm 6 delineates the steps in slashing.

5.3.6 Smart Contract

Smart contracts are an integral part of any general-purpose blockchain which intends to extend beyond the application of cryptocurrency. Traditionally, a smart contract is an executable piece of code that is immutable once created and executes on every node when invoked. This is extremely wasteful but imperative when contract execution cannot be verified by other nodes in the system. Conversely, since GreenCoin aims to be an energy-aware blockchain and stipulates that each node is equipped with a TEE, GreenCoin is capable of imposing a single-node execution model. In this, the account which invokes a smart contract can set a minimum threshold for the executor's energy mix score, and a block proposer can only add the invocation transaction to a block if their energy mix score is greater than or equal to the specified threshold. As discussed in Section 5.2, the block proposer sends supplementary data containing a verifiable report of the code executed by the proposer's TEE along with its input and output during block propagation. The other nodes in the system can verify the execution of the smart contract along with necessary state changes from this supplementary data.

Algorithm 7 getMaxExecutionTime

```
Require: gas in $(score, amount)$ format, $g$
    upper range of execution time, $t_{upper}$
    lower range of execution time, $t_{lower}$
Ensure: maximum execution time in seconds
1: if $g.score \times g.amount \leq 1$ then
2:     return $t_{lower}$
3: end if
4: $n = log(g.score \times g.amount)$
5: $d = log(g.amount)$
6: return $\lceil t_{upper} \times n/d \rceil$
```

Since the languages used to write smart contracts are expressive enough to be Turing complete, they are susceptible to the halting problem. Therefore, smart contract systems generally use the concept of *gas* to ensure termination. Gas refers to the unit of measurement for the computational effort required to execute a particular operation or transaction. If a node executing a smart contract runs out of gas before the computation finishes, the contract execution stops. Since only a finite amount of gas is made available to nodes, the execution is guaranteed to terminate.

In GreenCoin, Gas is an up-front remittance paid by a node R which requests the execution of a smart contract. This is a fee paid by R for using resources on the executor's machine. Gas is represented as a single coin $(score, amount)$. One unit of gas is essentially a coin that is burnt, i.e., not paid to any particular entity but removed from the wallet of R. At the time of execution, gas is used to calculate the CPU time allotted to the contract for execution.

Algorithm 7 shows how we determine the amount of time that can be purchased with a unit of gas. This procedure accepts a predefined range of time $[t_{lower}, t_{upper}]$ and returns a time within this range that can be purchased per unit of gas. If an execution runs out of the time that it paid for, the execution is halted and any state changes that were made are reverted. Given two different units of gas with the same value of $(score \times amount)$, the unit with a higher *score* component yields a longer execution time according to Algorithm 7. For example, consider the case in which $[t_{lower}, t_{upper}]$ is $[1, 10]$ and compare gas $(0.5, 10)$ to gas $(0.8, 6.25)$. Both gas tuples generate the same product of amount and score but the first can afford a maximum execution time of 7 seconds while the latter can afford a maximum execution time of 9 seconds, when computed on line 6 of Algorithm 7. Note that the conditional statement in line 1 of Algorithm 7 imposes a natural cutoff for the combination of *score* and *amount* below which the unit can buy the minimum execution time only.

In GreenCoin, contracts are effectively accounts, known as contract accounts, controlled by the logic in their code rather than by cryptographic key pairs. Contracts can receive coins and send coins programmatically from accounts and contract accounts. Contracts also have access to some additional information: account balances, contract account balances, the states of all contracts, and the invoking transaction's details. This allows contracts to act dynamically based on the current state of the blockchain.

5.3.7 Cryptoeconomics

GreenCoin's principal cryptoeconomic goal is to provide incentives that encourage nodes to use renewable energy for every interaction in the system. We provide four key incentives: (i) Given any two accounts with the same amount of coins, the account having coins with a higher score has a greater probability of proposing blocks and generating coins. (ii) Block rewards are generated based on the proposer's energy mix score and the average score of the transactions in the block. (iii) For this reason, nodes prefer including transactions with higher scores in a block, resulting in a higher probability of such transactions being included. (iv) Given any two units of gas with the same product of *score* and *amount*, more execution time for contracts can be purchased for the unit having a higher *score*.

We disincentivize nodes from exhibiting byzantine behavior by reducing their probability to participate in block proposition and, effectively, lowering their reputation. However, we acknowledge that slashing is a severe penalty as it halves wallet score, which has severe implications on the likelihood of getting selected as a proposer. Thus, slashable offenses must be provably attributable based on an invalid block proposal. An invalid proposal consists of an invalid block number, timestamp, state hash, transaction set, or signature. Additionally, to handle crash faults while disincentivizing being offline for extended periods, nodes incur a minor penalty if they fail to propose a block when they should.

Given these incentives and disincentives, the primary objective of GreenCoin is not to be a medium of exchange, store of value, or unit of account. Instead, GreenCoin's primary objective is to act as a carbon reputation credit, where an account that possesses more coins of a higher score is perceived as more reputable by the protocol. This reputation is reflected in its privileges. As a node continues to act honestly and consume a high amount of renewable energy, its reputation and privileges increase over time. Conversely, if a node exhibits negative behaviors, GreenCoin drastically reduces its reputation by first removing coins with higher scores during slashing.

5.4 Evaluation

In this section, we evaluate multiple aspects of GPoS and their impact on individual nodes in the GreenCoin environment. The prototype implementation of GreenCoin is designed for permissioned deployment in regional settings where users are not anonymized. That is, each user is authenticated into the system and only authenticated users may participate.

This use case originates with Internet of Things (IoT) applications for ecology (cf. [174]) and agriculture (cf. [119, 175]) where regional participants (who are concerned about carbon footprint) wish to use a tamper-proof ledger (with attribution) to share data, but not necessarily with an anonymized global community of users. Examples of data that these users wish to share include data on pest infestations (agriculture), predator movement (ecology), aquifer health, etc. Indeed, we plan to use the GreenCoin prototype with our collaborators as part of ongoing field deployments of IoT applications [176, 177, 178]. We note, however, that GreenCoin and GPoS are not specific to these use cases; we have developed this prototype implementation for these domains to facilitate "real world" validation by and feedback from a user community.

To evaluate GreenCoin, we first study how fuel mix affects block rewards and in turn wallet score of an account. To demonstrate the impact of slashing on wallet score, we then present experimental results for running with both honest and malicious nodes. Finally, we evaluate the impact of the fuel mix score of a region on the ability of a node to execute smart contracts.

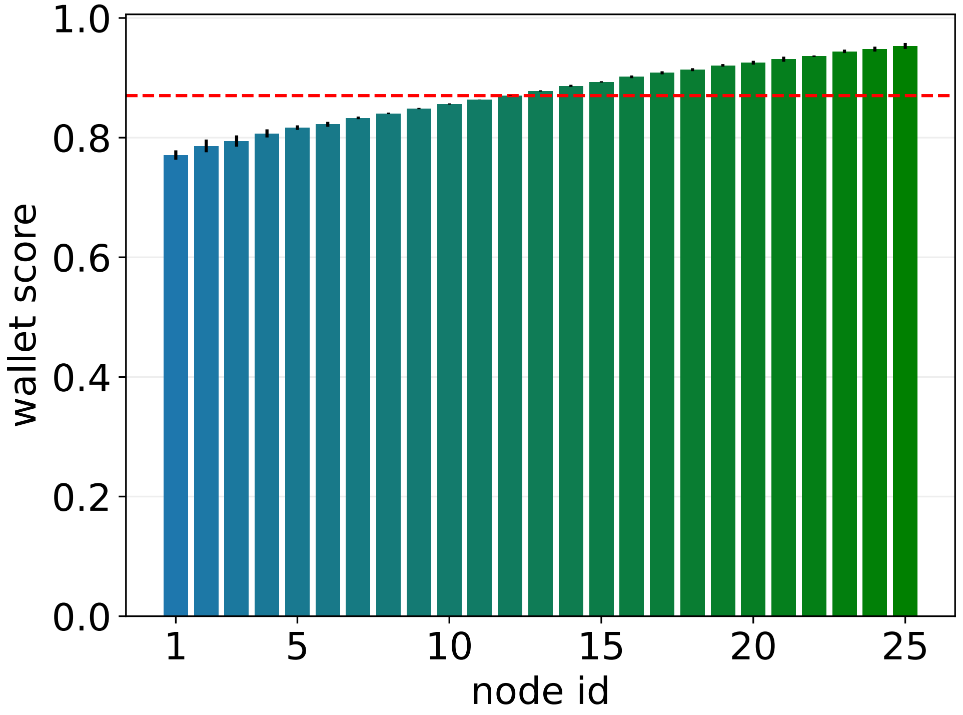


Figure 5.3: Wallet scores after mining 1000 blocks. The red dashed line represents the initial wallet score. Fuel mix range increases from node 1 ($[0.00, 0.04]$) to node 25 ($[0.96, 1.00]$) at an increment of 0.04.

We perform our experiments using 25 virtual machine instances in a private cloud running Eucalyptus [82]. Each instance has a 2GHz CPU and 4GB of memory. Each instance is a node in a GreenCoin deployment. We assume that there are 25 accounts, each tied to a different node for the duration of the experiments. Unless otherwise specified, we perform each experiment 10 times and present the average values. We also present the standard deviation along with the mean when applicable, as error bars.

5.4.1 Block Reward

To demonstrate how the fuel mix score of a region affects block reward, we assume each of the 25 nodes is located in a region with a different fuel mix score. The node with ID 1 has the lowest range of fuel mix scores ($[0.00, 0.04]$). Each subsequent node with a higher ID has a range of fuel mix scores that is 0.04 higher than the previous node. Hence, the node with ID 25 has the highest range of fuel mix scores ($[0.96, 1.00]$). We manipulate the interface to Depot to get a random fuel mix score for each node within the appropriate range. Irrespective of its location, each account starts with a collection of 100 coins total – 10 of each score ranging from 0.1 to 1.0 at an increment of 0.1. This amounts to a wallet score of 0.8702. We run the experiment until 1000 blocks are mined, each block having a single transaction. In each of these transactions, a node sends a coin with a score of 0.5 and an amount of 1.0 to the node with the next higher ID (the last node sends the coin to the first node). We keep the score and the amount of transferred coins constant so that the final wallet score is dependent exclusively on the fuel mix of a region rather than the executed transactions.

Figure 5.3 shows the final wallet score of the nodes. The red dashed line represents the initial wallet score. As expected, nodes situated in regions with higher fuel mix scores experience an increase in their wallet score. The node with the highest range of fuel mix score reaches a wallet score of 0.9533, an increase of 0.0831. Conversely, nodes situated in regions with lower fuel mix scores experience a decrease in their wallet score. The node with the lowest range of fuel mix score reaches a wallet score of 0.7709, a decrease of 0.0993.

5.4.2 Account Slashing

To demonstrate the effect of slashing, we divide the nodes into two groups – 13 honest nodes and 12 malicious nodes. We manipulate the interface to Depot to get random fuel mix scores that fall within the same range for all the nodes ($[0.96, 1.00]$), so that any change in the final wallet score is only due to the slashing of nodes without any effect from disparate block rewards. All the nodes start with the same collection of coins as

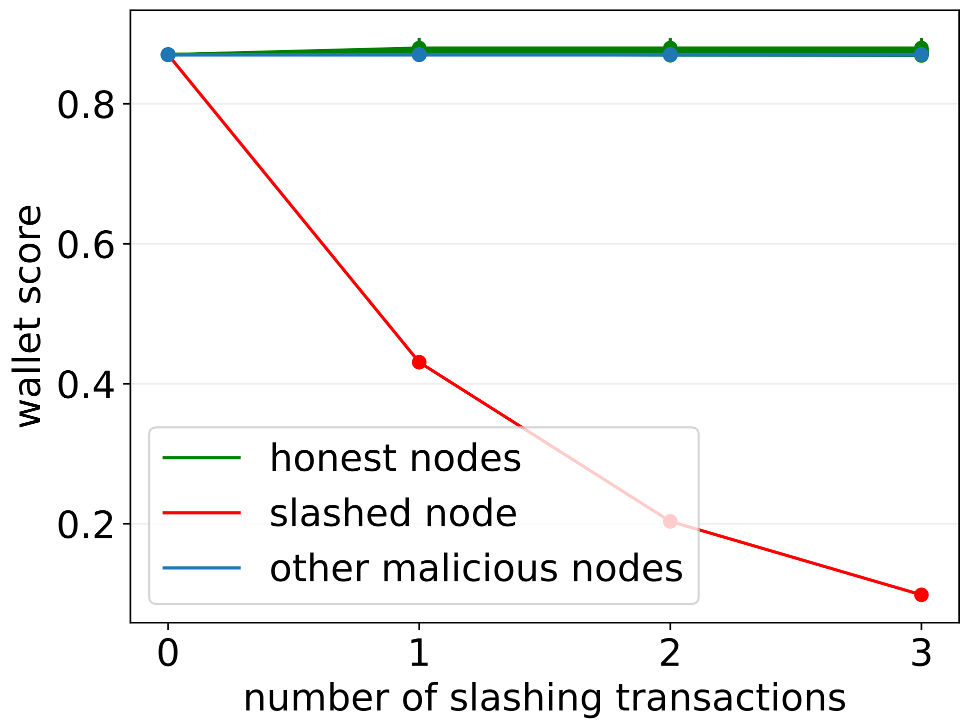


Figure 5.4: Effect of slashing on wallet score of a single malicious proposer. All the nodes have the same range of fuel mix scores.

described in Section 5.4.1. One of the malicious nodes proposes multiple faulty blocks. Each of the blocks contains a single transaction that transfers a coin with a score of 0.5 and an amount of 1.0 from the proposer to the next malicious node. The execution of this small number of transactions itself does not change the wallet score of the proposer significantly. All the malicious nodes vote dishonestly to declare the faulty blocks as valid and any valid block proposed by the honest nodes as invalid. We run the experiment until the wallet score of the faulty node drops below 0.2.

Figure 5.4 shows the wallet scores each time the faulty proposer is slashed. The red line represents the wallet score of the faulty proposer. As described in Section 5.3.5, each time an account is slashed its wallet score falls to half of its pre-slashed value or less. As the faulty proposer starts with a wallet score of 0.8702, four slashing events are enough to make the wallet score drop below 0.2. In this case, it drops to 0.0984. The wallet

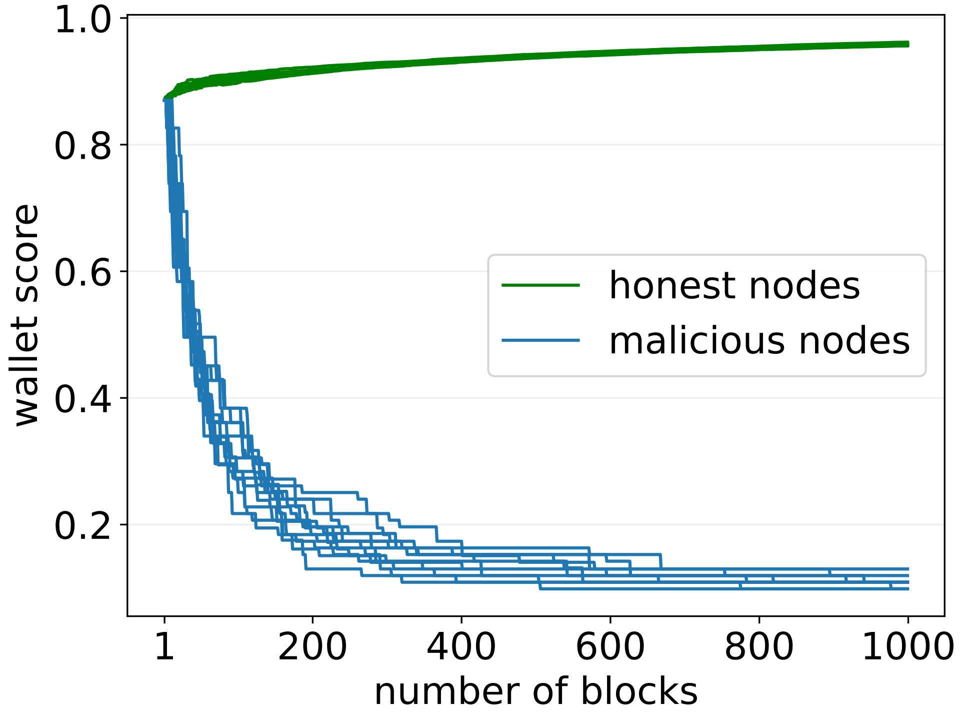


Figure 5.5: Deviation of wallet scores between honest nodes and malicious nodes over a prolonged period. All the nodes have the same range of fuel mix scores.

scores of all other nodes remain relatively unchanged.

To understand how wallet scores of honest and malicious nodes stabilize over time, we construct a second experiment with the same topology and seed account balance as the previous experiment. As in the previous experiment, all malicious nodes vote falsely. Additionally, whenever a malicious node becomes the proposer it proposes a faulty block. This behavior is different from the previous experiment, where only one specific malicious node was proposing faulty blocks and the other malicious nodes were following a more passive approach by only voting falsely. We run the experiment until 1000 blocks are generated. Figure 5.5 shows the deviation in wallet scores between the honest and the malicious nodes. Wallet scores of honest nodes gradually creep up to ~ 0.95 whereas those of malicious nodes fall drastically down to ~ 0.10.

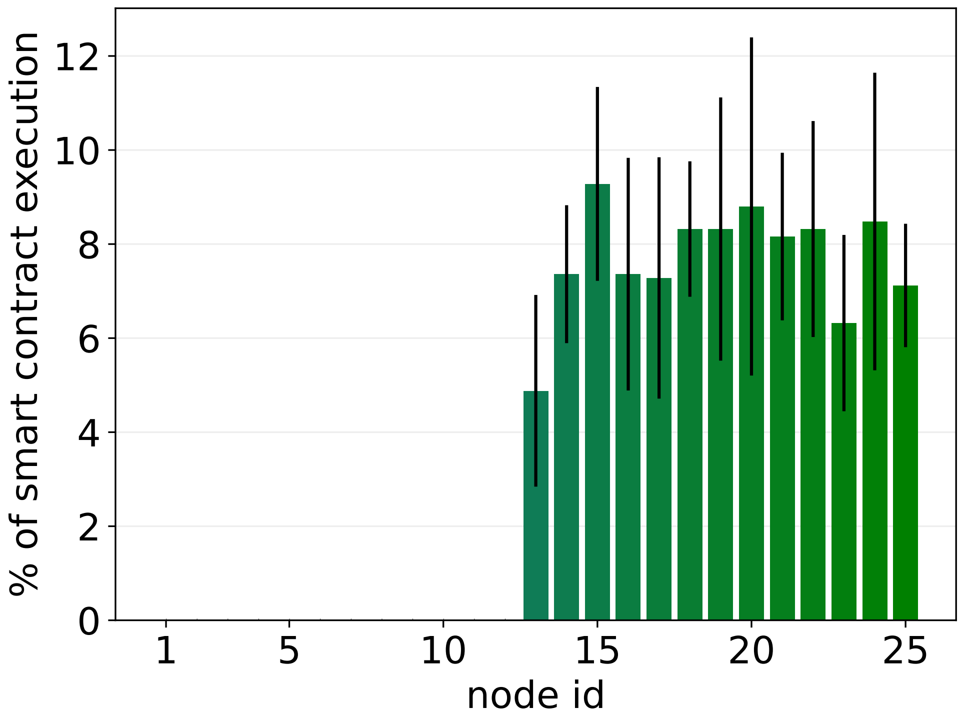


Figure 5.6: Percentage of smart contract execution by each node. The execution threshold of each smart contract is set to 0.5. Nodes with ID 1 to 12 have fuel mix scores falling below 0.5. Fuel mix scores of node with ID 13 straddles 0.5 while other nodes have fuel mix scores higher than 0.5.

5.4.3 Smart Contract Execution

As an example of smart contract execution, we use GreenCoin to compute the Pearson correlation coefficient between two time series of meteorological measurements (air temperature and humidity, in this example). For agricultural applications such as frost prevention where the activation of (possibly expensive) frost prevention measures depends on trustworthy meteorological data, we believe smart contracts will be useful.

We use the El Niño dataset (temperature and humidity) from UCI Machine Learning repository [179] for this experiment. We assume that each participant in the GreenCoin blockchain operates a set of local meteorological sensors and they use GreenCoin to publish the sensor results (taken from the dataset in this fictitious example) to the ledger via a local GreenCoin node.

As in section 5.4.1, we start with all the nodes having a different range of fuel mix scores but the same wallet score. One node creates a contract to log temperature data, one node creates a contract to log humidity data, and another node creates a contract to compute the correlation between temperature and humidity. To make sure all nodes end up creating the same number and similar types of transactions, the other nodes also create such contracts, although the experiment uses only three contracts. Each node creates a transaction to log two temperature and two humidity values, resulting in 50 data points each for temperature and humidity. Finally, each node creates a transaction to compute the Pearson correlation coefficient between temperature and humidity. Therefore, there are 125 transactions executing smart contracts. Each smart contract has an execution threshold of 0.5, i.e., a blockchain node must have a minimum fuel mix score of 0.5 to propose a block containing a transaction to execute a smart contract. As GreenCoin allows execution of a smart contract by the proposer only, it means a node must have a minimum fuel mix score of 0.5 to execute a smart contract. We run the experiment until all the smart contracts have been executed. Each block contains zero or one transaction.

Figure 5.6 shows the percentage of smart contracts executed by each node. Nodes with fuel mix lower than the execution threshold, i.e., 0.5 fail to execute any transaction involving execution of smart contracts. All other nodes end up executing a varied percentage of transactions involving execution of smart contracts as expected.

5.5 Summary

Cryptocurrencies provide new functionality that improves information trustworthiness and integrity. Their energy efficiency is the subject of ongoing research and commercial development. To this end, we have investigated GreenCoin – a cryptocurrency that uses attested location to attach indelible attributes to each coin representing renewable energy usage during its creation. GreenCoin relies on GPoS, which is a new Proof-of-Stake protocol for marking each coin with a "greenness" score, determining wallet score, slashing malicious behavior, and implementing smart contracts that only execute computations when the coins that pay for them have green scores above a specified threshold.

We validate our assumptions using a permissioned prototype of GreenCoin. Our results indicate that our approach effectively implements coin scoring, incentivizes both truthful and "green" participation, and implements a smart contract with "computational accountability" with respect to the use of renewable energy in cryptocurrency systems.

As part of our future work, we plan to deploy GreenCoin in IoT application deployments as part of ongoing collaborative work. We also plan to investigate its feasibility in implementing renewable accountability for permissionless blockchains and cryptocurrencies.

Chapter 6

Conclusion and Future Work

The proliferation of IoT applications in recent years has made IoT increasingly pervasive and ubiquitous. This has inevitably contributed to a data explosion. The availability of a high volume of data, along with recent advancements in fields like data analytics and machine learning, has made IoT applications data-driven. Due to their diverse nature and requirements, modern data-driven IoT deployments follow a multi-tiered architecture (cloud/edge/device). These IoT applications must tackle unique challenges such as heterogeneity, unstable network connectivity, unreliable power source, etc. Due to these challenges, we cannot make IoT deployments robust and fault-tolerant using the same techniques we use for cloud deployments only.

In this thesis, we study multiple ways to make IoT deployments reliable, fault-tolerant, and trustworthy. We present an overview of the current state-of-the-art technologies to achieve these goals and their limitations. In light of our discussion, we pose the thesis question: ***Can we leverage data versioning and replication to make distributed IoT systems reliable, fault-tolerant, and trustworthy?*** Our answer is positive After extensive system design, implementation, and evaluation. In particular, we identify and practically apply three core concepts to seamlessly make the next generation of data-driven IoT applications robust, fault-tolerant, and trustworthy. First, we propose using *versioned data* for efficiently tracking the evolution of data and facilitating system debugging. Second, we put forth novel formulations for *replicated data* geared towards availability that reduce coordination overhead. Finally, we propose a new energy-efficient and renewable energy-conscious blockchain technology to ensure *tamper-proof data*.

PEDaLS is our approach to making linked program data structures versioned. It translates an existing efficient versioning algorithm for single-machine, in-memory data structures to an algorithm for a distributed, storage-persistent one. PEDaLS uses append-only logs for its storage. Append-only logs provide additional robustness to the system due to their inherent features, such as immutability, coordination avoidance, etc. As PEDaLS exposes a simple API to the end-user, which is consistent across device scale, it can run on a wide variety of heterogeneous devices. Our results indicate that PEDaLS has the same space/time complexity as the original in-memory algorithm. Moreover, PEDaLS outperforms popular database storage systems for applications involving a high volume of temporal queries.

LSCRDT is our answer to the problem of efficiently replicating data in typical IoT environments where the network connections are unstable and power sources are unreliable. In such environments, protocols that require extensive coordination are inefficient and take longer to converge. LSCRDT prioritizes availability and efficiency over strong consistency by reducing communication overhead. As many IoT applications require only weaker consistency semantics, LSCRDT is favorable over other protocols. Moreover, LSCRDT improves upon extant CRDT protocols and removes several of their limitations. Our results suggest that in write-heavy workloads, LSCRDTs are favorable to CRDTs.

GreenCoin is our solution to provide tamper-proof data to the end user. It uses a novel renewable energy-aware PoS algorithm called GPoS which favors nodes located in regions with higher availability of renewable energy. Moreover, GreenCoin and GPoS allow us to use green smart contracts, where a node must possess a minimum greenness score to execute a contract. Our proof of concept implementation of a permissioned blockchain system using GreenCoin and GPoS indicates that our approach can effectively implement coin scoring, incentive, and slashing mechanisms.

In light of our research, we identify possible future work in each of the three fields of versioned data, replicated data, and tamper-proof data. PEDaLS follows the original in-memory algorithm for versioning linked data structures with constant in-degree. Future works can be done to formulate algorithms for versioning linked data structures with variable in-degree, e.g., graphs. Moreover, PEDaLS implements partially persistent data structures. Possible extensions to PEDaLS include fully persistent and confluently persistent data structures.

Although LSCRDT guarantees convergence of an arbitrary data structure by ordering operations, convergence itself does not mean the user expectations on the semantics are satisfied. As an example, the consistency of a conventional set can be achieved by consistent ordering of operations. However, the consistency for the more constrained OR-set following an add-wins strategy might violate the expected semantics, as a delete can supersede a concurrent add if the former is ordered after the latter. In the future, we plan to devise mechanisms to respect the expectations on the semantics of the underlying data structure as well.

Our current proof of concept GreenCoin implementation is deployed as a permissioned blockchain system. Although we envision the underlying principles of GreenCoin will also apply to permissionless blockchain systems, future work can involve empirical studies to either support or contradict this notion. Moreover, we plan to deploy GreenCoin at a broader scale as part of our ongoing collaborative work [134, 135]. Finally, future work on GreenCoin can comprise the deployment of GPoS in TEEs.

Bibliography

[1] P. Helland, *Immutability changes everything*, in *Conference on Innovative Data Systems Research*, 2015.
http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf Accessed 15-Sep-2019.

[2] P. Alvaro, S. Galwani, and P. Bailis, *Research for practice: Tracing and debugging distributed systems; programming by examples*, in *CACM*, Jan., 2017.

[3] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, *CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT*, in *ACM Symposium on Edge Computing*, 2019.

[4] Amazon, *S3 Object Versioning*, 2019.
https://docs.aws.amazon.com/AmazonS3/latest/dev/Versioning.html [Online; accessed 28-Sep-2019].

[5] *Amazon S3*, 2018. https://aws.amazon.com/s3/ [Online; accessed 28-Sep-2018].

[6] *Google Cloud: Versioned Object Storage*, 2018.
https://cloud.google.com/storage/docs/object-versioning [Online; accessed 12-Sep-2018].

[7] *Apache Kafka*, 2019. http://kafka.apache.org [Online; accessed Sep. 2019].

[8] *Amazon kinesis streams service*, 2020.
https://docs.aws.amazon.com/kinesis/index.html Accessed 15-Apr-2020.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, *Conflict-free replicated data types*, in *Symposium on Self-Stabilizing Systems*, pp. 386–400, Springer, 2011.

[10] N. Preguiça, C. Baquero, and M. Shapiro, *Conflict-free replicated data types crdts*, in *Encyclopedia of Big Data Technologies* (S. Sakr and A. Y. Zomaya, eds.), pp. 491–500. Springer International Publishing, Cham, 2019.

[11] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, *A comprehensive study of convergent and commutative replicated data types*, Tech. Rep. RR-7506, Inria, 2011.

[12] W. Yu, V. Elvinger, and C.-L. Ignat, *A generic undo support for state-based crdts*, in *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[13] D. Meissner, B. Erb, F. Kargl, and M. Tichy, *Retro-lambda: An event-sourced platform for serverless applications with retroactive computing support*, in *Intl. Conf. on Distributed and Event-based Systems*, 2018.

[14] W.-T. Lin, F. Bakir, C. Krintz, R. Wolski, and M. Mock, *Data repair for Distributed, Event-based IoT Applications*, in *ACM International Conference on Distributed and Event-Based Systems*, 2019.

[15] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, *Making data structures persistent*, *J. Comput. Syst. Sci.* **38** (1989), no. 1.

[16] A. Fiat and H. Kaplan, *Making data structures confluently persistent*, in *Symposium on Discrete Algorithms*, 2001.

[17] P. F. Dietz and R. Raman, *Persistence, amortization and randomization*, in *ACM-SIAM Symposium on Discrete Algorithms*, 1991.

[18] G. S. Brodal, *Partially persistent data structures of bounded degree with constant update time*, *Nord. J. Comput.* **3** (1996), no. 3.

[19] H. Kaplan, *Persistent Data Structures*, 2004.

[20] F. Pluquet, S. Langerman, A. Marot, and R. Wuyts, *Implementing partial persistence in object-oriented languages*, in *Meeting on Algorithm Engineering & Expermiments*, 2008.

[21] L. Ceze, C. von Praun, C. Cascaval, P. Montesinos, and J. Torrellas, *Programming and Debugging Shared Memory Programs with the Data Coloring*, in *Workshop on Compilers for Parallel Computing*, 2009.

[22] E. D. Demaine, J. Iacono, and S. Langerman, *Retroactive data structures*, *ACM Trans. Algorithms* **3** (May, 2007).

[23] E. D. Demaine, J. Iacono, and S. Langerman, *Retroactive data structures*, in *ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[24] H. Mannila and E. Ukkonen, *The set union problem with backtracking*, *International Colloquium on Automata, Languages and Programming* **226** (1986).

[25] J. Westbrook and R. E. Tarjan, *Amortized analysis of algorithms for set union with backtracking*, *SIAM J. Comput.* **18** (1989).

[26] Y. Zhan and D. E. Porter, *Versioned programming: A simple technique for implementing efficient, lock-free, and composable data structures*, in *ACM International on Systems and Storage Conference*, 2016.

[27] *Haskell*, 2019. "https://www.haskell.org" Accessed 17-Sep-2019.

[28] *Immutable.js*, 2019. "https://immutable-js.github.io/immutable-js/" Accessed 20-Sep-2019.

[29] John McClean, *Java Persistent Collections*, 2019. "https://medium.com/@johnmcclean/the-rise-and-rise-of-java-functional-data-structures-63782436f93b" Accessed 20-Sep-2019.

[30] C. Okasaki, *Purely Functional Data Structures*, Tech. Rep. CMU-CS-96-177, Carnegie Mellon University, 2019. `https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf` Accessed 20-Sep-2019.

[31] "Git." "https://git-scm.com/".

[32] "GitHub." `https://github.com/`. [Online; accessed 14-October-2013].

[33] L. Lamport *et. al.*, *Paxos made simple*, *ACM Sigact News* **32** (2001), no. 4 18–25.

[34] D. Ongaro and J. Ousterhout, *In search of an understandable consensus algorithm*, in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pp. 305–319, 2014.

[35] F. Houshmand and M. Lesani, *Hamsaz: replication coordination analysis and synthesis*, *Proceedings of the ACM on Programming Languages* **3** (2019), no. POPL 1–32.

[36] X. Li, F. Houshmand, and M. Lesani, *Hampa: Solver-aided recency-aware replication*, in *International Conference on Computer Aided Verification*, pp. 324–349, Springer, 2020.

[37] M. Simić, M. Stojkov, G. Sladić, and B. Milosavljević, *Crdts as replication strategy in large-scale edge distributed system: An overview*, 2020.

[38] J. Eberhardt, D. Ernst, and D. Bermbach, *Smac: State management for geo-distributed containers*, *Technische Universitaet Berlin* (2016).

[39] O. Qayyum and W. Yu, *Toward replicated and asynchronous data streams for edge-cloud applications*, in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pp. 339–346, 2022.

[40] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, *Smart locks: Lessons for securing commodity internet of things devices*, in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pp. 461–472, 2016.

[41] M. I. Naas, L. Lemarchand, P. Raipin, and J. Boukhobza, *Iot data replication and consistency management in fog computing*, *Journal of Grid Computing* **19** (2021), no. 3 1–25.

[42] N. Golubovic, R. Wolski, C. Krintz, and M. Mock, *Improving the accuracy of outdoor temperature prediction by iot devices*, in *2019 IEEE International Congress on Internet of Things (ICIOT)*, pp. 117–124, IEEE, 2019.

[43] M. Diogo, B. Cabral, and J. Bernardino, *Consistency models of nosql databases*, *Future Internet* **11** (2019), no. 2 43.

[44] A. Charapko, A. Ailijiang, and M. Demirbas, *Pigpaxos: Devouring the communication bottlenecks in distributed consensus*, in *Proceedings of the 2021 International Conference on Management of Data*, pp. 235–247, 2021.

[45] F. Nawab, D. Agrawal, and A. El Abbadi, *Dpaxos: Managing data closer to users for low-latency and mobile applications*, in *Proceedings of the 2018 International Conference on Management of Data*, pp. 1221–1236, 2018.

[46] B. Stopford, *Designing Event Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka.* O'Reilly Media, 2018. `https://drive.google.com/file/d/1NGst29pUjZwtn8pXTKvlSSuau2-to5dD/view` Accessed 15-Sep-2019.

[47] R. Kotla, L. Alvisi, and M. Dahlin, *Safestore: A durable and practical storage system*, in *USENIX Annual Technical Conference*, pp. 129–142, 2007.

[48] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, *The Hadoop Distributed File System*, in *IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[49] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li, *Storage management in asterixdb*, *VLDB* **7** (June, 2014).

[50] C. Gong, S. He, Y. Gong, and Y. Lei, *On integration of appends and merges in log-structured merge trees*, in *International Conference on Parallel Processing*, 2019.

[51] A. Twigg, A. Byde, G. Milos, T. Moreton, J. Wilkes, and T. Wilkie, *Stratified b-trees and versioned dictionaries*, in *USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, 2011.

[52] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. Davis, *Corfu: A shared log design for flash clusters*, in *USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[53] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, *Chariots: A scalable shared log for data management in multi-datacenter cloud environments.*, in *EDBT*, pp. 13–24, 2015.

[54] H. Vo, S. Wang, D. Agrawal, G. Chen, and B. Ooi, *Logbase: a scalable log-structured database system in the cloud*, *Proceedings of the VLDB Endowment* **5** (2012), no. 10 1004–1015.

[55] *Apache Samza*, 2019. `http://samza.apache.org` [Online; accessed Sep. 2019].

[56] P. Bailis and A. Ghodsi, *Eventual consistency today: Limitations, extensions, and beyond*, *ACM Queue* **11** (Mar., 2013).

[57] S. Burckhardt, *Principles of eventual consistency*, *Foundations and Trends in Programming Languages* **1** (2014), no. 1-2.

[58] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. Davis, S. Rao, T. Zou, and A. Zuck, *Tango: Distributed Data Structures over a Shared Log*, in *Symposium on Operating System Principles*, Nov., 2013.

[59] J. Kreps, N. Narkhede, J. Rao, *et. al.*, *Kafka: A distributed messaging system for log processing*, in *Proceedings of the NetDB*, vol. 11, pp. 1–7, 2011.

[60] Facebook, *LogDevice*, 2020. `https://engineering.fb.com/core-data/logdevice-a-distributed-data-store-for-logs/` Accessed 29-Feb-2020.

[61] "AWS Lambda." `https://aws.amazon.com/lambda/`, 2017. [Online; accessed 12-Sep-2017].

[62] A. Chien, P. Balaji, P. Beckman, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, R. Schreiber, J. Hammond, J. Dinan, I. Laguna, D. Richards, A. Dubey, B. van Straalen, M. Hoemmen, M. Heroux, K. Teranishi, and A. Siegel, *Versioned distributed arrays for resilience in scientific applications*, *Procedia Comput. Sci.* **51** (Sept., 2015).

[63] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, *Consistent and durable data structures for non-volatile byte-addressable memory*, in *USENIX Conference on File and Stroage Technologies*, 2011.

[64] M. Atkinson and R. Morrison, *Orthogonally persistent object systems*, *The VLDB Journal* **4** (1995), no. 3.

[65] Oracle, *Java Persistence API*, 2019.
"https://docs.oracle.com/cd/E19798-01/821-1841/6nmq2cpag/index.html"
Accessed 18-Sep-2019.

[66] Oracle, *JDBC*, 2020.
`https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/` Accessed 29-Feb-2020.

[67] R. Agarwal, *The c++ interface in objectivity*, *SIGPLAN Not.* **29** (Dec., 1994).

[68] T. Kelly, *Persistent Memory Programming on Conventional Hardware*, *ACMQUEUE* **17** (2019), no. 4.

[69] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, *Persistency for synchronization-free regions*, in *ACM Conference on Programming Language Design and Implementation*, PLDI 2018, 2018.

[70] M. Atkinson, P. Bailey, K. Chisholm, W. Cockshott, and R. Morrison, *An Approach to Persistent Programming*, *Computer Journal* **26** (1983), no. 4.

[71] M. Atkinson, L. Daynes, M. Jordan, T. Printezis, and S. Spence, *An orthogonally persistent Java*, in *SIGMOD*, 1996.

[72] S. Balzer, *Contracted Persistent Object Programming*, in *PhD Workshop, ECOOP*, 2005.

[73] P. Helland, *Data on the outside versus data on the inside*, in *Conference on Innovative Data Systems Research*, 2015.
`http://cidrdb.org/cidr2005/papers/P12.pdf` Accessed 15-Sep-2019.

[74] W. Lin, C. Krintz, R. Wolski, M. Zhang, X. Cai, T. Li, W. Xu, and R. Zhou, *Tracking Causal Order in AWS Lambda Applications*, in *IEEE International Conference on Cloud Engineering*, June, 2018.

[75] W.-T. Lin, C. Krintz, and R. Wolski, *Tracing Function Dependencies Across Clouds*, in *IEEE Cloud*, 2018.

[76] J. Mace, R. Roelke, and R. Fonseca, *Pivot tracing: Dynamic causal monitoring for distributed systems*, *ACM Trans. Comput. Syst.* **35** (Dec., 2018).

[77] I. Beschastnikh, P. Wang, Y. Brun, and M. Ernst, *Debugging distributed systems*, in *CACM*, June, 2016.

[78] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, *Mining temporal invariants from partially ordered logs*, *SIGOPS Oper. Syst. Rev.* **45** (Jan., 2012).

[79] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, *Friday: Global comprehension for distributed replay*, in *NSDI*, 2007.

[80] D. Bailis, "Coordination avoidance in distributed databases." Ph.D. Dissertation, University of California, Berkeley, `http://www.bailis.org/papers/bailis-thesis.pdf` Accessed 15-Sep-2019, 2015.

[81] "Redis." "http://redis.io".

[82] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, *The eucalyptus open-source cloud-computing system*, in *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pp. 124–131, IEEE, 2009.

[83] "CityPulse Smart City Datasets - Datasets." "http://iot.ee.surrey.ac.uk:8080/datasets.html" [Online; accessed 20-Sep-2022].

[84] A. R. Elias, N. Golubovic, C. Krintz, and R. Wolski, *Where's The Bear? – Automating Wildlife Image Processing Using IoT and Edge Cloud Systems*, Tech. Rep. UCSB-CS-2016-07, Computer Science Department at the University of California, Santa Barbara, October, 2016.

[85] S. A. Bishop, H. I. Okagbue, P. E. Oguntunde, A. A. Opanuga, and O. Odetunmibi, *Survey dataset on analysis of queues in some selected banks in ogun state, nigeria*, *Data in brief* **19** (2018) 835–841.

[86] M. Sharma, V. D. Sharma, and M. M. Bundele, *Performance analysis of rdbms and no sql databases: Postgresql, mongodb and neo4j*, in *2018 3rd International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*, pp. 1–5, IEEE, 2018.

[87] M.-G. Jung, S.-A. Youn, J. Bae, and Y.-L. Choi, *A study on data input and output performance comparison of mongodb and postgresql in the big data environment*, in *2015 8th international conference on database theory and application (DTA)*, pp. 14–17, IEEE, 2015.

[88] G. Harrison and M. Harrison, *Tuning aggregation pipelines*, in *MongoDB Performance Tuning*, pp. 155–183. Springer, 2021.

[89] P. Gupta, M. J. Carey, S. Mehrotra, and o. Yus, *Smartbench: a benchmark for data management in smart spaces*, *Proceedings of the VLDB Endowment* **13** (2020), no. 12 1807–1820.

[90] H. Homburger, M. K. Schneider, S. Hilfiker, and A. Lüscher, *Inferring behavioral states of grazing livestock from high-frequency position data alone*, *PLoS One* **9** (2014), no. 12 e114522.

[91] K. Zhao and R. Jurdak, *Understanding the spatiotemporal pattern of grazing cattle movement*, *Scientific reports* **6** (2016), no. 1 1–8.

[92] N. Saquib, C. Krintz, and R. Wolski, *Pedals: Persisting versioned data structures*, in *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 179–190, IEEE, 2021.

[93] P. S. Almeida, A. Shoker, and C. Baquero, *Delta state replicated data types*, *Journal of Parallel and Distributed Computing* **111** (2018) 162–173.

[94] M. Zawirski, C. Baquero, A. Bieniusa, N. Preguiça, and M. Shapiro, *Eventually consistent register revisited*, in *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, pp. 1–3, 2016.

[95] S. Dolan, *Brief announcement: The only undoable crdts are counters*, in *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pp. 57–58, 2020.

[96] G. Younes, P. S. Almeida, and C. Baquero, *Compact resettable counters through causal stability*, in *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, pp. 1–3, 2017.

[97] W. Yu and S. Rostad, *A low-cost set crdt based on causal lengths*, in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pp. 1–6, 2020.

[98] R. Brown, Z. Lakhani, and P. Place, *Big (ger) sets: decomposed delta crdt sets in riak*, in *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, pp. 1–5, 2016.

[99] V. Enes, P. S. Almeida, C. Baquero, and J. Leitão, *Efficient synchronization of state-based crdts*, in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 148–159, IEEE, 2019.

[100] J. Bauwens and E. G. Boix, *Improving the reactivity of pure operation-based crdts*, in *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, pp. 1–6, 2021.

[101] M. Kleppmann and A. R. Beresford, *A conflict-free replicated json datatype*, *IEEE Transactions on Parallel and Distributed Systems* **28** (2017), no. 10 2733–2746.

[102] M. Kleppmann, D. P. Mulligan, V. B. Gomes, and A. R. Beresford, *A highly-available move operation for replicated trees and distributed filesystems*, 2020.

[103] K. De Porre, F. Myter, C. De Troyer, C. Scholliers, W. De Meuter, and E. G. Boix, *Putting order in strong eventual consistency*, in *IFIP International Conference on Distributed Applications and Interoperable Systems*, pp. 36–56, Springer, 2019.

[104] S. Weiss, P. Urso, and P. Molli, *Logoot-undo: Distributed collaborative editing system on p2p networks*, *IEEE transactions on parallel and distributed systems* **21** (2010), no. 8 1162–1174.

[105] B. Nédelec, P. Molli, A. Mostefaoui, and E. Desmontils, *Lseq: an adaptive structure for sequences in distributed collaborative editing*, in *Proceedings of the 2013 ACM symposium on Document engineering*, pp. 37–46, 2013.

[106] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, *Replicated abstract data types: Building blocks for collaborative applications*, *Journal of Parallel and Distributed Computing* **71** (2011), no. 3 354–368.

[107] X. Lv, F. He, Y. Cheng, and Y. Wu, *A novel crdt-based synchronization method for real-time collaborative cad systems*, *Advanced Engineering Informatics* **38** (2018) 381–391.

[108] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski, *Specification and complexity of collaborative text editing*, in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pp. 259–268, 2016.

[109] V. A. Barros, J. C. Estrella, L. B. Prates, and S. M. Bruschi, *An iot-daas approach for the interoperability of heterogeneous sensor data sources*, in *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pp. 275–279, 2018.

[110] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, *A commutative replicated data type for cooperative editing*, in *2009 29th IEEE International Conference on Distributed Computing Systems*, pp. 395–403, IEEE, 2009.

[111] G. Oster, P. Urso, P. Molli, and A. Imine, *Data consistency for p2p collaborative editing*, in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pp. 259–268, 2006.

[112] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, in *Concurrency: the Works of Leslie Lamport*, pp. 179–196. ACM, 2019.

[113] Basho, "Bitcask." `https://docs.riak.com/riak/kv/2.2.3/setup/planning/backend/bitcask/index.html`.

[114] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, *Making geo-replicated systems fast as possible, consistent when necessary*, in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pp. 265–278, 2012.

[115] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, *Transactional storage for geo-replicated systems*, in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 385–400, 2011.

[116] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, *Consistency-based service level agreements for cloud storage*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 309–324, 2013.

[117] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, *Putting consistency back into eventual consistency*, in *Proceedings of the Tenth European Conference on Computer Systems*, pp. 1–16, 2015.

[118] C. Baquero, "Delta-enabled crdts." `https://github.com/CBaquero/delta-enabled-crdts`, 2015.

[119] N. Golubovic, R. Wolski, C. Krintz, and M. Mock, *Improving the Accuracy of Outdoor Temperature Prediction by IoT Devices*, in *IEEE Conference on IoT*, 2019.

[120] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, *Decentralized business review* (2008) 21260.

[121] A. Back *et. al.*, *Hashcash-a denial of service counter-measure*, .

[122] J. Reed, *Litecoin: An introduction to litecoin cryptocurrency and litecoin mining*, 2017.

[123] I. Eyal and E. G. Sirer, *Majority is not enough: Bitcoin mining is vulnerable*, *Communications of the ACM* **61** (2018), no. 7 95–102.

[124] "Dogecoin – an open-source peer-2-peer digital currency." `https://dogecoin.com/`. Accessed: 17-Apr-2023.

[125] "Home — monero – secure, private, untraceable." `https://www.getmonero.org/`. Accessed: 17-Apr-2023.

[126] M. Rosenfeld *et. al.*, *Overview of colored coins*, *White paper, bitcoil. co. il* **41** (2012) 94.

[127] "Solana — web3 infrastructure for everyone." `https://solana.com/`. Accessed: 17-Apr-2023.

[128] "Cardano — home." https://cardano.org/. Accessed: 17-Apr-2023.

[129] "Polkadot : Web3 interoperability — decentralized blockchain." https://polkadot.network/. Accessed: 17-Apr-2023.

[130] "The merge is here: Ethereum has switched to proof of stake." https://www.technologyreview.com/2022/09/15/1059520/the-merge-is-here-ethereum-has-switched-to-proof-of-stake/. Accessed: 17-Apr-2023.

[131] J. Ménétrey, C. Göttel, A. Khurshid, M. Pasin, P. Felber, V. Schiavoni, and S. Raza, *Attestation mechanisms for trusted execution environments demystified*, in *Distributed Applications and Interoperable Systems: 22nd IFIP WG 6.1 International Conference, DAIS 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, pp. 95–113, Springer, 2022.

[132] USEIA, *Us energy information administration web site*, 2023. https://www.eia.gov.

[133] *The depot data lake software repository*, 2023. https://github.com/MAYHEM-Lab/Depot.

[134] *The University of California, Santa Barbara, RiPiT Project*, 2023. https://sites.cs.ucsb.edu/ rich/ripit.html.

[135] *The University of Chicago RiPiT Project*, 2023. http://ripit.uchicago.edu.

[136] B. Sriman, S. Ganesh Kumar, and P. Shamili, *Blockchain technology: Consensus protocol proof of work and proof of stake*, in *Intelligent Computing and Applications: Proceedings of ICICA 2019*, pp. 395–406, Springer, 2021.

[137] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, *A survey of distributed consensus protocols for blockchain networks*, *IEEE Communications Surveys & Tutorials* **22** (2020), no. 2 1432–1465.

[138] B. Cao, Z. Zhang, D. Feng, S. Zhang, L. Zhang, M. Peng, and Y. Li, *Performance analysis and comparison of pow, pos and dag based blockchains*, *Digital Communications and Networks* **6** (2020), no. 4 480–485.

[139] O. Vashchuk and R. Shuwar, *Pros and cons of consensus algorithm proof of stake. difference in the network safety in proof of work and proof of stake*, *Electronics and Information Technologies* **9** (2018), no. 9 106–112.

[140] A. De Vries, *Bitcoin's growing energy problem*, *Joule* **2** (2018), no. 5 801–805.

[141] V. Kohli, S. Chakravarty, V. Chamola, K. S. Sangwan, and S. Zeadally, *An analysis of energy consumption and carbon footprints of cryptocurrencies and possible solutions*, *Digital Communications and Networks* **9** (2023), no. 1 79–89.

[142] Z. Wang, Q. Li, J. Song, H. Wang, and L. Sun, *Towards ip-based geolocation via fine-grained and stable webcam landmarks*, in *Proceedings of The Web Conference 2020*, pp. 1422–1432, 2020.

[143] P. Gill, Y. Ganjali, B. Wong, and D. Lie, *Dude, where's that ip? circumventing measurement-based ip geolocation*, in *Proceedings of the 19th USENIX conference on Security*, pp. 16–16, 2010.

[144] P. Callejo, M. Gramaglia, R. Cuevas, and A. Cuevas, *A deep dive into the accuracy of ip geolocation databases and its impact on online advertising*, *IEEE Transactions on Mobile Computing* (2022).

[145] "Ip geolocation and online fraud prevention — maxmind." `https://www.maxmind.com/en/home`. Accessed: 15-Apr-2023.

[146] "American registry for internet numbers." `https://www.arin.net/`. Accessed: 15-Apr-2023.

[147] "Ripe network coordination center." `https://www.ripe.net/`. Accessed: 15-Apr-2023.

[148] M. Cozar, D. Rodriguez, J. M. Del Alamo, and D. Guaman, *Reliability of ip geolocation services for assessing the compliance of international data transfers*, in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 181–185, IEEE, 2022.

[149] E. Katz-Bassett, J. P. John, A. Krishnamurthy, D. Wetherall, T. Anderson, and Y. Chawathe, *Towards ip geolocation using delay and topology measurements*, in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pp. 71–84, 2006.

[150] B. Eriksson, P. Barford, J. Sommers, and R. Nowak, *A learning-based approach for ip geolocation*, in *Passive and Active Measurement: 11th International Conference, PAM 2010, Zurich, Switzerland, April 7-9, 2010. Proceedings 11*, pp. 171–180, Springer, 2010.

[151] J. Saxon and N. Feamster, *Gps-based geolocation of consumer ip addresses*, in *Passive and Active Measurement: 23rd International Conference, PAM 2022, Virtual Event, March 28–30, 2022, Proceedings*, pp. 122–151, Springer, 2022.

[152] A. Vaish, A. Kushwaha, R. Das, and C. Sharma, *Data location verification in cloud computing*, *International Journal of Computer Applications* **68** (2013), no. 12.

[153] A. Noman and C. Adams, *Hardware-based dlas: Achieving geo-location guarantees for cloud data using tpm and provable data possession*, in *2014 17th International Conference on Computer and Information Technology (ICCIT)*, pp. 280–285, IEEE, 2014.

[154] S. Park, J. N. Yoon, C. Kang, K. H. Kim, and T. Han, *Tgvisor: A tiny hypervisor-based trusted geolocation framework for mobile cloud clients*, in *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pp. 99–108, IEEE, 2015.

[155] M. Sabt, M. Achemlal, and A. Bouabdallah, *Trusted execution environment: what it is, and what it is not*, in *2015 IEEE Trustcom/BigDataSE/Ispa*, vol. 1, pp. 57–64, IEEE, 2015.

[156] L. ARM, *Arm security technology-building a secure system using trustzone technology*, tech. rep., PRD-GENC-C. ARM Ltd. Apr.(cit. on p.), 2009.

[157] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, *Innovative instructions and software model for isolated execution.*, *Hasp@ isca* **10** (2013), no. 1.

[158] M. Crone, *Towards attack-tolerant trusted execution environments: Secure remote attestation in the presence of side channels*, 2021.

[159] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, *Trustworthy execution on mobile devices: What security properties can my mobile platform give me?*, in *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings 5*, pp. 159–178, Springer, 2012.

[160] H. Vill, *Sgx attestation process*, 2017.

[161] S. Weiser and M. Werner, *Sgxio: Generic trusted i/o path for intel sgx*, in *Proceedings of the seventh ACM on conference on data and application security and privacy*, pp. 261–268, 2017.

[162] J. Thalheim, H. Unnibhavi, C. Priebe, P. Bhatotia, and P. Pietzuch, *Rkt-io: A direct i/o stack for shielded execution*, in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 490–506, 2021.

[163] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, *Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems*, in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1416–1432, IEEE, 2020.

[164] A. Nilsson, P. N. Bideh, and J. Brorsson, *A survey of published attacks on intel sgx*, *arXiv preprint arXiv:2006.13598* (2020).

[165] D. A. Osvik, A. Shamir, and E. Tromer, *Cache attacks and countermeasures: the case of aes*, in *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pp. 1–20, Springer, 2006.

[166] Y. Yarom and K. Falkner, *Flush+ reload: A high resolution, low noise, l3 cache side-channel attack*, in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 719–732, 2014.

[167] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, *A systematic evaluation of transient execution attacks and defenses.*, in *USENIX Security Symposium*, pp. 249–266, 2019.

[168] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, *Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution*, in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 142–157, IEEE, 2019.

[169] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, *Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution*, in *USENIX Security Symposium (SEC)*, 2018.

[170] X. Xin, *Titan m makes pixel 3 our most secure phone yet*, *Google (Oct. 2018). url: https://www. blog. google/products/pixel/titan-m-makespixel-3-our-most-secure-phone-yet* (2018).

[171] "energy mix." `https://archive.unescwa.org/energy-mix`. Accessed: 14-Apr-2023.

[172] "California iso – supply, today's outlook." `https://www.caiso.com/todaysoutlook/Pages/supply.aspx`. Accessed: 14-Apr-2023.

[173] K. Mammadzada, M. Iqbal, F. Milani, L. García-Bañuelos, and R. Matulevičius, *Blockchain oracles: a framework for blockchain-based applications*, in *Business Process Management: Blockchain and Robotic Process Automation Forum: BPM 2020 Blockchain and RPA Forum, Seville, Spain, September 13–18, 2020, Proceedings 18*, pp. 19–34, Springer, 2020.

[174] A. R. Elias, N. Golubovic, C. Krintz, and R. Wolski, *Where's the Bear? – Automating Wildlife Image Processing Using IoT and Edge Cloud Systems*, in *Internet-of-Things Design and Implementation (IoTDI), 2017 IEEE/ACM Second International Conference on*, pp. 247–258, IEEE, 2017.

[175] N. Golubovic, A. Gill, C. Krintz, and R. Wolski, *CENTAURUS: A Cloud Service for K-means Clustering*, in *IEEE DataCom*, Nov., 2017.

[176] C. Krintz, R. Wolski, N. Golubovic, B. Lampel, V. Kulkarni, B. Sethuramasamyraja, B. Roberts, and B. Liu, *SmartFarm: Improving Agriculture Sustainability Using Modern Information Technology*, in *KDD Workshop on Data Science for Food, Energy, and Water*, Aug., 2016.

[177] "UCSB SmartFarm." `https://sites.cs.ucsb.edu/~ckrintz/projects/`. [Online; accessed 17-April-2023].

[178] "UCSB Edible Campus." `https://sustainability.ucsb.edu/ediblecampus`. [Online; accessed 17-April-2023].

[179] D. Dua and C. Graff, *UCI machine learning repository*, 2017.