University of California
Santa Barbara

From Controlled Data-Center Environments to Open Distributed Environments: Scalable, Efficient, and Robust Systems with Extended Functionality

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

Doctor of Philosophy
in
Computer Science

by

Victor I. F. Zakhary

December 2019

The Dissertation of Victor I. F. Zakhary is approved.

Professor Chandra Krintz

Professor Rich Wolski

Professor Divyakant Agrawal, Committee Co-Chair

Professor Amr El Abbadi, Committee Co-Chair

November 2019

From Controlled Data-Center Environments to Open Distributed Environments:

Scalable, Efficient, and Robust Systems with Extended Functionality

To my mom, Isis Awad, single moms, terrorist attack survivors, and underrepresented minorities.

Acknowledgements

I would like to express my utmost gratitude to my advisors, Divy Agrawal and Amr El Abbadi. Thank you, thank you, thank you! Divy and Amr are, by far, the best advisors I have ever met and I am so grateful that I worked with them during my PhD. I am about to leave graduate school to join a super exciting career path that I would have never dreamed of, if it was not through their unconditional support. Divy and Amr are extremely passionate about both teaching and research and I have developed several important skills under their supervision. I learned how to put my heart into my work and how to critically think in different research directions. I was always encouraged to investigate new research ideas and think outside the box and outside the comfort zone of both myself and the community to come up with novel research problems and solutions. I worked as a teaching assistant for both Amr and Divy and it was a great pleasure and learning experience. Amr loves to teach and loves to teach how to teach. I owe Amr the several teaching awards I have earned during my PhD. I owe my thanks to both Divy and Amr for their patience and their unconditional support during my low moments during my PhD. Overall, I believe my PhD experience was amazing; so, thank you!

I am also thankful to both Professor Chandra Krintz and Professor Rich Wolski for serving on my dissertation committee. I truly enjoyed taking their graduate classes and I have learned a lot from them. I am also so grateful for their continuous support and precious advice during my PhD ease and tough times. Their guidance significantly helped me shape my PhD research directions. I truly appreciate how they always have their doors open to listen and provide support and advice to me during my PhD and during my job search as well.

I would like to also thank my labmates. I enjoyed our discussions, brain storming sessions, and our peer reviews of several paper drafts. I enjoyed working with Cetin

Sahin, Faisal Nawab, Vaibhav Arora, Theodore Georgiou, Sujaya Maiyya, Mohammad Amiri, Ishani Gupta, Rey Tang, Lawrence Lim, Caitlin Scarberry, and Xiaofei Du.

There is no better source of support than my mother and my brother. I would like to express my sincere gratitude to my mom, Isis Awad, and my brother, Abraham Zakhary for their unconditional love and support through my life and during my PhD. They both are my role models and I am so grateful to have them in my life.

I would like to also thank my friends from Egypt who gave me support during my PhD. I would like to mention Mark, Sultan, Ramzy, Emiliooo, Joe, Solio, Wagdy, Zew, Peter, Mary, Nosa, Eve, Yasmine, Saeed, Kosba, and Fouad for being there for me during my good times and my tough times as well. I also would like to thank my friends in Santa Barbara who made the PhD time a pleasurable experience. I would like to mention Dev, Nick, Leida, Nadim, Wahba, Shadi, Mahmoud, Karim, Mai, Nevena, Deeksha, Sujaya, Sarah, Rema, Adam, Gehad, and Rita. Thank you for all the great time we had during this journey.

Finally, I would like to thank the January 25th 2011 Egyptian revolution for inspiring our generation in Egypt to become the best of ourselves. Although this revolution has failed on the nation level, I believe that it succeeded significantly on our personal level.

Abstract

From Controlled Data-Center Environments to Open Distributed Environments:
Scalable, Efficient, and Robust Systems with Extended Functionality

by

Victor I. F. Zakhary

The past two decades have witnessed several paradigm shifts in computing environments. Starting from cloud computing which offers on-demand allocation of storage, network, compute, and memory resources, as well as other services, in a pay-as-you-go billing model. Ending with the rise of permissionless blockchain technology, a decentralized computing paradigm with lower trust assumptions and limitless number of participants. Unlike in the cloud, where all the computing resources are owned by some trusted cloud provider, permissionless blockchains allow computing resources owned by possibly malicious parties to join and leave their network without obtaining permission from some centralized trusted authority. Still, in the presence of malicious parties, permissionless blockchain networks can perform general computations and make progress. Cloud computing is powered by geographically distributed data-centers controlled and managed by trusted cloud service providers and promises theoretically infinite computing resources. On the other hand, permissionless blockchains are powered by open networks of geographically distributed computing nodes owned by entities that are not necessarily known or trusted. This paradigm shift requires a reconsideration of distributed data management protocols and distributed system designs that assume low latency across system components, inelastic computing resources, or fully trusted computing resources.

In this dissertation, we propose new system designs and optimizations that address scalability and efficiency of distributed data management systems in cloud environments.

We also propose several protocols and new programming paradigms to extend the functionality and enhance the robustness of permissionless blockchains. The work presented spans global-scale transaction processing, large-scale stream processing, atomic transaction processing across permissionless blockchains, and extending the functionality and the use-cases of permissionless blockchains. In all these directions, the focus is on rethinking system and protocol designs to account for novel cloud and permissionless blockchain assumptions. For global-scale transaction processing, we propose GPlacer, a placement optimization framework that decides replica placement of fully and partial geo-replicated databases. For large-scale stream processing, we propose Cache-on-Track (CoT) an adaptive and elastic client-side cache that addresses server-side load-imbalances that occur in large-scale distributed storage layers. In permissionless blockchain transaction processing, we propose AC3WN, the first correct cross-chain commitment protocol that guarantees atomicity of cross-chain transactions. Also, we propose TXSC, a transactional smart contract programming framework. TXSC provides smart contract developers with transaction primitives. These primitives allow developers to write smart contracts without the need to reason about the anomalies that can arise due to concurrent smart contract function executions. In addition, we propose a forward-looking architecture that unifies both permissioned and permissionless blockchains and exploits the running infrastructure of permissionless blockchains to build global asset management systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Paradigm Shifts in Global Data Management

Social networks, the web, and mobile applications have attracted hundreds of millions of end-users [29, 39]. These users share their relationships and exchange images and videos in timely personalized experiences [59]. To enable this real-time experience, the underlying data management systems have to provide efficient, scalable, and highly available access to big data. Social networks, the web, and mobile applications are examples of Global-Scale Data Management (GSDM) systems that aim to provide efficient and scalable big data access to hundreds of millions of end-users around the globe. The good news is that cloud computing provides the infrastructure needed to power GSDM systems. Cloud computing, a term that was initially coined by Compaq Computer [182] and popularized upon Amazon's release of Elastic Compute Cloud (EC2) [1], offers theoretically infinite infrastructure-as-a-service resources in a pay-as-you-go billing model. Cloud computing is powered by geographically distributed core data-centers and edge data-centers that offer elastic compute, memory, storage, network, and several services to GSDM systems. The unfortunate news is that moving GSDM systems to cloud infrastructures raises several challenges.

First, cloud applications strive for high-performance 24/7 service to clients dispersed around the world. Achieving this is threatened by complete datacenter scale outages; either planned or unplanned. Second, GSDM systems such as social networks aim to serve hundreds of millions of end-users with sub-second response latency. Social network users consume several orders of magnitude more data than they produce [51]. In addition, a single page load requires hundreds of data object lookups that need to be served in a fraction of a second [59]. Therefore, traditional disk-based storage systems are not suitable to handle requests at this scale due to the high access latency of disks and I/O throughput bounds [222]. Third, moving end-user data to the cloud and allowing social network giants to have control over massive amounts of end-user personal data introduce several data privacy threats. Microtargeting (e.g., the Cambridge Analytica scandal [25]), surveillance, and discriminating ads are examples of threats to user privacy caused by social network end-user data mining. These three challenges, namely availability, scalability, and end-user data privacy require rethinking and building GSDM systems in novel ways that address these challenges while leveraging cloud infrastructure.

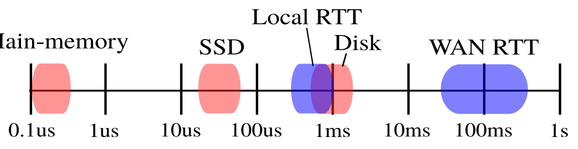


Figure 1.1: Latency of Wide-Area Network Round-Trip Time communication (WAN RTT) compared to memory access latency [181] and network latency within a datacenter (local RTT).

To address the availability challenge, several GSDM systems and their backend databases are increasingly being deployed on multiple datacenters spanning large geographic regions (*geo-replication*). F1 [195], Spanner [79], and Tao [59] are examples of deployed systems that are geographically replicated for fault-tolerance and performance reasons. Geo-replication serves *two* important goals. First, it brings data copies closer to end-users to serve data lookup requests with low latency. Second, it allows applications to serve end-user requests even in the presence of datacenter scale outage. However, geo-replication raises the important question of replica **placement**: *at which datacenters should data be placed?*. Current cloud providers offer hundreds of datacenters and thousands of edge datacenters that are globally distributed all over the world. Unlike networks within a datacenter, the topology of the Wide-Area Network (WAN) is asymmetric and diverse—the latency connecting a pair of datacenters can be an order of magnitude larger than the latency connecting another pair. This makes placement a significant factor in performance. Figure 1.1 illustrates the latency difference between communication messages that occur within the same machine, among different machines in one datacenter, or in multiple datacenters in different geographical regions. This large communication latency of the WAN motivates systems like Yahoo's PNUTS [77], Facebook's Tao [59] and others [138, 167] to trade off replica consistency and/or multi-row transaction support with high availability and scalability. However, enterprise applications and applications with complex and evolving schemas have more interest in data management systems that provide transactional ACID properties [196, 79, 53]. Application developers spend significant time to build transaction semantics and complex mechanisms, which are error-prone, on top of eventually consistent datastores in order to handle stale data items and reason about inconsistency [195, 79]. Therefore, the first question we ask in this dissertation is: *"Can we optimize the placement of geo-replicated databases to minimize transaction latency while achieving the* ***ACID transaction*** *guarantees and the required availability level?"*.

To address the scalability challenge, distributed caching services have been widely deployed on top of persistent storage in order to efficiently serve user requests at scale [211]. Distributed caching systems such as Memcached [32] and Redis [35] are widely adopted by cloud service providers such as Amazon ElastiCache [20] and Azure Redis Cache [22]. These caching services offer significant latency and throughput improvements to systems that directly access the persistent storage layer. Redis and Memcached use consistent hashing [128] to distribute keys among several caching servers. Although consistent hashing ensures a fair distribution of the number of keys assigned to each caching shard, it does not consider the workload per key in the assignment process. Real-world workloads are typically skewed with few keys being significantly hotter than other keys [122]. This skew causes load-imbalance among caching servers. Load imbalance in the caching layer can have significant impact on the overall application performance. In particular, it may cause drastic increases in the latency of operations at the tail end of the access frequency distribution [121]. In addition, the average throughput decreases and the average latency increases when the workload skew increases [71]. This increase in the average and tail latency is amplified for real workloads when operations are executed in chains of dependent data objects [150]. A single Facebook page-load results in retrieving hundreds of objects in multiple rounds of data fetching operations [167, 59]. Finally, solutions that equally overprovision the caching layer resources to handle the most loaded caching server suffer from resource under-utilization in the least loaded caching servers. Therefore, the second question we ask in this dissertation is: *"Can we exploit the geo-distribution and elasticity of edge data-centers to design* ***adaptive*** *and* ***decentralized*** *load-balancing solutions for GSDM imbalanced distributed storage layers?"*.

The third challenge of preserving social networks and web end-user data privacy raises complicated tension between user utility and user data privacy. On one hand, end-users share their data and postings with social network and web giants. On the other hand, end-user data privacy must be preserved. Social network users develop, over time, online persona [220] that reflect their overall interests, activism, and diverse orientations. Users have numerous followers that are specifically interested in their personas and their postings which are aligned with these personas. Due to the rise of machine learning and deep learning techniques, user posts and social network interactions can be used to accurately and automatically infer many user persona attributes such as gender, ethnicity, age, political interest, and location [133, 177, 225, 223, 72, 68]. Although social networks use end-user attributes to provide personalised services, recent news about the *Cambridge Analytica scandal* [25] and similar *data breaches* [34] suggest that users cannot depend on the social network providers to preserve their privacy. User sensitive attributes such as gender, ethnicity, and location have been widely misused in *illegally discriminating ads*, *microtargeting*, and *surveillance.* Privacy risks vary from discrimination in job [19] and housing [31] ads, election manipulation [16, 17], activists of color targeting [11], to becoming a danger for our democracy [97]. To protect end-user data privacy, recent legislation like GDPR [12] and CCPA [24] have been proposed and passed to limit the usage of end-user personal data by social network and web giants. Although we believe that protecting end-user personal data by laws is the ideal path to follow, the legislation path is long and time-consuming. Until social network and web giants are forced to protect end-user personal data privacy, we believe that end-users should have fine control over which personal attributes to make public and which ones to keep private. Therefore, the third question we ask in this dissertation is: ”*Can we develop **decentralized**, user-centric, and trust-free systems that allow end-users to have fine control of the privacy of their persona attributes?*”

In 2008, the *mysterious Nakamoto* came up with a novel decentralized peer-to-peer computation model that enabled Bitcoin [164], the first successful global scale peer-to-peer cash system or cryptocurrency. The Bitcoin original protocol allows financial transactions to be transacted among participants without the need for a trusted third party, e.g., banks, credit card companies, or PayPal. Bitcoin eliminates the need for such a trusted third party by replacing it with a distributed ledger that is fully replicated among a decentralized open network of computing nodes in the cryptocurrency system. This distributed ledger is referred to as *blockchain* and the open network is usually referred to as permissionless blockchain.

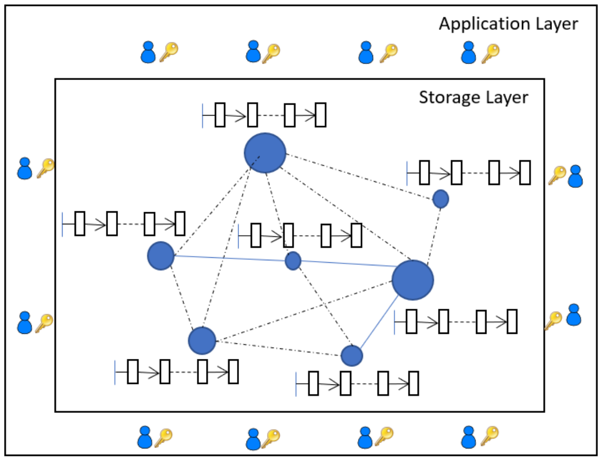


Figure 1.2: Open Blockchain Architecture Overview

An open permissionless blockchain system [155] (e.g., Bitcoin, Ethereum [205]) typically consists of two layers: a storage layer and an application layer as illustrated in Figure 1.2. **The storage layer** comprises a decentralized distributed ledger managed by an open network of computing nodes. A blockchain system is permissionless if computing nodes can join or leave the network of its storage layer at any moment without obtaining a permission from a centralized authority. Each computing node, also called a miner, maintains a copy of the ledger. The ledger is a tamper-proof chain of blocks, hence named blockchain. Each block contains a set of valid transactions that transfer assets among end-users. **The application layer** comprises end-users who communicate with the storage layer via *message passing* through a client library. End-users have identities, defined by their public keys, and signatures, generated using their private keys. Digital signatures are the end-users' way to generate transactions. End-users submit their transactions to the storage layer through a client library. Transactions are used to transfer assets from one identity to another. End-users multicast their transaction messages to mining nodes in the storage layer. A mining node validates the transactions it receives and valid transactions are added to the current block of a mining node. Miners run a consensus protocol through mining to agree on the next block to be added to the chain. A miner who mines a block gets the right to add their block to the chain and multicasts it to other miners. To make progress, miners accept the first received mined block after verifying it and start mining the next block.

Since the publishing of the Bitcoin paper, many efforts have been developed to transfer the usage of permissionless blockchains from a cryptocurrency management system to the new public cloud [205, 64]. Instead of using the blockchain distributed ledger to store cryptocurrency transactions, end-users can store generic data and code within the distributed ledger through smart contracts [200]. Smart contracts extend the simple abstract data type notion of blockchain transactions to include complex data type classes with end-user defined variables and functions. Also, instead of using the mining node to verify and validate cryptocurrency transactions, the compute power of the mining nodes can be used to process generic logic expressed in smart contract functions. Seeing permissionless blockchains as the new public cloud introduces several challenges and interesting problems that need to be addressed. First, there is the trust issue. Most cloud data management protocols assume trusted infrastructure owned by a known cloud service provider. Also, cloud replication and transaction management protocols assume that all protocol participants are known and their number is fixed. On the other hand, permissionless blockchains run on top of an open network with unknown number of untrusted computing nodes. This requires the redevelopment of common cloud data management protocols such as 2-Phase Commit (2PC) [112, 57] for atomic distributed transaction commitment and Paxos [140] for replication. Second, there is a lack of abstraction in smart contract programming languages. Modern programming languages provide application developers with many abstractions such as transaction abstractions. This allows application developers to leverage ACID transaction properties without rewriting the transaction management logic in each application. Currently, the transaction abstraction is not supported in smart contract languages like solidity [38]. This requires smart contract developers to reason about transaction semantics, concurrency, and isolation in every smart contract they write. The distributed database literature [79, 195] has shown that putting the burden of implementing transaction logic in the application layer is problematic. This is no simple task and serious smart contract concurrency bugs have been highlighted in the blockchain literature [132, 151, 193, 86]. In fact, from a financial point-of-view, two such famous anomalies in the context of blockchains, TheDAO [4, 62] and the BlockKing [9] have resulted in the loss of tens of millions of investors' dollars [151]. Third, there is a disconnection between tangible assets and intangible assets in blockchain systems. In a cloud setting, a trusted third party can be leveraged to trade tangible assets such as houses and cars. The responsibility of verifying the existence of such assets is put on this trusted third party. However, in the absence of trust in permissionless blockchains, neither mining nodes nor end-users can be trusted to verify the existence of tangible assets. This requires the development of novel protocols and systems that leverage the infrastructure of permissionless blockchains to trade tangible assets. Finally, there is the issue of user data privacy. Since permissionless blockchains are built on transparency and all end-user transaction information are public, permissionless blockchains introduce several end-user privacy challenges that need to be addressed.

These challenges motivated us to ask several questions in this dissertation. First, *Can we develop atomic blockchain-based distributed transaction management protocols, like 2PC, that do not assume a trusted coordinator?*. Second, *Can the ACID transaction abstraction be supported in smart contract programming languages?*. Finally, *Can we design protocols and systems that leverage permissionless blockchain infrastructures to trade tangible assets?*.

This dissertation addresses several of the aforementioned questions. Given the assumptions of cloud computing and permissionless blockchains as the new public cloud, *Can we develop* ***decentralized*** *systems and protocols that provide* ***ACID transaction*** *guarantees for scalable, fault-tolerant, and privacy preserving data management systems?*. All the solutions explained in this dissertation focus on one or two of the following aspects: decentralization and ACID transaction support on different environments and under different assumptions. In Section 1.2, we provide an overview of the works and solutions explained in this dissertation and which aspects are the focus of each solution. Section 1.3 provides an organization of the rest of this dissertation.

1.2 Dissertation Overview

Cloud computing and permissionless blockchains are two computing paradigm shifts that have shaken many traditional distributed system design assumptions. On one hand, the elasticity and the asymmetric and diverse Wide-Area-Network (WAN) latency across cloud data-centers require rethinking traditional transactional and stream processing system designs. On another hand, open permissionless blockchains powered by a highly decentralized, untrusted, and limitless open network of compute nodes require rethinking transaction management protocols that either assume trust among protocol participants or centralization of a protocol coordinator. The research in this dissertation leverages cloud and permissionless blockchain infrastructures to build scalable, efficient, and robust GSDM systems with extended functionality.

Figure 1.3 summarized the system and protocol solutions proposed in this dissertation. As shown, solutions in this dissertation focus on decentralization and/or ACID transaction support. These solutions provide answers to the questions raised in Section 1.1. First, we present GPlacer [218], a placement framework that optimizes the

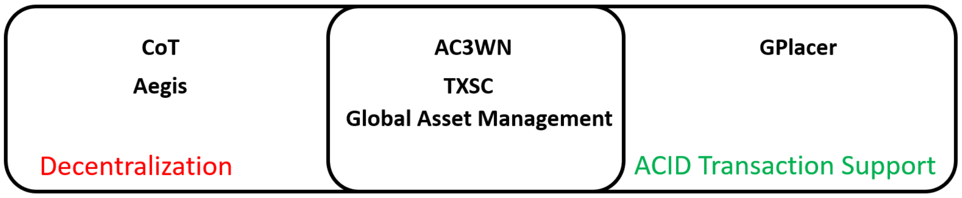


Figure 1.3: Overview of the system and protocol solutions proposed in this dissertation.

placement of geo-replicated databases to minimize the transaction latency while achieving the **ACID transaction** guarantees and the required availability level. Then, we propose CoT [213], a load-balancing solution for GSDM's imbalanced distributed storage layers. CoT is a **decentralized** system that leverages the geo-distribution and elasticity of core and edge data-centers. Afterwards, we introduce Aegis [216]. Aegis is a **decentralized**, user-centric, and trust-free system that allows social network and web end-users to have fine control of the privacy of their persona attributes.

In permissionless blockchain environments, we present several decentralized primitives, building blocks, and abstractions that support correct ACID transaction semantics under the trust-free and the open network assumptions. First, we present AC^{3}WN [212], the first correct atomic cross blockchain commitment protocol. AC^{3}WN is decentralized and does not require a trusted coordinator to ensure the ACID guarantees of cross blockchain transactions. Then, we propose TXSC [214], a framework that provide ACID transaction support for smart contract programming languages. Finally, we present a forward looking platform that enables global asset management in blockchain systems [215]. We summarize the aforementioned solutions in the following Sections.

1.2.1 GPlacer

Current cloud providers offer hundreds of datacenters and thousands of edge datacenters that are globally distributed all over the world. Unlike networks within a datacenter, the topology of the Wide-Area Network (WAN) is asymmetric and diverse—the latency connecting a pair of datacenters can be an order of magnitude larger than the latency connecting another pair. This makes placement a significant factor in performance. However, it is not only placement. The specifics of the transaction management protocol play a crucial role in deciding which placement is ideal. GPlacer is a placement optimization framework that embeds the transaction protocol constraints into an optimization to derive both the data placement and the transaction protocol configuration that minimize the overall transaction latency. In developing GPlacer, we discover counter-intuitive lessons about data placement and transaction execution practices. Our evaluation shows that applying these lessons in addition to known best practices generate deployments that reduce the average transaction latency by up to 68%. GPlacer optimizes the replica placement and the transaction execution plan for both leader-based protocols and non-leader-based protocols (e.g., quorum based protocols). In addition, GPlacer optimizes the placement for *multi-row transactional workloads with strong consistency requirements*.

1.2.2 CoT

Cache-on-Track (CoT) is a **decentralized**, **elastic**, and **predictive** front-end caching mechanism to address the load-imbalance across GSDM distributed storage layers. CoT uses a small front-end cache to solve back-end load-imbalance as introduced in [96]. However, CoT does not assume perfect caching at the front-end as assumed in [96]. CoT proposes a new cache replacement policy specifically tailored for small front-end caches that serve **skewed workloads**. CoT uses the space saving algorithm [158] to track the **top-k** *heavy hitters*. The tracking information allows CoT to *cache* the exact top C hot-most keys out of the approximate top-k tracked keys preventing cold and noisy keys from the long tail to replace hot keys in the cache. CoT is decentralized in the sense that each front-end independently determines its hot key set based on the key access distribution served at this specific front-end. This allows CoT to address back-end load-imbalance without introducing single points of failure or bottlenecks that typically come with centralized solutions. In addition, this allows CoT to scale to thousands of front-end servers, a common requirement of social network and modern web applications. CoT is elastic in that each front-end uses its local load information to monitor its contribution to the back-end load-imbalance. Each front-end elastically adjusts its tracker and cache sizes to reduce the load-imbalance caused by this front-end. In the presence of workload changes, CoT dynamically adjusts front-end tracker to cache ratio in addition to both the tracker and cache sizes to eliminate any back-end load-imbalance.

1.2.3 Aegis

Aegis is a decentralized and user-centric system that allows social network and web end-users to have fine control of the privacy of their persona attributes. First, we propose *multifaceted privacy*, a novel privacy model that aims to obfuscate a user's sensitive attributes while revealing the user's public persona attributes. Multifaceted privacy allows users to freely express their online public personas without revealing any sensitive attributes **of their choice**.

To achieve multifaceted privacy, we build *Aegis*, a prototype user-centric social network stream processing system that enables social network users to take charge of protecting their own privacy, instead of depending on the social network providers. Our philosophy in building Aegis is that social network users need to introduce some noisy interactions and obfuscation posts to confuse **content based attribute inferences**. Choosing this noise introduces a challenging *dichotomy* and tension between the utility of the user persona and her privacy. Obfuscation posts need to be carefully chosen to achieve obfuscation of private attributes without damaging the user's public persona. The main goal of Aegis is to automatically find and suggest the noise (obfuscation) postings that achieve the multifaceted privacy.

1.2.4 AC^{3}WN

An **A**tomic **C**ross-**C**hain **T**ransaction, AC^{2}T, is a distributed transaction that spans multiple blockchains. This distributed transaction consists of sub-transactions and each sub-transaction is executed on some blockchain. An **A**tomic **C**ross-**C**hain **C**ommitment, AC^3, protocol is required to execute AC^{2}Ts. This protocol is a variation of traditional distributed atomic commitment protocols (e.g., 2PC [112, 57]). This protocol should guarantee both *atomicity* and *commitment* of AC^{2}Ts. **Atomicity** ensures the **all-or-nothing** property where either all sub-transactions take place or none of them is executed. **Commitment** guarantees that any changes caused by a cross-chain transaction must eventually take place if the transaction is decided to commit. Unlike in 2PC and other traditional distributed atomic commitment protocols, atomic cross-chain commitment protocols are also trust-free and therefore must **tolerate** maliciousness [117].

We propose **AC^{3}WN**, the first correct and decentralized all-or-nothing **A**tomic **C**ross-**C**hain **C**ommitment protocol that uses an open **W**itness **N**etwork. The commit and the abort decision of all sub-transactions in AC^{2}T are modeled as conflicting decisions. A decentralized open network of witnesses is used to coordinate AC^{2}T. The witness network guarantees that conflicting decision must never simultaneously take place and either all sub-transactions in an AC^{2}T commit or all of them abort.

1.2.5 TXSC

Smart contracts have their own variables and multiple functions that may be executed by different end-users results in transactions which might be incorporated in different blocks by different miners. This clearly results in complex concurrency challenges which need to be handled by smart contract developers. We advocate leveraging the traditional transactional approach to address the concurrency violations in the context of smart contract executions in large scale blockchain systems. In particular, we propose Transactional Smart Contracts (TXSC) as a framework that allows developers to write smart contracts with correct transaction isolation semantics. Unlike previous works [132, 151, 193] that propose smart contract analysis tools to detect concurrency bugs in smart contracts, TXSC aims to free smart contract developers from the burden of implementing correct concurrency control semantics for each smart contract. Instead, developers can focus on the smart contract application semantics and leave the concurrency semantics to TXSC.

TXSC addresses concurrency challenges in smart contract development. In particular,

1. We model smart contract concurrency anomalies as transaction isolation problems. Examples illustrate how different smart contract concurrency anomalies can be mapped to the problem of transaction isolation of either single domain or distributed cross-domain transactions.

2. TXSC is the first framework to provide smart contract developers with transactional primitives *start transaction* and *end transaction*. TXSC takes a smart contract that contains these primitives as an input and translates it to a transactionally correct smart contract using the smart contract native language.

1.2.6 Asset Management in Blockchain Systems

Permissionless blockchains (e.g., Bitcoin) have shown a wide success in implementing global scale peer-to-peer cryptocurrency systems. In such blockchains, new currency units are generated through the mining process and are used in addition to transaction fees to incentivize miners to maintain the blockchain. Although it is clear how currency units are generated and transacted on, it is unclear how to use the infrastructure of permissionless blockchains to manage other assets than the blockchain's currency units (e.g., cars, houses, etc.). We propose a global asset management system that leverages the infrastructure of permissionless blockchains as a marketplace for global assets. This proposal unifies both permissioned (trusted) and permissionless blockchains in order to build generic asset management system. A governmental permissioned blockchain authenticates the registration of end-user assets through smart contract deployments on a permissionless blockchain. Afterwards, end-users can transact on their assets through smart contract function calls (e.g., sell a car, rent a room in a house, etc). In return, end-users get paid in currency units of the same blockchain or other blockchains through atomic cross-chain transactions and governmental offices receive taxes on these transactions in cryptocurrency units.

1.3 Organization

The rest of the dissertation is organized as follows. Part I provides solutions to build scalable and efficient cloud data management systems. First, GPlacer is presented in Chapter 2. Chapter 3 describes the details of CoT. Afterwards, we propose multifaceted privacy and Aegis in Chapter 4. Part II includes robust primitives and abstractions that extend the functionality of permissionless blockchains. Chapter 5 presents the details and the correctness proofs of the AC^{3}WN protocol. Chapter 6 describes how to leverage the infrastructure of permissionless blockchains to build global asset managenment system. Chapter 7 explains the details of the TXSC framework. Part III summarizes the dissertation and discusses future directions in GSDM systems research in Chapter 8.

Part I

Building Scalable and Efficient Cloud Data Management Systems

Chapter 2

GPlacer: Global-Scale Placement of Transactional Data Stores

2.1 Overview

Internet applications strive for high-performance 24/7 service to clients dispersed around the world. Achieving this is threatened by complete datacenter outages; either planned or unplanned. To overcome these challenges, application services and their backend databases are increasingly being deployed on multiple datacenters spanning large geographic regions (*geo-replication*). F1 [195], Spanner [79], and Tao [59] are examples of deployed systems that are geographically replicated for fault-tolerance and performance reasons.

Moving to Global-Scale Data Management (GSDM), despite its benefits, raises many challenges that are not faced by traditional deployments. The large WAN communication latency is orders of magnitude larger than the traditional LAN communication latency. Figure 2.1 illustrates the latency difference between communication messages that occur within the same machine, among different machines in one datacenter, or

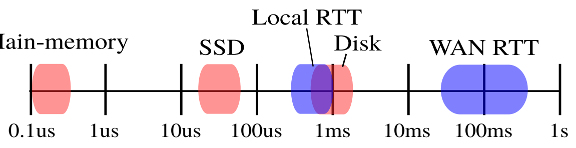


Figure 2.1: Latency of Wide-Area Network Round-Trip Time communication (WAN RTT) compared to memory access latency [181] and network latency within a datacenter (local RTT).

in multiple datacenters in different geographical regions. This large communication latency of the WAN motivates systems like Yahoo's PNUTS [77], Facebook's Tao [59] and others [138, 167] to trade off replica consistency and/or multi-row transaction support with high availability and scalability. However, enterprise applications and applications with complex and evolving schemas have more interest in data management systems that provide transactional ACID properties [196, 79, 53]. Application developers spend significant time to build transaction semantics and complex mechanisms, that are error-prone, on top of the eventual consistent datastores in order to handle stale data items and reason about inconsistency [195, 79]. Therefore, in the past few years, many solutions have emerged to provide strongly consistent transactions for geo-replicated databases [79, 154, 134, 166, 165, 145]. These solutions use different replication and isolation techniques in order to minimize the number of WAN messages required to achieve strong ACID transactional guarantees for geo-replicated databases, hence reducing the transaction latency.

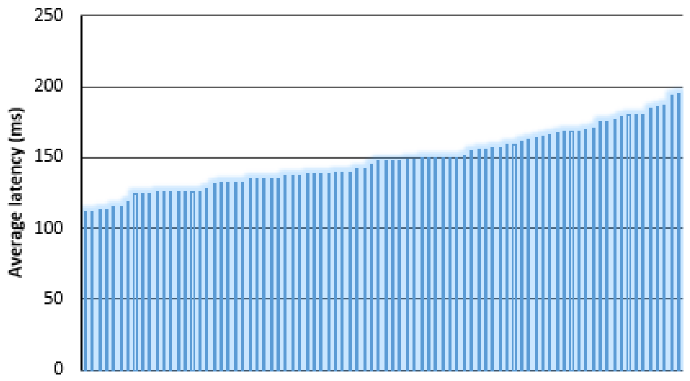
Data placement is the problem of deciding the subset of datacenters to host a full or a partial replica of the data to achieve a certain objective such as minimizing the transaction latency, minimizing the deployment monetary costs, and any combination of these and other user-defined objective functions.

We propose GPlacer; an optimization framework that solves the data placement problem. GPlacer embeds the commit protocol constraints into an optimization framework to derive both the data placement and the commit protocol configurations that minimize the overall transaction latency. In developing GPlacer, we discover counter-intuitive lessons about data placement and transaction execution practices. These lessons exploit the latency diversity and asymmetry of the WAN links and are widely applicable to Paxos-based commitment protocols [111, 154] and leader-based commitment protocols [79, 53]. GPlacer incorporates these lessons, the commitment protocol constraints, and the application requirements in an optimization framework to find the placement that minimizes the average transaction latency.

	C	O	V	I	Si	T	Se	Sy	SP
C	0(1)	22(2)	65(13)	136(5)	189(12)	113(5)	142(12)	159(2)	185(11)
O	22(2)	1(1)	88(14)	125(2)	166(13)	101(11)	131(13)	178(3)	182(11)
V	65(13)	88(14)	1(16)	73(13)	220(22)	156(16)	179(20)	219(13)	121(16)
I	136(5)	125(2)	73(13)	0(0)	180(18)	211(10)	233(14)	301(5)	185(12)
Si	189(11)	166(12)	220(22)	180(17)	1(9)	68(8)	97(13)	169(8)	329(21)
T	113(5)	101(11)	156(18)	211(10)	68(9)	0(3)	32(9)	104(2)	263(15)
Se	142(9)	131(13)	179(20)	233(13)	97(13)	32(10)	1(9)	133(8)	290(16)
Sy	159(2)	178(3)	219(12)	301(5)	169(10)	104(2)	133(8)	1(0)	338(11)
SP	185(13)	182(12)	121(17)	185(13)	329(23)	263(16)	290(18)	338(14)	1(11)

Table 2.1: The average RTT latencies between different datacenters in milliseconds and the standard deviation inside parentheses.

WAN links are diverse and asymmetric; a link connecting a pair of datacenters can be an order of magnitude larger than a link connecting another pair. Table 2.1 shows the average measured Round-Trip Time (RTT) between every pair of nine Amazon AWS datacenters in **C**alifornia (C), **O**regon (O), **V**irginia (V), **S**ão **P**aulo (SP), **I**reland (I), **Sy**dney (Sy), **Si**ngapore (Si), **T**okyo (T), and **Se**oul (Se). As shown, the average RTT between California and Oregon datacenters is 22ms while the average RTT between Singapore and São Paulo datacenters is 329ms. Therefore, the number of required WAN messages per transaction is not the only factor that dominates the transaction latency. Transaction latency is a product of both the transaction commit protocol, which controls



r0.6

Figure 2.2: The average latency, of all the clients in 9 datacenters, to reach the closest quorum (2 out 3) for all the possible $\binom{9}{3} = 84$ different placements sorted by latency.

the number of required WAN messages per transaction, and the locations of the replicas, hence the placement, which controls the latency per WAN message.

To illustrate the placement effect on the average obtained transaction latency, we conduct the following experiment. We equally distribute clients among the nine AWS datacenters. Three out of the nine datacenters are chosen to host a data *replica*. The time to reach the closest quorum, two replicas out of these three, is measured for all the clients for all the **possible placements** and the average latency is reported. Figure 2.2 shows the effect of only changing the placement on the average obtained latency for all the clients while fixing the protocol. As seen in Figure 2.2, changing only the placement while fixing all the other parameters (the protocol, the workload distribution, etc.) can lead to a significant change of 1.75x between the minimum and the maximum reported average latency. This latency difference amplifies for real workloads when transactions are executed in chains [150].

Unlike GPlacer that optimizes the placement for *multi-row transactional workload with strong consistency requirements*, many works focus on optimizing the placement for weaker consistency levels and single-row operations. SPANStore [208] develops an optimization framework to minimize the monetary cost of a geo-replicated *key/value* store deployment. This framework optimizes the total cost of processing, storage, bandwidth, and I/O and finds the placement that achieves the minimum overall cost while meeting the application requirements. Liu et al. [146], like SPANStore, optimize the monetary cost of geo-replicated key/value store deployments. However, they consider cost savings exploiting resource reservation payment model instead of the pay-as-you-go payment model while avoiding over reservation. Ping et al. [178] propose the use of a utility function to derive a placement that achieves a balance between the availability and the speed of data access. Volley [44] analyzes data access logs and generates a migration plan for data partitions to minimize the access latency.

Sharov et al. [194] optimize the placement for *strong consistent transactions* using *leader-based* protocols. Sharov assumes that a database is sharded into multiple *partitions* and each partition is independently **replicated**. Each partition has a **leader replica** that serializes all the transactions that span this partition to achieve isolation. This leader replicates the updates to a *majority quorum* of the partition replicas to achieve fault tolerance. Although they provide placements for strong consistent transactional workloads, their optimizations are tightly coupled with leader-based protocols and it does not apply to the many non-leader-based protocols that are widely used such as [111, 176, 154, 134, 54]. Also, their resulting optimal placement allocates all the partition leaders together in one datacenter. Placing all partition leaders in one datacenter introduces a single point of failure. A datacenter outage can lead to a temporal loss of access to the entire data until all partition leaders are re-elected. In addition, transactions that span a single partition might incur higher latency than the latency observed when the leader of each partition is placed closer to the clients that frequently access this partition.

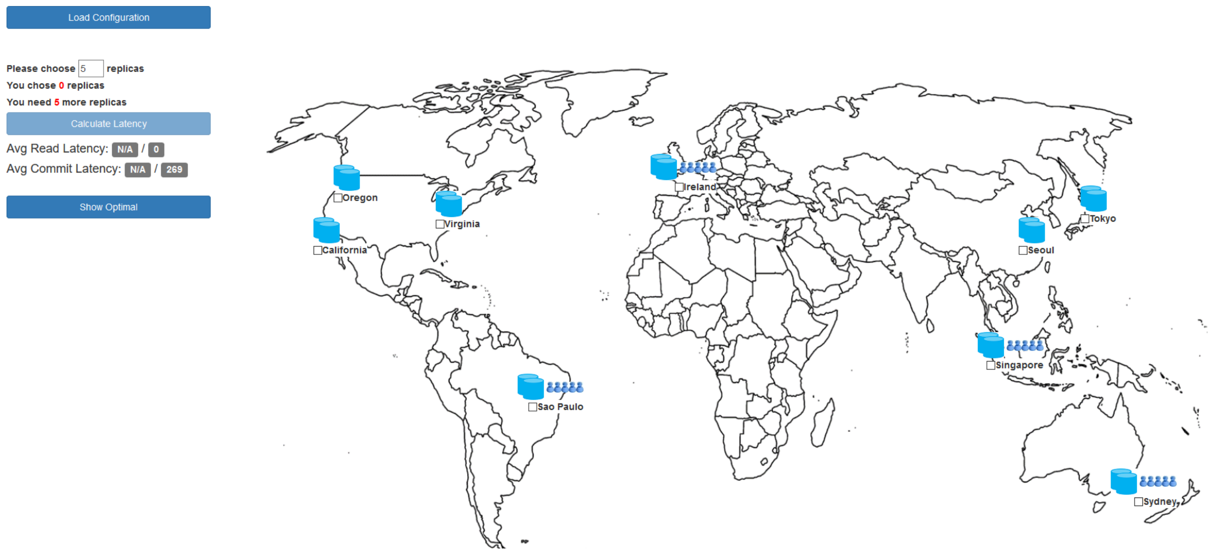
The rest of the chapter is organized as follows. Section 2.2 builds the case for the importance of replica placement frameworks by showing the common placement mistakes committed by practitioners and theoreticians during our replica placement demonstration in [217]. Section 2.3 explains the transaction model, the client requests, and the assumptions and limitations of application requirements. Although GPlacer can optimize the placement for different classes of commitment protocols, a Paxos-based protocol is used to explain the details of GPlacer. Section 2.4 formalizes the placement problem into an exhaustive search problem. Although the exhaustive search finds the optimal placement, it does not efficiently scale with the number of datacenters. Therefore, we introduce several placement heuristics that find sub-optimal placements while efficiently scale with the number of datacenters. Section 2.5 describes the counter-intuitive lessons learned during the development of GPlacer and their impact on the transaction latency. In Section 2.6, we evaluate the effect of the placement lessons on the transaction latency and the abort rate. We also evaluate the output and the performance of the proposed heuristics compared to the exhaustive search. In Section 2.7, we explain the changes that need to be done to extend GPlacer to optimize for other protocols. The chapter is concluded in Section 2.8.

2.2 DB-Risk: The Game of Global Database Placement

DB-Risk [217] is a game that is designed to motivate deeper understanding of the challenges of data placement in geo-replicated environments. It also showcases the need for optimization frameworks to optimize the placement of geo-replicated databases. The game is designed to be played by the demo participants to introduce a competitive element. A live version of the DB-Risk game can be found in [10].

DB-Risk System model. The DB-Risk model of geo-replication consists of a topology of datacenters and clients executing transactional workload. Data is fully replicated to a subset of AWS datacenters. Users issue transactions that consist of read and write operations. Each client executes transactions back-to-back. The execution of transactions depends on the replication protocol. DB-Risk focuses on majority based replication and commitment protocols explained as follows. A client executes a transaction by performing the reads and buffering the writes. Read requests are sent to a majority of datacenters. The highest version read is used. After executing reads and writes a vote request is sent to datacenters. The vote request consists of the read versions and the buffered write operations. Each datacenter, upon receiving a vote request, attempts to lock all objects that are being written. Additionally, it verifies that the read versions were not overwritten. If both are successful, the datacenter sends back a positive vote. Otherwise, a negative vote is sent. The client commits the transaction if a majority of positive votes is received and aborts the transaction otherwise. The client sends the decision *to all replicas*. Once a majority acknowledges the receipt of the decision, the transaction terminates. The transaction latency is the time from the beginning of executing operations until terminating the transaction. We define the commit latency as the time spent committing the transaction, which is equivalent to the transaction latency without the time spent reading the data values.

DB-Risk Game Details. In DB-Risk, players are asked to place replicas in datacenters around the world with the aim of minimizing the transaction latency. Each player gets a set of placement optimizations to choose from to enhance the transaction latency. The winner is the player with a placement and set of optimizations that achieve the lowest average transaction latency. Figure 2.3 shows a screenshot of the DB-Risk game. Participants are shown a map of nine AWS datacenters in California, Oregon, Virginia, Ireland, Sao Paulo, Singapore, Seoul, Tokyo, and Sydney. In addition, pairwise average Round-Trip Time (RTT) among the nine datacenters is provided. The workload at each datacenter is represented by the number of blue clients. Participants are asked



Please choose 5 replicas
You chose 0 replicas
You need 5 more replicas
Avg Read Latency: N/A / 0
Avg Commit Latency: N/A / 269

Figure 2.3: A screenshot of the DB-Risk game.

to place *five* data replicas in the nine available datacenters. In addition, participants are given the option to use some protocol optimizations like *optimistic reads, passive replica reads, and request handoff.* Optimistic reads allow transaction clients to read data values from any replica and validate the data version at commit time. Similarly, passive replica reads permit transaction clients to read data values from local caches that are asynchronously updated in each datacenter. Read versions must be validated at commit time as well. Request handoff enables transaction clients to handoff transaction read and commit operations to other datacenters aiming to reduce operation latency. Read optimizations are formally explained in Section 2.3.2 and request handoff is formalized in Section 2.5.1.

Data replica placement has twofold effect on transaction latency as follows:

- The distances between transaction clients and data replicas affect read and commit latency as clients have to reach a replica or a set of replicas to execute each operation.

- The distances among different data replicas significantly affect transaction latency as data replicas have to synchronously coordinate among each other to ensure transaction strong consistency.

Currently, there are hundreds of core datacenters and thousands of edge datacenters [13] distributed around the global and ready to host data replicas of different applications. Therefore, Db-Risk's search space for the optimal placement is considered small. However, many of the DB-Risk participants focused on one of the placement effects on transaction latency and ignore the other. Participants choose to place replicas in datacenters that have most of the clients ignoring the effect of coordination among replicas on transaction latency. Although this strategy is intuitive and reduces the time for a client to reach a replica, it does not find the optimal placement in many of DB-Risk's scenarios. This is because the datacenters that have larger number of clients happen to be far from each other and the time to reach a quorum of replicas is maximized using this strategy. We use the following DB-Risk scenario to point out the problems of placing replicas near clients without considering the distance among different replicas.

Consider the DB-Risk scenario shown in Figure 2.4. Clients are located in Ireland, Sao Paulo, Singapore, and Sydeny. However, the optimal placement of five replicas for DB-Risk's majority-base protocol is to place data replicas in California, Oregon, Virginia, Seoul, and Tokyo. This placement places quorums of replicas close to each other (quorums are shown using the green and red dotted curves). Clients in both Ireland and Sao Paulo handoff their commit requests to Virginia (green lines) and clients in both Singapore and Sydney handoff their requests to Seoul (red lines). This placement allows the replica in Virginia to quickly form a quorum (*3 of 5*) with the replicas in California and Oregon. Also, the replica in Seoul can rapidly form a quorum with the replicas in Tokyo and California. The intersection between the two quorums (in California) guarantees serializability. **Surprisingly, in this example, none of the chosen replicas**

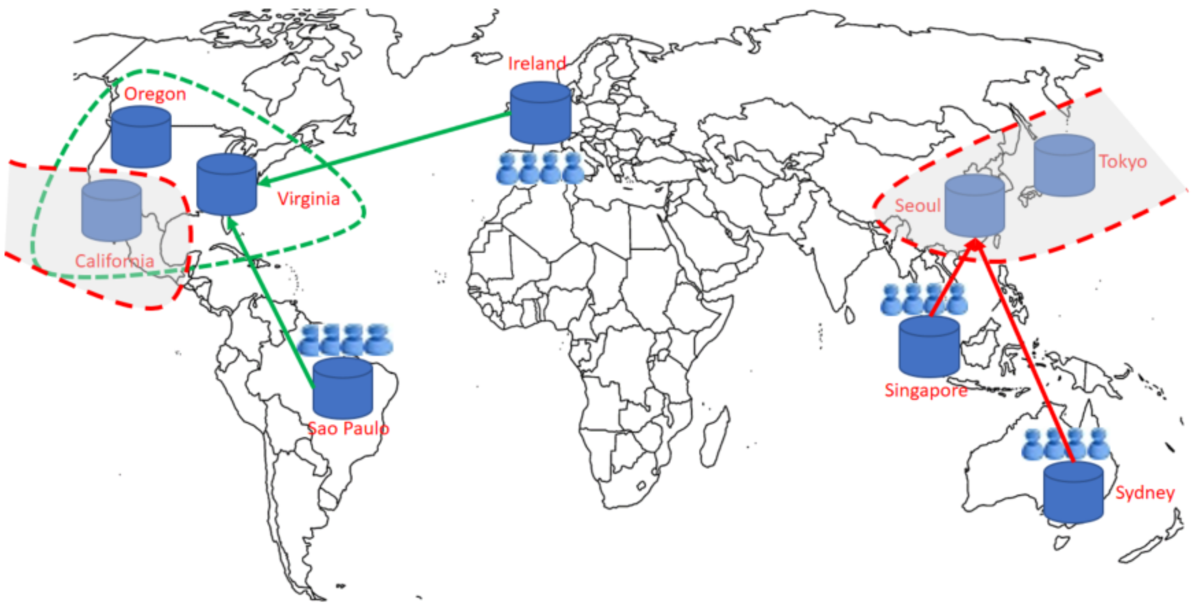


Figure 2.4: A placement scenario that shows the importance of considering different optimization aspects to minimize the average transaction latency.

are placed in datacenters that have clients.

We summarize the placement lessons we learned during DB-Risk as follows:

- The placement search space is currently huge considering the number of core and edge datacenters.
- Even with a restricted search spaces, manual approaches tend to use intuitive but not optimal placement strategies that result in higher average transaction latency.
- Placement optimization frameworks are necessary to find the replica placement that achieves the minimum average transaction latency

Hence, this chapter presents GPlacer, a placement optimization framework for multi-row transactional workloads with strong consistency requirements.

2.3 Background

Global-scale placement is the problem of deciding which datacenters should store a full or a partial replica of an application's data subject to a certain objective function. Objective functions can vary between minimizing the deployment monetary cost [208, 146] or minimizing the data access latency for a defined set of client operations [194]. Objective functions are always constrained by the application requirements (e.g., availability, upper bound access time, or bandwidth usage). In this section, we present our storage model and our assumptions about the workload distribution, the application requirements, and the objective function.

The universe of datacenters, denoted by DC, is defined as all the datacenters that can host an application[1] instance and/or a replica[2] of the database. We assume that the application is deployed on a subset of the datacenters $DC_{app} \subseteq DC$. The clients of the application are scattered around the globe and for simplicity, we assume that clients are collocated with their closest datacenter. The application is deployed in all the datacenters that have clients. However, these datacenters can be different from the datacenters that host replicas. $DC_{db} \subseteq DC$ is the subset of datacenters that host a database replica. We assume that the database is *partitioned* and all the partitions are *fully replicated* in DC_{db}.

2.3.1 The Transaction Management Protocol

The clients of the application access the globally-distributed storage by issuing transactions, which are collections of read and write operations followed by a commit or an abort. GPlacer considers transactions with *strong guarantees*, *i.e.*, serializability [57]. Strong consistent transactions on globally-distributed data require more coordination than weaker forms of access like eventual consistency or single-key atomicity— thus mak-

[1] *Application* refers to the middle tier logic.
[2] *Replica* refers to a copy of the backend database.

ing strongly consistent transactions more expensive. A strong consistency transactional interface is more natural to programmers and is required by many applications. Thus, we adopt such strong access semantics for GSDM as others did from both academia [154, 134] and industry [79].

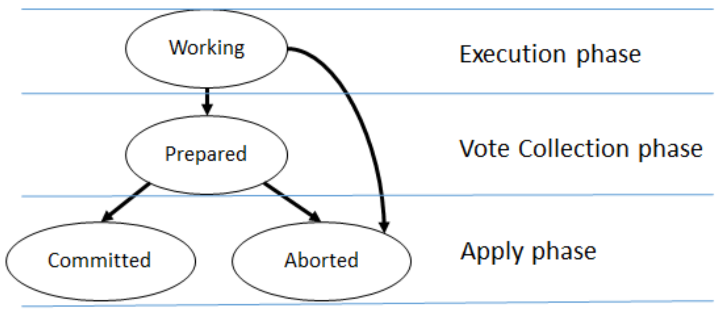


Figure 2.5: The state-transition diagram of a distributed transaction adopted from [111].

We adopt the distributed transaction model proposed by Gray and Lamport [111]. Figure 2.5 shows the different states of a transaction and the corresponding execution phases. A client drives the execution of a transaction in three phases. The details of these three phases differ across different transaction management protocols. However, the abstract semantics behind these phases are the same for all the protocols that provide the same strong transactional guarantees. The three phases of a transaction are: the execution phase, the vote collection phase, and the apply phase.

During the execution phase, the transaction is in the working state when read and write operations are processed. We assume that writes are locally buffered at the client and the updates are sent to the data replicas in the second phase. This assumption is widely used in many geo-replicated transaction management protocols [79, 154, 134]. For a read operation, clients communicate with their read coordinator, r_c. r_c processes a read request and responds back to the client. The RTT between a client c and r_c is denoted by RTT_{c-to-r_c} and the time for r_c to process the read request is denoted by P_{r_c}. The total execution phase latency is denoted by $L_e = n_r.(RTT_{c-to-r_c} + P_{r_c})$ where n_r is the average number of read requests per transaction. The transaction management protocol determines the values of n_r, RTT_{c-to-r_c}, and P_{r_c}. Some protocols assume that the client is the read coordinator. In such case, $RTT_{c-to-r_c} = 0$. Also, some protocols require that the client issues read requests one by one and others require that the client should batch all the reads in one request. The processing time P_{r_c} depends on how many replicas r_c should communicate with to serve a read request. In our model protocol, r_c has to communicate with a majority quorum to serve each read (we also consider read optimizations later in Section 2.3.2). As write requests are locally buffered, their effect on the execution phase latency is negligible. During the execution phase, a client might decide to abort the transaction by simply moving the transaction to the aborted state. However, if the client decides to commit the transaction, the transaction is moved to the prepared state and the vote collection phase starts.

During the vote collection phase, the client sends the transaction's details to the commit coordinator c_c which is responsible for coordinating with the other replicas to decide either to commit or to abort the transaction. Typically, the c_c uses either two-phase commit (2PC) with two-phase locking (2PL) [79] or quorum-based approaches (*e.g.*, Paxos) with 2PL [154]. The vote collection phase can be mapped to the first phase of the 2PC or the first round of Paxos. The latency of the voting phase is denoted by $L_v = \frac{RTT_{c-to-c_c}}{2} + RTT_{c_c-to-p}$ where RTT_{c-to-c_c} is the RTT between c and c_c and RTT_{c_c-to-p} is the round-trip time between the c_c and the furthest participant p included in the voting process.

If the decision of the vote collection phase is to abort, the client is notified, the transaction is moved to the aborted state, the other participants are asynchronously updated, and the obtained locks are released. However, if the decision is to commit, c_c starts the apply phase by sending the apply message to all the participants. Upon receiving the apply message, the participants commit the transaction, release the locks, and respond back to the coordinator. c_c notifies the client and the transaction is moved to the committed state. The latency of the apply phase is denoted by $L_a = RTT_{c_c-to-p} + \frac{RTT_{c-to-c_c}}{2}$ as the apply phase takes a round of communication with the participants in addition to the time to inform the client about the decision. A transaction commit latency L_c is the time spent in the vote collection phase and the apply phase combined: $L_c = RTT_{c-to-c_c} + 2 \cdot RTT_{c_c-to-p}$. The transaction latency L_t is the time from the beginning till the end of a transaction: $L_t = L_e + L_c = n_r \cdot (RTT_{c-to-r_c} + P_{r_c}) + RTT_{c-to-c_c} + 2RTT_{c_c-to-p}$.

GPlacer optimizes the average overall transaction latency over all the clients in different datacenters. It is designed to optimize the placement for a wide class of transaction commitment protocols.GPlacer focuses on optimizing for multi-master Paxos-based protocols [154, 111] and for leader-based protocols [79] both on *partitioned fully replicated databases*. In multi-master Paxos, each replica can act as the commitment coordinator role and uses the two rounds of Paxos for both transaction isolation and replication. However, in leader-based protocols, a transaction can fall into one of two categories: single-partition transactions or multi-partition transactions. Single-partition transactions span only one partition and the isolation between transactions that span this partition is managed by the leader of this partition. Multi-partition transactions span multiple partitions and typically 2PC is used between the leaders of the partitions involved in a transaction to achieve isolation. In both categories, partition leaders replicate the updates of committed transactions to a majority quorum of their partition replicas using only the second round of Paxos.

In Section 2.4, we formalize GPlacer. We use Replicated Commit [154] as our protocol model where reads are served from a majority of the replicas and commits are done using the two rounds of Paxos for isolation and replication. In Section 2.3.2, we explain some commonly used optimization to reduce the execution phase latency. In Section 2.7, we explain how to extend GPlacer to optimize placement for leader-based protocols like Spanner [79].

2.3.2 Read optimizations

In this section, we present two widely-used read optimizations that are considered in GPlacer. A read request latency $L_r = (RTT_{c-to-r_c} + P_{r_c})$. The first optimization, **optimistic read**, aims to eliminate the read processing time P_{r_c}. The second optimization, **passive replica read** aims to eliminate the time to reach the coordinator RTT_{c-to-r_c} and the processing time P_{r_c} by bringing a copy of the data to the client's datacenter. We define two different types of replicas a datacenter can host: *active replica* or *passive replica*. An active replica contributes synchronously in the vote collection and the apply phases and can act the coordinator and the participant roles. However, a passive replica is a read-only replica. It is asynchronously updated after the transactions are committed.

Optimistic read aims to eliminate the read request processing time by optimistically reading data values from the closest active replica without any coordination with other active replicas. This optimization has been introduced before as early as in Postgres-R local reads [130] and as fast reads in Zookeeper [123]. Applying optimistic reads require validating the value read in the commit phase to guarantee the freshness of the optimistically read values in the execution phase. In Spanner [79], reads are served by the leader of each partition. However, optimistic reads can be beneficial by reading from the closest partition replica instead from the partition leader. In Replicated Commit [154], a client is required to read from a quorum of the replicas. Applying optimistic read reduces the read latency by reading from one replica instead of a quorum.

Passive replica read aims to completely eliminate the read latency by processing read requests from a local read-only replica or a **passive replica**. The reason behind this naming is that a passive replica does not participate actively in the commit decision. Therefore, adding more passive replicas does not affect the commit latency. However, these replicas need to be asynchronously updated which increases the bandwidth required per committed transaction. Also, having many passive replicas increases the deployment cost. Data read from a passive replica needs to be validated in the commit phase to guarantee freshness. If the data is frequently updated at the active replicas, the data values read from a passive replica will be stale which increases the transaction abort rate. The concept of passive replica read has also been introduced in [194] as *weak reads*.

GPlacer chooses the set of active replicas and the set of passive replicas. In addition, it assigns r_c and c_c for clients in every datacenter. Application requirements are given as inputs to the framework. GPlacer takes as an input the fault tolerance level f, the total number of replicas t, and the workload distribution. f determines the number of active replicas and t determines the number of passive replicas. The workload distribution determines which datacenters should have active replicas, which should have passive replicas, and which should not have a replica at all. GPlacer finds placements that optimize the overall average transaction latency for strongly consistent multi-row transaction workloads. However, systems that require non-transactional or weakly consistent operations can easily be tuned in GPlacer's prototype but we do not discuss them since they were treated in previous works [44, 208].

2.3.3 Notation

Table 2.2 summarizes the notation used throughout the chapter.

2.4 Framework formulation

GPlacer finds the placement that minimizes the average transaction latency for partitioned fully replicated databases. As explained in Section 2.3, Paxos-based protocols use majority quorums for both transaction isolation and replication while leader-based protocols use majority quorums only for replication. Placement for Paxos-based protocols

DC	the set of all datacenters of size —DC—
DC_{app}	$DC_{app} \subseteq DC$ is the subset of datacenters that host an application
DC_{db}	$DC_{db} \subseteq DC$ is the subset of datacenters that host a database replica
c	a client who submits transactions
r_c	a transaction read coordinator replica
P_{r_c}	the time for r_c to process a read request
n_r	the average number of read requests per transaction
c_c	a transaction commit coordinator replica
p	the furthest participant replica in a transaction voting phase
RTT_{a-to-b}	the round-trip time from site a to site b
L_e	$L_e = n_r.(RTT_{c-to-r_c} + P_{r_c})$ is the latency of a transaction execution phase
L_c	$L_c = RTT_{c-to-c_c} + 2 \cdot RTT_{c_c-to-p}$ is the latency of a transaction commit phase
L_t	$L_t = L_e + L_c$ is the overall transaction latency
L_p	a request (either read or commit) processing time at a coordinator
f	the number of datacenter scale outages that should be tolerated
c_i	the number of clients at datacenter i
P_{sp-txn}	percentage of single-partition transactions
P_{mp-txn}	percentage of multi-partition transactions ($P_{mp-txn} = 100 - P_{sp-txn}$)

Table 2.2: Summary of GPlacer notation.

requires finding the subset of datacenters that should host replicas and the majority quorums used by the protocol. Leader-based protocols requires an additional step of placing the leaders of different database partitions on the replicas chosen in the first step. In Section 2.4.1, we formulate the placement problem into an exhaustive search model for Paxos-based protocols. This model evaluates all the possible placement combinations and returns one placement that achieves the minimum average transaction latency for a given workload. The model finds the placement $DC_{db} \subseteq DC$ and the majority quorums for each replica in this placement that optimizes the objective function. Although the model finds the optimal placement, due to the model complexity, it does not scale with the number of datacenters *when multiple cloud providers and edge datacenters are considered*. Therefore, in Section 2.4.2, we introduce two replica-placement heuristics to find placements that are close to optimal among hundreds of datacenters. The performance and the resulting placements of these heuristics are evaluated in Section 2.6.

2.4.1 Model formulation

The **inputs** of GPlacer fall into *two* categories:

- **Datacenter information**: this includes the number of datacenters $|DC|$ and the average RTT between every pair of the datacenters.
- **Application information**: this includes the number of datacenter scale outages f the deployment should tolerate and the application workload distribution. The workload distribution is denoted by c_i and represents the number of clients c at datacenter i.

The **outputs** of GPlacer include:

- The list of datacenters that should host a database replica.
- The read and the commit coordinator of clients at each datacenter. Clients at one datacenter share the same read and commit coordinators.

As the placement problem can be represented as an optimization model, we first implemented the placement model as an integer program and used the open source GLPK solver [14]. However, the solver could not efficiently scale with the number of datacenters. Many of the optimization constraints are conditional and to convert them to linear constraints, multiple binary output variables are introduced. The binary outputs and their related constraints are quadratic in the number of the datacenters $O(|DC|^2)$. In addition, GLPK solver introduced performance overhead. Therefore, to conduct a fair comparison with the replica-placement heuristics, we implement both the exhaustive search and the heuristics in Java. The **objective function** of the placement model is to minimize the average transaction latency of all the clients in all the datacenters. Algorithm 1 shows the details of the exhaustive search model.

Algorithm 1 Evaluates all the possible placement combinations and returns the one that achieves the minimum average latency for given application requirements.

```
Input: $f$, $|DC|$, $RTT_{ij}\ \forall i, j \in DC$ and the Set $C = \{c_i\ \forall i \in DC\}$ Output: $DC_{db}$, $DC_{rc}$, and $DC_{cc}$
1: $DC_{db}, DC_{rc}, DC_{cc} \leftarrow \{\}$, $minL \leftarrow MaxInt$
2: for each Set $S \subset DC, |S| = 2f + 1$ do
3:     $l, S_{rc}, S_{cc} \leftarrow evalLat(S, RTT, C)$
4:     if $l < minL$ then
5:         $minL \leftarrow l$, $DC_{db} \leftarrow S$
6:         $DC_{rc} \leftarrow S_{rc}$, $DC_{cc} \leftarrow S_{cc}$
7:     end if
8: end for
```

Algorithm 1 evaluates all the possible subsets of the input datacenters of size $2f + 1$ and returns the one that minimizes the average transaction latency. The function *evalLat*, in line 3, has different implementations based on the enabled read optimizations. When all the read optimizations are disabled, *evalLat* assumes that the read coordinator and the commit coordinator are collocated with the client who issues a transaction and reads are served from a majority quorum of replicas. However, if *optimistic read* is enabled, the read latency is updated to the RTT to the closest chosen replica from the client. Also, if *passive replica read* is enabled, the read latency is updated to zero as all the clients perform read operations from a local replica.

2.4.2 Replica-placement heuristics

Although Algorithm 1 finds the optimal placement among all the possible placements, it does not efficiently scale when the total number of the datacenters, $|DC|$, or the number of the replicas, $|DC_{db}|$, increases. Our experiments show that choosing 7 replicas out of 60 datacenters $\binom{60}{7}$ takes 2 hours while choosing 7 replicas out of hundreds of datacenters (which is the case when we consider edge datacenters) could take years. Therefore, we present two replica-placement heuristics that efficiently find placements with sub-optimal average transaction latency. These replica-placement heuristics consider the *two main aspects* that affect the transaction latency; the latency between the clients and the replicas and the latency between the replicas themselves. The running-time of these heuristics is polynomial in the total number of the datacenters. The performance and the resulting placements of these heuristics are compared to the exhaustive search results in Section 2.6.2.

The first replica-placement heuristic is shown in Algorithm 2. It uses an iterative greedy algorithm to choose the replicas. It starts with an empty set of chosen replicas $DC_{db} \leftarrow \{\}$, line 1, and at each iteration, it adds one replica to DC_{db} until $2f+1$ replicas are chosen. The inner loop, lines 4-14, evaluates the effect of adding each unchosen replica to DC_{db} on the average transaction latency and the replica that achieves the minimum latency is added to DC_{db}, line 15. *evalLat* is the same evaluation function introduced in Algorithm 1 line 3. The intuition behind this heuristic is that choosing the best candidate at each step should lead to a solution that is optimal or close to the optimal.

The second replica-placement heuristic is presented in Algorithm 3. It is based on the K-Means algorithm. It assigns weights to every datacenter, initially equals to the number of clients in this datacenter; line 4. A datacenter weight is updated according to the number of quorums it participates at; line 13. Datacenter weights are iteratively updated and datacenters are sorted by their weights. The top $2f+1$ datacenters are chosen to host replicas in lines 5 and 18. The algorithm evaluates the placement in every iteration and stops when the average transaction latency converges. To avoid fast convergence to a local minimum, a minimum iteration count is required before terminating the algorithm; lines 1 and 7. The minimum evaluated placement is saved to make sure that the final placement does not achieve higher transaction latency than any placement that has been

Algorithm 2 Greedily adds one replica at a time achieving the minimum average latency at each iteration.

Input: f, $|DC|$, $RTT_{ij}\ \forall i,j \in DC$ and the Set $C = \{c_i\ \forall i \in DC\}$ **Output:** DC_{db}, DC_{rc}, and DC_{cc}

```
1: $DC_{db}, DC_{rc}, DC_{cc} \leftarrow \{\}$
2: while $|DC_{db}| < 2f + 1$ do
3:     $S \leftarrow DC_{db}$, $minL \leftarrow MaxInt$, $minDC \leftarrow \phi$
4:     for all $dc \in DC$ do
5:         if $dc \notin S$ then
6:             $S \leftarrow S \cup \{dc\}$
7:             $l, S_{rc}, S_{cc} \leftarrow evalLat(S, RTT, C)$
8:             if $l < minL$ then
9:                 $minL \leftarrow l$, $minDC \leftarrow dc$
10:                $DC_{rc} \leftarrow S_{rc}$, $DC_{cc} \leftarrow S_{cc}$
11:            end if
12:            $S \leftarrow S \setminus \{dc\}$
13:        end if
14:    end for
15:    $DC_{db} \leftarrow DC_{db} \cup \{minDC\}$
16: end while
```

evaluated before.

2.5 Surprising Placement Lessons

During the development of GPlacer, we learned some counter-intuitive lessons about data placement that exploit the diversity and the asymmetry of the WAN links to decrease the execution and the commit latencies, hence the transaction latency. (The transaction latency L_t is sum of the execution latency L_e and the commit latency L_c.) In this Section, we explain the details of these placement lessons and their effect on transaction latency.

2.5.1 Request handoff as a Transaction Execution Optimization

A client executes either read or commit requests. The latency of these two requests can be abstracted as the sum of: RTT_{c-to-r_c} or RTT_{c-to-c_c}, the round-trip time between

Algorithm 3 Assigns weights to datacenters and iteratively chooses the top weighted $2f+1$ to host replicas.

Input: f, $|DC|$, RTT_{ij} $\forall i, j \in DC$, t and the Set $C = \{c_i \ \forall i \in DC\}$

Output: DC_{db}, DC_{rc}, and DC_{cc}

```
1: $minIter \leftarrow t$, $iter \leftarrow 0$
2: $DC_{db}, DC_{rc}, DC_{cc} \leftarrow \{\}$
3: $l_{n-1}, l_n \leftarrow MaxInt$
4: $Weights \leftarrow \{c_0, c_1, ..., c_{|DC|}\}$ // Initialize weights with the number of clients at each datacenter.
5: $DC_{db} \leftarrow top(sort(Weights), 2f+1)$ // Sort on weights and choose a placement of the top $2f+1$.
6: $l_n, DC_{rc}, DC_{cc} \leftarrow evalLat(DC_{db}, RTT, C)$
7: while $l_n < l_{n-1}$ || $iter++ < minIter$ do
8:     $l_{n-1} \leftarrow l_n$
9:     $NewW \leftarrow \{0, 0, ..., 0\}$ // New Weights
10:    for all $dc1 \in DC$ do
11:        for all $dc2 \in DC$ do
12:            if $dc2 \in nearestQuorum(dc1)$ then
13:                $NewW[dc2] += Weights[dc1]$
14:            end if
15:        end for
16:    end for
17:    $Weights \leftarrow NewW$
18:    $DC_{db} \leftarrow top(sort(Weights, 2f+1)$
19:    $l_n, DC_{rc}, DC_{cc} \leftarrow evalLat(DC_{db}, RTT, C)$
20: end while
```

the client and a read or a commit coordinator and L_p, the time for the coordinator to process this request.

Therefore, the request latency is mainly affected by the distance between the client and the coordinator, the distance between the coordinator and the participants, and finally the number of communication rounds required between the coordinator and the participants to serve the request. Different transaction management protocols choose the coordinator based on some intuitive heuristics. In [154], Mahmoud et al. assume that the *client* is the coordinator of a transaction. In Spanner [79], the 2PC coordinator is randomly chosen from the leaders of the partitions involved in a multi-partition transaction. In [166], Nawab et al. choose the coordinator to be the closest replica to the client. However, the choice of the coordinator can drastically affect the request latency. To illustrate this effect, we provide two examples of 2PC and Paxos deployments to show that carefully choosing the coordinator can save up to 48% of the average latency.

Two-phase commit: assume there are three data partitions X, Y, and Z deployed in three AWS datacenters in SP, V, and I respectively. Now, assume a client in datacenter I wants to commit a transaction t that updates the elements $x_1 \in X$, $y_1 \in Y$, and $z_1 \in Z$. The commit latency at any coordinator equals to double the RTT between the coordinator and the furthest involved partition leader. Therefore, if the client chooses the leader of partition Z in datacenter I to be the commit coordinator, the resulting commit latency is $2 \cdot max(RTT_{IV}, RTT_{ISP}) = 2 * 185 = 370ms$. Although the time between the client and the coordinator is reduced to zero, the latency is still high because the coordinator is relatively far from SP. However, if the client chooses the leader of partition Y in datacenter V to be the commit coordinator, the resulting commit latency is $RTT_{IV} + 2 \cdot max(RTT_{VI}, RTT_{VSP}) = 73 + 2 * 121 = 315ms$ saving around 15% of the commit latency without modifying any constraint of the original 2PC protocol. Also, when datacenter V is the 2PC coordinator, the participant at datacenter I will be notified about the commit decision after $\frac{RTT_{IV}}{2} + max(RTT_{VI}, RTT_{VSP}) + \frac{RTT_{IV}}{2} = 36.5 + 121 + 36.5 = 194ms$. The participant at datacenter I can directly inform the client with the decision saving around 48% of the latency obtained when I is chosen to be the coordinator.

Paxos: assume there are five replicas of the database in datacenters I, V, SP, O, and C as shown in Figure 2.6. A client in datacenter SP wants to commit a transaction that requires to execute the two rounds of Paxos to reach a consensus about the commit decision. The latency of the two rounds of Paxos equals to double the RTT between the coordinator and the furthest replica in the closest majority to the coordinator. Therefore, if the client in SP chooses the replica in SP to be the coordinator, the resulting commit latency equals to $2 \cdot max(RTT_{SPSP}, RTT_{SPV}, RTT_{SPO}) = 2 \cdot max(1, 121, 182) = 2*182 = 364ms$. However, if the client in SP, delegates the coordination to the replica in V, the resulting commit latency will be $RTT_{SPV} + 2 \cdot max(RTT_{VV}, RTT_{VI}, RTT_{VC}) = 121 + 2 * 73 = 267ms$ saving around 26.6% of the commit latency obtained when SP is chosen to be the coordinator.

We presented a primitive version of the handoff idea in [217]. To generalize, for any request R from a client at datacenter A, it might be beneficial to handoff this request to a replica at datacenter B if the summation of RTT_{AB} and the time for datacenter B to serve this request L_B are less than L_A, the time to serve this request at datacenter A. In other words, request handoff from datacenter A to datacenter B is beneficial if $L_A > RTT_{AB} + L_B$. This optimization is widely applicable on different protocols and different request types.

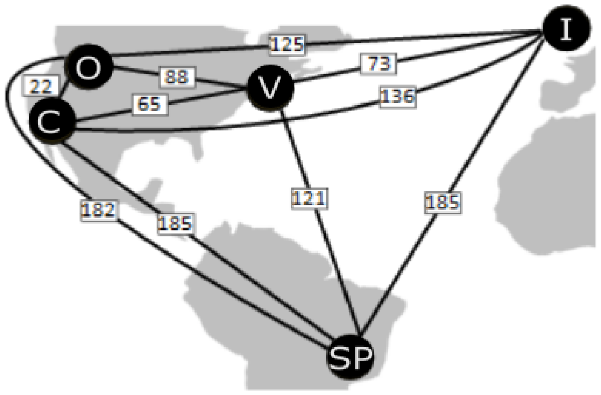
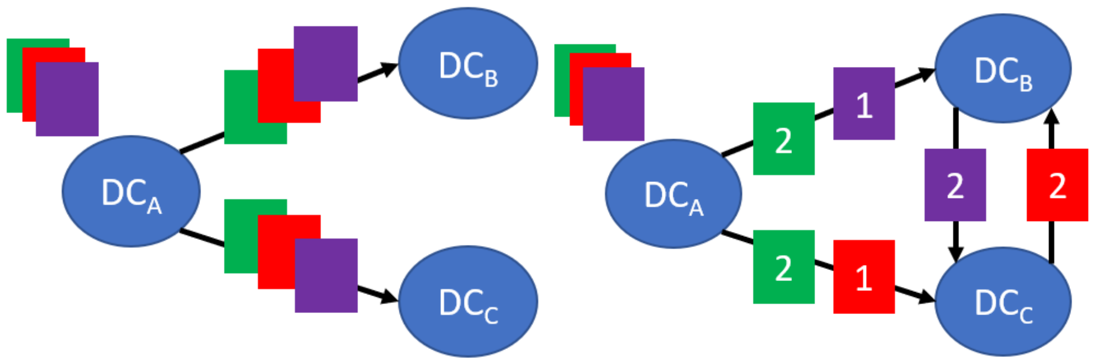


Figure 2.6: Five replicas of the database are deployed in datacenters I, V, SP, O, and C.

2.5.2 Request Handoff for inter-datacenter Big Data Replication

In Section 2.5.1, we explain how request handoff leverages asymmetry and diversity in network latency to reduce the latency of distributed transactions. In this section, we explain how Zhang et al. [224] use request handoff to reduce the latency of inter-datacenter big data replication. Finally, we explain how request handoff can be presented



(a) Replication over direct links (b) Replication leveraging network overlay paths

Figure 2.7: An example to illustrate how overlay network paths can be used to reduce big data transfer time.

as a general inter-datacenter network optimization.

Zhang et al. [224] show that inter-datacenter multicast forms 91% of the inter-datacenter traffic in Baidu. Therefore, the authors present BDS, a centralized application-level multicast service that leverages overlay network paths to accelerate large-scale inter-datacenter big data replication. We use the following example shown in Figure 2.7 from [224] to illustrate the link between BDS's overlay network paths and request hand-off. Suppose datacenter DC_A shown in Figure 2.7 wants to replicate a 3 GB file to both datacenter DC_B and datacenter DC_C. Also, assume that the bandwidth capacity between any two datacenter pair is 1GB/s. As shown in Figure 2.7a, if DC_A uses direct links with both DC_B and DC_C to transfer the file, it takes 3 seconds to transfer the file to both datacenters considering only transmission delays. However, if DC_A uses overlay network paths, the transmission time can be significantly reduced. Figure 2.7b shows an example of how DC_A can use overlay network paths to shorten the transmission latency. First, DC_A divides the file into three 1GB segments (shown in purple, red and green in Figure 2.7b) . Then, DC_A sends the first 1GB segment (purple) to DC_B and the second 1GB segment (red) in parallel to DC_C in 1 second. Now DC_A can send the third 1GB segment (green) to both DC_B and DC_C while DC_B and DC_C exchange the first and the second segments using overlay network paths in 1 additional second. Considering only transmission time and assuming that the links between DC_B and DC_C are available and free, DC_A can leverage the overlay network paths to reduce the transmission time to 2 seconds saving 33% of the transmission time. In this example, DC_A leverages the available links between DC_B and DC_C to *hand off* the transmission of the purple segment from DC_B to DC_C instead of from DC_A to DC_C. Similarly, the transmission of the red segment is handed off to DC_C where DC_C sends the red segment to DC_B instead of doing the sending from DC_A to DC_B.

It is important to highlight the differences between the handoff examples in Section 2.5.1 and these examples in Section 2.5.2. In Section 2.5.1, both Paxos and 2PC protocols exchange small sized messages over wide area inter-datacenter networks. Therefore, propagation delays represented by the length of the links are the main bottleneck while transmission delays represented by link bandwidth are neglected. Handoff reduces the propagation latency by introducing little bandwidth overhead by the handoff messages. On the other hand, the examples in Section 2.5.2 mainly suffer from bandwidth limitation and hence transmission delays are the main bottleneck while propagation delays are neglected. In this case, handoff reduces the transmission latency by introducing little propagation delays on the overlay network paths. The examples in both Section 2.5.1 and Section 2.5.2 show that request handoff can be leveraged to significantly reduce the latency of an inter-datacenter network request. Request handoff is a network optimization that leverages either diversity and asymmetry in inter-datacenter propagation latency or in inter-datacenter transmission latency to optimize the overall latency of an inter-datacenter request.

2.6 GPlacer Evaluation

A performance evaluation study of the request handoff, the read optimizations, and the proposed heuristics is conducted in this section. In our study, we first evaluate the effect of the read optimizations and the request handoff on execution and commit latencies in Section 2.6.1. In Section 2.6.2, we evaluate the performance of the replica-placement heuristics introduced in Section 2.4.2. We compare the running time and the resulting placement latencies of these heuristics to the running time and the placement latencies of the exhaustive search algorithm in Algorithm 1.

2.6.1 Placement optimizations

Experimental setup

We use the placement scenario in Figure 2.6 to evaluate the effect of read optimizations and request handoff on the transaction latency. Request handoff exploits the diversity of the WAN links to decrease the transaction latency and this scenario shows a good example of this diversity, $RTT_{SPC} > 8RTT_{OC}$. Amazon EC2 machines in Ireland (I), Virginia (V), São Paulo (SP), Oregon (O), and California (C) datacenters are leveraged as infrastructure for our experiments. Larger machines are used in datacenters C and O so that we can measure the handoff effect without causing throttling in datacenters C and O. Compute optimized machines are used because computing is the main source of contention in our experiments. We use one compute optimized (c4.large) machine with 2 vCPUs and 3.75 GB of RAM in datacenters V, I, and SP while we use one compute optimized (c3.4xlarge) machine with 16 vCPUs and 30 GB of RAM in datacenters C and O. We assign active replicas to servers in C, O, and V while we assign passive replicas to servers in I and SP. These machines use HBase [15] as the underlying persistent data store. The average RTTs observed between different datacenters are shown in Table 2.1.

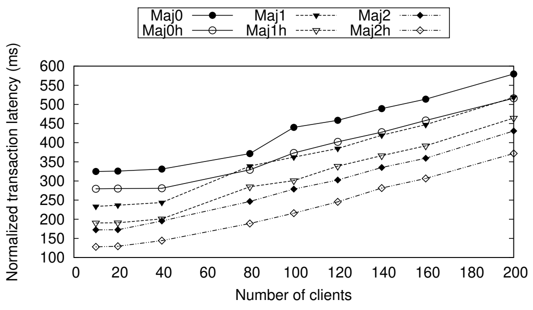
The observed RTTs are sampled over 48 hours using AWS nano machines pinging each other. The data is fully replicated in all five datacenters and an optimistic Paxos-based concurrency control protocol is used. A transaction requires two majority rounds to commit and the read-set is validated at commit time. We implemented multiple versions of the protocol based on how read requests are processed in the execution phase. $Maj0$ is conservative and requires read requests to be processed from a majority of the active replicas. $Maj1$ implements the optimistic read optimization and requires read requests to be processed from one active replica. $Maj2$ implements the optimistic read and passive replica optimizations and processes read requests from either active or passive replica. Transaction commitment is implemented the same way in all three versions. The commit handoff optimization is applied on all three versions and it only changes the way a commit coordinator is chosen. For this, we implemented $Maj0h$, $Maj1h$, and $Maj2h$ to apply the handoff optimization on the three protocol implementations. We compare the average obtained commit and transaction latencies for all the three implementations with and without applying the handoff optimization. In addition, we compare transaction throughputs and abort rates for all three implementations.

Dedicated client machines in each datacenter generate client workloads. Each client machine is configured with a read coordinator and a commit coordinator. Also, client machines execute a workload thread per client. Clients are uniformly distributed among the 5 datacenters in all the experiments unless otherwise stated. Client machines use YCSB [78] to generate workloads. Since YCSB is not designed to generate multi-record transactions, we use Transactional YCSB (T-YCSB) [84], an extended version of YCSB that generates multi-record transactions, for this purpose. T-YCSB generates transactions that consist of read and write operations on different data records followed by a commit. Each transaction is configured to have five operations. The ratio of read to write operations is 1:1 unless otherwise specified. Read and write operations choose a key from a pool of 50000 keys following a zipfian distribution. This small number of keys enables us to observe the performance of the system under contention. Each client can have only one outgoing transaction. Clients submit a new transaction as soon as they receive a decision for their outgoing transaction. Each experiment runs for 10 minutes.

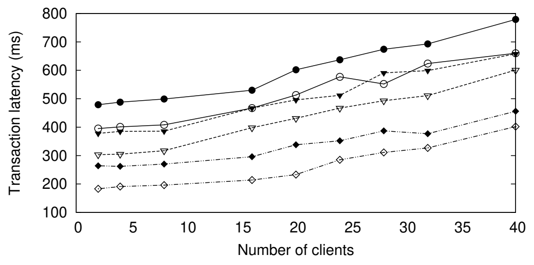
Experimental results

Transaction latency. Active replicas are placed in only three datacenters C, O, and V. Therefore, a majority quorum consists of *two* active replicas. The $Maj0$ implementation assumes that clients at each datacenter drive their transactions(no handoff). Also, it assumes that reads have to be processed from at least two active replicas and commits have to be accepted by and applied to at least two active replicas. $Maj0h$ allows clients in SP to handoff their commit to O and clients in I to handoff their commits to C. $Maj1h$ and $Maj2h$ allow the same handoff plans while enabling optimistic reads in $Maj1h$ and optimistic reads and passive replica reads in $Maj2h$.

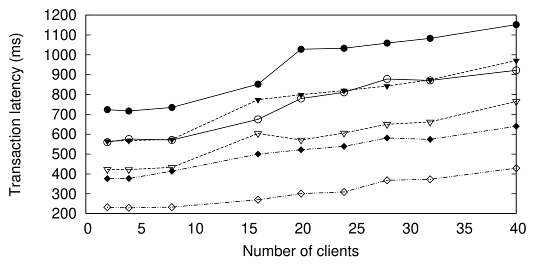
Figure 2.8 shows the effect of increasing the number of clients from 10 to 200 on transaction latency. As a transaction requires two round trips to a quorum of two active replicas to commit, clients in C and O have their location as an advantage that they can always achieve lower transaction latency and higher throughput than clients in other sites as long as the number of clients is equal in all the datacenters. Therefore, to measure the effect of the placement optimizations on transaction latency in isolation from the throughput, we use the normalized transaction latency as a comparison metric between different implementations. The normalized transaction latency L_{norm} is the average of the average transaction latency in all the datacenters $L_{norm} = \frac{L_C+L_O+L_V+L_I+L_{SP}}{5}$ where L_i is the average transaction latency at datacenter i. As shown in Figure 2.8a, applying read optimizations in $Maj2$ significantly enhances L_{norm} by 48% compared to $Maj0$. Also, the handoff in $Maj2h$ enhances L_{norm} by 26% compared to $Maj2$ leading to a



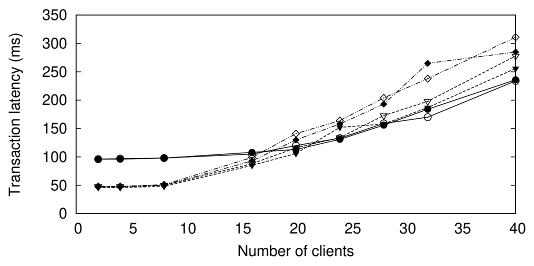
(a) Overall normalized transaction latency.



(b) Transaction latency in Ireland.



(c) Transaction latency in São Paulo.



(d) Transaction latency in California.

Figure 2.8: Transaction latency as number of clients increases. Figures 2.8a, 2.8b, 2.8c, and 2.8d share one plotting legend.

total enhancement of 60% compared to $Maj0$. Increasing the number of clients beyond 100 (20 at each datacenter) causes throttling in server machines.

This throttling leads to an increase in the overall transaction latency and a decrease in the benefit obtained from the applied optimizations. Figures 2.8b and 2.8c shows the effect of applying read optimizations and handoff on transaction latencies at I and SP respectively. As shown, read optimizations and handoff together in $Maj2h$ enhances transaction latency compared to $Maj0$ by 62% and 68% in I and SP respectively. Also, handoff in $Maj2h$ saves 30% and 38% of the transaction latency compared to $Maj2$ for clients in I and SP. Figure 2.8d presents the effect of the placement optimizations on the transaction latency in C. As shown, read optimizations significantly reduce the transaction latency in C by 49% as reads are served locally. This applies until throttling happens. After throttling, the transaction latency in C increases for $Maj2$ because

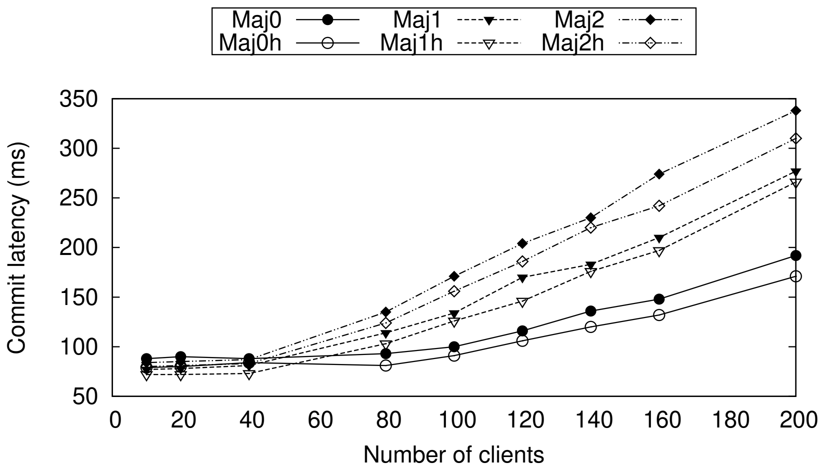
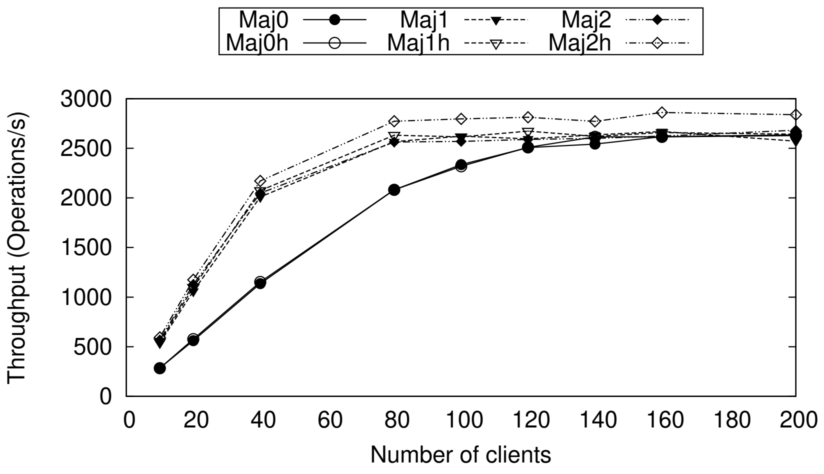


Figure 2.9: Overall average commit latency as number of clients increases.

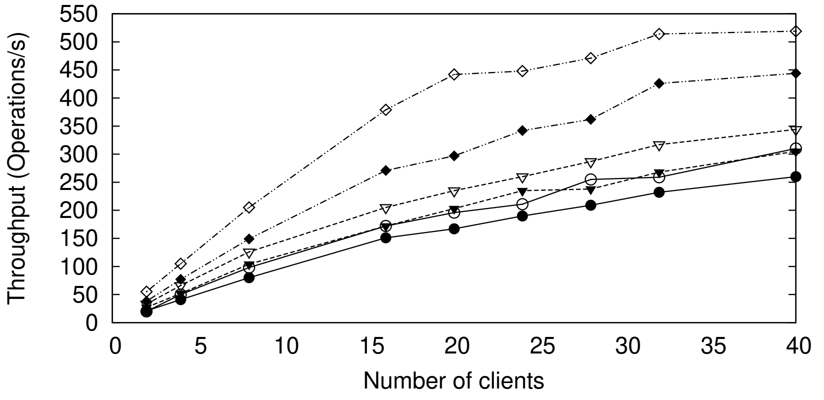
serving reads locally in all the datacenters increases the frequency of the transactions that are ready to commit in the system causing more contention in datacenters C and O. The handoff slightly increases the transaction latency in C and its negative effect is negligible before the throttling happens.

Commit latency. Figure 2.9 shows the effect of the placement optimizations on the overall average commit latency. While the normalized transaction latency is significantly enhanced by applying read optimizations, read optimizations negatively affect the overall commit latency. By reducing the execution phase latency, the number of active transactions that are ready to commit increases and leads to an increase in the commit latency. However, applying handoff enhances the overall average commit latency by $10 - 15\%$ in $Maj0h$, $Maj1h$, and $Maj2h$ compared to $Maj0$, $Maj1$, and $Maj2$ respectively.

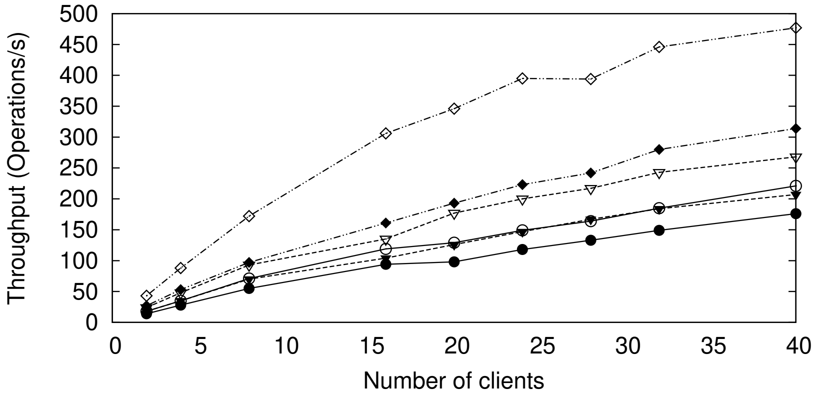
Throughput. The throughput, measured by number of operations per second, is presented in Figure 2.10. Figure 2.10a shows that applying read optimizations in $Maj1$ and $Maj2$ achieves $2x$ the throughput in $Maj0$ until hitting the thrashing point(≥ 100 clients). After that, throughput is slightly higher in $Maj1$, $Maj2$, and $Maj1h$ and about 8% higher in $Maj2h$. Throughput results in I and SP are shown in Figures 2.10b



(a) Overall throughput.



(b) Throughput in Ireland.



(c) Throughput in São Paulo.

Figure 2.10: Throughput as number of clients increases. Figures 2.10a, 2.10b, and 2.10c share one plotting legend.

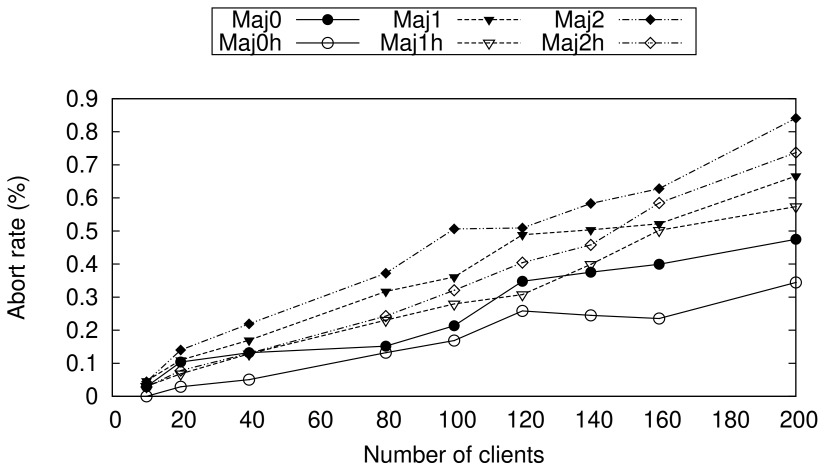


Figure 2.11: Overall abort rate as number of clients increases.

and 2.10c. These figures show a significant increase of 100% between $Maj0$ and $Maj2h$ in I and 170% between the same implementations in SP. Applying handoff not only significantly benefits I and SP but also benefits the overall throughput.

Abort rate. The abort rates are shown in Figure 2.11. The abort rate is a result of many factors, such as the amount of contention, the number of concurrent transactions, the lifetime of a transaction, among others. As shown, the overall abort rate is below 1% for all six different implementations. However, we observed two important patterns that are worth analyzing. First, read optimizations increase the abort rate by 100% for some experiment runs. Obtaining the read-set from a local copy increases the chances of reading a stale value and hence increasing transaction aborts. However, these stale values has a small life-time as all the passive replicas are asynchronously updated. Second, handoff decreases the abort rate by $25-30\%$ because a transaction's lifetime is shortened by reducing the overall transaction latency and specifically the high latency transactions in I and SP.

2.6.2 Replica-placement heuristics

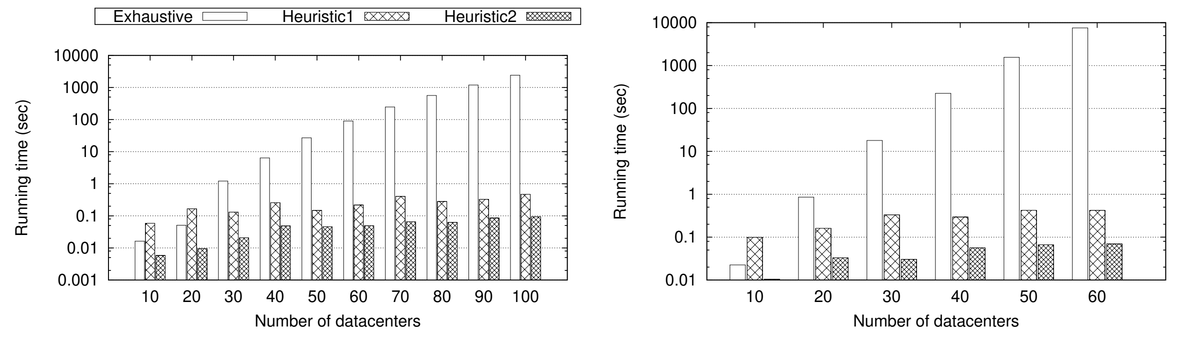
We evaluate the replica-placement heuristics in this section. This evaluation tries to answer two questions: *How fast can these heuristics find a placement? and how good is this placement compared to the optimal placement?*. For that, we compare the performance and the resulting placements of the proposed heuristics in Algorithms 2 and 3 to the performance and the resulting placements of the exhaustive search in Algorithm 1 at scale. We assume that optimistic reads and passive replica reads are enabled. Therefore, we use commit latency as a comparison metric as the transaction execution latency is negligible when reads and writes are served locally and none of the replicas are overloaded with requests. The proposed heuristics and the exhaustive search programs are all implemented in Java which allows us to conduct a fair comparison. The exhaustive search algorithm evaluates all possible placement combinations and returns the placement that achieves the minimum average commit latency for a certain workload. Algorithm 2 introduces a greedy heuristic that adds one replica at a time achieving the minimum average transaction latency at each iteration. Algorithm 3 is inspired by the K-Means algorithm and it assigns initial weight to each datacenter equals to the number of clients at this datacenter. Weights are updated based on the quorums a datacenter participates at and based on handoff.

Finding the optimal placement of five replicas within ten datacenters can be efficiently done. It requires the evaluation of only 252 different placements and the exhaustive search is sufficient in this case. However, in a more realistic setting, the number of datacenters around the globe, including edge datacenters, may easily exceed 4000 datacenters [13]. Also, it has been shown in [208] that it is economically efficient to deploy storage in datacenters of different cloud providers as none of them provides cheaper storage in all the deployment regions. To choose five datacenters out of 4000 datacenters requires to evaluate 8.5e+15 different placements. To get a sense of the space size and the running time, we evaluated the exhaustive search algorithm and the heuristics proposed in Section 2.4.2 using different datasets.

First, we generate multiple datasets of datacenters DC distributed around the globe with randomly chosen round-trip time $0 \leq RTT \leq 500$ ms. We also make sure that the triangle inequality holds among any three datacenters such that $\forall_{A,B,C \in DC}\ RTT_{AB} + RTT_{BC} \geq RTT_{AC}$. Second, we distribute the workload around the generated datacenters with ratios between $0 - 10$. We use the generated data as inputs to both the exhaustive search program and the placement heuristics. These experiments are run locally on an Intel Core i5-3210M CPU 2.50GHz with 8GB of RAM.

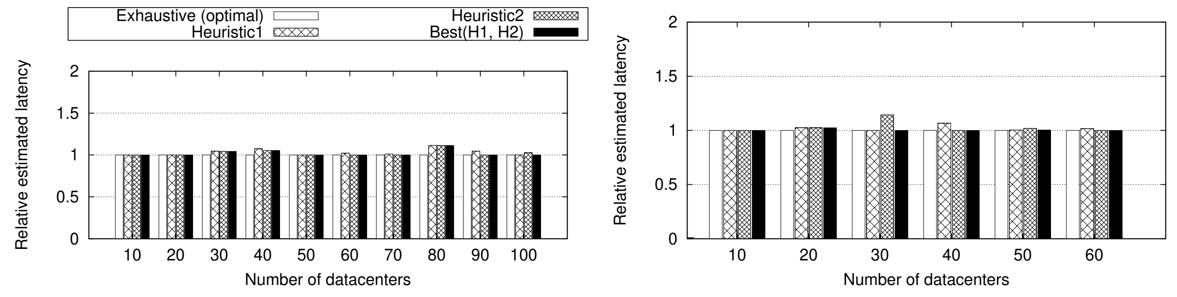
Running time. In this part of the evaluation, we answer the first question, namely *How fast can these heuristics find a placement?*. Figures 2.12a and 2.12b show a running time comparison between the exhaustive search and the placement heuristics when the number of replicas are 5 and 7 respectively. As shown, the running time of the exhaustive search grows exponentially with the number of datacenters while the running time of both heuristics are negligible(< 1 second). Also, the exponential power significantly increases as the number of replicas required to be placed increases. This shows that it is infeasible to use the exhaustive search when the datacenter set size exceeds few tens.

Resulting placements. The second part of the evaluation answers the second question about the quality of the placements found by the proposed heuristics. We compare the commit latency of the resulting placement to the optimal commit latency of the resulting placement decided by the exhaustive search. Figures 2.13a and 2.13b show the relative commit latency of the heuristics compared to the optimal commit latency when the number of placed replicas are 5 and 7 respectively. In these figures, the optimal commit latency is represented by 1.0. A relative comparison of the commit latency is shown for both the placement heuristics and the best of the two heuristics as well. As



(a) 5 replicas. (b) 7 replicas.

Figure 2.12: The running time (seconds), in log scale, of exhaustive search and placement heuristics as number of datacenters increases. Figures 2.12a and 2.12b show the running time when 5 replicas and 7 replicas are chosen respectively. Both figures share one plotting legend.



(a) 5 replicas. (b) 7 replicas.

Figure 2.13: A comparison of the resulting commit latency of placements by exhaustive search and placement heuristics as number of datacenters increases. Figures 2.13a and 2.13b compare the estimated latency when 5 replicas and 7 replicas are chosen respectively. Both figures share one plotting legend.

shown, the best of the two heuristics is optimal in 70% of the cases and within 5%−11% of the optimal in the rest of the cases. As the running times of both heuristics is negligible, we can always run both the heuristics and choose the best placement out of the two results. Figure 2.13 suggests that neither heuristics beats the other in all cases.

2.7 GPlacer Extensions

In this section, we discuss how GPlacer can be extended to optimize the placement for leader-based protocols. In leader-based protocols, a transaction can fall into one of two categories: single-partition transactions or multi-partition transactions. Single-partition transactions span only one partition and the isolation between transactions that span this partition is managed by the leader of this partition. Multi-partition transactions span multiple partitions and typically 2PC is used between the leaders of the partitions involved in a transaction to achieve isolation. In both categories, partition leaders replicate the updates of committed transactions to a majority quorum of their partition replicas using only the second round of Paxos.

The average transaction latency for leader-based protocols is affected by the following factors:

- The distance between the client and the partition leader.
- The distance between the partition leader and its replicas.
- The distance between different partition leaders involved in multi-partition transactions.
- The percentage of multi-partition transactions P_{mp-txn} (how often 2PC is required to be executed).

The first two factors mainly affect single-partition transactions while the last two factors mainly affect multi-partition transactions. Finding the optimal placement for leader-based protocols can easily become impractical. Consider a database with p partitions and we want to place the leaders of these partitions on r replicas. There are r^p different placement combinations and finding the optimal placement by checking all the combinations is impractical. For example, if a database has 500 partitions and we want to place these partitions among 5 replicas. To find the optimal leader placements, 5^{500} different combinations need to be evaluated. Therefore, different heuristics are usually used to limit the search space.

2.7.1 Leader-placement heuristics

A solution that *considers all the optimization aspects* should adapt the placement based on the percentage of the multi-partition transactions P_{mp-txn}, and the client distribution. Sharov et al. [194] place the leaders of all the partitions in one datacenter. Algorithm 4 implements the leader-placement heuristic introduced in [194] (**heuristic 1**). It iterates over all the replicas, Lines 2 and 4, and evaluates the latency assuming that all the partition leaders are placed in the currently evaluated replica (Line 5). The replica that achieves the minimum latency (Line 8) is returned (Line 11). This heuristic optimizes the placement when P_{mp-txn} is high. However, when P_{mp-txn} is low, placing the leaders of all the partitions in one datacenter can hurt the performance in addition to introducing a single point of failure.

The second heuristic (**heuristic 2**) is to *independently* place the leaders of different partitions. For every partition, place its leader at the same datacenter where it is accessed the most. This heuristic optimizes the placement when P_{mp-txn} is low and partitions are mostly accessed from one datacenter.

Placing all the partition leaders in one datacenter favors multi-partition transactions while independently placing them in multiple datacenters favors single-partition transactions. When the workload is a mixture of both transaction categories, both heuristics fail to optimize the placement. Therefore, we present a third heuristic (**heuristic 3**) that optimizes the placement when the workload is divided between the two categories.

Algorithm 4 Finds datacenter $dc_l \in DC$ that achieves the minimum average transaction latency assuming all the partition leaders are put together in one datacenter.

Input: DC, RTT_{ij} $\forall i, j \in DC$, and c_i $\forall i \in DC$
Output: dc_l

```
1: $dc_l \leftarrow \emptyset$, $l \leftarrow MaxInt$
2: for all $j \in DC$ do
3:     $tempL \leftarrow 0$
4:     for all $i \in DC$ do
5:         $tempL += c_i \cdot (RTT_{ij} + q_j)$ // $q_j$ is the time for replica $j$ to reach its closest quorum $q \subset DC$.
6:     end for
7:     if $tempL < l$ then
8:         $l \leftarrow tempL$, $dc_l \leftarrow j$
9:     end if
10: end for
11: return $dc_l$
```

This heuristic uses GPlacer to find a set of $2f + 1$ datacenters that should host a replica DC_{db} according to the workload distribution. Heuristic 3's idea is to place partition leaders among a set of datacenters DC_{db} that are centered among all clients. After using GPlacer to find DC_{db}, heuristic 3 runs Algorithm 5 to independently place the leaders of each partition among the set DC_{db}. Although, heuristic 2 independently places partition leaders in the universe of all datacenters DC, heuristic 3 limits the placement options to the set DC_{db} chosen by GPlacer. In Section 2.7.2, we compare the resulting placements of the three heuristics.

Algorithm 5 Places the leader of each partition among the chosen replicas and closer to the clients who access this partition the most.

Input: DC_{db}, and p_i $\forall i \in DC$ and $\forall p \in P$ // P is the set of all partitions and p_i is the percentage of access for partition p from datacenter i.
Output: $\forall_{p \in P}$ l_p

```
1: for all $p \in P$ do
2:     $i \leftarrow max(\forall_{j \in DC}\ p_j)$
3:     $l_p \leftarrow nearest(dc \in DC_{db}, i)$ // returns the nearest datacenter $dc \in DC_{db}$ to datacenter $i$.
4: end for
```

2.7.2 Leader-placement heuristics evaluation

We compare the expected average commit latency of the resulting leader placements using the three heuristics. The percentage of distributed multi-partition transactions is varied and the expected commit latency is calculated for the three heuristics. Before placing partition leaders in heuristic 3, we use GPlacer's exhaustive search algorithm to find DC_{db}, the set of datacenters that are allowed to host partition leaders. Then, heuristic 3 uses Algorithm 5 to place partition leaders among the set DC_{db}.

The commit latency of a single partition transaction is estimated as the RTT from the client datacenter to the partition leader datacenter plus the RTT from the partition leader datacenter to a majority of this partition replicas. The commit latency of a multi-partition transaction requires an additional 2PC among the involved partitions. In our evaluation, the 2PC's latency is neglected if all the involved partition leaders are placed together in the same datacenter and two round-trips to the furthest involved partition leader if partition leaders are not in the same datacenter. When heuristic 2 or heuristic 3 are used to placed partition leaders, the resulting placement typically places partition leaders in different datacenters. Heuristic 3 limits the placement of partition leaders among the set DC_{db} found by GPlacer. As heuristic 1 places all partition leaders in one datacenter, heuristic 1 is considered a special case of heuristic 3 where the set of datacenters that can host partition leaders DC_{db} is of size $|DC_{db}| = 1$. On the other hand, heuristic 2 allows the placement of partition leaders in any datacenter in DC. Therefore, heuristic 2 is considered a special case of heuristic 3 where the set of datacenters that can host partition leaders DC_{db} is of size $|DC_{db}| = |DC|$. We measure the effect of varying the size of DC_{db} on heuristic 3's placements. The results are compared to both heuristic 1's and heuristic 2's placements while varying the percentage of distributed transactions.

Figure 2.14 shows the expected commit latencies when the three heuristics are used

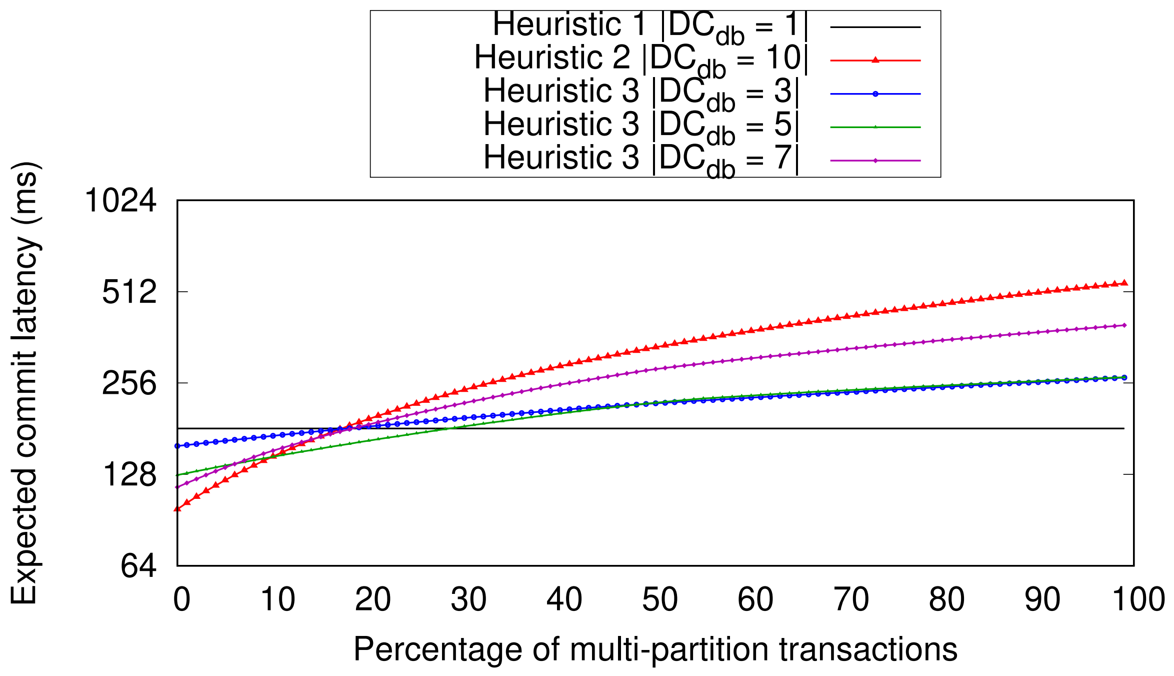
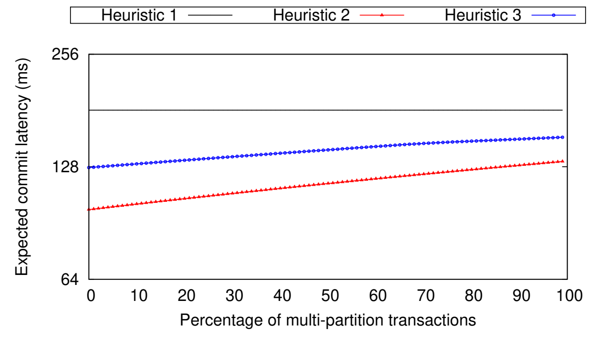
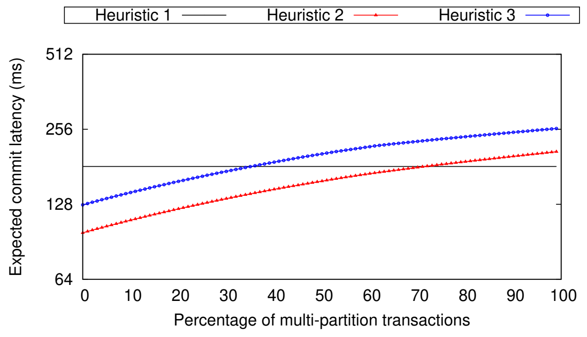


Figure 2.14: Comparison of the expected commit latency of the resulting leader placements of heuristic 1 $|DC_{db}| = 1$, heuristic 2 $|DC_{db}| = 10$, and heuristic 3 when $|DC_{db}| = 3$, $|DC_{db}| = 5$, and $|DC_{db}| = 7$ while varying the percentage of distributed transactions.

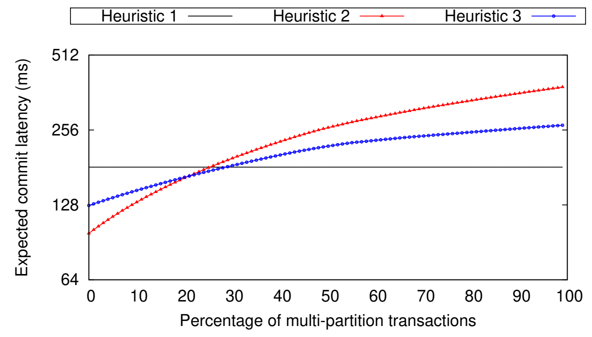
to place partition leaders. The expected commit latency is shown in a log scale in the y-axis and the percentage of the multi-partition transaction is shown in the x-axis. In this experiment, clients are distributed among 10 datacenters where the RTT among each pair varies from $0ms$ to $500ms$ and each transaction has to be replicated to a quorum of size 3 before it is committed. Figure 2.14 reports the expected commit latencies of both heuristic 1 and heuristic 2 in addition to heuristic 3 when the size of DC_{db} is set to 3, 5, and 7. This aims to show the effect of varying $|DC_{db}|$ on heuristic 3's placement while changing the percentage of distributed transaction. As heuristic 1 places all partition leaders in one datacenters, the average commit latency does not change with the percentage of multi-partition transactions. Placing all partition leaders in one datacenter encounters the same average commit latency for both single-partition and multi-partition transactions. Therefore, the average estimated commit latency of heuristic 1 is shown in Figure 2.14 as a horizontal solid black line. However, for heuristics 2 and 3, increasing the percentage of multi-partition transactions increases the average commit latency as the cost of the 2PC between all the involved partition leaders increases. Figure 2.14 suggests that heuristic 2 has an advantage over both heuristic 1 and 3 when the percentage of distributed transactions is low ($P_{mp-txn} \leq 9\%$). However, heuristic 3 with $DC_{db} = 5$ has an advantage over other heuristics when $9\% < P_{mp-txn} \leq 29\%$. Heuristic 1 gains advantage over other heuristics when the percentage of distributed transactions is high ($P_{mp-txn} > 29\%$). Typically, the percentage of multi-partition transactions is around 10% [201, 203].



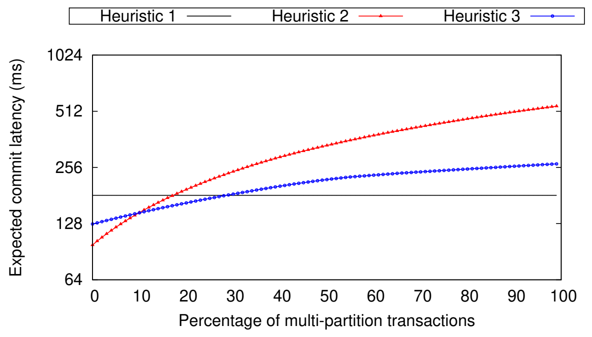
(a) $d = 1$



(b) $d = 3$



(c) $d = 7$



(d) $d = 9$

Figure 2.15: The effect of varying the distance d among partition leaders of distributed transactions while varying the percentage of multi-partition transactions.

Also, Figure 2.14 suggests that limiting the number of datacenters to host partition leaders hurts the latency of single-partition transactions. As shown, when the percentage of distributed transactions is low, heuristic 2 ($|DC_{db}| = 10$) outperforms both heuristic

1 ($|DC_{db}| = 1$) and heuristic 3 when $|DC_{db}| = 3$, $|DC_{db}| = 5$, and $|DC_{db}| = 7$. This happens because in both heuristics 1 and 3, partition leaders cannot be hosted in every available datacenter and hence many single-partition transactions have to travel to their partition leaders in other datacenters. On the other hand, limiting the number of datacenters to host partition leaders significantly benefits the average commit latency when the percentage of multi-partition transactions is high. Hosting partition leaders in one or few datacenters limits the cost of 2PC that needs to be paid for multi-partition transactions.

Another parameter that affects multi-partition transactions' 2PC latency is the distance among the involved partition leaders. For this, we introduce a distance parameter d that determines how far a distributed transaction can span. $d = 1$ means that a distributed transaction coordinated by partition leader i can only span partition leader j where j is the nearest partition leader to i. If d is set to $|DC|$, it means a distributed transaction coordinated by partition leader i can span any other partition leader $j \in DC$. d determines the locality of a distributed transaction. When d is low, it means that distributed transactions only span nearby partition leaders and hence the cost of 2PC among the involved partition leaders is low. We evaluate the effect of varying d only on leader placement heuristics 2 and 3 as heuristic 1 places all partition leaders together and therefore d has no effect on heuristic 1's expected commit latency. Also, as heuristic 3 places partition leaders in the set DC_{db}, the distance parameter has the limits $0 \leq d \leq |DC_{db}| - 1$. $d = 0$ results in a single-partition transaction while $d = |DC_{db}| - 1$ results in a distributed transaction that spans all partition leaders in DC_{db}. As shown in Figure 2.14, both heuristics 2 and 3 favor single-partition transactions and when $d = 1$, distributed transactions observe low 2PC overhead and hence lower average commit latency. On the other hand, the benefit of placing all partition leaders in one datacenter is reduced when d is low. Figure 2.15 shows the effect of varying the distance d among partition leaders of distributed transactions. In this experiment, clients are distributed among 10 datacenters where the RTT among each pair varies from $0ms$ to $500ms$ and each transaction has to be replicated to a quorum of size 3 before it is committed. In addition, heuristic 3 is configured with $|DC_{db}| = 5$. Figure 2.15a clearly demonstrates that heuristic 1 obtains higher commit latency than both heuristics 2 and 3 when $d = 1$. As d increases and distributed transactions are allowed to span more partition leaders, the cost of 2PC among partition leaders increases. Therefore, the resulting commit latency increases for both heuristic 2 and 3 as shown in Figures 2.15b for $d = 3$, 2.15c for $d = 7$, and 2.15d for $d = 9$. This result suggests to use heuristic 1 when the percentage of multi-partition transactions is high and distributed transactions are allowed to span all partition leaders. When distributed transactions are limited to local geographical regions and the percentage of multi-partition transactions is low, it is preferable to use either heuristic 2 or heuristic 3 for partition leaders placement.

It is important to mention that the reported results differ for different scenarios and estimates should be calculated *a priori* to decide which leader placement achieves the minimum commit latency for a given scenario. Our framework evaluates the outcomes of the three heuristics and chooses the placement that achieves the minimum latency.

2.8 GPlacer Conclusion

In this chapter, we address the data placement problem of geo-replicated databases with strong consistency guarantees. We present different placement optimizations to reduce transactions execution latency and commit latency. These placement optimizations are widely applied on different distributed transaction management protocols. Our evaluation shows that applying the read optimizations and the request handoff optimization could reduce transaction latency by 68% and increases throughput by 170%. To address the placement problem at scale, we propose different placement heuristics that can efficiently find sub-optimal placements within $5 - 10\%$ of the optimal placements. Experiments show that these heuristics are able to scale without significantly reducing the quality of the resulting placements from the optimal placement. Finally, we discuss three partition leader placement heuristics to place partition leaders. Experiments show that none of the three heuristics is superior when the percentage of multi-partition transactions varies. Unlike in [194] which uses one heuristic to place partition leaders regardless of the percentage of multi-partition transactions, our framework switches between different heuristics when the percentage of multi-partition transactions varies.

Chapter 3

CoT: Decentralized Elastic Caches for Cloud Environments

3.1 Introduction

Social networks, the web, and mobile applications have attracted hundreds of millions of users [29, 39]. These users share their relationships and exchange images and videos in timely personalized experiences [59]. To enable this real-time experience, the underlying storage systems have to provide efficient, scalable, and highly available access to big data. Social network users consume several orders of magnitude more data than they produce [51]. In addition, a single page load requires hundreds of object lookups that need to be served in a fraction of a second [59]. Therefore, traditional disk-based storage systems are not suitable to handle requests at this scale due to the high access latency of disks and I/O throughput bounds [222].

To overcome these limitations, distributed caching services have been widely deployed on top of persistent storage in order to efficiently serve user requests at scale [211]. Distributed caching systems such as Memcached [32] and Redis [35] are widely adopted by cloud service providers such as Amazon ElastiCache [20] and Azure Redis Cache [22]. These caching services offer significant latency and throughput improvements to systems that directly access the persistent storage layer. Redis and Memcached use consistent hashing [128] to distribute keys among several caching servers. Although consistent hashing ensures a fair distribution of the number of keys assigned to each caching shard, it does not consider the workload per key in the assignment process. Real-world workloads are typically skewed with few keys being significantly hotter than other keys [122]. This skew causes load-imbalance among caching servers.

Load imbalance in the caching layer can have significant impact on the overall application performance. In particular, it may cause drastic increases in the latency of operations at the tail end of the access frequency distribution [121]. In addition, the average throughput decreases and the average latency increases when the workload skew increases [71]. This increase in the average and tail latency is amplified for real workloads when operations are executed in chains of dependent data objects [150]. A single Facebook page-load results in retrieving hundreds of objects in multiple rounds of data fetching operations [167, 59]. Finally, solutions that equally overprovision the caching layer resources to handle the most loaded caching server suffer from **resource underutilization** in the least loaded caching servers.

Various approaches have been proposed to solve the load-imbalance problem using centralized load monitoring [43, 207], server side load monitoring [121], or front-end load monitoring [96]. Adya et al. [43] propose Slicer that separates the data serving plane from the control plane. The control plane is a centralized system component that collects metadata about shard accesses and server workload. It periodically runs an optimization algorithm that decides to redistribute, repartition, or replicate slices of the key space to achieve better back-end load-balance. Hong et al. [121] use a distributed server side load monitoring to solve the load-imbalance problem. Each back-end server independently tracks its hot keys and decides to distribute the workload of its hot keys among other back-end servers. Solutions in [43, 207] and [121] require the back-end to change the key-to-caching-server mapping and announce the new mapping to all the front-end servers. Fan et al. [96] use a distributed front-end load-monitoring approach. This approach shows that adding a small cache in the front-end servers has significant impact on solving the back-end load-imbalance. Caching the heavy hitters at front-end servers reduces the skew among the keys served from the caching servers and hence achieves better back-end load-balance. Fan et al. *theoretically* show through analysis and simulation that a small *perfect cache* at each front-end solves the back-end load-imbalance problem. However, perfect caching is practically hard to achieve. Determining the cache size and the replacement policy that achieve near perfect caching at the front-end for dynamically changing and evolving workloads is challenging.

We propose **Cache-on-Track** (CoT); a **decentralized**, **elastic**, and **predictive** heavy hitter caching at front-end servers. CoT proposes a new cache replacement policy specifically tailored for small front-end caches that serve **skewed workloads**. CoT uses a small front-end cache to solve back-end load-imbalance as introduced in [96]. However, CoT does not assume perfect caching at the front-end. CoT uses the space saving algorithm [158] to track the **top-k** *heavy hitters*. The tracking information allows CoT to *cache* the exact top C hot-most keys out of the approximate top-k tracked keys preventing cold and noisy keys from the long tail to replace hot keys in the cache. CoT is decentralized in the sense that each front-end independently determines its hot key set based on the key access distribution served at this specific front-end. This allows CoT to address back-end load-imbalance without introducing single points of failure or bottlenecks that typically come with centralized solutions. In addition, this allows CoT to scale to thousands of front-end servers, a common requirement of social network and modern web applications. CoT is elastic in that each front-end uses its local load information to monitor its contribution to the back-end load-imbalance. Each front-end elastically adjusts its tracker and cache sizes to reduce the load-imbalance caused by this front-end. In the presence of workload changes, CoT dynamically adjusts front-end tracker to cache ratio in addition to both the tracker and cache sizes to eliminate any back-end load-imbalance.

In traditional architectures, memory sizes are static and caching algorithms strive to achieve the best usage of all the available resources. However, in a cloud setting where there are theoretically infinite memory and processing resources and cloud instance migration is the norm, cloud end-users aim to achieve their SLOs while reducing the required cloud resources and thus decreasing their monetary deployment costs. CoT's main goal is to reduce the necessary front-end cache size at each front-end to eliminate server-side load-imbalance. Reducing front-end cache size is crucial for the following reasons: 1) it reduces the monetary cost of deploying front-end caches. For this, we quote David Lomet in his recent works [149, 148, 147] where he shows that cost/performance is usually more important than sheer performance: *"the argument here is not that there is insufficient main memory to hold the data, but that there is a less costly way to manage data."*. 2) In the presence of data updates and when data consistency is a requirement, increasing front-end cache sizes significantly increases the cost of the data consistency management technique. Note that social networks and modern web applications run on thousands of front-end servers. Increasing front-end cache size not only multiplies the cost of deploying bigger cache by the number of front-end servers, but also increases several costs in the consistency management pipeline including a) the cost of tracking key incarnations in different front-end servers and b) the network and processing costs to propagate updates to front-end servers. 3) Since the workload is skewed, our experiments clearly demonstrate that the relative benefit of adding more front-end cache-lines, measured by the average cache-hits per cache-line and back-end load-imbalance reduction, drastically decreases as front-end cache sizes increase.

CoT's resizing algorithm dynamically increases or decreases front-end allocated memory in response to dynamic workload changes. CoT's dynamic resizing algorithm is valuable in different cloud settings where 1) all front-end servers are deployed in the same datacenter and obtain the same dynamically evolving workload distribution, 2) all front-end servers are deployed in the same datacenter but obtain different dynamically evolving workload distributions, and finally 3) front-end servers are deployed at different *edge-datacenters* and obtain different dynamically evolving workload distributions. In particular, CoT aims to capture local trends from each individual front-end server perspective. In social network applications, front-end servers that serve different geographical regions might experience different key access distributions and different local trends (e.g., #miami vs. #ny). Similarly, in large scale data processing pipelines, several applications are deployed on top of a shared caching layer. Each application might be interested in different partitions of the data and hence experience different key access distributions and local trends. While CoT operates on a *fine-grain key* level at front-end servers, solutions like Slicer [43] operate on coarser grain slices or shards at the caching servers. Server side solutions are complementary to CoT. Although capturing local trends alleviates the load and reduces load-imbalance among caching servers, other factors can result in load-imbalance and hence using server-side load-balancing, e.g., Slicer, might still be beneficial.

We summarize the contributions of CoT as follows.

- Cache-on-Track (CoT) is a decentralized, elastic, and predictive front-end caching framework that reduces back-end load-imbalance and improves overall performance.
- CoT dynamically minimizes the required front-end cache size to achieve back-end load-balance. CoT's built-in elasticity is a key novel advantage over other replace-

ment policies.

- Extensive experimental studies that compare CoT's replacement policy to both traditional as well as state-of-the-art replacement policies, namely, LFU, LRU, ARC, and LRU-2. The experiments demonstrate that CoT achieves server size load-balance for different workload with **50% to 93.75%** less front-end cache in comparison to other replacement policies.
- The experimental study demonstrates that CoT successfully auto-configures its tracker and cache sizes to achieve back-end load-balance.
- In our experiments, we found a *bug* in YCSB's [78] ScrambledZipfian workload generator. This generator generates workloads that are significantly less-skewed than the promised Zipfian distribution.

The rest of the chapter is organized as follows. In Section 3.2, the system and data models are explained. In Section 3.3, we motivate CoT by presenting the main advantages and limitations of using LRU, LFU, ARC, and LRU-k caches at the front-end. We present the details of CoT in Section 3.4. In Section 3.5, we evaluate the performance and the overhead of CoT. The related work is discussed in Section 3.6 and the chapter is concluded in Section 3.7.

3.2 System and Data Models

This section introduces the system and data access models. Figure 3.1 presents the system architecture where user-data is stored in a distributed back-end storage layer in the cloud. The back-end storage layer consists of a distributed in-memory caching layer deployed on top of a distributed persistent storage layer. The caching layer aims

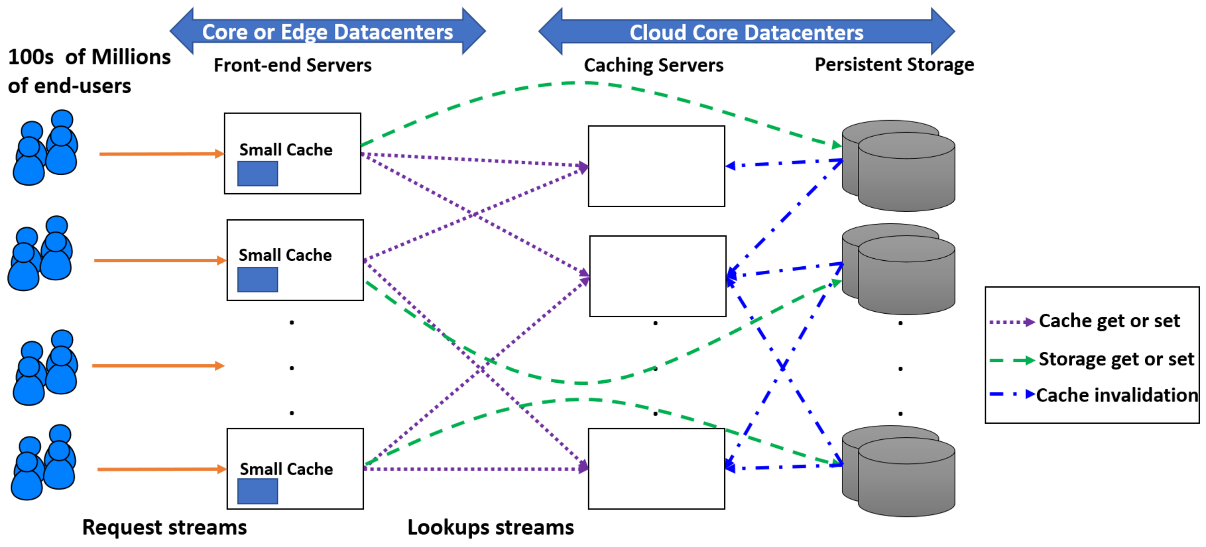


Figure 3.1: Overview of the system architecture.

to improve the request latency and system throughput and to alleviate the load on the persistent storage layer. As shown in Figure 3.1, hundreds of millions of end-users send streams of page-load and page-update requests to thousands of stateless front-end servers. These front-end servers are either deployed in the same core datacenter as the back-end storage layer or distributed among other core and edge datacenters near end-users. Each end-user request results in hundreds of data object lookups and updates served from the back-end storage layer. According to Facebook Tao [59], 99.8% of the accesses are reads and 0.2% of them are writes. Therefore, the storage system has to be **read optimized** to efficiently handle end-user requests at scale.

The front-end servers can be viewed as the clients of the back-end storage layer. We assume a typical key/value store interface between the front-end servers and the storage layer. The API consists of the following calls:

- $v = get(k)$ retrieves value v corresponding to key k.
- $set(k, v)$ assigns value v to key k.

- *delete(k)* deletes the entry corresponding key *k*.

Front-end servers use *consistent hashing* [128] to locate keys in the caching layer. Consistent hashing solves the *key discovery* problem and reduces key churn when a caching server is added to or removed from the caching layer. We extend this model by adding an additional layer in the cache hierarchy. As shown in Figure 3.1, each front-end server maintains a small cache of its hot keys. This cache is populated according to the accesses that are served by this front-end server.

We assume a client driven caching protocol similar to the protocol implemented by **Memcached** [32]. A cache client library is deployed in the front-end servers. *Get* requests are initially attempted to be served from the local cache. If the requested key is in the local cache, the *value* is returned and the request is marked as served. Otherwise, a *null* value is returned and the front-end has to request this key from the caching layer **at the back-end storage layer**. If the key is cached in the caching layer, its value is returned to the front-end. Otherwise, a *null* value is returned and the front-end has to request this key from the persistent storage layer and upon receiving the corresponding value, the front-end inserts the value in its front-end local cache and in the server-side caching layer as well. As in [167], a *set*, or an update, request invalidates the key in both the local cache and the caching layer. Updates are directly sent to the persistent storage, local values are set to null, and delete requests are sent to the caching layer to invalidate the updated keys. The Memcached client driven approach allows the deployment of a *stateless* caching layer. As requests are driven by the client, a caching server does not need to maintain the state of any request. This simplifies scaling and tolerating failures at the caching layer. Although, we adopt the Memcached client driven request handling protocol, our model works as well with write-through request handling protocols.

Our model is not tied to any replica consistency model. Each key can have multiple incarnations in the storage layer and the caching layer. Updates can be synchronously propagated if *strong consistency* guarantees are needed or asynchronously propagated if *weak consistency* guarantees suffice. Achieving strong consistency guarantees among replicas of the same object has been widely studied in [71, 121]. Ghandeharizadeh et al. [102, 103] propose several complementary techniques to CoT to deal with consistency in the presence of updates and configuration changes. These techniques can easily be adopted in our model according to the application requirements. We understand that deploying an additional vertical layer of cache increases potential data inconsistencies and hence increases update propagation and synchronization overheads. Therefore, our goal in building CoT is to reduce the front-end cache size in order to limit the inconsistencies and the synchronization overheads that result from deploying front-end caches, while maximizing their benefits.

3.3 Front-end Cache Alternatives

Fan et al. [96] show that a small cache in the front-end servers has big impact on the caching layer load-balance. Their analysis assumes perfect caching in front-end servers for the hottest keys. A **perfect cache** of C cache-lines is defined such that accesses for the C hot-most keys always hit the cache while other accesses always miss the cache. However, the perfect caching assumption is impractical especially for dynamically changing and evolving workloads. Different replacement policies have been developed to approximate perfect caching for different workloads. In this section, we discuss the workload assumptions and various client caching objectives. This is followed by a discussion of the advantages and limitations of common caching replacement policies such as Least Recently Used (LRU), Least Frequently Used (LFU), Adaptive Replacement Cache (ARC [157]) and LRU-k [171].

Workload assumptions: Real-world workloads are typically skewed with few keys being significantly hotter than other keys [122]. Zipfian distribution is a common example of a key hotness distribution. However, key hotness can follow different distributions such as *Gaussian* or different variations of Zipfian [58, 114]. In this work, we assume skewed workloads with periods of stability (where hot keys remain hot during these periods).

Client caching objectives: Front-end servers construct their perspective of the key hotness distribution based on the requests they serve. Front-end servers aim to achieve the following caching objectives:

- The cache replacement policy should prevent cold keys from replacing hotter keys in the cache.
- Front-end caches should adapt to the changes in the workload. In particular, front-end servers should have a way to retire hot keys that are no longer accessed. In addition, front-end caches should have a mechanism to expand or shrink their local caches in response to changes in workload distribution. For example, front-end servers that serve uniform access distributions should dynamically shrink their cache size to *zero* since caching is of no value in this situation. On the other hand, front-end servers that serve highly skewed Zipfian (e.g., s = 1.5) should dynamically expand their cache size to capture all the hot keys that cause load-imbalance among the back-end caching servers.

A popular policy for implementing client caching is the LRU replacement policy. Least Recently Used (LRU) costs O(1) per access and caches keys based on their recency of access. This may allow cold keys that are recently accessed to replace hotter cached keys. Also, LRU cannot distinguish well between frequently and infrequently accessed keys [141]. For example, this access sequence (A, B, C, D, A, B, C, E, A, B, C, F, ...) would always have a cache miss for an LRU cache of size 3. Alternatively, Least

Frequently Used (LFU) can be used as a replacement policy. LFU costs $O(log(C))$ per access where C is the cache size. LFU is typically implemented using a min-heap and allows cold keys to replace hotter keys at the root of the heap. Also, LFU cannot distinguish between old references and recent ones. For example, this access sequence (A, A, B, B, C, D, E, C, D, E, C, D, E ....) would always have a cache miss for an LFU cache of size 3 except for the 2^{nd} and 4^{th} accesses. This means that LFU cannot adapt to changes in workload. Both LRU and LFU are limited in their knowledge to the content of the cache and cannot develop a wider perspective about the hotness distribution outside of their static cache size. Our experiments in Section 3.5 show that replacement policies that track more keys beyond their cache sizes (e.g., ARC, LRU-k, and CoT) beat the hit-rates of replacement policies that have no access information of keys beyond their cache size especially for periodically stable skewed workloads.

Adaptive Replacement Cache (ARC) [157] tries to realize the benefits of both LRU and LFU policies by maintaining two caching lists: one for *recency* and one for *frequency*. ARC dynamically changes the number of cache-lines allocated for each list to either favor recency or frequency of access in response to workload changes. In addition, ARC uses shadow queues to track more keys beyond the cache size. This helps ARC to maintain a broader perspective of the access distribution beyond the cache size. ARC is designed to find the fine balance between recent and frequent accesses. As a result, ARC pays the cost of caching every new cold key in the recency list evicting a hot key from the frequency list. This cost is significant especially when the cache size is much smaller than the key space and the workload is skewed favoring frequency over recency.

LRU-k tracks the last k accesses for each key in the cache, in addition to a pre-configured fixed size history that include the access information of the recently evicted keys from the cache. New keys replace the key with the least recently k^{th} access in the cache. The evicted key is moved to the history, which is typically implemented using a

LRU like queue. LRU-k is a suitable strategy to mock perfect caching of periodically stable skewed workloads when its cache and history sizes are perfectly pre-configured for this specific workload. However, due to the lack of LRU-k's dynamic resizing and elasticity of both its cache and history sizes, we choose to introduce CoT that is designed with native resizing and elasticity functionality. This functionality allows CoT to adapt its cache and tracker sizes in response to workload changes.

3.4 Cache on Track (CoT)

Front-end caches serve two main purposes: *1) decrease the load on the back-end caching layer* and *2) reduce the load-imbalance among the back-end caching servers.* CoT focuses on the latter goal and considers back-end load reduction a complementary side effect. CoT's design philosophy is to track more keys beyond the cache size. This tracking serves as a filter that prevents cold keys from populating the small cache and therefore, only hot keys can populate the cache. In addition, the tracker and the cache are dynamically and adaptively resized to ensure that the load served by the back-end layer follows a load-balance target.

The idea of tracking more keys beyond the cache size has been widely used in replacement policies such as 2Q [127], MQ [226], LRU-k [171, 172], ARC [157], and in other works like Cliffhanger [75] to solve other cache problems. Both 2Q and MQ use multiple LRU queues to overcome the weaknesses of LRU of allowing cold keys to replace warmer keys in the cache. Cliffhanger uses shadow queues to solve a different problem of memory allocation among cache blobs. All these policies are desgined for fixed memory size environments. However, in a cloud environment where elastic resources can be requested on-demand, a new cache replacement policy is needed to take advantage of this elasticity.

CoT presents a new cache replacement policy that uses a *shadow heap* to track more keys beyond the cache size. Previous works have established the efficiency of heaps in tracking frequent items [158]. In this section, we explain how CoT uses tracking beyond the cache size to achieve the caching objectives listed in Section 3.3. In particular, CoT answers the following questions: 1) *how to prevent cold keys from replacing hotter keys in the cache?*, 2) *how to reduce the required front-end cache size that achieves lookup load-balance?*, 3) *how to adaptively resize the cache in response to changes in the workload distribution?* and finally 4) *how to dynamically retire old heavy hitters?*.

First, we develop the notation in Section 3.4.1. Then, we explain the space saving tracking algorithm [158] in Section 3.4.2. CoT uses the space saving algorithm to track the approximate *top-k* keys in the lookup stream. In Section 3.4.3, we extend the space saving algorithm to capture the exact top C keys out of the approximately tracked *top-k* keys. CoT's cache replacement policy dynamically captures and caches the exact top C keys thus preventing cold keys from replacing hotter keys in the cache. CoT's adaptive cache resizing algorithm is presented in Section 3.4.4. CoT's resizing algorithm exploits the elasticity and the migration flexibility of the cloud and minimizes the required front-end memory size to achieve back-end load-balance. Section 3.4.4 explains how CoT expands and shrinks front-end tracker and cache sizes in response to changes in workload.

3.4.1 Notation

The key space, denoted by S, is assumed to be large in the scale of trillions of keys. Each front-end server maintains a cache of size $C <<< S$. The set of cached keys is denoted by S_c. To capture the hot-most C keys, each front-end server tracks $K > C$ keys. The set of tracked key is denoted by S_k. Front-end servers cache the hot-most C keys where $S_c \subset S_k$. A key hotness h_k is determined using the dual cost model introduced in [85]. In this model, read accesses increase a key hotness by a read weight

S	key space
K	number of tracked keys at the front-end
C	number of cached keys at the front-end
h_k	hotness of a key k
$k.r_c$	read count of a key k
$k.u_c$	update count of a key k
r_w	the weight of a read operation
u_w	the weight of an update operation
h_{min}	the minimum key hotness in the cache
S_k	the set of all tracked keys
S_c	the set of tracked and cached keys
S_{k-c}	the set of tracked but not cached keys
I_c	the current local lookup load-imbalance
I_t	the target lookup load-imbalance
α	the average hit-rate per cache-line

Table 3.1: Summary of notation.

r_w while update accesses decrease it by an update weight u_w. As update accesses cause cache invalidations, frequently updated keys should not be cached and thus an update access decreases key hotness. For each tracked key, the read count $k.r_c$ and the update count $k.u_c$ are maintained to capture the number of read and update accesses of this key. Equation 3.1 shows how the hotness of key k is calculated.

$$h_k = k.r_c \times r_w - k.u_c \times u_w \tag{3.1}$$

h_{min} refers to the minimum key hotness in the cache. h_{min} splits the tracked keys into *two* subsets: 1) the set of tracked and cached keys S_c of size C and 2) the set of tracked but not cached keys S_{k-c} of size $K-C$. The current local load-imbalance among caching servers lookup load is denoted by I_c. I_c is a local variable at each front-end that determines the current contribution of this front-end to the back-end load-imbalance. I_c is defined as the workload ratio between the most loaded back-end server and the least loaded back-end server as observed at a front-end server. For example, if a front-end server sends, during an epoch, a maximum of 5K key lookups to some back-end server and, during the same epoch, a minimum of 1K key lookups to another back-end server then I_c, at this front-end, equals 5. I_t is the target load-imbalance among the caching servers. I_t is the only input parameter set by the system administrator and is used by front-end servers to dynamically adjust their cache and tracker sizes. Ideally I_t should be set close to 1. $I_t = 1.1$ means that back-end load-balance is achieved if the most loaded server observe at most 10% more key lookups that the least loaded server. Finally, we define another **local auto-adjusted** variable α. α is the average hits per cache-line and it determines the quality of the cached keys. α helps detect changes in workload and adjust the cache size accordingly. Note that CoT automatically infers the value of α based on the observed workload. Hence, the system administrator does not need to set the value of α. Table 3.1 summarizes the notation.

3.4.2 Space-Saving Hotness Tracking Algorithm

We use the *space-saving* algorithm introduced in [158] to track the key hotness at front-end servers. Space-saving uses a min-heap to order keys based on their hotness and a hashmap to lookup keys in the tracker in *O(1)*. The space-saving algorithm is shown in Algorithm 6. If the accessed key k is not in the tracker (Line 1), it replaces the key with minimum hotness at the root of the min-heap (Lines 2, 3, and 4). The algorithm gives the newly added key the benefit of doubt and assigns it the hotness of the replaced key. As a result, the newly added key gets the opportunity to survive immediate replacement in the tracker. Whether the accessed key k was in the tracker or is newly added to the tracker, the hotness of the key is updated based on the access type according to Equation 3.1 (Line 6) and the heap is accordingly adjusted (Line 7).

Algorithm 6 The space-saving algorithm: track_key(key k, access_type t).

State: S_k: keys in the tracker.
Input: (key k, access_type t)

```
1: if $k \notin S_k$ then
2:     let $k^{'}$ be the root of the min-heap
3:     replace $k^{'}$ with $k$
4:     $h_k := h_{k'}$
5: end if
6: $h_k$ := update_hotness(k, t)
7: adjust_heap(k)
8: return $h_k$
```

3.4.3 CoT: Cache Replacement Policy

CoT's tracker captures the approximate top K hot keys. Each front-end server should cache the exact top C keys out of the tracked K keys where $C < K$. The exactness of the top C cached keys is considered with respect to the approximation of the top K tracked keys. Caching the exact top C keys prevents cold and noisy keys from replacing hotter keys in the cache and achieves the first caching objective. To determine the exact top C keys, CoT maintains a cache of size C in a min-heap structure. Cached keys are partially ordered in the min-heap based on their hotness. The root of the cache min-heap gives the minimum hotness, h_{min}, among the cached keys. h_{min} splits the tracked keys into *two unordered* subsets S_c and S_{k-c} such that:

- $|S_c| = C$ and $\forall_{x \in S_c} h_x \geq h_{min}$
- $|S_{k-c}| = K - C$ and $\forall_{x \in S_{k-c}} h_x < h_{min}$

Figure 3.2: CoT: a key is inserted to the cache if its hotness exceeds the minimum hotness of the cached keys.

For every key access, the hotness information of the accessed key is updated in the tracker. If the accessed key is cached, its hotness information is updated in the cache as well. However, if the accessed key is not cached, its hotness is compared against h_{min}. As shown in Figure 3.2, the accessed key is inserted into the cache only if its hotness exceeds h_{min}. Algorithm 7 explains the details of CoT's cache replacement algorithm.

Algorithm 7 CoT's caching algorithm

State: S_k: keys in the tracker and S_c: keys in the cache.
Input: (key k, access_type t)

```
1: $h_k$ = track_key(k, t) as in Algorithm 6
2: if $k \in S_c$ then
3:     let v = access($S_c$, k) // local cache access
4: else
5:     let v = server_access(k) // caching server access
6:     if $h_k > h_{min}$ then
7:         insert($S_c$, k, v) // local cache insert
8:     end if
9: end if
10: return v
```

For every key access, the *track_key* function of Algorithm 6 is called (Line 1) to update the tracking information and the hotness of the accessed key. Then, a key access is served from the local cache only if the key is in the cache (Lines 3). Otherwise, the access is served from the caching server (Line 5). Serving an access from the local cache implicitly updates the accessed key hotness and location in the cache min-heap. If the accessed key is not cached, its hotness is compared against h_{min} (Line 6). The accessed key is inserted to the local cache if its hotness exceeds h_{min} (Line 7). This happens only if there is a tracked but not cached key that is hotter than one of the cached keys. Keys are inserted to the cache together with their tracked hotness information. Inserting keys into the cache follows the LFU replacement policy. This implies that a local cache insert (Line 7) would result in the replacement of the coldest key in the cache (the root of the cache heap) if the local cache is full.

3.4.4 CoT: Adaptive Cache Resizing

This section answers the following questions: *how to reduce the necessary front-end cache size that achieves front-end lookup load-balance? How to shrink the cache size when the workload's skew decreases?* and *How to detect changes in the set of hot keys?* As explained in Section 3.1, Reducing the front-end cache size decreases the front-end cache monetary cost, limits the overheads of data consistency management techniques, and maximizes the benefit of front-end caches measured by the average cache-hits per cache-line and back-end load-imbalance reduction.

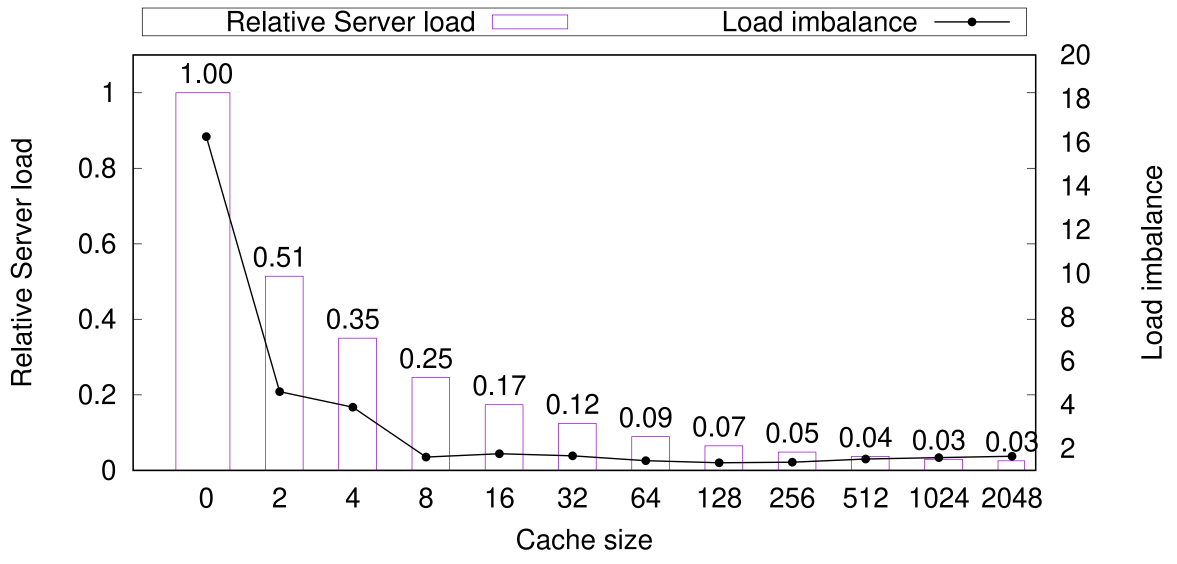


Figure 3.3: Reduction in relative server load and load-imbalance among caching servers as front-end cache size increases.

The Need for Cache Resizing:

Figure 3.3 experimentally shows the effect of increasing the front-end cache size on both back-end *load-imbalance reduction* and decreasing the workload at the back-end. In this experiment, 8 memcached shards are deployed to serve back-end lookups and 20 clients send lookup requests following a significantly skewed Zipfian distribution (s = 1.5).

The size of the key space is 1 million and the total number of lookups is 10 millions. The front-end cache size at each client is varied from 0 cachelines (no cache) to 2048 cachelines (≈0.2% of the key space). Front-end caches use CoT's replacement policy and a ratio of 4:1 is maintained between CoT's tracker size and CoT's cache size. We define back-end load-imbalance as the workload ratio between the most loaded server and the least loaded server. The target load-imbalance I_t is set to 1.5. As shown in Figure 3.3, processing all the lookups from the back-end caching servers (front-end cache size = 0) leads to a significant load-imbalance of 16.26 among the caching servers. This means that the most loaded caching server receives 16.26 times the number of lookup requests received by the least loaded caching server. As the front-end cache size increases, the server size load-imbalance drastically decreases. As shown, a front-end cache of size 64 cache lines at each client reduces the load-imbalance to 1.44 (an order of magnitude less load-imbalance across the caching servers) achieving the target load-imbalance $I_t = 1.5$. Increasing the front-end cache size beyond 64 cache lines only reduces the back-end aggregated load but not the back-end load-imbalance. The *relative server load* is calculated by comparing the server load for a given front-end cache size to the server load when there is no front-end caching (cache size = 0). Figure 3.3 demonstrates the reduction in the relative server load as the front-end cache size increases. However, the benefit of doubling the cache size proportionally decays with the key hotness distribution. As shown in Figure 3.3, the first 64 cachelines reduce the relative server load by 91% while the second 64 cachelines reduce the relative server load by only 2% more.

The failure of the ”one size fits all” design strategy suggests that statically allocating fixed cache and tracker sizes to all front-end servers is not ideal. Each front-end server should independently and adaptively be configured according to the key access distribution it serves. Also, changes in workloads can alter the key access distribution, the skew level, or the set of hot keys. For example, social networks and web front-end servers that serve different geographical regions might experience different key access distributions and different local trends (e.g., #miami vs. #ny). Similarly, in large scale data processing pipelines, several applications are deployed on top of a shared caching layer. Front-end servers of different applications serve accesses that might be interested in different partitions of the data and hence experience different key access distributions and local trends. Therefore, CoT's cache resizing algorithm learns the key access distribution independently at each front-end and dynamically resizes the cache and the tracker to achieve lookup load-imbalance target I_t. CoT is designed to reduce the front-end cache size that achieves I_t. Any increase in the front-end cache size beyond CoT's recommendation mainly decreases back-end load and should consider other conflicting parameters such as the additional cost of the memory cost, the cost of updates and maintaining the additional cached keys, and the percentage of back-end load reduction that results from allocating additional front-end caches.

CoT: Cache Resizing Algorithm:

Front-end servers use CoT to minimize the cache size that achieves a target load-imbalance I_t. Initially, front-end servers are configured with no front-end caches. The system administrator configures CoT by an input *target load-imbalance* parameter I_t that determines the maximum tolerable imbalance between the most loaded and least loaded back-end caching servers. Afterwards, CoT expands both tracker and cache sizes until the current load-imbalance achieves the inequality $I_c \leq I_t$.

Algorithm 8 describes CoT's cache resizing algorithm. CoT divides the timeline into epochs and each epoch consists of E accesses. Algorithm 8 is executed at the end of each epoch. The epoch size E is proportional to the tracker size K and is dynamically updated to guarantee that $E \geq K$ (Line 4). This inequality is required to guarantee that CoT does not trigger consecutive resizes before the cache and the tracker are filled with keys. During each epoch, CoT tracks the number of lookups sent to every back-end caching server. In addition, CoT tracks the total number of cache hits and tracker hits during this epoch. At the end of each epoch, CoT calculates the current load-imbalance I_c as the ratio between the highest and the lowest load on back-end servers during this epoch. Also, CoT calculates the current average hit per cached key α_c. α_c equals the total cache hits in the current epoch divided by the cache size. Similarly, CoT calculates the current average hit per tracked but not cache key α_{k-c}. CoT compares I_c to I_t and decides on a resizing action as follows.

1. $I_c > I_t$ (Line 1), this means that the target load-imbalance is not achieved. CoT follows the binary search algorithm in searching for the front-end cache size that achieves I_t. Therefore, CoT decides to double the front-end cache size (Line 2). As a result, CoT doubles the tracker size as well to maintain a tracker to cache size ratio of at least 2, $K \geq 2 \cdot C$ (Line 3). In addition, CoT uses a local variable α_t to capture the quality of the cached keys when I_t is first achieved. Initially, $\alpha_t = 0$. CoT then sets α_t to the average hits per cache-line α_c during the current epoch (Line 5). In subsequent epochs, α_t is used to detect changes in workload.

2. $I_c \leq I_t$ (Line 6), this means that the target load-imbalance has been achieved. However, changes in workload could alter the quality of the cached keys. Therefore, CoT uses α_t to detect and handle changes in workload in future epochs as explained below.

α_t is reset whenever the inequality $I_c \leq I_t$ is violated and Algorithm 8 expands cache and tracker sizes. Ideally, when the inequality $I_c \leq I_t$ holds, keys in the cache (the set S_c) achieve α_t hits per cache-line during every epoch while keys in the tracker but not in the cache (the set S_{k-c}) do not achieve α_t. This happens because keys in the set S_{k-c} are less hot than keys in the set S_c. α_t represents a target hit-rate per cache-line for future

Algorithm 8 CoT's elastic resizing algorithm.

State: S_c: keys in the cache, S_k: keys in the tracker, C: cache capacity, K: tracker capacity, α_c: average hits per key in S_c in the current epoch, α_{k-c}: average hits per key in S_{k-c} in the current epoch, I_c: current load-imbalance, and α_t: target average hit per key
Input: I_t

```
1: if $I_c > I_t$ then
2:     resize($S_c$, $2 \times C$)
3:     resize($S_k$, $2 \times K$)
4:     E := max (E, $K$)
5:     Let $\alpha_t = \alpha_c$
6: else
7:     if $\alpha_c < (1-\epsilon).\alpha_t$ and $\alpha_{k-c} < (1-\epsilon).\alpha_t$ then
8:         resize($S_c$, $\frac{C}{2}$)
9:         resize($S_k$, $\frac{K}{2}$)
10:    else if $\alpha_c < (1-\epsilon).\alpha_t$ and $\alpha_{k-c} > (1-\epsilon).\alpha_t$ then
11:        half_life_time_decay()
12:    else
13:        do_nothing()
14:    end if
15: end if
```

epochs. Therefore, if keys in the cache do not meet the target α_t in a following epoch, this indicates that the quality of the cached keys has changed and an action needs to be taken as follows.

1. Case 1: keys in S_c, on the average, do not achieve α_t hits per cacheline and keys in S_{k-c} do not achieve α_t hits as well (Line 7). This indicates that the quality of the cached keys decreased. In response. CoT shrinks both the cache and the tracker sizes (Lines 8 and 9). If shrinking both cache and tracker sizes results in a violation of the inequality $I_c < I_t$, Algorithm 8 doubles both tracker and cache sizes in the following epoch and α_t is reset as a result. In Line 7, we compare the average hits per key in both S_c and S_{k-c} to $(1-\epsilon) \cdot \alpha_t$ instead of α_t. Note that ϵ is a small constant <<< 1 that is used to avoid unnecessary resizing actions due to

insignificant statistical variations.

2. Case 2: keys in S_c do not achieve α_t while keys in S_{k-c} achieve α_t (Line 10). This signals that the set of hot keys is changing and keys in S_{k-c} are becoming hotter than keys in S_c. For this, CoT triggers a half-life time decaying algorithm that halves the hotness of all cached and tracked keys (Line 11). This decaying algorithm aims to forget old trends that are no longer hot to be cached (e.g., Gangnam style song). Different decaying algorithms have been developed in the literature [80, 81, 76]. Therefore, this chapter only focuses on the resizing algorithm details without implementing a decaying algorithm.

3. Case 3: keys in S_c achieve α_t while keys in S_{k-c} do not achieve α_t. This means that the quality of the cached keys has not changed and therefore, CoT does not take any action. Similarly, if keys in both sets S_c and S_{k-c} achieve α_t, CoT does not take any action as long as the inequality $I_c < I_t$ holds (Line 13).

3.5 CoT Experimental Evaluation

In this section, we evaluate CoT's caching algorithm and CoT's adaptive resizing algorithm. We choose to compare CoT to traditional and widely used replacement policies like LRU and LFU. In addition, we compare CoT to both ARC [157] and LRU-k [171]. As stated in [157], ARC, in its online auto-configuration setting, achieves comparable performance to LRU-2 (which is the most responsive LRU-k) [171, 172], 2Q [127], LRFU [141], and LIRS [125] even when these policies are perfectly tuned offline. Also, ARC outperforms the online adaptive replacement policy MQ [226]. Therefore, we compare with ARC and LRU-2 as representatives of these different polices. The experimental setup is explained in Section 3.5.1. First, we compare the hit rates of CoT's cache algorithm to LRU, LFU, ARC, and LRU-2 hit rates for different front-end cache sizes in Section 3.5.2. Then, we compare the required front-end cache size for each replacement policy to achieve a target back-end load-imbalance I_t in Section 3.5.3. In Section 3.5.4, we provide an end-to-end evaluation of front-end caches comparing the end-to-end performance of CoT, LRU, LFU, ARC, and LRU-2 on different workloads with the configuration where no front-end cache is deployed. Finally, CoT's resizing algorithm is evaluated in Section 3.5.5.

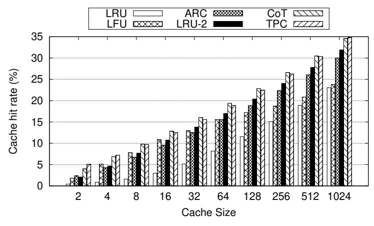
3.5.1 Experiment Setup

We deploy 8 instances of memcached [32] on a small cluster of 4 caching servers (2 memcached instance per server). Each caching server has an Intel(R) Xeon(R) CPU E31235 with 4GB RAM dedicated to each memcached instance.

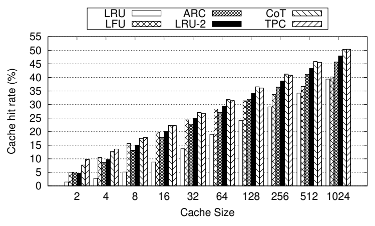
Dedicated client machines are used to generate client workloads. Each client machine executes multiple client threads to submit workloads to caching servers. Client threads use Spymemcached 2.11.4 [37], a Java-based memcached client, to communicate with memcached cluster. Spymemcached provides communication abstractions that distribute workload among caching servers using *consistent hashing* [128]. We slightly modified Spymemcached to monitor the workload per back-end server at each front-end. Client threads use Yahoo! Cloud Serving Benchmark (YCSB) [78] to generate workloads for the experiments. YCSB is a standard key/value store benchmarking framework. YCSB is used to generate key/value store requests such as *Get*, *Set*, and *Insert*. YCSB enables configuring the ratio between read (Get) and write (Set) accesses. Also, YCSB allows the generation of accesses that follow different access distributions. As YCSB is CPU-intensive, client machines run at most 20 client threads per machine to avoid contention among client threads. During our experiments, we realized that YCSB's ScrambledZipfian workload generator has a bug as it generates Zipfian workload distributions with significantly less skew than the skew level it is configured with. Therefore, we use YCSB's ZipfianGenerator instead of YCSB's ScrambledZipfian.

Our experiments use different variations of YCSB core workloads. Workloads consist of 1 million key/value pairs. Each key consists of a common prefix *"usertable:"* and a unique ID. We use a value size of 750 KB making a dataset of size 715GB. Experiments use read intensive workloads that follow Tao's [59] read-to-write ratio of 99.8% reads and 0.2% updates. Unless otherwise specified, experiments consist of 10 million key accesses sampled from different access distributions such as Zipfian (s = 0.90, 0.99, or 1.2) and uniform. Client threads submit access requests back-to-back. Each client thread can have only one outgoing request. Clients submit a new request as soon as they receive an acknowledgement for their outgoing request.

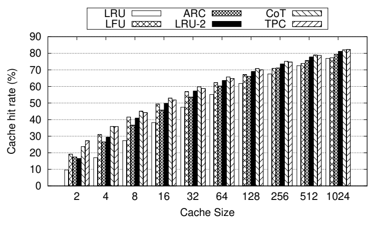
3.5.2 Hit Rate



(a) Zipfian 0.90



(b) Zipfian 0.99



(c) Zipfian 1.20

Figure 3.4: Comparison of LRU, LFU, ARC, LRU-2, CoT and TPC's hit rates using Zipfian access distribution with different skew parameter values (s= 0.90, 0.99, 1.20)

The first experiment compares CoT's hit rate to LRU, LFU, ARC, and LRU-2 hit rates using equal cache sizes for all replacement policies. 20 client threads are provisioned on one client machine and each cache client maintains its own cache. The cache size is varied from a very small cache of 2 cache-lines to 1024 cache-lines. The hit rate is compared using different Zipfian access distributions with skew parameter values s = 0.90, 0.99, and 1.2 as shown in Figures 3.4a, 3.4b, and 3.4c respectively. CoT's tracker to cache size ratio determines how many tracking nodes are used for every cache-line. CoT automatically detects the ideal tracker to cache ratio for any workload by fixing the cache size and doubling the tracker size until the observed hit-rate gains from increasing the tracker size are insignificant i.e., the observed hit-rate saturates. The tracker to cache size ratio decreases as the workload skew increases. A workload with high skew simplifies the task of distinguishing hot keys from cold keys and hence, CoT requires a smaller tracker size to successfully filter hot keys from cold keys. Note that LRU-2 is also configured with the same history to cache size as CoT's tracker to cache size. In this experiment, for each skew level, CoT's tracker to cache size ratio is varied as follows: 16:1 for Zipfian 0.9, 8:1 for Zipfian 0.99, and 4:1 for Zipfian 1.2. Note that CoT's tracker maintains only the meta-data of tracked keys. Each tracker node consists of a read counter and a write counter with 8 bytes of memory overhead per tracking node. In real-world workloads, value sizes vary from few hundreds KBs to few MBs. For example, Google's Bigtable [69] uses a value size of 64 MB. Therefore, a memory overhead of at most $\frac{1}{8}$ KB (16 tracker nodes * 8 bytes) per cache-line is negligible.

In Figures 3.4, the *x-axis* represents the cache size expressed as the number of cache-lines. The *y-axis* represents the front-end cache hit rate (%) as a percentage of the total workload size. At each cache size, the cache hit rates are reported for LRU, LFU, ARC, LRU-2, and CoT cache replacement policies. In addition, TPC represents the theoretically calculated hit-rate from the Zipfian distribution CDF if a perfect cache with the same cache size is deployed. For example, a perfect cache of size 2 cache-lines stores the hot most 2 keys and hence any access to these 2 keys results in a cache hit while accesses to other keys result in cache misses.

As shown in Figure 3.4a, CoT surpasses LRU, LFU, ARC, and LRU-2 hit rates at all cache sizes. In fact, CoT achieves almost similar hit-rate to the TPC hit-rate. In Figure 3.4a, CoT outperforms TPC for some cache size which is counter intuitive. This happens as TPC is theoretically calculated using the Zipfian CDF while CoT's hit-rate is calculate out of YCSB's sampled distributions which are approximate distributions. In addition, CoT achieves higher hit-rates than both LRU and LFU with **75% less cache-lines**. As shown, CoT with 512 cache-lines achieves 10% more hits than both LRU and LFU with 2048 cache-lines. Also, CoT achieves higher hit rate than ARC using **50% less cache-lines**. In fact, CoT configured with 512 cache-lines achieves 2% more hits than ARC with 1024 cache-lines. Taking tracking memory overhead into account, CoT maintains a tracker to cache size ratio of 16:1 for this workload (Zipfian 0.9). This means that CoT adds an overhead of 128 bytes (16 tracking nodes * 8 bytes each) per cache-line. The percentage of CoT's tracking memory overhead decreases as the cache-line size increases. For example, CoT introduces a tracking overhead of 0.02% when the cache-line size is 750KB. Finally, CoT consistently achieves 8-10% higher hit-rate than LRU-2 configured with the same history and cache sizes as CoT's tracker and cache sizes.

Similarly, as illustrated in Figures 3.4b and 3.4c, CoT outpaces LRU, LFU, ARC, and LRU-2 hit rates at all different cache sizes. Figure 3.4b shows that a configuration of CoT using 512 cache-lines achieves 3% more hits than both configurations of LRU and LFU with 2048 cache-lines. Also, CoT consistently outperforms ARC's hit rate with 50% less cache-lines. Finally, CoT achieves 3-7% higher hit-rate than LRU-2 configured with the same history and cache sizes. Figures 3.4b and 3.4c highlight that increasing workload skew decreases the advantage of CoT. As workload skew increases, the ability of LRU, LFU, ARC, LRU-2 to distinguish between hot and cold keys increases and hence CoT's preeminence decreases.

3.5.3 Back-End Load-Imbalance

In this section, we compare the required front-end cache sizes for different replacement policies to achieve a back-end load-imbalance target I_t. Different skewed workloads are used, namely, Zipfian s = 0.9, s = 0.99, and s = 1.2. For each distribution, we first measure the back-end load-imbalance when no front-end cache is used. A back-end load-imbalance target I_t is set to $I_t = 1.1$. This means that the back-end is load balanced if the most loaded back-end server processes at most 10% more lookups than the least loaded back-end server. We evaluate the back-end load-imbalance while increasing the front-end cache size using different cache replacement policies, namely, LRU, LFU, ARC, LRU-2, and CoT. In this experiment, CoT uses the same tracker-to-cache size ratio as in Section 3.5.2. For each replacement policy, we report the minimum required number of cache-lines to achieve I_t.

Dist.	Load-imbalance	Number of cache-lines to achieve $I_t = 1.1$				
	No cache	LRU	LFU	ARC	LRU-2	CoT
Zipf 0.9	1.35	64	16	16	8	8
Zipf 0.99	1.73	128	16	16	16	8
Zipf 1.20	4.18	2048	2048	1024	1024	512

Table 3.2: The minimum required number of cache-lines for different replacement policies to achieve a back-end load-imbalance target $I_t = 1.1$ for different workload distributions.

Table 3.2 summarizes the reported results for different distributions using LRU, LFU, ARC, LRU-2, and CoT replacement policies. For each distribution, the initial back-end load-imbalance is measured using no front-end cache. As shown, the initial load-imbalances for Zipf 0.9, Zipf 0.99, and Zipf 1.20 are 1.35, 1.73, and 4.18 respectively. For each distribution, the minimum required number of cache-lines for LRU, LFU, ARC, and CoT to achieve a target load-imbalance of $I_t = 1.1$ is reported. As shown, CoT requires **50% to 93.75% less cache-lines** than other replacement policies to achieve I_t. Since

LRU-2 is configured with a history size equals to CoT's tracker size, LRU-2 requires the second least number of cache-lines to achieve I_t.

3.5.4 End-to-End Evaluation

In this section, we evaluate the effect of front-end caches using LRU, LFU, ARC, LRU-2, and CoT replacement policies on the overall running time of different workloads. This experiment also demonstrates the overhead of front-end caches on the overall running time. In this experiment, we use 3 different workload distributions, namely, uniform, Zipfian (s = 0.99), and Zipfian (s = 1.2) distributions as shown in Figure 3.5. For all the three workloads, each replacement policy is configured with 512 cache-lines. Also, CoT and LRU-2 maintains a tracker (history) to cache size ratio of 8:1 for Zipfian 0.99 and 4:1 for both Zipfian 1.2 and uniform distributions. In this experiment, a total of 1M accesses are sent to the caching servers by 20 client threads running on one client machine. Each experiment is executed 10 times and the average overall running time with 95% confidence intervals are reported in Figure 3.5.

In this experiment, the front-end servers are allocated in the same cluster as the back-end servers. The average Round-Trip Time (RTT) between front-end machines and back-end machines is 244μs. This small RTT allows us to fairly measure the overhead of front-end caches by minimizing the performance advantages achieved by front-end cache hits. In real-world deployments where front-end servers are deployed in edge-datacenters and the RTT between front-end servers and back-end servers is in order of **10s of ms**, front-end caches achieve more significant performance gains.

The uniform workload is used to measure the overhead of front-end caches. In a uniform workload, all keys in the key space are equally hot and front-end caches cannot take any advantage of workload skew to benefit some keys over others. Therefore, front-

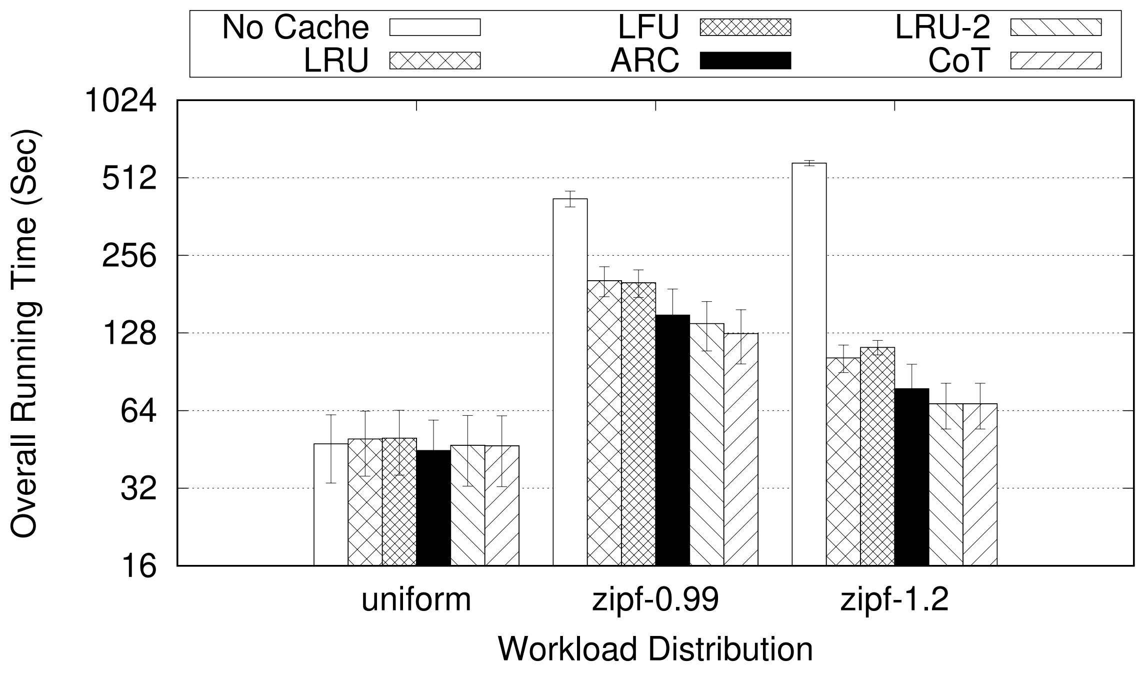


Figure 3.5: The effect of front-end caching on the end-to-end overall running time of 1M lookups using different workload distributions.

end caches only introduce the overhead of maintaining the cache without achieving any significant performance gains. As shown in Figure 3.5, there is no significant statistical difference between the overall running time when there is no front-end cache and when there is a small front-end cache with different replacement policies. Adding a small front-end cache does not incur running time overhead even for replacement policies that use a heap (e.g., LFU, LRU-2, and CoT).

The workloads Zipfian 0.99 and Zipfian 1.2 are used to show the advantage of front-end caches even when the network delays between front-end servers and back-end servers are minimal. As shown in Figure 3.5, workload skew results in significant overall running time overhead in the absence of front-end caches. This happens because the most loaded server introduces a performance bottleneck especially under thrashing (managing 20 connections, one from each client thread). As the load-imbalance increases, the effect of this bottleneck is worsen. Specifically, in Figure 3.5, the overall running time of

Zipfian 0.99 and Zipfian 1.2 workloads are respectively 8.9x and 12.27x of the uniform workload when no front-end cache is deployed. Deploying a small front-end cache of 512 cachelines significantly reduces the effect of back-end bottlenecks. Deploying a CoT small cache in the front-end results in 70% running time reduction for Zipfian 0.99 and 88% running time reduction for Zipfian 1.2 in comparison to having no front-end cache. Other replacement policies achieve running time reductions of 52% to 67% for Zipfian 0.99 and 80% to 88% for Zipfian 1.2. LRU-2 achieves the second best average overall running time after CoT with no significant statistical difference between the two policies. Since both policies use the same tracker (history) size, this again suggests that having a bigger tracker helps separate cold and noisy keys from hot keys. Since the ideal tracker to cache size ratio differs from one workload to another, having an automatic and dynamic way to configure this ratio at run-time while serving workload gives CoT a big leap over statically configured replacement policies.

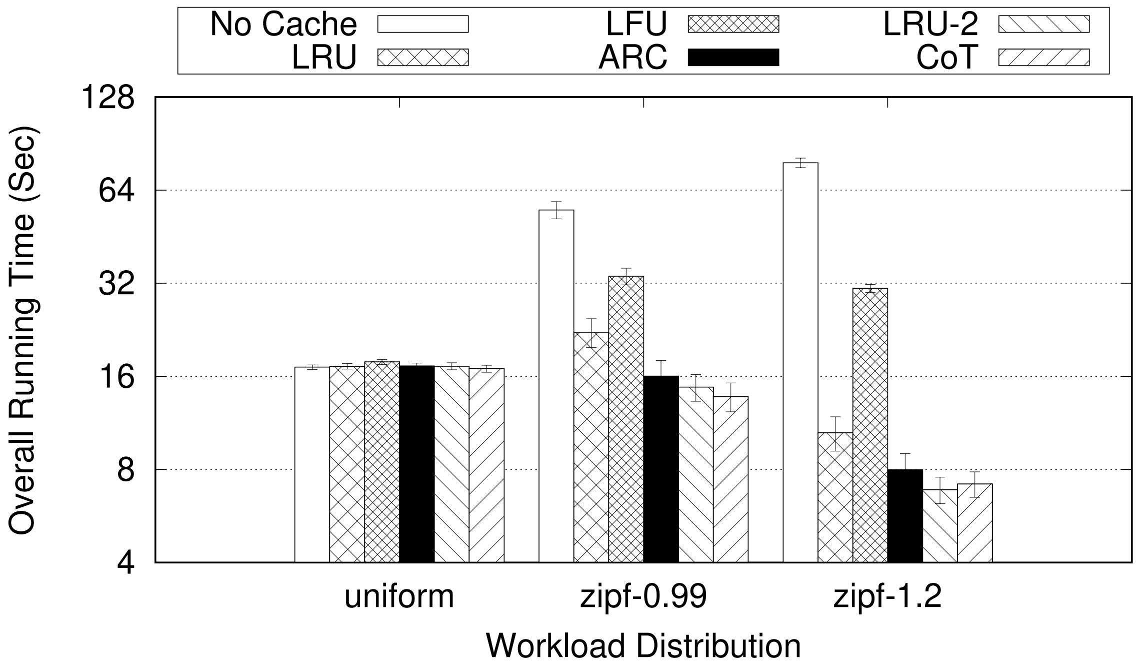


Figure 3.6: The effect of front-end caching on the end-to-end overall running time of 50K lookups using different workload distributions sent by only **one** client thread.

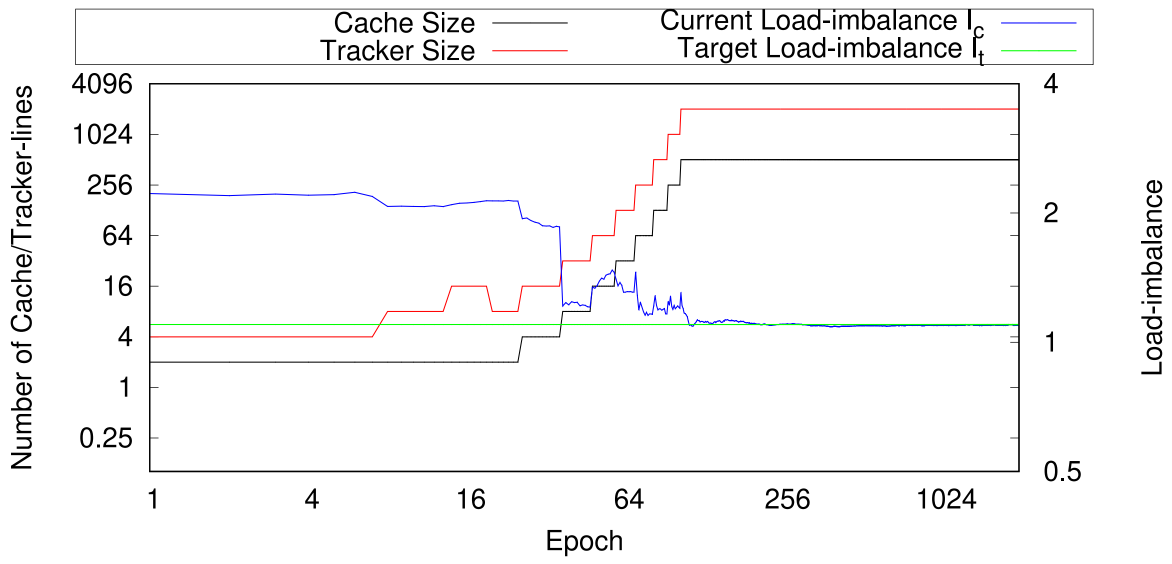
To isolate the effect of both front-end and back-end thrashing on the overall running time, we run the same experiment with only one client thread that executes 50K lookups (1M/20) and we report the results of this experiment in Figure 3.6. The first interesting observation of this experiment is that the overall running time of Zipfian 0.99 and Zipfian 1.2 workloads are respectively 3.2x and 4.5x of the uniform workload when no front-end cache is deployed. These numbers are proportional to the load-imbalance factors of these two distributions (1.73 for Zipfian 0.99 and 4.18 for Zipfian 1.2). These factors are significantly worsen under thrashing as shown in the previous experiment. The second interesting observation is that deploying a small front-end cache in a non-thrashing environment results in a lower overall running time for skewed workload (e.g., Zipfian 0.99 and Zipfian 1.2) than for a uniform workload. This occurs because front-end caches eliminate back-end load-imbalance and locally serve lookups as well.

3.5.5 Adaptive Resizing

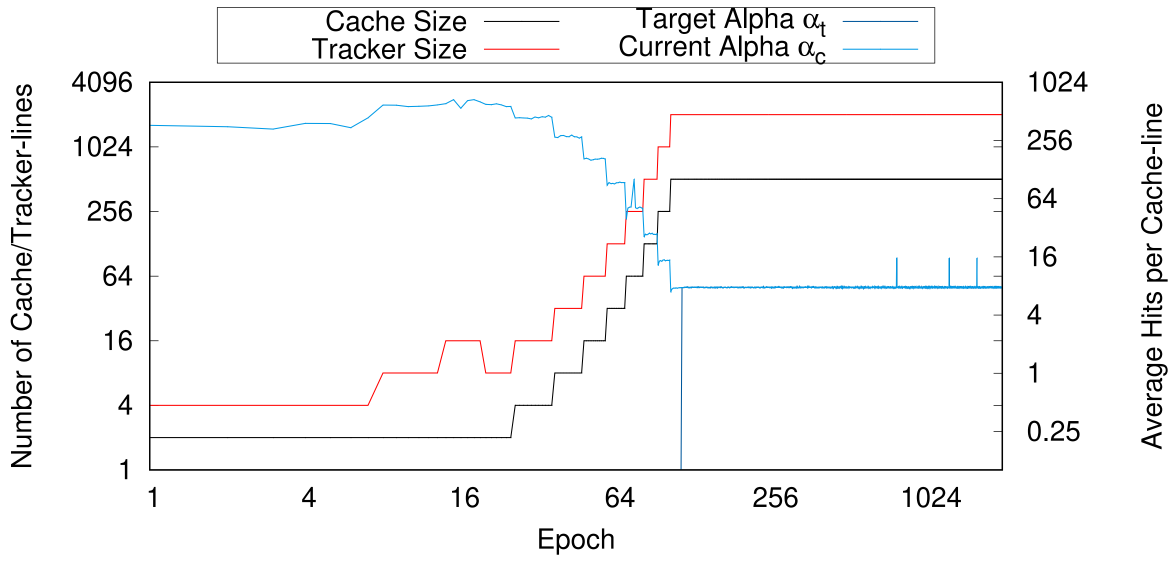
This section evaluates CoT's auto-configure and resizing algorithms. *First*, we configure a front-end client that serves a Zipfian 1.2 workload with a tiny cache of size *two* cachelines and a tracker of size of *four* tracking entries. This experiment aims to show how CoT expands cache and tracker sizes to achieve a target load-imbalance I_t as shown in Figure 3.7. After CoT reaches the cache size that achieves I_t, the average hit per cache-line α_t is recorded as explained in Algorithm 8. *Second*, we alter the workload distribution to uniform and monitors how CoT shrinks tracker and cache sizes in response to workload changes without violating the load-imbalance target I_t in Figure 3.8. In both experiments, I_t is set to 1.1 and the epoch size is 5000 accesses. In both Figures 3.7a and 3.8a, the x-axis represents the epoch number, the left y-axis represents the number of tracker and cache lines, and the right y-axis represents the load-imbalance.

The black and red lines represent cache and tracker sizes respectively with respect to the left y-axis. The blue and green lines represent the current load-imbalance and the target load-imbalance respectively with respect to the right y-axis. Same axis description applies for both Figures 3.7b and 3.8b except that the right y-axis represents the average hit per cache-line during each epoch. Also, the light blue and the dark blue lines represent the current average hit per cache-line and the target hit per cache-line at each epoch with respect to the right y-axis.

In Figure 3.7a, CoT is initially configured with a cache of size 2 and a tracker of size 4. CoT's resizing algorithm runs in 2 phases. In the first phase, CoT discovers the ideal tracker-to-cache size ratio that maximizes the hit rate for a fixed cache size for the current workload. For this, CoT fixes the cache size and doubles the tracker size until doubling the tracker size achieves no significant benefit on the hit rate. This is shown in Figure 3.7b in the first 15 epochs. CoT allows a warm up period of 5 epochs after each tracker or cache resizing decision. Notice that increasing the tracker size while fixing the cache size reduces the current load-imbalance I_c (shown in Figure 3.7a) and increases the current observed hit per cache-line α_c (shown in Figure 3.7b). Figure 3.7b shows that CoT first expands the tracker size to 16 and during the warm up epochs (epochs 10-15), CoT observes no significant benefit in terms of α_c when compared to a tracker size of 8. In response, CoT therefore shrinks the tracker size to 8 as shown in the dip in the red line in Figure 3.7b at epoch 16. Afterwards, CoT starts phase 2 searching for the smallest cache size that achieves I_t. For this, CoT doubles the tracker and caches sizes until the target load-imbalance is achieved and the inequality $I_c \leq I_t$ holds as shown in Figure 3.7a. CoT captures α_t when I_t is first achieved. α_t determines the quality of the cached keys when I_t is reached for the first time. In this experiment, CoT does not trigger resizing if I_c is within 2% of I_t. Also, as the cache size increases, α_c decreases as the skew of the additionally cached keys decreases. For a Zipfian 1.2 workload and



(a) Changes in cache and tracker sizes and the current load-imbalance I_c over epochs.

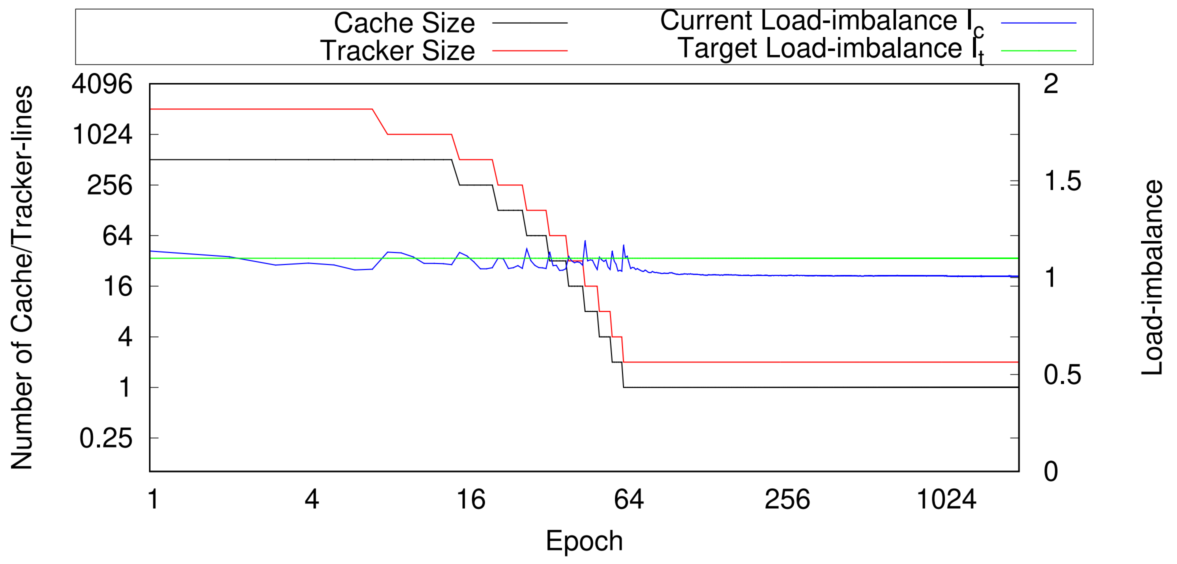


(b) Changes in cache and tracker sizes and the current hit rate per cacheline α_c over epochs.

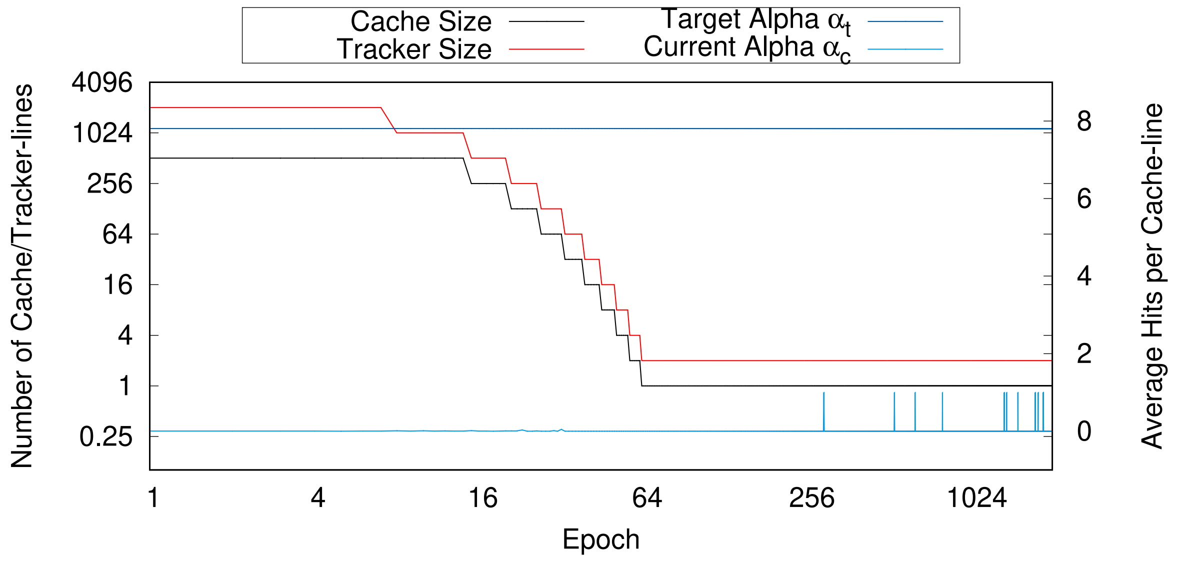
Figure 3.7: CoT adaptively expands tracker and cache sizes to achieve a target load-imbalance $I_t = 1.1$ for a Zipfian 1.2 workload.

to achieve $I_t = 1.1$, CoT requires 512 cache-lines and 2048 tracker lines and achieves an average hit per cache-line of $\alpha_t = 7.8$ per epoch.

Figure 3.8 shows how CoT successfully shrinks tracker and cache sizes in response to



(a) Changes in cache and tracker sizes and the current load-imbalance I_c over epochs.



(b) Changes in cache and tracker sizes and the current hit rate per cache-line α_c over epochs.

Figure 3.8: CoT adaptively shrinks tracker and cache sizes in response to changing the workload to uniform.

workload skew drop without violating I_t. After running the experiment in Figure 3.7, we alter the workload to uniform. Therefore, CoT detects a drop in the current average hit per cache-line as shown in Figure 3.8b. At the same time, CoT observe that the current load-imbalance I_c achieves the inequality $I_c \leq I_t = 1.1$. Therefore, CoT decides to shrink both the tracker and cache sizes until either $\alpha_c \approx \alpha_t = 7.8$ or I_t is violated or until cache and tracker sizes are negligible. First, CoT resets the tracker to cache size ratio to 2:1 and then searches for the right tracker to cache size ratio for the current workload. Since the workload is uniform, expanding the tracker size beyond double the cache size achieves no hit-rate gains as shown in Figure 3.8b. Therefore, CoT moves to the second phase of shrinking both tracker and cache sizes as long α_t is not achieved and I_t is not violated. As shown, in Figure 3.8, CoT shrinks both the tracker and the cache sizes until front-end cache size becomes negligible. As shown in Figure 3.8a, CoT shrinks cache and tracker sizes while ensuring that the target load-imbalance is not violated.

3.6 Related Work

Distributed caches are widely deployed to serve social networks and the web at scale [59, 167, 211]. Real-world workloads are typically skewed with few keys that are significantly hotter than other keys [122]. This skew can cause load-imbalance among the caching servers. Load-imbalancing negatively affects the overall performance of the caching layer. Therefore, many works in the literature have addressed the load-imbalacing problem from different angles. Solutions use different load-monitoring techniques (e.g., centralized tracking [43, 124, 42, 207], server-side tracking [121, 71], and client-side tracking [96, 126]). Based on the load-monitoring, different solutions redistribute keys among caching servers at different granularities. The following paragraphs summarize the related works under different categories.

Centralized load-monitoring: Slicer [43] separates the data serving plane from the control plane. The key space is divided into *slices* where each slice is assigned to one or more servers. The control plane is a centralized system component that collects the access information of each slice and the workload per server. The control plane periodically runs an optimization that generates a new slice assignment. This assignment might result in redistributing, repartitioning, or replicating slices among servers to achieve better load-balancing. Unlike in Centrifuge [42], Slicer does not use consistent hashing to map keys to servers. Instead, Slicer distributes the generated assignments to the front-end servers to allow them to locate keys. Also, Slicer highly replicates the centralized control plane to achieve high availability and to solve the fault-tolerance problem in both Centrifuge [42] and in [71]. CoT is complementary to systems like Slicer. Our goal is to cache heavy hitters at front-end servers to reduce key skew at back-end caching servers and hence, reduce Slicer's initiated re-configurations. Our focus is on developing a replacement policy and an adaptive cache resizing algorithm to enhance the performance of front-end caches. Also, our approach is distributed and front-end driven that does not require any system component to develop a global view of the workload. This allows CoT to scale to thousands of front-end servers without introducing any centralized bottlenecks.

Server side load-monitoring: Another approach to load-monitoring is to distribute the load-monitoring among the caching shard servers. In [121], each caching server tracks its own hot-spots. When the hotness of a key surpasses a certain threshold, this key is replicated to γ caching servers and the replication decision is broadcast to all the front-end servers. Any further accesses on this hot key shall be equally distributed among these γ servers. This approach aims to distribute the workload of the hot keys among multiple caching servers to achieve better load balancing. Cheng et al. [71] extend the work in [121] to allow moving coarse-grain key cachelets (shards) among threads and caching servers. Our approach reduces the need for server side load-monitoring. Instead, load-monitoring happens at the edge. This allows individual front-end servers to independently identify their local trends.

Client side load-monitoring: Fan et al. [96] theoretically show through analysis and simulation that a small cache in the client side can provide load balancing to n caching servers by caching only $O(n\ log(n))$ entries. Their result provides the theoretical foundations for our work. Unlike in [96], our approach does not assume perfect caching nor *a priori* knowledge of the workload access distribution. Gavrielatos et al. [99] propose *symmetric caching* to track and cache the hot-most items at every front-end server. Symmetric caching assumes that all front-end servers obtain the same access distribution and hence allocates the same cache size to all front-end servers. However, different front-end servers might serve different geographical regions and therefore observe different access distributions. CoT discovers the workload access distribution independently at each front-end server and adjusts the cache size to achieve a target load-imbalance I_t. NetCache [126] uses programmable switches to implement heavy hitter tracking and caching at the network level. Like symmetric caching, NetCache assumes a fixed cache size for different access distributions. To the best of our knowledge, CoT is the first front-end caching algorithm that exploits the cloud elasticity allowing each front-end server to independently reduce the necessary required front-end cache memory to achieve back-end load-balance.

Other works in the literature focus on maximizing cache hit rates for fixed memory sizes. Cidon et al. [74, 75] redistribute available memory among memory slabs to maximize memory utilization and reduce cache miss rates. Fan et al. [95] use cuckoo hashing [174] to increase memory utilization. Lim et al. [144] increase memory locality by assigning requests that access the same data item to the same CPU. Bechmann et al. [56] propose Least Hit Density (LHD), a new cache replacement policy. LHD predicts the expected hit density of each object and evicts the object with the lowest hit density. LHD aims to evict objects that contribute low hit rates with respect to the cache space they occupy. Unlike these works, CoT does not assume a static cache size. In contrast, CoT maximizes the hit rate of the available cache and exploits the cloud elasticity allowing front-end servers to independently expand or shrink their cache memory sizes as needed.

3.7 CoT Concluding Remarks

This chapter presents *Cache on Track* (CoT), a decentralized, elastic, and predictive cache at the edge of a distributed cloud-based caching infrastructure. CoT proposes a new cache replacement policy specifically tailored for small front-end caches that serve skewed workloads. Using CoT, system administrators do not need to statically specify cache size at each front-end in-advance. Instead, they specify a target back-end load-imbalance I_t and CoT dynamically adjusts front-end cache sizes to achieve I_t. Our experiments show that CoT's replacement policy outperforms the hit-rates of LRU, LFU, ARC, and LRU-2 for the same cache size on different skewed workloads. CoT achieves a target server size load-imbalance with 50% to 93.75% less front-end cache in comparison to other replacement policies. Finally, our experiments show that CoT's resizing algorithm successfully **auto-configures** front-end tracker and cache sizes to achieve the back-end target load-imbalance I_t in the presence of workload distribution changes.

Chapter 4

Express Your Online Persona without Revealing Your Sensitive Attributes

4.1 Overview

Over the past decade, social network platforms such as Facebook, Twitter, and Instagram have attracted hundreds of millions of users [29, 39, 33]. These platforms are widely and pervasively used to communicate, create online communities [100], and socialize. Social media users develop, over time, online persona [220] that reflect their overall interests, activism, and diverse orientations. Users have numerous followers that are specifically interested in their personas and their postings which are aligned with these personas. However, due to the rise of machine learning and deep learning techniques, user posts and social network interactions can be used to accurately and automatically infer many user persona attributes such as gender, ethnicity, age, political interest, and location [133, 177, 225, 223]. Recent works show that it is possible to predict an individual user's location solely using content-based analysis of the user's posts [72, 68]. Zhang et al. [223] show that hashtags in user posts can alone be used to precisely infer a user's location with accuracy of 70% to 76%. Also, Facebook likes analysis has been successfully used to distinguish between Democrats and Republicans with 85% accuracy [133].

Social network giants have widely used attribute inference to serve personalized trending topics, to suggest pages to like and accounts to follow, and to notify users about hyper-local events. In addition, social networks such as Facebook use tracking [30] and inference techniques to classify users into categories (e.g. Expats, Away from hometown, Politically Liberal, etc.). These categories are used by advertisers and small businesses to enhance directed advertising campaigns. However, recent news about the *Cambridge Analytica scandal* [25] and similar *data breaches* [34] suggest that users cannot depend on the social network providers to preserve their privacy. User sensitive attributes such as ***gender, ethnicity, and location*** have been widely misused in *illegally discriminating ads*, *microtargeting*, and *surveillance*. A recent ACLU report [19] shows that Facebook illegally allowed employers to exclude women from receiving their job ads on Facebook. Also, several reports have shown that Facebook allows discrimination against some ethnic groups in housing ads [31]. News about the Russian-linked Facebook Ads during the 2016 election suggests that the campaign targeted voters in swing states [16] and specifically in Michigan and Wisconsin [17]. In addition, location data collected from Facebook, Twitter, and Instagram has been used to target activists of color [11].

An **online-persona** can be thought of as the set of user attributes that can be inferred about a user from their online postings and interactions. These attributes fall into two categories: *public* and *private* persona attributes. Users should **decide** which attributes fall in each category. Some attributes (e.g., political orientation and ethnicity) should be publicly revealed as a user's followers might follow her because of her public persona attributes. Other attributes (e.g., gender and location) are private and sensitive, and the user would not like them to be revealed. However, with the above mentioned inference methods, the social media providers, as well as any adversary receiving the user posting can reveal a user's sensitive attributes.

To remedy this situation, we propose *multifaceted privacy*, a novel privacy model that aims to obfuscate a user's sensitive attributes while revealing the user's public persona attributes. Multifaceted privacy allows users to freely express their online public personas without revealing any sensitive attributes **of their choice**. For example, a *#BLM* activist might want to hide her location from the police and from discriminating advertisers while continuing to post about topics specifically related to her political movement. This activist can try to hide her location by disabling the geo-tagging feature of her posts and hiding her IP address using an IP obfuscation browser like Tor [18]. However, recent works have shown that content-based location inferences can successfully and accurately predict a user's location solely based on the content of her posts [72, 68, 223]. If this activist frequently posts about topics that discuss BLM events in Montpelier, Vermont, she is most probably a resident of Montpelier (Montpelier has a low African American population).

To achieve multifaceted privacy, we build *Aegis*[1], a prototype client-centric social network stream processing system that enables social network users to take charge of protecting their own privacy, instead of depending on the social network providers. Our philosophy in building Aegis is that social network users need to introduce some noisy interactions and obfuscation posts to confuse **content based attribute inferences**. This idea is inspired from Rivest's chaffing and winnowing privacy model in [185]. Unlike in [185] where the sender and receiver exchange a secret that allows the receiver to easily distinguish the chaff from the wheat, in social networks, a user (sender) posts to an open world of followers (receivers) and it is not feasible to exchange a secret with every

[1]Aegis: a shield in the Greek mythology.

recipient to distinguish real posts from the obfuscation ones. In addition, a subset of the recipients could be adversaries who try to infer the user's sensitive attributes from their postings. In fact, our approach, and unlike other approaches in [185, 204] is designed to hide private attributes from all recipients of a user's postings, irrespective whether they are humans who follow the user for their writing reflecting their public on-line persona attributes, which are preserved, or automated profiling systems trying to discern the user's private attributes.

Choosing this noise introduces a challenging *dichotomy* and tension between the utility of the user persona and her privacy. Similar notions of dichotomy between sensitive and non sensitive personal attributes have been explored in sociology and are referred to as contextual integrity [168]. Obfuscation posts need to be carefully chosen to achieve obfuscation of private attributes without damaging the user's public persona. For example, a #NoBanNoWall activist loses persona utility if she writes about #BuildTheWall to hide her location. Multifaceted privacy represents a continuum between privacy and persona utility. Figure 4.1 captures this continuum. Both the x-axis and the y-axis represent the same persona attributes that can be derived from a user's posting. The particular example in Figure 4.1 only considers the location, the gender, the ethnicity, and the political interest attributes of a user (as shown in the x-axis left to right or on the y-axis top to bottom). The user needs to pick which attributes, among the four considered attributes, should be revealed and which should be hidden. This is represented by the diagonal line, which explicitly captures this trade-off. In Figure 1, the x-axis represents the attributes the user would like to hide, while the y-axis represents the attributes the user wants to reveal. Clearly they must be mutually exclusive. In this particular example, the user wants to obfuscate and hide her location and gender, while at the same time revealing her political interest and ethnicity. Figure 4.1 shows that privacy is the *reciprocal* of the persona utility. Any attribute that needs to be kept private cannot be preserved in the public persona. As illustrated, the more persona attributes are kept private, the more obfuscation overhead is needed to achieve the privacy of these persona attributes. A user who chooses to publicly reveal all her persona attributes achieves no attribute privacy and hence requires no obfuscation posting overhead. Note that users can reorder the attributes on the axes of Figure 4.1 in order to achieve their intended public/private attribute separation. The main goal of Aegis is to automatically find and suggest the noise (obfuscation) postings that achieve the multifaceted privacy to this user.

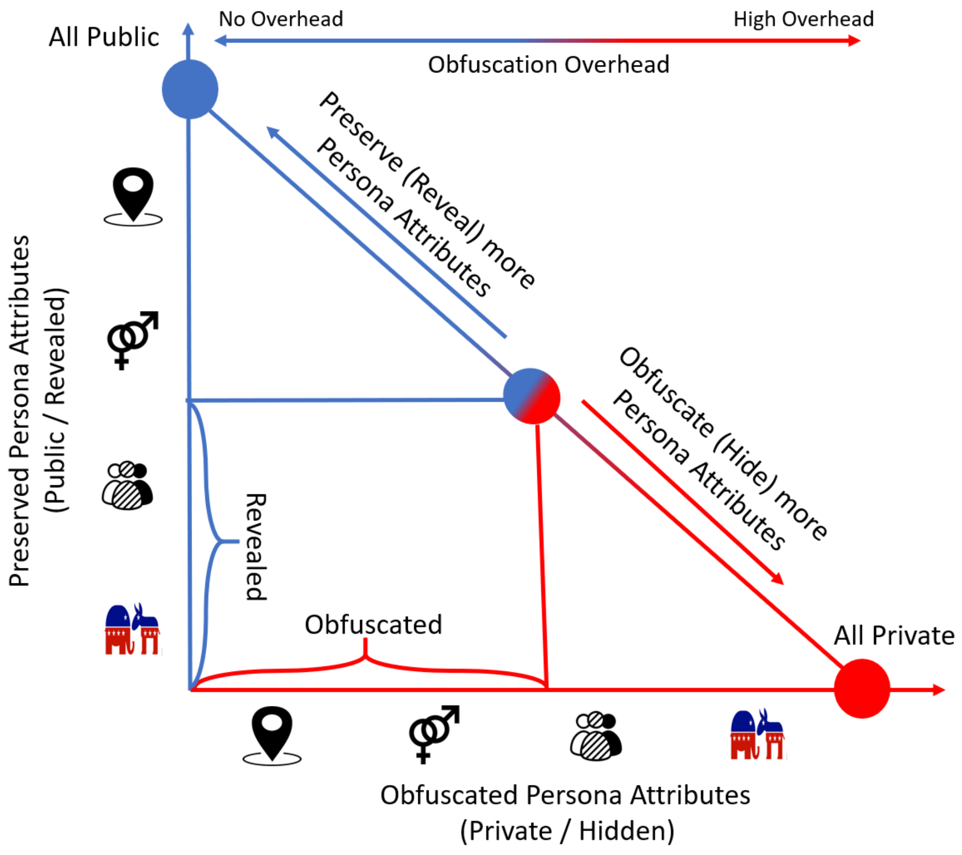

All Public
No Overhead
High Overhead
Obfuscation Overhead
Preserve (Reveal) more Persona Attributes
Obfuscate (Hide) more Persona Attributes
Preserved Persona Attributes (Public / Revealed)
Revealed
Obfuscated
All Private
Obfuscated Persona Attributes (Private / Hidden)

Figure 4.1: The dichotomy of multifaceted privacy: persona vs. privacy.

Unlike previous approaches that require users to change their posts and *hashtags* [223] to hide their sensitive attributes, Aegis allows users to publish their original posts without changing their content. Our experiments show that adding obfuscation posts successfully preserves multifaceted privacy. Aegis considers the added noise as the cost to pay for achieving multifaceted privacy. Therefore, Aegis targets users who are willing to write additional posts to hide their sensitive attributes.

Aegis is user-centric, as we believe that users need to take control of their own privacy concerns and cannot depend on the social media providers. This is challenging as it requires direct user engagement and certain sacrifices. However, we believe Aegis will help better understand the complexity of privacy as well as the role for individual engagement and responsibility. Aegis represents a first step in the long path to better understanding the tensions between user privacy, the utility of social media, and trust in public social media providers. This is an overdue discussion that needs to be discussed by the scientific community, and we believe Aegis will help provide the medium for this discussion.

Our contributions are summarized as follows:

- We propose *multifaceted privacy*, a novel privacy model that represents a continuum between the privacy of sensitive private attributes and public persona attributes.

- We build Aegis, a prototype user-centric social network stream processing system that preserves multifaceted privacy. Aegis continuously analyzes social media streams to suggest topics to post that are aligned with the user's public persona but hide their sensitive attributes.

- We conduct an extensive experimental study to show that Aegis can successfully achieve multifaceted privacy.

The rest of the chapter is organized as follows. We explain the models of user, topic, and security in Section 4.2. Topic classification algorithms and data structures that achieve multifaceted privacy are described in Section 4.3 and Aegis's system design is explained in Section 4.4. Afterwards, an experimental evaluation is conducted in Section 4.5 to evaluate the effectiveness of Aegis in achieving the multifaceted privacy. The related work is presented in Section 4.6 and the chapter is concluded in Section 4.7. Future extensions are presented in Section 4.8.

4.2 Models

In this section, we present the user, topic, and security models. The user and topic models explain how users and topics are represented in the system. The security model presents both the privacy and the adversary models.

4.2.1 User Model

Our user model is similar to the user model presented in [101]. The set U is the set of social network users where $U = \{u_1, u_2, ...\}$. A user u_i is represented by a vector of attributes V_{u_i} (e.g., gender $V_{u_i}[g]$, ethnicity $V_{u_i}[e]$, age $V_{u_i}[a]$, political interest $V_{u_i}[p]$, location $V_{u_i}[l]$, etc). Each attribute a has a domain $a.d$ and the attribute values are picked from this domain. For example, the gender attribute g has domain $g.d = \{\text{male, female}\}$[2] and $\forall_{u_i \in U} V_{u_i}[g] \in g.d$. An example user u_x is represented by the vector V_{u_x} where $V_{u_x} =$ (g: female, e: African American, a: 23, p: Democrat, l: New York). Attribute domains can form a hierarchy (e.g., location: city $\rightarrow$ county $\rightarrow$ state $\rightarrow$ country) and an attribute can be generalized by climbing up this hierarchy. A user who lives in Los Angeles is also a resident of Orange County, California, and the United States. Other attributes can form trivial hierarchies (e.g., gender: male or female $\rightarrow$ * (no knowledge)).

The user attribute vector V_{u_i} is divided into two main categories: 1) the set of public

[2]Due to the limitation of the inference models, the gender attribute is considered only binary. However, better models can be used to infer non binary gender attribute values.

persona attributes $V^p_{u_i}$ and 2) the set of private *sensitive* attributes $V^s_{u_i}$. Multifaceted privacy aims to publicly reveal all persona attributes in $V^p_{u_i}$ while hiding all sensitive attributes in $V^s_{u_i}$. Each user defines her $V^s_{u_i}$ and $V^p_{u_i}$ *a priori*. As shown in Figure 4.1, attributes in $V^p_{u_i}$ are the complement of the attributes in $V^s_{u_i}$. Therefore, each attribute either belongs to $V^p_{u_i}$ or $V^s_{u_i}$.

4.2.2 Topic Model

The set T represents the set of all topics that are discussed by all the social network users in U. $T^\tau_i \subset T$ represents the set of all the topics posted by user u_i's upto time τ where $T^\tau_i = \{t^1_i, t^2_i, ..., t^n_i\}$. A topics is identified by a hashtag or a keyword. Each topic is characterized by a set of attributes, which are identical to user attributes, e.g., ethnicity, location, gender, and political interest. Topic attributes captures the attribute distributions of the users who post about this particular topic. Hence, the attributes of a topic are collectively interfered from the attributes of the users who write posts that include the topic's hashtag or keyword. For example, an analysis of the ethnicity of the users who post about the topic `#BLM` can result in the distribution 10% Asian, 25% White, 15% Hispanic, and 50% Black for the ethnicity attribute of the topic `#BLM`. This distribution means that Asians, Whites, Hispanics, and Blacks post about the topic `#BLM` and 50% of the users who post about this topic are Black. A topic t_i is represented by a vector of attribute distributions V_{t_i} where $V_{t_i}[g]$, $V_{t_i}[e]$, $V_{t_i}[p]$, and $V_{t_i}[l]$ are respectively the gender, the ethnicity, the political interest, and the location distributions of the users who post about t_i.

4.2.3 Security Model

An approach that is commonly used for attribute obfuscation is *generalization*. The idea behind attribute generalization is to report a generalized value of a user's sensitive attribute in order to hide the actual attribute value within. Consider location as a sensitive attribute example. Many works [163, 223] have used location generalization in different contexts. Mokbel et al. [163] use location generalization to hide a user's exact location from Location Based Services (LBS). A query that asks *"what is the nearest gas station to my exact location in Stanford, CA?"* should be altered to *"list all gas stations in California"*. Notice that the returned result of the altered query has to be filtered at the client side to find the answer of the original query. Similarly, Andres et al. [48] propose geo-indistinguishability, a location privacy model that uses differential privacy to hide a user's exact location in a circle of radius r from LBS providers. The wider the generalization range, the more privacy achieved, and the more network and processing overhead are incurred at the client side. Similarly, in the context of social networks, Zhang et al. [223] require Twitter users to generalize their location revealing hashtags in order to hide their exact location. For example, a user whose post includes "#WillisTower" should be generalized to "#Chicago" to hide a user's exact location. Notice that generalization requires users to alter their original posts or queries.

Rather than generalization, we adopt the notion of k-indisting-uishability, where the values of a user's sensitive attributes are indistinguishable among k other values chosen from the sensitive attribute domain. When the user posts about a topic that reveals the value of one of her sensitive attributes, she can hide this post among other posts that reveal another $k-1$ values of the same sensitive attributes thus significantly reduce the probability of inferring the user's actual sensitive attribute values. The purpose of the obfuscation posts is to equalize the probability of inferring a sensitive attribute value among the k attribute values and hence achieving k-indistinguishability. We adopt k-indistinguishability rather than generalization as it allows end-users to discuss topics that could reveal their sensitive attributes at a fine-grain level while hiding their actual sensitive attribute values at this fine-grain level. For example, generalization hides a user's city level location by requiring the user to write topics that are connected to a state or a country level locations but not connected to a specific city. On the other hand, k-indistinguishability allows a user to post topics that reveal a specific city location. Still, the actual city location of the user is hidden among $k-1$ other city locations. K-indistinguishability allows end-users to have more fine control over their attribute privacy.

For every sensitive attribute $s \in V^s_{u_i}$, the user defines an indistinguishability parameter k_s. k_s determines the number of attribute values among which the real value of parameter s is hidden. For example, a user who lives in LA can set $k_l = 3$ in order to hide her original city location, LA, among 3 different cities (e.g., LA, SF, and NYC). This means that a content-based inference attack should not be able to distinguish the user's real city location among the set {LA, SF, NYC}. As explained in 4.2.1, attribute domains either form multi-level hierarchies (e.g., location) or trivial hierarchies (e.g., gender and ethnicity). Unlike in attribute generalization where a user's attribute value is generalized by climbing up the attribute hierarchy, k-attribute-indistinguishability achieves the privacy of an attribute value by hiding it among $k-1$ attribute values chosen from the siblings of the actual attribute value in the same hierarchy level (e.g., a user's city level location is hidden among $k-1$ other cities instead of generalizing it to the state or the country levels). The following inference attack explains when k-attribute-indistinguishability is achieved or violated.

The adversary assumptions: the adversary model and the inference attacks are similar to the ones presented in [223]. However, unlike in [223], our inference attack is not only limit to the location attribute but can be extended to infer every user sensitive attribute in $V^s_{u_i}$. The adversary has access to the set of all topics T and all the public posts related to each topic. This assumption covers any adversary who can crawl or get access to the public posts of every topic in the social network. As proposed in Section 4.1, the target user u_i does not reveal her sensitive attribute values to the public (e.g., a user who wants to hide her location must obfuscates her IP address and disable the geo-tagging feature for her posts). Therefore, the adversary can only see the content of the public posts published by u_i. The adversary uses their knowledge about the set of all topics T and the set of topics T^τ_i discussed by u_i to infer her sensitive attributes. Multifaceted privacy protects users against an adversary who performs *content-based inference* attacks. Therefore, multifaceted privacy assumes that the adversary does not have any *side channel knowledge* that can be used to reveal a user's sensitive attribute value (e.g., another online profile that is directly linked to the user u_i where sensitive attributes such as gender, ethnicity, or location are revealed).

Inference attack: the adversary's ultimate goal is to reveal or at least have high confidence in the knowledge of the sensitive attribute values of the target user u_i. For this, the adversary runs a content-based attack as follows. First, the adversary crawls the set of topics T^τ_i that user u_i wrote about. For each topic, the adversary infers the demographics of the users who wrote about this topic. Then, the adversary aggregates the demographics of all the topics in T^τ_i. The adversary uses the aggregated demographics to estimate the sensitive attributes of user u_i. The details of the inference attack is explained as follows.

For each topic $t_j \in T^\tau_i$, an adversary crawls the set of posts P_{t_j} that discusses topic t_j and for each post $p_i \in P_{t_j}$, the adversary uses some models to infer the gender, ethnicity, political interest, and location of the user who wrote this post. Then, the adversary uses the inferred attributes of each post in topic t_j to populate t_j's distribution vector V_{t_j}. For example, $V_{t_j}[g]$ is the gender distribution of all users who wrote about topic t_j. Similarly, $V_{t_j}[e]$, $V_{t_j}[p]$, and $V_{t_j}[l]$ are the ethnicity, political interest, and location distributions of the users who posted about topic t_j. We define $V^*_{u_i}$ as a vector of attribute distributions that is used to estimate the attributes of user u_i. $V^*_{u_i}$ is the result of aggregating the normalized V_{t_j} for every topic $t_j \in T_i^\tau$ as shown in Equation 4.1.

$$V^*_{u_i} = \frac{\sum_{t_j \in T_i^\tau} \frac{V_{t_j}}{|P_{t_j}|}}{|T_i^\tau|} \tag{4.1}$$

Equation 4.1 shows that the topic's attribute distribution vector V_{t_j} is first normalized by dividing V_{t_j} by the number of posts in topic t_j. This normalization equalizes the effect of every topic $t_j \in T_i^\tau$ on the user's attribute estimations in $V^*_{u_i}$. $V^*_{u_i}$ is the summation of the normalized V_{t_j} for every topic $t_j \in T_i^\tau$ divided by the number of topics in T_i^τ. $V^*_{u_i}[a]$ is the distribution of attribute a for user u_i. For example, a user might have a gender distribution $V^*_{u_i}[g] = \{$female:0.8, male:0.2$\}$. This means that the inference attack using u_i's posted topics suggests that the probability u_i is a female is 80% while u_i is a male is only 20%. For every attribute a, an attacker uses the maximum attribute value $max(V^*_{u_i}[a])$ as an estimation of the actual value $V_{u_i}[a]$. In the previous example, an attacker would infer $V^*_{u_i}[g] =$ female as an estimate of the gender of user u_i. An inference attack succeeds in estimating an attribute a if the attacker can have sufficient confidence in estimating the actual value $V_{u_i}[a]$. This confidence is achieved if the difference between the maximum estimated attribute value of a and the top-k^{th} estimated attribute value of a is greater than Δ_a. For example, if $\Delta_g = 0.1$ and $k_g = 2$ (assuming gender is a binary attribute and it needs to be hidden among the 2 gender attribute values), then an attacker successfully estimates u_i's gender if the $max(V^*_{u_i}[g])$ is distinguishable from the 2^{nd} highest value in $V^*_{u_i}[g]$ by more than 10%. In the previous example where $V^*_{u_i}[g]$ $= \{$female:0.8, male:0.2$\}$, an attacker succeeds to estimate u_i's gender = female as the difference between $V^*_{u_i}[g].female - V^*_{u_i}[g].male \geq \Delta_g$.

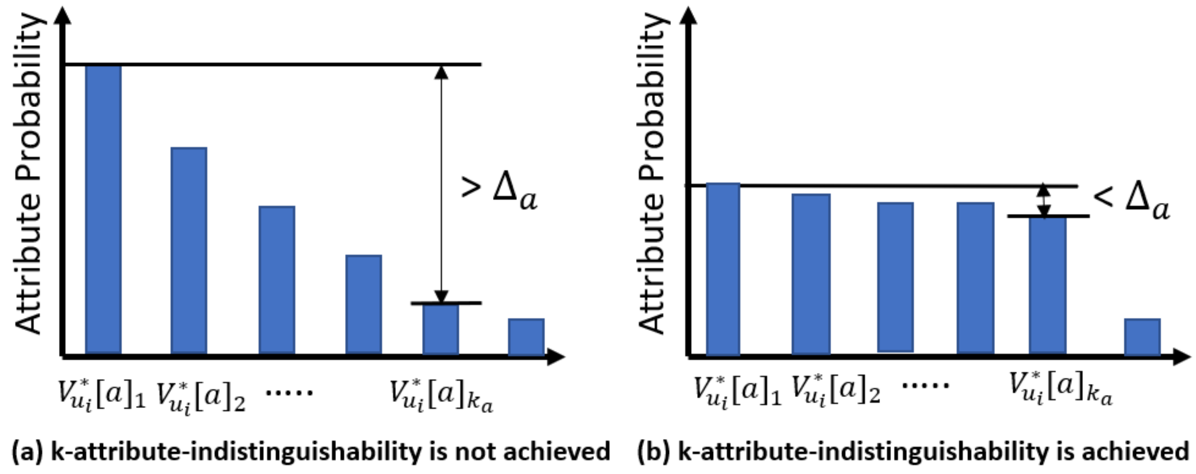


Figure 4.2: Illustrating k-attribute-indistinguishability

Figure 4.2 show an example of a successful inference attack and another of a failed inference attack on attribute a of user u_i. In this example, the domain of attribute a has at least 6 values (e.g., ethnicity or location). As shown in Figure 4.2.a, the maximum estimated attribute value $V^*_{u_i}[a]_1$ is distinguishable from the top-k^{th} (k_a for attribute a) attribute values in $V^*_{u_i}[a]$ by more than Δ_a. In this scenario, an attacker can conclude with high confidence that $V^*_{u_i}[a]_1$ is a good estimate for $V_{u_i}[a]$. However, in Figure 4.2.b, $V^*_{u_i}[a]_1$ is indistinguishable from the top-k^{th} attribute values in $V^*_{u_i}[a]$. In this scenario, the attack is marked failed and k-attribute-indistinguishability is achieved.

The parameter k is used to determine the number of attribute values within which the user's actual attribute value is hidden. The bigger the k, the less the attacker's confidence about the user's actual attribute value. As a result, increasing k introduces uncertainty in the attacker's inference and hence boosts the adversary cost to micro-target users who hide their actual attribute values among k different attribute values. For example, an adversary who wants to target a user in location CA has to pay 3 times the advertisement cost to reach the same user if the user equally hides her state location among 3 other state locations (e.g., CA, IL, and NY).

We understand that the requirement to determine the sensitive attributes and an indistinguishability parameter value k_s for every sensitive attribute $s \in V^s_{u_i}$ could be challenging for many users. Users might not have a sense of the number of attribute values to obfuscate the actual value of a particular attribute. One possible solution to address this usability challenge is to design a questionnaire for Aegis's first time users. This questionnaire could help Aegis understand which persona attributes are sensitive and how critical the privacy of every sensitive attribute is to each specific user. This allows Aegis to auto-configure k_s of every sensitive attribute s of this specific user. The details of such an approach is out of the scope of this work. This work assumes that Aegis is preconfigured with the set $V^s_{u_i}$ and for every attribute $s \in V^s_{u_i}$, the value of k_s is determined.

4.3 Multifaceted Privacy

Multifaceted privacy aims to obfuscate a user's sensitive attributes for every attribute in $V^s_{u_i}$. This has to be done while publicly revealing every attribute in the user's public persona in $V^p_{u_i}$. Various approaches have been used to obfuscate specific sensitive attributes, in particular, *Tagvisor* [223] protects users against content-based inference attacks by requiring users to alter their posts by changing or replacing hashtags that reveal their sensitive attributes, in their case location. Our approach is different, as it is paramount to not only preserving the privacy of the sensitive attributes, but also to preserve the on-line persona of the user, and hence reveal their public attributes. It is critical for a user to post their posting in their own words that reflect their persona. We therefore preserve multifaceted privacy by hiding a specific post among other obfuscation posts. Our approach needs to suggest posts that are aligned with the user's public persona but linked to alternative attribute values of their sensitive attributes in order to obfuscate them. This requires a topic classification model that simplifies the process of suggesting obfuscation posts. For example, consider state level location as a sensitive attribute. To achieve k-location indistinguishability, a user's exact state should be hidden among $k-1$ other states. This requires suggesting obfuscation postings about topics that are mainly discussed in these other $k-1$ states. Users in NY state can use topics that are mainly discussed in IL to obfuscate their location among NY and IL. To discover such potential topics, all topics that are discussed on a social network need to be classified by the sensitive attributes that need to be obfuscated, state level location in this example. A topic is linked to some state if the maximum estimated state location of this topic, $max(V_{t_i}[l])$, is distinguishable from other state location estimates in $V_{t_i}[l]$ by more than Δ_l. For example, if $\Delta_l = 10\%$, a topic that has a state location distribution of {NY=0.6, IL=0.2, CA=0.1, Others=0.1} is linked to NY state while a topic that does not have a distinguishable state location inference by more than Δ_l is not linked to any state. In this section, we first explain a simple but incorrect topic classification model that successfully suggests obfuscation posts that hide a user's sensitive attributes but does not preserve her public persona. Then, we explain how to modify the topic classification model to suggest obfuscation posts that do achieve multifaceted privacy.

A simple incorrect proposal: in this proposal, topics are classified **independently** by every sensitive attribute. As shown in Figure 4.3, each attribute forms an independent hierarchy. The root of the hierarchy has the topics that are not linked to a specific attribute value. Topics that are strongly linked to some attribute value fall down in the hierarchy node that represents this attribute value. For example, state level location attribute forms a hierarchy of two levels. The first level, the root, has all the topics that do not belong to a specific state. A topic like #Trump is widely discussed in all the states and therefore it resides on the root of the location attribute. However,

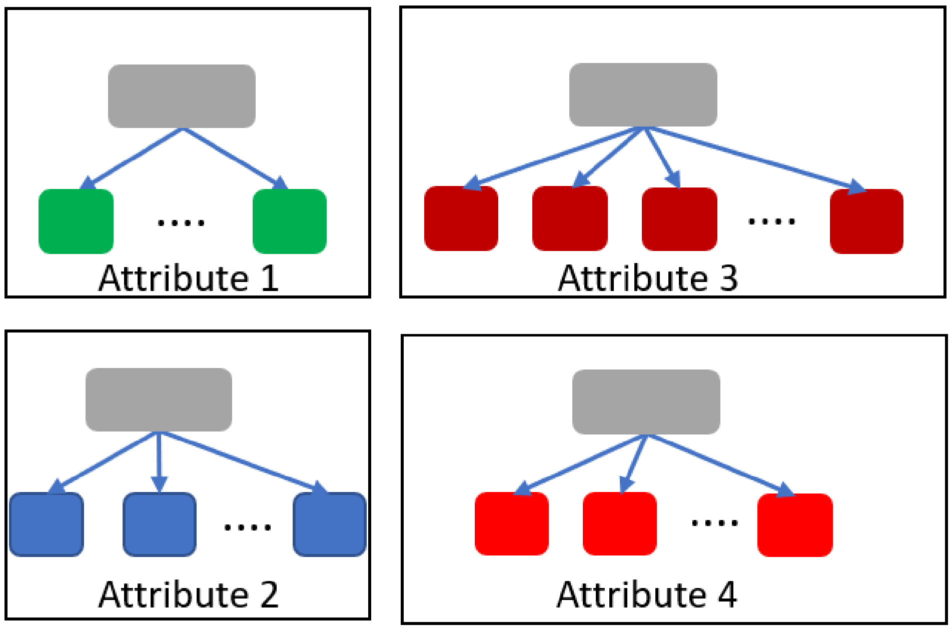


Figure 4.3: Each attribute forms an independent hierarchy

#cowboy is mainly discussed in TX and therefore it falls down in the hierarchy to the TX node. To obfuscate a user's state location, topics need to be selected from the sibling nodes of the user's state in the state level location hierarchy. These topics belong to other locations and can be used to achieve k-location-indistinguishability privacy. Although this proposal successfully achieves location privacy, the suggested posts are not necessarily aligned with the user's public online persona. For example, this obfuscation technique could suggest the topic #BuildTheWall (from TX) to an activist (from NY) who frequently posts about #NoBanNoWall in order to hide her location. This misalignment between the suggested obfuscation posts and the user's public persona discourages users from seeking privacy fearing the damage to their public online persona.

A persona preserving proposal: To overcome the independent classification shortcomings, obfuscation postings need to be suggested from a tree hierarchy that captures relevant dependencies among the specific user's attributes including both public persona and private sensitive attributes. Public attributes need to reside on the upper

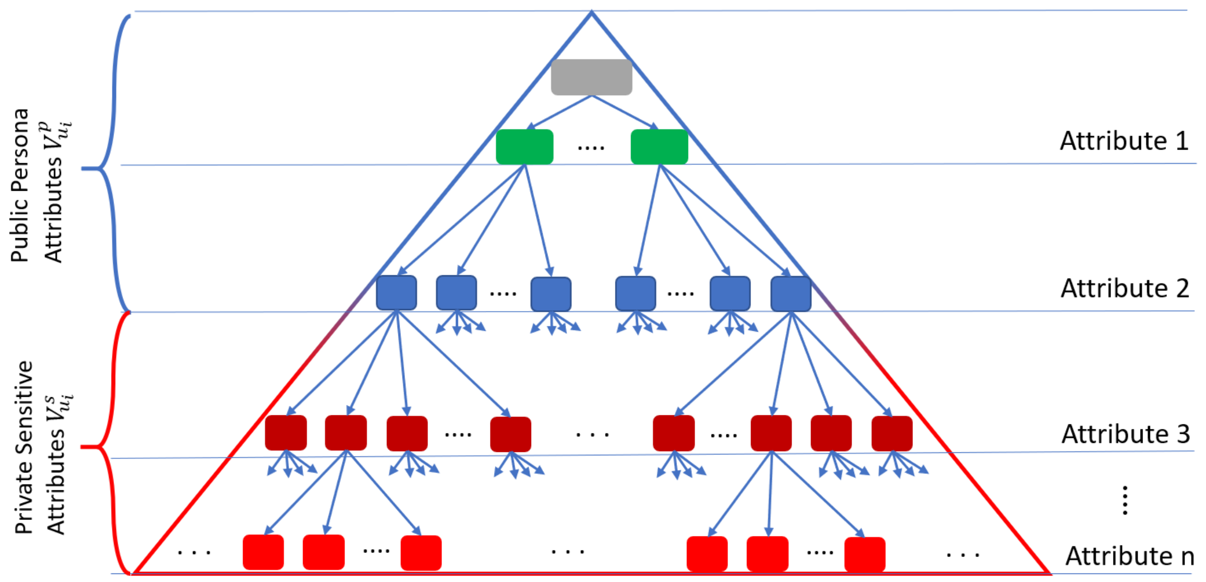

Public Persona Attributes $V_{u_i}^p$
Private Sensitive Attributes $V_{u_i}^s$
Attribute 1
Attribute 2
Attribute 3
Attribute n

Figure 4.4: A dependent topic tree where public attributes are at the top while private attributes are at the bottom

levels of the classification tree while private ones reside on the bottom levels of the tree as shown in Figure 4.4. To achieve k-attribute-indistinguishability, sibling values of the sensitive attributes are used to hide the actual value of these sensitive attributes. By placing the public attributes higher up in the hierarchy, we ensure that the suggest topics adhere to the public persona. Finally, multifaceted privacy only requires all the public attributes, regardless of their order, to reside on the upper levels of the hierarchy while all the private attributes, regardless of their order, to reside on the lower levels of the hierarchy.

Social network topics are **dependently** classified in the tree by the attribute domain values at each level. For example, if the top most level of the tree is the political party attribute, topics that are mainly discussed by Democrats are placed in the left green child while topics that are mainly discussed by Republicans are place in the right green child. Note that topics that have no inference reside in the root of the hierarchy. Now, if the second public persona attribute is ethnicity (shown as blue nodes in Figure 4.4), topics in both the Democratic and Republican nodes are classified by the ethnicity domain attribute values. For example, topics that are mainly discussed by White Democrats are placed under the Democratic node in the White ethnicity node while topics that are mainly discussed by Asian Republicans are put under the Republican node in the Asian ethnicity node. This classification is applied at every tree level for every attribute. Now, assume a user is White, Female, Democrat, who lives in CA and wants to hide her location (shown as red nodes in Figure 4.4) while publicly revealing her ethnicity, gender, and political party. In this case, topics that reside in the sibling nodes of the leaf of her persona path, e.g., topics that are mainly discussed by White Female Democrats who live in locations other than CA (e.g., NY and IL) can be suggested as obfuscation topics. This dependent classification **guarantees** that the suggested topics are aligned with the user's public persona but belong to other sensitive attribute values (different locations in this example). Note that this technique is generic enough to obfuscate any attribute and any number of attributes. Each user defines her sensitive attribute(s) and the classification hierarchy would be constructed with these attributes to the bottom thus guaranteeing that the suggested posts do not violate multifaceted privacy.

4.4 Aegis System Design

This section presents Aegis, a prototype social network stream processing system that implements multifaceted privacy and overcomes the adversarial content-based attribute inference attacks discussed in Section 4.2.3. Aegis achieves k-attribute-indistinguishability by suggesting topics to post that are aligned with the social network user's public persona while hiding the user's sensitive attributes among their other domain values. To achieve this, Aegis uses the classification and suggestion models discussed in Section 4.3. Aegis is designed in a *user-centric* manner which is configured on the user's local machine.

In fact, Aegis can be developed as a browser extension where all the user interactions with the social network are handled through this extension. Every local deployment of Aegis only needs to construct a partition of the attribute-based topic classification hierarchy developed in Figure 4.4. This partition (or sub-hierarchy) include the user's attribute path from the root to a leaf in addition to the sibling nodes of the user's sensitive attribute nodes. For example, a user whose attributes are Female, White, Democrat, and CA and wants to obfuscate her state location only requires Aegis to construct the Female, White, Democrat, CA path in addition to $k-1$ other paths with the shared prefix Female, White, Democrat but linked to $k-1$ other states. Aegis chooses these $k-1$ states with the most Female, White, Democrat user presence. This is important for Aegis to be able to find enough obfuscation topics to suggest that are aligned with the original user's public persona attributes. These $k-1$ states are used to hide the user's true state in order to achieve k-location-indistinguishability. Note that if another user's public persona is specifically associated with their location while they consider their ethnicity to be sensitive, then the hierarchy needs to be reordered to reflect this criterion.

Although Aegis can be integrated with different online social network platforms, our prototype implementation of Aegis only supports Twitter to illustrate Aegis's functionality. Twitter provides developers with several public APIs [28] that allow them to stream tweets that discuss certain topics. In addition, Twitter streaming APIs allow developers to sample 1% of all the tweets posted on Twitter. In Twitter, a topic is represented by either a *hashtag* or a *keyword*. Aegis is built to work for *new* Twitter profiles in order to continuously confuse an adversary about a profile's true sensitive attribute values from the genesis of this profile. Aegis is not designed to work with existing old profiles as an adversary could have already used their existing posts to reveal their sensitive attribute values. Even though Aegis suggests obfuscation posts, an adversary can distinguish the

old original posts from the **newly** added original posts accompanied by their obfuscation posts and hence reveals the user's sensitive attributes true values.

Aegis is designed to achieve the following goals:

1. to automate the process of streaming and classifying Twitter topics according to their attributes,
2. to construct and continuously maintain the topic classification sub-hierarchy,
3. and finally to use the topic classification sub-hierarchy to suggest topics to the user that achieve multifaceted privacy.

To achieve these goals, Aegis consists of two main processes:

- a **T**witter analyzer **P**rocess TP and
- a topic **S**uggestion **P**rocess SP.

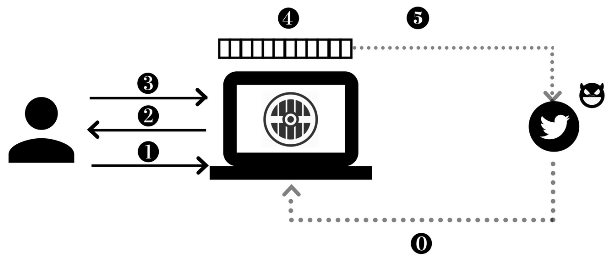


Figure 4.5: Aegis System Design and User Interaction Flow

TP continuously analyzes the topics that are being discussed on Twitter and for each topic t_i, TP uses the topic attribute inference models to infer t_i's attribute distribution vector V_{t_i}. The accuracy of the topic attribute inference increases as the number of posts that discuss topic t_j, $|P_{t_j}|$, increases. TP uses a local key-value store as a topic repository where the key is the topic id $t_j \in T$ and the value is the topic attribute distribution vector

V_{t_i}. In addition, TP constructs and continuously maintains the topic classification sub-hierarchy that classifies the topics based on their attributes. This topic classification sub-hierarchy is used for suggesting obfuscation topics. For this, the user provides TP with both their attribute vector values V_{u_i} and their sensitive attribute vector $V_{u_i}^s$. V_{u_i} and $V_{u_i}^s$ determine the sub-hierarchy that TP needs to maintain in order to obfuscate the attributes in $V_{u_i}^s$. Figure 4.5 shows the interactions among the user, Aegis, and the social network. As shown, step 0 represents the continuous Twitter stream analysis performed by TP. As TP analyzes Twitter streams, it continuously updates the topic repository and the classification sub-hierarchy.

The *topic suggestion process* SP mainly handles user interactions with Twitter. SP uses the topic classification sub-hierarchy constructed and maintained by TP to suggest obfuscation topics. For every sensitive attribute $s \in V_{u_i}^s$, the user provides the indistinguishability parameter k_s that determines how many attribute values from the domain of s should be used to hide the true value of s. Aegis allows users to configure the privacy parameter Δ_s for every attribute s. However, to enhance usability, Aegis maintains a default value for the privacy parameter $\Delta_s = 10\%$. This means that the privacy of attribute s is achieved if the inference attack cannot distinguish the maximum inferred attribute value from the k_s^{th} inferred attribute value by more than 10%.

SP uses k_s, and Δ_s for every attribute $s \in V_{u_i}^s$ to generate the topic suggestion set S_i. Note that SP is locally deployed at the user's machine. Therefore, the user does not have to trust any service outside of her machine. Aegis is designed to transfer user trust from the social network providers to the local machine. For every sensitive attribute s, SP selects a fix set of $k_s - 1$ attribute domain values. These $k_s - 1$ attribute values are used to obfuscate the true value of attribute s, $V_{u_i}[s]$.

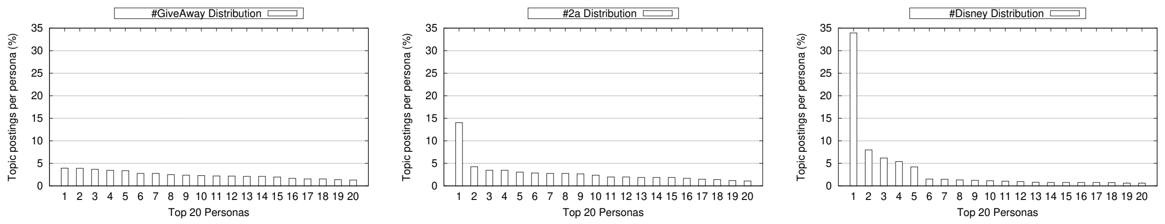
As shown in Figure 4.5, in Step 1, the user writes a post to publish on Twitter. SP receives this post and queries TP about the attributes of all the topics mentioned in this post. SP uses these topic attributes to simulate the adversarial attack. If TP's topic inference indicates that k-attribute-indistinguishability is violated for any attribute $s \in V^s_{u_i}$, SP queries TP for topics with public persona $V^p_{u_i}$ but linked to the other attribute values of s in the set of $k_s - 1$ attribute values. For every returned topic, SP ensures that writing about this topic enhances the aggregated inference of the original post and the obfuscation posts towards k_s-attribute-indistinguishability. SP adds these topics to the set S_i and returns them to the user (Step 2). The user selects a few topics from S_i to post in Step 3 and submits the posts to SP. Note that users are **required to write** the obfuscation posts using their personal writing styles to ensure that the original posts and the obfuscation posts are indistinguishable [108, 47]. Afterwards, SP ensures that the aggregated inference of submitted obfuscation posts in addition to the original post lead to k-attribute-indistinguishability. Otherwise, SP keeps suggesting more topics. As every original post along with its obfuscation posts achieve k-attribute-indistinguishability, the aggregated inference over the whole user's posts achieve k-attribute-indistinguishability. In Step 4, SP queues the original and the obfuscation posts and publishes them on the user's behalf in random order and intervals to prevent *timing attacks* (Step 5). An adversary can perform a timing attack if the original posts and the obfuscation posts are distinguishable. Queuing and randomly publishing the posts prevents the adversary from distinguishing original posts from the obfuscation posts and hence prevents timing attacks.

We understand that the obfuscation writing overhead might alienate users from Aegis. As a future extension, Aegis can exploit deep neural network language models to learn the user's writing style [89]. Aegis can use such a models to generate [109] full posts instead of hashtags and users can either directly publish these posts or edit them before publishing.

4.5 Aegis Experimental Evaluation

In this section, we experimentally evaluate the effectiveness of Aegis in achieving multifaceted privacy. We first present the experimental setup and analyze some properties of the used dataset in Section 4.5.1. Then, Sections 4.5.2 and 4.5.3 present illustrative inference and obfuscation examples that show the functionality of Aegis using real Twitter topics. We experimentally show how Aegis can be used to hide user location in Section 4.5.4 and measure the effect of changing the indistinguishability parameter k on the obfuscation overhead in Section 4.5.5. Finally, Section 4.5.6 illustrates the efficiency of Aegis on hiding the user gender while preserving other persona attributes.

4.5.1 Experimental Setup



(a) #GiveAway (Negligible Connection) (b) #2a (Weak Connection) (c) #Disney (Strong Connection)

Figure 4.6: Examples of negligible, weak, and strong connection distribution topics for top-20 personas

For our experiments, we use the 1% random sampling of the Twitter stream during the year 2017. The attributes gender, ethnicity, and location are used to build a three level topic classification hierarchy that classifies all topics according to their attribute distribution. For simplicity and without loss of generality, we use the language models in [192] to infer both gender and ethnicity attributes of a post writer. In addition, we infer the location distribution of different topics using the explicitly geo-tagged posts about these topics. In the 1% of Twitter's 2017 postings, our models were able to extract

2,126,791 unique topics. Our classification hierarchy suggests that 66% of the dataset tweets are posted by males. This analysis is consistent with the statistics published in [27]. In addition, the dataset has 6,864,300 geo-tagged posts, 15% of which originated in California. Finally, the predominant ethnicity extracted from the dataset is White.

As the classification hierarchy is built using only gender, ethnicity, and state location attributes, this results in a hierarchy of 500 different paths from the root to a leaf of the hierarchy. These 500 paths result from all the possible combinations of gender (male, female), ethnicity (White, Black, Asian, Hispanic, Native American), and state location (50 States). The 500 paths represent the different 500 personas considered in our experiments. Our topic classification hierarchy suggests that topics vary significantly in their connection to a specific persona path (a gender, ethnicity, and location combination). For example, #GiveAway is widely discussed among the 500 personas across the 50 States with very little skew towards specific personas over others. For topics that are widely discussed across all different persona, their skew is usually proportional to the population density of different States. For example, the top five highly populated states (CA, TX, FL, NY, and PA) usually appear as the top locations where widely discussed topics are posted. On the other hand, other topics show strong connection to specific personas. For example, 33.93% of the personas who write about #Disney are Male, Asian, and live in Florida where Disney World is located. Figure 4.6 shows 3 examples of topics that have trivial, weak, and strong connection to specific personas. In Figure 4.6, the x-axis represents the top-20 personas who post about a topic and the y-axis represents the percentage of postings for each persona. As shown in Figure 4.6a, #GiveAway has slight skew (negligible connection) towards some personas over other personas. Also, the top most persona who post about #GiveAway represent only 3.94% of the overall postings about the topic. On the other hand, Figure 4.6b shows that the persona distribution for #2a (refers to the second amendment) has more skew (weak connection) towards some personas over others. As shown, the top posting persona on the topic #2a contributes 14.03% of the overall postings of this topic. Finally, a topic like #Disney has remarkable skew (strong connection) towards some personas over others. As shown in Figure 4.6c, the top posting persona on the topic #Disney contributes 33.93% of the overall postings of this topic.

Strength	**Minimum δ**	**Maximum δ**
Negligible	0%	10%
Weak	10%	20%
Mild	20%	30%
Strong	30%	100%

Table 4.1: Topic to persona connection strength categories and their corresponding δ ranges

Topic	**Freq**	**M W CA**	**F W CA**	**M W TX**	**F W TX**
#teen	7094	7	**1**	15	4
#hot	7478	5	**1**	13	3
#etsy	2739	6	5	27	**1**
#diy	1987	3	7	11	**1**
#actor	725	**1**	2	9	11
#cowboys	797	5	16	**1**	2

Table 4.2: Topic Analysis By Persona

We define a *topic to persona connection strength* parameter δ. δ is defined as the difference in posting percentage between the top-1 posting persona and the top-k posting persona. For #2a and k = 3, $\delta = 14.03 - 3.45 = 10.6$ while for #Disney and for k=3, $\delta = 33.93 - 6.19 = 27.74$. We categorize topics into 4 categories according to their δ value. As shown in Table 4.1, a topic to persona connection that ranges from 0% to 10% represents a *negligible connection* and hence this topic does not reveal the persona attributes of the users who write about it. As the topic to persona connection increases, the potential of revealing the attributes of the users who write about this topic increases. In our experiments, we measure the overhead of obfuscation for three different distributions with *weak, mild* and *strong* connections having a topic to persona strength connection ranges that are shown in Table 4.1

4.5.2 Illustrative Inference Example

Using our hierarchical data structure we can infer interesting information about different topics on Twitter in our dataset, specifically regarding correlations between persona and topics. In Table 4.2 we analyze 7 topics and their connection to 4 of the most prominent personas in our dataset (Male-White-CA, Female-White-CA, Male-White-TX, Female-White-TX). *Frequency* denotes the number of times the topic was observed and the number under a persona for a particular topic denotes the order or rank in which a persona discusses this topic most. For example, among all the persona we analyze in our dataset, #actor is most discussed by White Male Californians (rank 1) closely followed by White Female also from California (rank 2). This is followed by other persona out of the focus of Table 4.2, until White Female Texans are reached at rank 9 and White Male Texans at rank 11. Also, Table 4.2 shows that both #teen and #hot are discussed the most by Female White Californians and that overall Females (in both CA and TX) who discuss this topic are more than Males in both CA and TX. We can also observe correlations across topics that have semantic connections like #etsy and #diy. Etsy is an online platform for users to sell DIY (Do It Yourself) projects. Female White Texans are much more interested in such DIY specific topics than any other of the personas. Lastly, high correlations of certain topics can be observed with specific locations such as #actor with California and #cowboys with Texas. As Table 4.2 reveals, the topics you post on social media significantly reveal your attributes, even private sensitive attributes you are unwilling to share. As such, we need a tool like Aegis to prevent adversaries from inferring private attributes while preserving others.

4.5.3 Illustrative Obfuscation Example

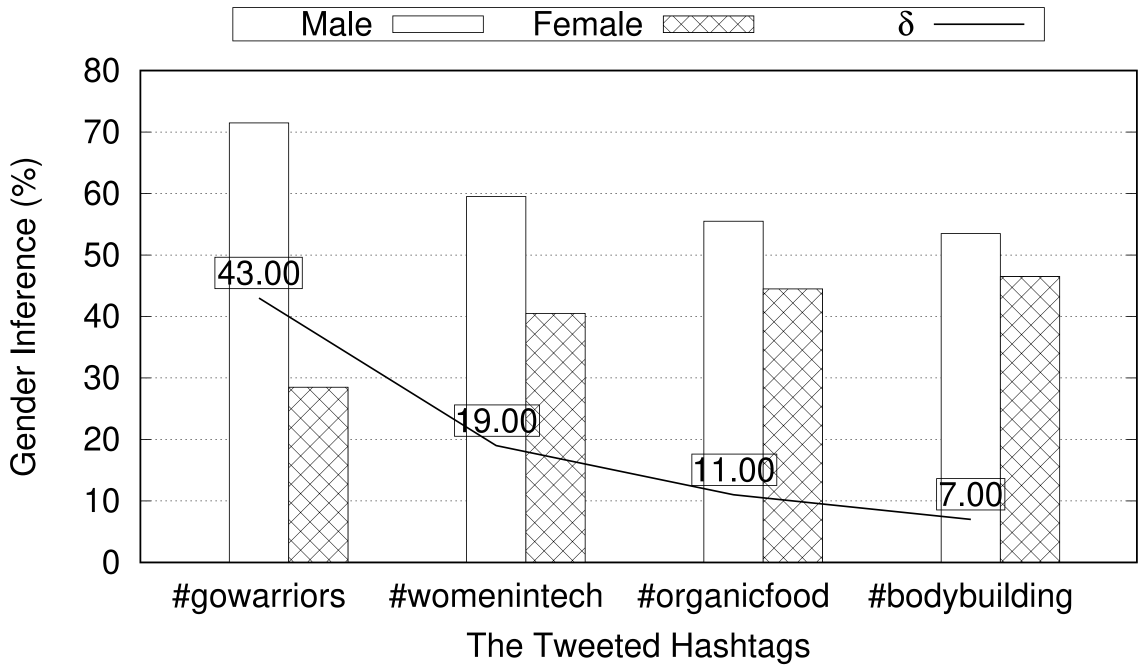


Figure 4.7: Illustrative Gender Obfuscation Example.

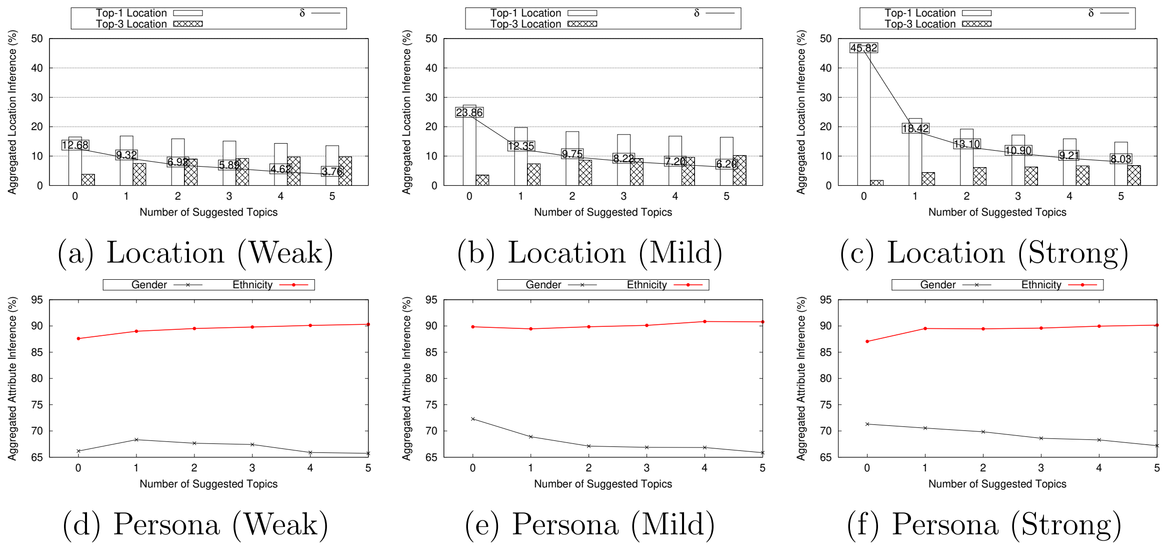


Figure 4.8: Effect of obfuscation posts on location and user persona given weak, mild and strong connected topics to locations

In this section, we provide an illustrative obfuscation example that shows how Aegis achieves multifaceted privacy. This example begins with a newly created Twitter profile of a Male, White user who lives in California. The user wants to obfuscate his gender among the gender domain values {male, female} achieving 2-gender-indistinguishably. For this, Aegis constructs a hierarchy with ethnicity at the top level, then location at the second level, and finally gender (the sensitive attribute) at the bottom level of the hirarchy. Indistinguishability holds if the privacy parameter $\Delta_g = 0.1$ is achieved. Δ_g is set to 0.1 (or 10%) to ensure that users who write topics with only negligible topic to persona strength connection do not have to add any obfuscation posts to their timelines as the topics they post do not reveal their sensitive attributes. In addition, the user wants to preserve his ethnicity and location attributes as his public persona. Now, assume that the user tweets #gowarriors to show his support for his favorite Californian basketball team, the Golden State Warriors.

Unfortunately, as shown in Figure 4.7, #gowarriors has a strong connection to the male gender attribute value. In Figure 4.7, the x-axis represents the posted hashtags one after the other and the y-axis represents the *aggregated* gender inferences for both male and female attribute values over all the posted hashtags. In addition, δ represents the difference of the aggregated gender inference between male and female attribute values. As shown, initially, $\delta = 43\%$ which indicates a strong link between the user's gender and the male attribute value. As $\delta \geq \Delta_g$, this indicates that 2-gender-indistinguishably is violated.

Therefore, Aegis suggests to post topics that are mainly discussed by White people who live in California but linked to the female gender attribute value. Figure 4.7 shows the effect of posting subsequent topics on the aggregated gender inference. The topic #womenintech helps to reduce the aggregate inference difference to 19%. #organicfood brings the difference down to 11% and #bodybuilding reduces it to 7%. Notice that $\delta = 7\%$ achieves $\delta \leq \Delta_g$ and hence 2-gender-indistinguishably is achieved. Notice that the same example holds if the user's true gender is female. The goal of Aegis is not to invert the gender attribute value but to achieve inference indistinguishability among $k_g = 2$ different gender attribute values.

4.5.4 User Location Obfuscation

The number of obfuscation topics largely depend on the topic to persona *connection strength* of the original posts. In this experiment, we show how Aegis is used to hide user location while preserving their gender and ethnicity. To hide location, Aegis constructs a hierarchy with gender at the top level, ethnicity at the second level, and location (the sensitive attribute) at the bottom level. In this experiment, we set $k_l = 3$ where user location is hidden among three locations. Δ_l is set to 0.1 to indicate that 3-location-indistinguishability is achieved if the difference between the highest (top-1) aggregated location inference and the 3^{rd} (top-3) aggregated location inference is less than 10%. The number of obfuscation posts needed and their effect on the user persona are reported. This experiment uses 4707 weak topics, 1984 mild topics, and 1106 strong topics collected from Twitter over several weeks. This experiment assumes a newly created twitter profile simulated with one of the 4 most prominent personas in our data set, namely M W CA, F W CA, M W TX, and F W TX as presented in Table 4.2. First, a post with a topic to location connection strength (weak, mild, or strong) is added to the user profile. Then, we add the suggested obfuscation posts one at a time to the user's timeline. After every added obfuscation post, the location and persona inference are reported. The reported numbers are aggregated and averaged for every topic to location connection strength category. Figure 4.8 shows the effect of adding obfuscation posts on both the location inference and the persona inference for weak, mild, and strong topics. As shown in Figure 4.8a, the 4707 weak topics on average need just *one* obfuscation suggestion to achieve

3-location-indistinguishability when $\Delta_l = 0.1$. Note that adding more obfuscation posts achieves 3-location-indistinguishability for smaller Δ_ls (e.g., $\Delta_l = 0.05$ (4 suggestions), $\Delta_l = 0.04$ (5 suggestions)). Also, adding one suggestion post achieves 26.5% reduction in δ. The same experiment is repeated for mild and strong topics and the results are reported in Figures 4.8b and 4.8c respectively. Notice that strong topics requires *four* suggestions on average to achieve 3-location-indistinguishability when $\Delta = 0.1$. Also, in Figures 4.8c, adding a single suggestion post achieves 49.8% reduction in δ.

Figures 4.8d, 4.8e, and 4.8f show the effect on user persona after adding obfuscation posts to weak, mild, and strong posts respectively. User persona is represented by gender and ethnicity. As obfuscation posts are carefully chosen to align with the user persona, we observe negligible changes on the average gender and ethnicity inferences after adding obfuscation posts.

4.5.5 The Effect of Changing k_l

In this experiment, we measure the effect of changing the parameter k_l on the number of obfuscation posts required to achieve k-location-indistinguishability. k_l determines the number of locations within which user u_i wants to hide her true location. Increasing k_l increases the achieved privacy and boosts the required obfuscation overhead to achieve k-location-indistinguishability. Assume users in Texas hide their State level location among 3 States: Texas, Alabama, and Arizona. A malicious advertiser who wants to target users in Texas is uncertain about their location and now has to pay 3 times the cost of the original advertisement campaign to reach the same target audience. Therefore, increasing k inflates the cost of micro-targeting.

This experiment uses 621 strong topics collected from Twitter over several weeks. The average $\delta = V_{u_i}[l]_1 - V_{u_i}[l]_{k_l}$ is reported for all topics. In addition, the effect of

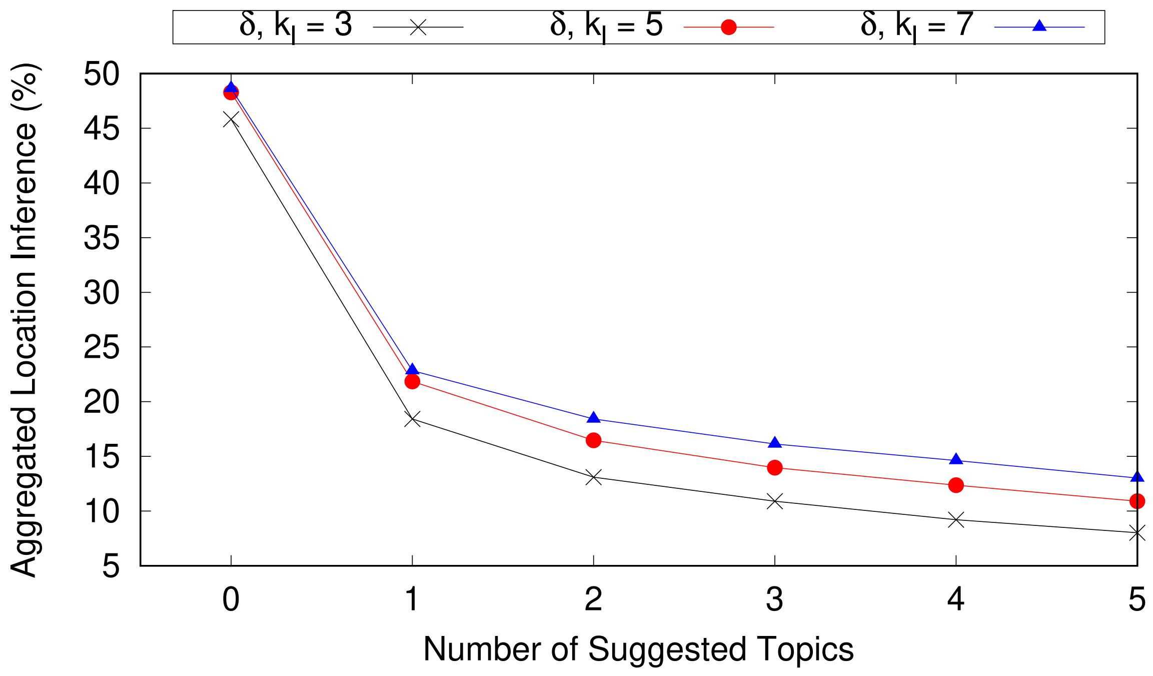


Figure 4.9: Change in δ as obfuscation posts are added

adding obfuscation posts on the average δ for different values of $k_l = 3, 5,$ and 7 is reported. Figure 4.9 shows the effect of adding obfuscation posts on the aggregated location inference for different values of k_l. The privacy parameter Δ_l is set to $\Delta_l = 0.1$.

As shown in Figure 4.9, achieving 3-location indistinguishability for strong topics requires 4 obfuscation posts on the average for $\Delta_l = 0.1$. On the other hand, 7-location indistinguishability requires more than 5 obfuscation posts for the same value of Δ_l. This result highlights the trade-off between privacy and obfuscation overhead. Achieving higher privacy levels by increasing k_l or lowering Δ_l requires more obfuscation posts and hence more overhead. Obfuscation posts are carefully chosen to align with the user persona. Therefore, we observe negligible changes on the average gender and ethnicity inferences after adding obfuscation posts for different values of k_l.

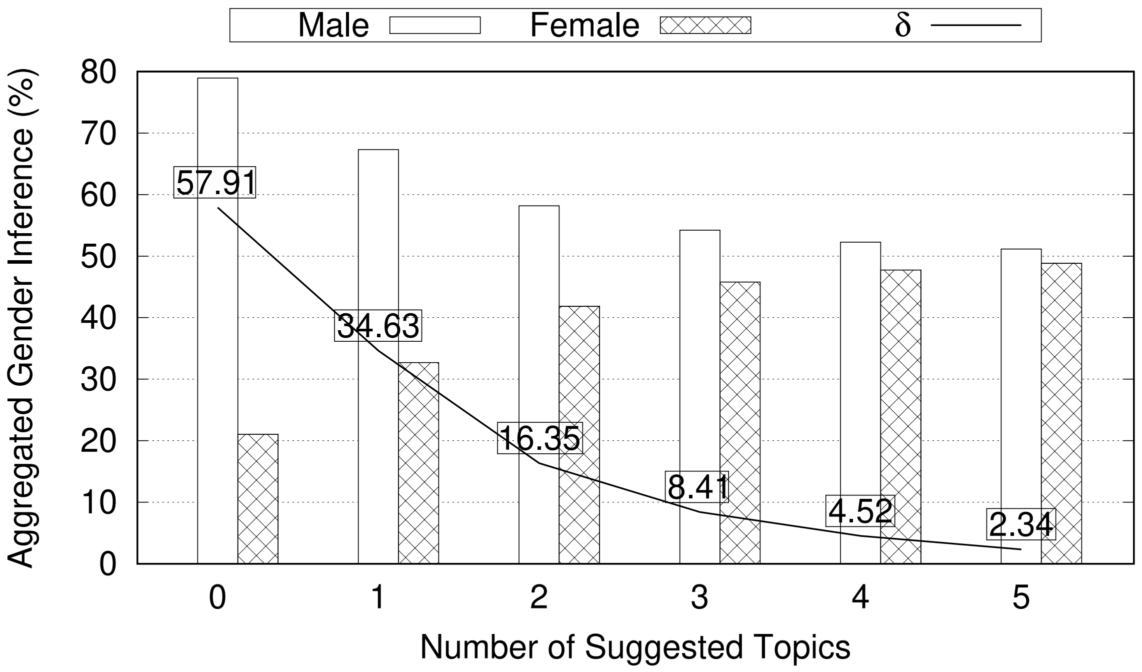


Figure 4.10: The effect of obfuscation posts on gender inference for strong connected topics to gender.

4.5.6 User Gender Obfuscation

This experiment shows how Aegis is used to hide user gender while preserving their ethnicity and location. In this experiment, we set $k_g = 2$ where user gender should be hidden among male and female gender domain values. Δ_g is set to $\Delta_g = 0.1$ to indicate that 2-gender-indistinguishability is achieved if the difference between the highest (top-1) aggregated gender inference and the 2^{nd} (top-2) aggregated gender inference is less than 10%. This experiment was executed on 40 gender strongly connected topics. The aggregated gender inference is reported when adding the original strong post to the user's timeline and after adding every obfuscation post one at a time. In addition, δ, the difference between the male gender inference and the female gender inference is reported. 2-gender-indistinguishability is achieved if $\delta \leq 10\%$. As shown in Figure 4.10, gender strongly connected topics result in high δ that violates the 2-gender-indistinguishability privacy target. Therefore, Aegis suggests obfuscation posts that results in δ reduction. Figure 4.10 shows that strong topics need on the average 3 obfuscation posts to achieve the gender privacy target. This result is quite consistent with the location obfuscation experiments in Section 4.5.4. This shows Aegis's obfuscation mechanism is quite generic and can be efficiently used to hide different user sensitive attributes. Finally, as the obfuscation posts are carefully chosen to align with the user's public persona attributes, we observed negligible changes on the average location and ethnicity inferences after adding obfuscation posts.

4.6 Related Work

The problem of sensitive attribute privacy of social network users has been extensively studied in the literature from different angles. *k-anonymity* [190, 198, 197] and its successors *l-diversity* [153] and *t-closeness* [143] are well-known and widely used privacy models in publishing dataset to hide user information among a set of indistinguishable users in the dataset. Also, differential privacy [90, 91, 92] has been widely used in the context of dataset publishing to hide the identity of a user in a published dataset. These models focus on hiding user *identity* among other users in the published dataset. Another variation of differential privacy is *pan-privacy* [93]. Pan-privacy is designed to work for data streams and hence it is more suitable for social network streams privacy. However, these models are service centric and assume trusted service providers. This work tackles the privacy problem from the end-user angle where the user identity is known and all their online social network postings are public and connected to their identity. Our goal is to confuse *content-based sensitive attribute inference* attacks by hiding the user's original public posts among other obfuscation posts. Our multifaceted privacy achieves the privacy of user sensitive attributes without altering their public persona.

In the context of social networks privacy, earlier works focus on sensitive attribute inferences due to the structure of social networks. In [225], Zhelava and Getoor attempt to infer the user's sensitive attributes using public and private user profiles. However, the authors do not provide a solution to prevent such inference attacks. Georgiou et al. [101, 100] study the inference of sensitive attributes in the presence of community-aware trending topic reports. An attacker can increase their inference confidence by consuming these reports and the corresponding community characteristics of the involved users. In [101], a mechanism is proposed to prevent social network services from publishing trending topics that reveal information about individual users. However, this mechanism is service centric and it is not suitable for hiding a user's sensitive attributes against content-based inference attacks. Ahmad et al.[45] introduce a client-centered obfuscation solution for protecting user privacy in personalized web searches. The privacy of a search query is achieved by hiding it among other obfuscation search queries. Although this work is client-centered, it is not suitable for social networks privacy where the user online persona has to be preserved.

Recent works have focused on the privacy of some sensitive attributes such as location of social network users. Ghufran et al. [104] show that social graph analysis can reveal user location from friends and followers locations. Although, it is important to protect user sensitive attributes like location against this attack, Aegis focuses only on content-based inference attacks. Yakout et al. [210] proposed a system called Privometer, which measures how much privacy leaks from certain user actions (or from their friends' actions) and creates a set of suggestions that could reduce the risk of a sensitive attribute being successfully inferred. Similar to Privometer, [116] proposes sanitation techniques to the structure of the social graph by introducing noise, and obfuscating edges in the social graphs to prevent sensitive information inference. Andres et al. [48] propose geo-indistinguishability, a location privacy model that uses differential privacy to hide a user's exact location in a circle of radius r from locaction based service providers. In a recent work, Zhang et al. [223] introduce *Tagvisor*, a system to protect users against content-based inference attacks. However, Tagvisor requires users to alter their posts by changing or replacing hashtags that reveal their location. Other works [107, 60] depend on user collaboration to hide an individual's exact location among the location of the collaboration group. This approach requires group members to collaborate and synchronously change their identities to confuse adversaries. However, these techniques are prone to content-based inference attacks and collaboration between users might be hard to achieve in the social network context. For location based services, Mokbel et al. [163] use location generalization and k-anonymity to hide the exact location of a query. These works do not preserve the user online persona while achieving location privacy. In addition, these works do not provide a generic mechanism to hide other sensitive attributes such as user gender and ethnicity.

Viejo et al. [204] propose a similar solution to Aegis that automatically generates fake posts in order to alter the attribute distribution of a user's sensitive attributes to a uniform distribution. This solution differs from Aegis in many folds. First, this solution only targets automatic profiling systems and assumes that a user's human followers are not adversaries who might also try to infer a user's actual sensitive attribute values. Therefore, the generated fake posts are meaningless to human followers but confuses automatic profiling systems. Although, meaningless posts are easy to generate, they might hurt the utility of a user's profile as followers might unfollow accounts that post random meaningless posts. Finally, fake posts are not generated in a way that preserves a user's public personality.

This chapter presents Aegis, the first persona friendly system that enables users to hide their sensitive attributes while preserving their online persona. Aegis is a client-centric solution that can be used to hide any user specified sensitive attribute. Aegis does not require users to alter their original posts or topics. Instead, Aegis hides the user's original posts among other obfuscation posts that are aligned with their persona but linked to other sensitive attribute values achieving k-attribute-indistinguishability.

4.7 Aegis Conclusion

This work proposes *multifaceted privacy*, a novel privacy model that obfuscates a user's sensitive attributes while publicly revealing their public online persona. To achieve the multifaceted privacy, we build *Aegis*, a prototype client-centric social network stream processing system that achieves multifaceted privacy. Aegis is user-centric and allows social network users to control which persona attributes should be publicly revealed and which should be kept private. Aegis is designed to transfer user trust from the social network providers to her local machine. For this, Aegis continuously suggests *topics* and *hashtags* to social network users to post in order to obfuscate their sensitive attributes and hence confuse content-based sensitive attribute inferences. The suggested topics are carefully chosen to preserve the user's publicly revealed persona attributes while hiding their private sensitive persona attributes. Our experiments show that Aegis is able to achieve sensitive attributes privacy such as location and gender. Adding as few as 0 to 4 obfuscation posts (depending on how strongly connected the original post is to a persona) successfully hides the user specified sensitive attributes without altering the user's public persona attributes.

4.8 Aegis Future Extensions

Research in the social network privacy has focused on dataset publishing and obfuscating user information among other users. Such focus is *service centric* and assumes that service providers are trusted. However, Aegis is *user centric* and aims to give the users control over their own privacy. This control comes with a cost represented by the obfuscation posts that need to be posted by users. The role of Aegis is to automate the topic suggestion process and to ensure that k-attribute-indistinguishability holds against content-base inference attacks. There are several directions where these obfuscation and privacy models can evolve.

Obfuscation Post Generation. Aegis suggests topics as keywords or hashtags and requires users to write the obfuscation posts using their personal writing styles to ensure that the original posts and the obfuscation posts are indistinguishable. However, the obfuscation writing overhead might alienate users from Aegis. Instead, Aegis can exploit deep neural network language models to learn the user's writing style. Aegis can use such a model to suggest full posts instead of hashtags and users can either directly publish these posts or edit them before publishing. This extension aims to reduce the overhead on the users by automating the obfuscation post generation.

Social Graph Attack Prevention. Aegis is mainly designed to obfuscate the user sensitive attributes against content-based inference attacks. An orthogonal attack is to use the attribute values of friends and followers to infer a user's real sensitive attribute value. Take location as a sensitive attribute example. Ghufran et al., [104] show that user location can be inferred from the locations of followers and friends. A user whose friends are mostly from NYC is highly probable to be from NYC. Aegis can be extended to prevent this attack. Users of similar persona but different locations can create an indistinguishability network where users in this network have followers and friends from different locations. Similar to the obfuscation topics, Aegis could suggest users to follow with similar persona but different locations.

Part II

Robustness and Extending the Functionality of Permissionless Blockchains

Chapter 5

Atomic Commitment Across Blockchains

5.1 Overview

The wide adoption of permissionless open blockchain networks by both industry (e.g., Bitcoin [164], Ethereum [205], etc) and academia (e.g., Bzycoin [131], Elastico [152], BitcoinNG [94], Algorand [160], etc) suggests the importance of developing protocols and infrastructures that support peer-to-peer atomic cross-chain transactions. Users, who usually do not trust each other, should be able to directly exchange their tokens and assets that are stored on different blockchains (e.g., Bitcoin and Ethereum) without depending on trusted third party intermidiaries. Decentralized permissionless [155] blockchain ecosystems require infrastructure enablers and protocols that allow users to atomically exchange tokens without giving up trust-free decentralization, the main reason behind using permissionless blockchain. We motivate the problem of atomic cross-chain transactions and discuss the current available solutions and their limitations through the following example.

Suppose Alice owns X bitcoins and she wants to exchange them for Y ethers. Luckily, Bob owns ether and he is willing to exchange his Y ethers for X bitcoins. In this example, Alice and Bob want to atomically exchange assets that reside in different blockchains. In addition, both Alice and Bob **do not trust** each other and in many scenarios, they might not be co-located to do this atomic exchange in person. Current infrastructures do not support these direct peer-to-peer transactions. Instead, both Alice and Bob need to **independently** exchange their tokens through a trusted centralized exchange, Trent (e.g., Coinbase [26] and Robinhood [36]) either through fiat currency or directly. Using Fiat, both Alice and Bob first exchange their tokens with Trent for a fiat currency (e.g., USD) and then use the earned fiat currency to buy the other token also from Trent or from another trusted exchange. Alternatively, some exchanges (e.g., Coinbase) allow their customers to directly exchange tokens (ether for bitcoin or bitcoin for ether) without going through fiat currencies.

These solutions have many drawbacks that make them unacceptable solutions for atomic peer-to-peer cross-chain transactions. *First*, both solutions require both Alice and Bob to trust Trent. This centralized trust requirement risks to derail the whole idea of blockchain's trust-free decentralization [164]. *Second*, both solutions require Trent to trade in all involved resources (e.g., bitcoin and ether). This requirement is unrealistic especially if Alice and Bob want to exchange commodity resources (e.g., transfer a car ownership for bitcoin assuming car titles are stored in a blockchain [117]). *Third*, both solutions do not achieve atomicity of the transaction among the involved participants. Alice might trade her bitcoin directly for ether or through a fiat currency while Bob has no obligation to execute his part of the swap. Finally, both solutions significantly increase the number of required transactions to achieve the intended cross-chain transaction, and hence drastically increases the imposed fees. One cross-chain transaction between Alice and Bob results in either four transactions (two between Alice and Trent and two between

Bob and Trent) if fiat is used or at best two transactions (one between Alice and Trent and one between Bob and Trent) if assets are directly swapped.

An **A**tomic **C**ross-**C**hain **T**ransaction , AC^{2}T, is a distributed transaction that spans multiple blockchains. This distributed transaction consists of sub-transactions and each sub-transaction is executed on some blockchain. An **A**tomic **C**ross-**C**hain **C**ommitment, AC^3, protocol is required to execute AC^{2}Ts. This protocol is a variation of traditional distributed atomic commitment protocols (e.g., 2PC [112, 57]). This protocol should guarantee both *atomicity* and *commitment* of AC^{2}Ts. **Atomicity** ensures the **all-or-nothing** property where either all sub-transactions take place or none of them is executed. **Commitment** guarantees that any changes caused by a cross-chain transaction must eventually take place if the transaction is decided to commit. Unlike in 2PC and other traditional distributed atomic commitment protocols, atomic cross-chain commitment protocols are also trust-free and therefore must **tolerate** maliciousness [117].

A two-party atomic cross-chain commitment protocol was originally proposed by Nolan [169, 21] and generalized by Herlihy [117] to process multi-party atomic cross-chain transactions, or swaps. Both Nolan's protocol and its generalization by Herlihy use smart contracts, hashlocks, and timelocks to execute atomic cross-chain transactions. A smart contract is a self executing contract (or a program) that gets executed in a blockchain once all the terms of the contract are satisfied. A hashlock is a cryptographic one-way hash function $h = H(s)$ that locks assets in a smart contract until a hash secret s is provided. A timelock is a time bounded lock that triggers the execution of a smart contract function after a pre-specified time period.

The atomic swap between Alice and Bob, explained in the earlier example, is executed using Nolan's protocol as follows. Let a participant be the leader of the swap, say Alice. Alice creates a secret s, only known to Alice, and a hashlock $h = H(s)$. Alice uses h to lock X bitcoins in a smart contract SC_1 and publishes SC_1 in the Bitcoin network.

SC_1 states to transfer X bitcoins to Bob if Bob provides the secret s to SC_1 such that $h = H(s)$. In addition, SC_1 is locked with a timelock t_1 that refunds the X bitcoins to Alice if Bob fails to provide s to SC_1 before t_1 expires. As SC_1 is published in the Bitcoin network and made public to everyone, Bob can verify that SC_1 indeed transfers X bitcoins to the public address of him if he provides s to SC_1. In addition, Bob learns h from SC_1. Using h, Bob publishes a smart contract SC_2 in the Ethereum network that locks Y ethers in SC_2 using h. SC_2 states to transfer Y ethers to Alice if Alice provides the secret s to SC_2. In addition, SC_2 is locked with a timelock $t_2 < t_1$ that refunds the Y ethers to Bob if Alice fails to provide s to SC_2 before t_2 expires.

Now, if Alice wants to redeem her Y ethers from SC_2, Alice must reveal s to SC_2 before t_2 expires. Once s is provided to SC_2, Alice redeems the Y ethers and s gets revealed to Bob. Now, Bob can use s to redeem his X bitcoins from SC_1 before t_1 expires. Notice that $t_1 > t_2$ is a necessary condition to ensure that Bob has enough time to redeem his X bitcoins from SC_1 after Alice provides s to SC_2 and before t_1 expires. If Bob provides s to SC_1 before t_1 expires, Bob successfully redeems his X bitcoins and the atomic swap is marked completed.

The case against the current proposals: If Bob fails to provide s to SC_1 before t_1 expires due to a crash failure or a network denial of service attack at Bob's site, Bob loses his X bitcoins and SC_1 refunds the X bitcoins to Alice. This violation of the atomicity property of the protocol penalizes Bob for a failure that happens out of his control. Although a crashed participant is the only participant who ends up being worse off (Bob in this example), this protocol does not guarantee the atomicity of AC^{2}Ts in asynchronous environments where crash failures, network delays, and network denial of service attacks are the norm.

Another important drawback in Nolan's and Herlihy's protocols is the requirement to sequentially deploy the smart contracts in an atomic swap before the leader (Alice in our example) reveals the secret s. This requirement is necessary to ensure that the deployment events of all the smart contracts in the atomic swap *happen before* the redemption of any of the smart contracts. This causality requirement ensures that any malicious participant who declines to deploy her payment smart contract cannot take advantage of the protocol. However, the sequential publishing of smart contracts, especially in atomic swaps that include many participants, proportionally increases the latency of the swap to the number of sequentially published contracts.

This chapter proposes **AC^{3}WN**, the first decentralized all-or-nothing **A**tomic **C**ross-**C**hain **C**ommitment protocol that uses an open **W**itness **N**etwork. The redemption and the refund events of smart contracts in AC^{2}T are modeled as conflicting events. A decentralized open network of witnesses is used to guarantee that conflicting events must never simultaneously take place and either all smart contracts in an AC^{2}T are redeemed or all of them are refunded. Recent and concurrent work by Herlihy *et al.* [119] proposes the CBC protocol, a protocol that uses an additional blockchain to coordinate cross-chain deals. Although, this additional network performs a similar role to the witness network in AC^{3}WN, the CBC protocol focuses on cross-chain deals that do not require the all-or-nothing atomicity property. Unlike in Nolan's and Herlihy's protocols [169, 117], AC^{3}WN allows all participants to concurrently publish their contracts in a swap resulting in a drastic decrease in an atomic swap's latency. Our contribution is summarized as follows:

- We present AC^{3}WN, the first all-or-nothing atomic cross-chain commitment protocol. AC^{3}WN is decentralized and does not require to trust any centralized intermediary.
- We prove the correctness of AC^{3}WN showing that AC^{3}WN achieves both atomicity and commitment of AC^{2}Ts.
- Finally, we analytically evaluate AC^{3}WN in comparison to Herlihy's protocol in [117].

Unlike in Herlihy's protocol where the latency of an atomic swap proportionally increases as the number of the sequentially published smart contracts in the atomic swap increases, our analysis shows that the latency of an atomic swap in AC^{3}WN is constant irrespective of the number of smart contracts involved.

The rest of the chapter is organized as follows. In Section 5.2, we discuss the open blockchain data and transactional models. Section 5.3 explains the cross-chain distributed transaction model and Section 5.4 presents our atomic cross-chain commitment protocol. An analysis of the atomic cross-chain commitment protocol is presented in Section 5.5. The protocol is evaluated in Section 5.6 and the chapter is concluded in Section 5.7.

5.2 Open Blockchain Models

5.2.1 Architecture Overview

An open permissionless blockchain system [155] (e.g., Bitcoin and Ethereum) typically consists of two layers: a storage layer and an application layer. **The storage layer** comprises a decentralized distributed ledger managed by an open network of computing nodes. A blockchain system is permissionless if computing nodes can join or leave the network of its storage layer at any moment without obtaining a permission from a centralized authority. Each computing node, also called a miner, maintains a copy of the ledger. The ledger is a tamper-proof chain of blocks, hence named blockchain. Each block contains a set of valid transactions that transfer assets among end-users. **The application layer** comprises end-users who communicate with the storage layer via *message passing* through a client library. End-users have identities, defined by their public keys, and signatures, generated using their private keys. Digital signatures are the end-users' way to generate transactions as explained later in Section 5.2.3. End-users submit their transactions to the storage layer through a client library. Transactions are used to transfer assets from one identity to another. End-users multicast their transaction messages to mining nodes in the storage layer.

A mining node validates the transactions it receives and valid transactions are added to the current block of a mining node. Miners run a consensus protocol through mining to agree on the next block to be added to the chain. A miner who mines a block gets the right to add its block to the chain and multicasts it to other miners. To make progress, miners accept the first received mined block after verifying it and start mining the next block[1]. Sections 5.2.2 and 5.2.3 explain the data model and the transactional model of open blockchain systems respectively.

5.2.2 Data Model

The storage layer stores the ownership information of assets in the system in the blockchain. The ownership is determined through identities and identities are typically implemented using public keys. In addition, the blockchain stores transactions that transfer the ownership of an asset from one identity to another. Therefore, an asset can be tracked from its registration in the blockchain, the first owner, to its last owner in the blockchain. For example, the Bitcoin blockchain stores the information of the most recent owner of every bitcoin in the Bitcoin blockchain. A bitcoin that is linked to Alice's public key is owned by Alice. Also, new bitcoins are generated and registered in the Bitcoin blockchain through mining. Asset ownership transfers are implemented through transactions.

[1]Forks and fork resolutions are discussed in later Sections.

5.2.3 Transaction Model

A transaction is a digital signature that transfers the ownership of assets from one identity to another. End-users, in the application layer, use their private keys [186] to digitally sign assets linked to their identity to transfer these assets to other identities, identified by their public keys. These digital signatures are submitted to the storage layer via message passing through a client library. It is the responsibility of the miners to validate that end-users can transact only on their own assets. If an end-user digitally signs an asset that is not owned by this end-user, the resulting transaction is not valid and is rejected by the miners. In addition, miners validate that an asset cannot be spent twice and hence prevent double spending of assets.

Another way to perform transactions in blockchain systems is through **smart contracts**. A smart contract is a program written in some scripting language (e.g., Solidity for Ethereum smart contracts [38]) that allows general program executions on a blockchain's mining nodes. End-users publish a smart contract in a blockchain through a deployment message, msg, that is sent to the mining nodes in the storage layer. The deployment message includes the smart contract code in addition to some implicit parameters that are accessible to the smart contract code once the smart contract is deployed. These parameters include the sender public key, accessed through $msg.sender$, and an optional asset value, accessed through $msg.val$. This optional asset value allows end-users to send some of their blockchain assets to a deployed smart contract. Like transactions, a smart contract is published in a blockchain if it is included in a mined block in this blockchain. We adopt Herlihy's notion of a smart contract as an object in programming languages [118, 86]. A smart contract has a state, a constructor that is called when a smart contract is first deployed in the blockchain, and a set of functions that could alter the state of the smart contract. The constructor initializes the smart contract's state and uses the implicit parameters to initialize the owner of the smart contract and the assets value sent to this smart contract. Miners verify that the end-user who deploys a smart contract indeed owns these assets. Once assets are sent to a smart contract, the ownership of these assets is moved to the smart contract itself. Smart contract assets can only be transacted on within the smart contract logic until these assets are unlocked from the smart contract as a result of a smart contract function call. To execute a smart contract function, end-users submit their function call accompanied by the function parameters through messages to miners. These messages could include implicit parameters as well (e.g., *msg.sender*). Miners execute[2] the function on the current state of the contract and record any contract state changes in their current block in the blockchain. Therefore, a smart contract object state might span many blocks after the block where the smart contract is first deployed.

5.3 Atomic Cross-Chain Transaction Model

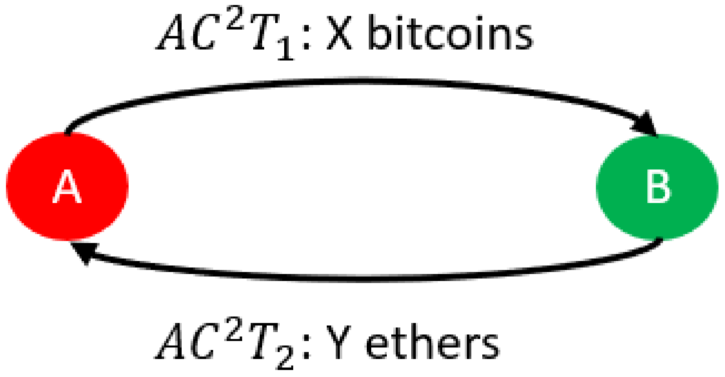


Figure 5.1: An atomic cross-chain transaction graph to swap X bitcoins for Y ethers between Alice (A) and Bob (B).

An Atomic Cross-Chain Transaction, AC^2T, is a distributed transaction to trans-

[2]End-users pay to miners a smart contract deployment fee plus a function invocation fee for every function call.

fer the ownership of assets stored in multiple blockchains among two or more participants. This distributed transaction consists of sub-transactions and each sub-transaction transfers an asset on some blockchain. An AC^{2}T is modeled using a directed graph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ [117] where $\mathcal{V}$ is the set of vertexes and $\mathcal{E}$ is the set of edges in $\mathcal{D}$. $\mathcal{V}$ represents the participants in AC^{2}T and $\mathcal{E}$ represents the sub-transactions in AC^{2}T. A directed edge $e = (u, v) \in \mathcal{E}$ represents a sub-transaction that transfers an asset $e.a$ from a source participant $u \in \mathcal{V}$ to a recipient participant $v \in \mathcal{V}$ in some blockchain $e.BC$. Figure 5.1 shows an example of an AC^{2}T graph between Alice (A) and Bob (B). As shown, the edge (A, B) represents the sub-transaction AC^2T_1 that transfers X bitcoins from A to B while the edge (B, A) represents the sub-transaction AC^2T_2 that transfers Y ethers from B to A.

An atomic cross-chain commitment protocol is required in order to correctly execute an AC^{2}T. This protocol must ensure the atomicity and the commitment of all sub-transactions in AC^{2}T as follows.

- Atomicity: either all asset transfers of all sub-transactions in the AC^{2}T take place or none of them do.
- Commitment: once the atomic cross-chain commitment protocol decides the commitment of an AC^{2}T, all asset transfers of all sub-transactions in this AC^{2}T must eventually take place.

An atomic cross-chain commitment protocol is a variation of the two phase commit protocol (2PC) [112, 57]. Therefore, we use the analogy of 2PC to explain an abstraction of an atomic cross-chain commitment protocols. In 2PC, a distributed transaction spans multiple data partitions and each partition is responsible for executing a sub-transaction. A coordinator sends a vote request to all involved data partitions. Upon receiving a vote request, a data partition votes back *yes* only if it succeeds in executing all the operations of its sub-transaction on the involved data objects. Otherwise, a data partition votes *no* to the coordinator. A coordinator decides to commit a distributed transaction if all involved data partitions vote yes, otherwise it decides to abort the distributed transaction. If a commit decision is reached, all data partitions commit their sub-transactions. However, if an abort is decided, data partitions abort their sub-transactions. 2PC assumes that the coordinator and the data partitions are trusted. The main challenge in blockchain systems is how to design a trust-free variation of 2PC where participants do not trust each other and a protocol cannot depend on a centralized trusted coordinator.

An atomic cross-chain commitment protocol requires that for every edge $e = (u, v) \in \mathcal{E}$, the source participant u to lock an asset $e.a$ in Blockchain $e.BC$. This asset locking is necessary to temporarily prevent the participant u from spending $e.a$ through other transactions in $e.BC$. If every source participant u locks $e.a$ in $e.BC$, the atomic cross-chain commitment protocol can decide to commit the AC^{2}T. Once the protocol decides to commit the AC^{2}T, every recipient participant v should be able to *redeem* the asset $e.a$. However, if the protocol decides to abort the AC^{2}T because some participants do not comply to the protocol or a participant requests the transaction to abort, every source participant u should be able to get a refund of their locked assets $e.a$.

In blockchain systems, smart contracts are used to implement this logic. Participant u deploys a smart contract SC_e in Blockchain $e.BC$ to lock an asset $e.a$ owned by u in SC_e. SC_e ascertains to conditionally transfer $e.a$ to v if a commitment decision is reached, otherwise $e.a$ is refunded to u. A smart contract SC_e exists in one of three states: *published* (P), *redeemed* (RD), or *refunded* (RF). A smart contract SC_e is *published* if it gets deployed to $e.BC$ by u. Publishing the smart contract SC_e serves *two* important goals towards the atomic execution of an AC^{2}T. First, it represents a *yes* vote on the sub-transaction corresponding to the edge e. Second, it locks the asset $e.a$ in blockchain $e.BC$. A smart contract SC_e is *redeemed* if participant v successfully redeems the asset $e.a$ from SC_e. Finally, a smart contract SC_e is refunded if the asset $e.a$ is refunded to participant u.

Now, if for every edge $e = (u, v) \in \mathcal{E}$, the participant u publishes a smart contract SC_e in $e.BC$, it means that all participants vote yes on AC^{2}T, lock their involved assets in AC^{2}T, and hence the AC^{2}T can be committed. However, if some participants decline to publish their smart contracts, the AC^{2}T has to be aborted. The commitment of AC^{2}T requires the redemption of **every** smart contract SC_e in AC^{2}T. On the other hand, if the AC^{2}T aborts, this requires the refund of **every** smart contract SC_e in AC^{2}T.

To implement conditional smart contract redemption and refund, a cryptographic commitment scheme primitive based on [105] is used. A *commitment scheme* allows a user to commit to some chosen value without revealing this value. Once this hidden value is revealed, other users can verify that the revealed value is indeed the one that is used in the commitment. A *hashlock* is an example of a commitment scheme. A hashlock is a cryptographic one-way hash function $h = H(s)$ that is used to conditionally lock assets in a smart contract using h, the lock, until a hash secret s, the key, is revealed. Once s is revealed, everyone can verify that the lock h equals to $H(s)$ and hence unlocks the assets locked in the smart contract.

An atomic cross-chain commitment protocol should ensure that smart contracts in AC^{2}T are *either all redeemed or all refunded.* For this, a protocol uses *two* mutually exclusive commitment scheme instances: a redemption commitment scheme and a refund commitment scheme. All smart contracts in AC^{2}T commit their redemption action to the redemption commitment scheme instance and their refund action to the refund commitment scheme instance. If the protocol decides to commit the AC^{2}T, the protocol must publish the redemption commitment scheme secret. This allows all participants in AC^{2}T to redeem their assets. However, if the protocol reaches an abort decision, the protocol must publish the refund commitment scheme secret. This allows participants in AC^{2}T to refund the locked assets in every published smart contract. A protocol must ensure that once the secret of one commitment scheme instance is revealed, the secret of the other instance cannot be revealed. This guarantees the *atomic* execution of an AC^{2}T.

Algorithm 9 illustrates a smart contract template that can be used in implementing an atomic cross-chain commitment protocol. Each smart contract has a sender s and recipient r (Line 2), an asset a (Line 3) to be transferred from s to r through the contract, a state (Line 4), and a redemption and refund commitment scheme instances rd and rf (Lines 5 and 6). A smart contract is published in a blockchain through a deployment message. When published, its constructor (Line 7) is executed to initialize the contract. The deployment message of a smart contract typically includes some implicit parameters like the sender's address (msg.sender, Line 8) and the asset value (msg.value, Line 9) to be locked in the contract. The constructor initializes the addresses, the asset value, the refund and redemption commitment schemes, and sets the contract state to P (Line 11).

In addition, each smart contract has a redeem function (Line 13) and a refund function (Line 17). A redeem function takes evidence parameter. This evidence parameter proves that the AC^{2}T is decided to commit. The redeem function requires the smart contract to be in state P and that the provided evidence is a valid redemption commitment scheme secret (Line 14). If all these requirements hold, the asset a is transferred from the contract to the recipient and the contract state is changed to *RD*. However, if any requirement is violated, the redeem function fails and the smart contract state remains unchanged.

Similarly, the refund function requires the smart contract to be in state P and that the provided evidence is a valid refund commitment scheme secret (Line 18). If all these requirements hold, the asset a is refunded from the contract to the sender and the contract state is changed to *RF*.

The redeem and the refund functions use *two* helper functions: IsRedeemable (Line 21)

Algorithm 9 An atomic swap smart contract template

```
abstract class AtomicSwapSC
 1: enum State {Published (P), Redeemed (RD), Refunded (RF)}
 2: Address s, r // Sender and recipient public keys.
 3: Asset a
 4: State state
 5: CS rd // Redemption commitment scheme
 6: CS rf // Refund commitment scheme
 7: procedure CONSTRUCTOR(Address r, CS rd, CS rf)
 8:     this.s = msg.sender, this.r = r
 9:     this.a = msg.value
10:     this.rd = rd, this.rf = rf
11:     state = P
12: end procedure
13: procedure REDEEM(Evidence $e_{rd}$)
14:     requires(state == P and IsRedeemable($e_{rd}$))
15:     transfer a to r, state = RD
16: end procedure
17: procedure REFUND(Evidence $e_{rf}$)
18:     requires(state == P and IsRefundable($e_{rf}$))
19:     transfer a to s, state = RF
20: end procedure
21: procedure ISREDEEMABLE(Evidence $e_{rd}$)
22:     return verify(rd, $e_{rd}$)
23: end procedure
24: procedure ISREFUNDABLE(Evidence $e_{rf}$)
25:     return verify(rf, $e_{rf}$)
26: end procedure
```

and IsRefundable (Line 24). IsRedeemable verifies that the provided evidence is a valid redemption commitment scheme secret and hence the smart contract can be redeemed. Similarly, IsRefundable verifies that the provided evidence is a valid refund commitment scheme secret and hence the smart contract can be refunded.

5.4 AC^3: Atomic Cross-Chain Commitment

This section presents an **A**tomic **C**ross-**C**hain **C**ommitment, AC^3, protocol, AC^{3}WN, that achieves both **atomicity** and **commitment** of an AC^{2}T. First, we present an important building block on how miners of one blockchain validate the publishing of a transaction or a smart contract in another blockchain in Section 5.4.1. Then, we present AC^{3}WN, an AC^3 protocol that uses a permissionless **W**itness **N**etwork to coordinate AC^{2}Ts in 5.4.2. Using a permissionless network of witnesses does not require more trust in the witness network than the required trust in the blockchains used to exchange the assets in an AC^{2}T. Furthermore, the AC^{3}WN protocol overcomes the vulnerability of centralized solutions that are subject to failures and denial of service attacks.

5.4.1 Cross-Chain Validation

This section explains different techniques of how the miners of one blockchain, *the validators*, can validate the publishing and verify the state of a smart contract deployed in another blockchain, *the validated* blockchain. A **simple but impractical** solution is to require all the miners of every blockchain to serve as validators to all other blockchains. A blockchain validator maintains a copy of the validated blockchain and for every new mined block, a validator validates the mined block and adds it to its local copy of the validated blockchain. If all mining nodes mine one blockchain and validate all other blockchains, mining nodes can consult their local copies of these blockchains to validate the publishing and hence verify the state of any smart contract in any blockchain. If a participant needs the miners of the validator blockchain to validate the publishing of a smart contract in the validated blockchain, this participant submits evidence that comprises a block id and a transaction id of the smart contract in the validated blockchain to the miners of the validator blockchain. This evidence is easily verified by the mining node of the validator blockchain by consulting their copy of the validated blockchain. However, this full replication of all the blockchains in all the mining nodes is impractical. Not only does it require massive processing power to validate all blockchains, but also it requires significant storage and network capabilities at each mining node.

Alternatively, miners of one blockchain, the validators, can run *light nodes* [64] of other blockchains, the validated blockchains. A light node, as defined in [64], is a node that downloads only block headers of the validated blockchain, verifies the proof of work of these block headers, and downloads only the blockchain branches that associate with transactions of interest to this node. This solution requires the validators to mine for one blockchain and run light nodes for every validated blockchains. The validators can consult their local light node copy of the validated blockchain to validate the publishing and hence verify the state of a smart contract in the validated blockchain. Although the cost of maintaining a light node is much cheaper than maintaining a blockchain full copy, running a light node for all blockchains does not scale as the number of blockchains increases.

The previous two techniques put the onus of validating one blockchain on the miners of another blockchain. In addition, they require changes in the current infrastructure by requiring the miners of one blockchain to either maintain a full copy or a light node of other blockchains.

Our proposal: Another way to allow miners of one blockchain, the validators, to validate the publishing and verify the state of a smart contract in another blockchain, the validated, is to push the validation logic into the code of a smart contract in the validator blockchain. A smart contract in the validator blockchain is deployed and stores the header of a *stable block* in the validated blockchain. A stable block is a block at depth d from the current head of the validated blockchain such that the probability of forking the blockchain at this block is negligible (i.e., a block at depth ≥ 6 in the Bitcoin

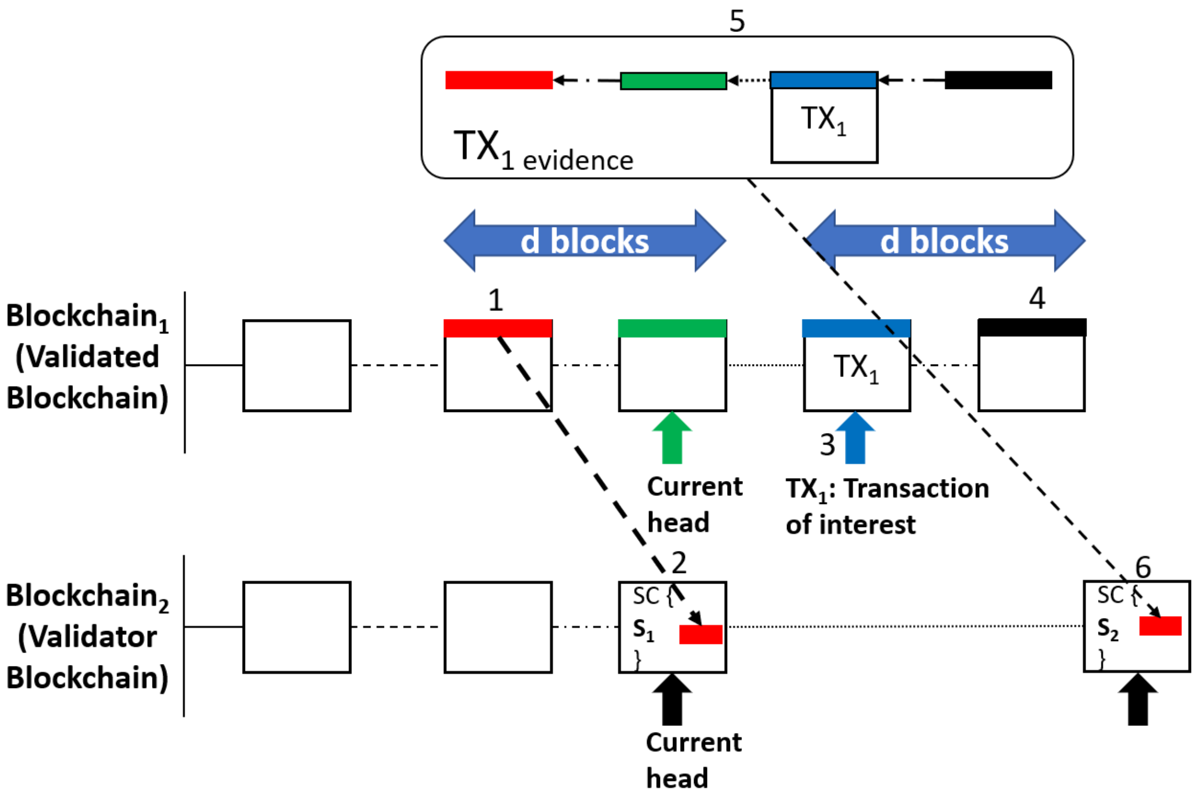

5
TX1
TX1 evidence
d blocks
d blocks
1
4
Blockchain1 (Validated Blockchain)
TX1
Current head
3
TX1: Transaction of interest
2
6
Blockchain2 (Validator Blockchain)
SC {
S1
}
SC {
S2
}
Current head

Figure 5.2: How miners of one blockchain could validate transactions in another blockchain.

blockchain [23]). A participant who deploys the smart contract in the validator blockchain stores the block header of a stable block of the validated blockchain as an attribute in the smart contract object in the validator blockchain. When the transaction or the smart contract of interest takes place in a block in the validated blockchain and after this block becomes a stable block, at depth d, a participant can submit evidence of the transaction occurrence in the validated blockchain to the miners of the validator blockchain. This evidence comprises the headers of all the blocks that follow the stored stable block in the smart contract of the validator blockchain in addition to the block where the transaction of interest took place. The evidence is submitted to the validator smart contract via a function call. This smart contract function validates that the passed headers follow the header of the stable block previously stored in the smart contract object and that the proof of work of each header is valid. In addition, the function verifies that the transaction of interest indeed took place and that the block of this transaction is stable and buried under d blocks in the validated blockchain.

Figure 5.2 shows an example of a validator blockchain, blockchain$_2$, that validates the occurrence of transaction TX_1 in the validated blockchain, blockchain$_1$. In this example, there exists a smart contract SC that gets deployed in the current head block of blockchain$_2$ (labeled by number 2 in Figure 5.2). SC has an initial state S_1 and stores the header of a stable block, at depth d, in blockchain$_1$ (labeled by number 1). This header is represented by a red rectangle inside SC. SC's state is altered from S_1 to S_2 if evidence is submitted to miners of blockchain$_2$ that proves that TX_1 took place in blockchain$_1$ in some block after the stored stable block in SC. When TX_1 takes place in blockchain$_1$ (labeled by number 3) and its block becomes a stable block at depth $\geq d$ (labeled by number 4), a participant submits the evidence (labeled by number 5) to the miners of blockchain$_2$ through SC's function call (labeled by number 6). This function takes the evidence as a parameter and verifies that the submitted blocks took place after the stored stable block in SC. This verification ensures that the header of each submitted block includes the hash of the header of the previous block starting from the stored stable block in SC. In addition, this function verifies the proof of work of each submitted block header. Finally, the function validates that TX_1 took place in some block in the submitted blocks and that this block has already become a stable block. If this verification succeeds, the state of SC is altered from S_1 to S_2. This technique allows miners of one blockchain to verify transactions and smart contracts in another blockchain without maintaining a copy of this blockchain. In addition, this technique puts the evidence validation responsibility on the developer of the validator smart contract.

5.4.2 AC^{3}WN: Permissionless Witness Network

This section presents AC^{3}WN, an AC^3 protocol that uses a *permissionless blockchain network* of witnesses to decide whether an AC^2T should be committed or aborted. **Miners** of this blockchain are collectively the **witnesses** on AC^2Ts. The main design challenge of the AC^{3}WN protocol is how to use a permissionless network of witnesses to implement the redemption and refund commitment scheme instances used by every smart contract in AC^{2}T. In addition, how to ensure that the two instances are *mutually exclusive*.

When a set of participants want to execute an AC^{2}T, they deploy a smart contract SC_w in the witness network where SC_w is used to coordinate the AC^{2}T. SC_w has a state that determines the state of the AC^{2}T. SC_w exists in one of three states: *Published* (P), *Redeem_Authorized* (RD_{auth}), or *Refund_Authorized* (RF_{auth}). Once SC_w is deployed, SC_w is initialized to the state P. If the witness network decides to commit the AC^{2}T, the witnesses set SC_w's state to RD_{auth}. However, if the witness network decides to abort the AC^{2}T, the witnesses set SC_w's state to RF_{auth}.

Figure 5.3 shows an AC^{2}T that exchanges assets among blockchains, $blockchain_1, ...,$ $blockchain_n$ and uses a *witness blockchain* for coordination. Also, it illustrates the AC^{3}WN protocol steps. For every AC^{2}T, a directed graph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ is constructed at some timestamp t and multisigned by all the participants in the set $\mathcal{V}$ generating a graph multisignature $ms(\mathcal{D})$ as shown in Equation 5.1. The timestamp t is important to distinguish between identical AC^2Ts among the same participants. The order of participant signatures in $ms(\mathcal{D})$ is not important. Any signature order indicates that all participants in the AC^{2}T agree on the graph $\mathcal{D}$ at some timestamp t.

$$ms(\mathcal{D}) = sig(..., sig((\mathcal{D}, t), p_1), ..., p_{|\mathcal{V}|}) \tag{5.1}$$

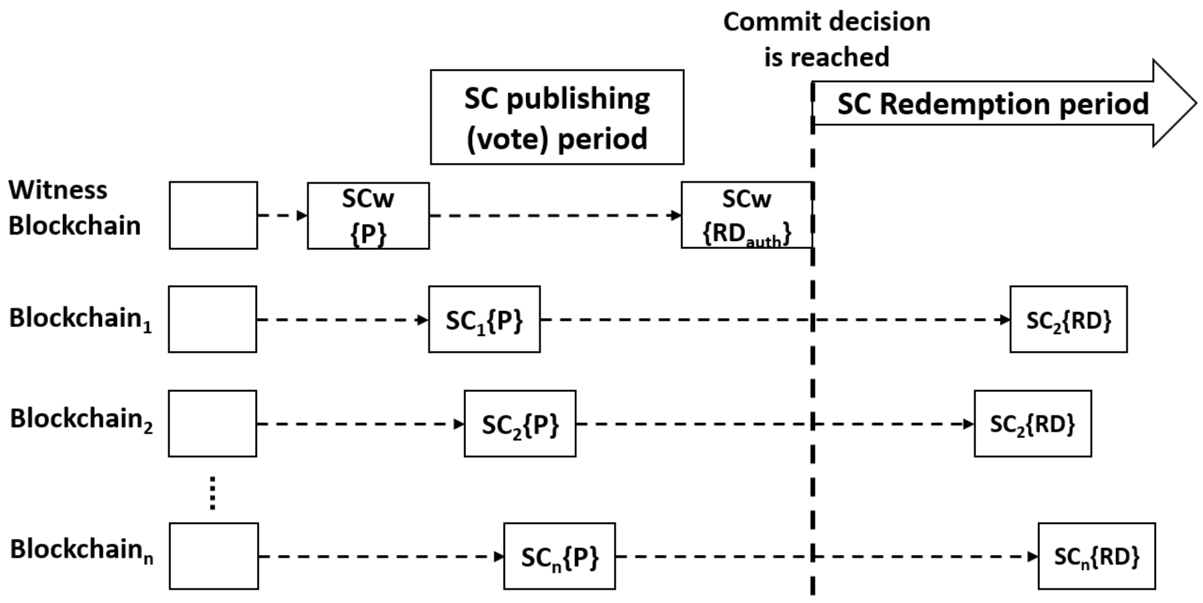


Figure 5.3: Coordinating AC^2Ts using a permissionless witness network.

A participant registers $ms(\mathcal{D})$ in a smart contract SC_w in the witness network where SC_w's state is initialized to P. The state P indicates that participants of the AC^{2}T agreed on $\mathcal{D}$. In addition, participants agree to conditionally link the redeem and the refund actions of their smart contracts in the AC^{2}T to SC_w's states RD_{auth} and RF_{auth} respectively. Afterwards, the participants *parallelly* deploy their smart contracts in the blockchains, $blockchain_1, ..., blockchain_n$, as shown in Figure 5.3. After all the participants deploy their smart contracts in the AC^{2}T, a participant may submit a state change request to the witness network miners to alter SC_w's state from P to RD_{auth}. This request is accompanied by evidence that all smart contracts in the AC^{2}T are deployed and correct. Upon receiving this request, witness network miners verify that SC_w's state is P and that participants of the AC^{2}T have indeed deployed their smart contracts in the AC^{2}T in their corresponding blockchains. In addition, miners verify that all these smart contracts are in state P and that the redemption and the refund of these smart contracts are conditioned on SC_w's states RD_{auth} and RF_{auth} respectively. If this verification succeeds, witness network miners record SC_w state change to RD_{auth} in their current block. Once a block that reflects the state change of SC_w to RD_{auth} is mined in the witness network, the commitment of the AC^{2}T is decided and participants can use this block as a commitment evidence to redeem their assets in the smart contracts of the AC^{2}T. The commit decision is illustrated in Figure 5.3 using the vertical dotted line.

Similarly, if some participants decline to deploy their smart contracts in the AC^{2}T or a participant changes her mind before the commitment of the AC^{2}T, a participant can submit a state change request to the witness network miners to alter SC_w's state from P to RF_{auth}. The miners of the witness network only verify that SC_w's state is P. If this verification succeeds, the miners of the witness network record SC_w's state change to RF_{auth} in their current block. Once a block that reflects the state change of SC_w to RF_{auth} is mined in the witness network, the AC^{2}T is decided to abort and the participants can use this block as evidence of the abort to refund their assets in the deployed smart contracts of the AC^{2}T. Note that SC_w is programmed to ensure that SC_w's state can only be changed either from P to RD_{auth} or from P to RF_{auth} but no other state transition is allowed. This ensures that SC_w's states RD_{auth} and RF_{auth} are mutually exclusive. Miners use the cross-chain evidence validation techniques presented in Section 5.4.1 to validate smart contracts in other blockchains.

Algorithm 10 presents the details of SC_w. SC_w consists of *four* functions: Constructor (Line 5), AuthorizeRedeem (Line 10), AuthorizeRefund (Line 14), and VerifyContracts (Line 18). The Constructor initializes SC_w with the participants public keys and the multisigned graph of the AC^{2}T. This information is necessary to the witness network miners to later verify the publishing of all smart contracts in the AC^{2}T. AuthorizeRedeem alters SC_w's state from P to RD_{auth}. To call AuthorizeRedeem, a participant provides evidence that shows where the smart contracts of the AC^{2}T are published (Line 10). AuthorizeRedeem first verifies that SC_w's state is currently P. In addition, AuthorizeRedeem verifies that all smart contract in the AC^{2}T are published and correct through a VerifyContracts function call (Line 11). If this verification succeeds, SC_w's state is altered to RD_{auth} (Line 12). On the other hand, AuthorizeRefund verifies only that the state of SC_w is P (Line 15). If true, SC_w's state is altered to RF_{auth} (Line 16).

Algorithm 10 Witness network smart contract as an AC^{2}T Coordinator.

```
class WitnessSmartContract
1: enum State {Published (P), Redeem_Authorized ($RD_{auth}$), Refund_Authorized ($RF_{auth}$)}
2: Address [] pk // Addresses of all participants in AC$^2$T
3: Mutlisignature ms // The multisigned graph $\mathcal{D}$
4: State state
5: procedure Constructor(Address[] pk, MS ms($\mathcal{D}$))
6:     this.pk = pk
7:     this.ms = ms($\mathcal{D}$)
8:     this.state = P
9: end procedure
10: procedure AuthorizeRedeem(Evidence e )
11:     requires (state == P and VerifyContracts(e))
12:     this.state = $RD_{auth}$
13: end procedure
14: procedure AuthorizeRefund
15:     requires (state == P)
16:     this.state = $RF_{auth}$
17: end procedure
18: procedure VerifyContracts(Evidence e)
19:     if e validates all the smart contracts in AC$^2$T (Check Section 5.4.1 for details) then
20:         return true
21:     end if
22:     return false
23: end procedure
```

VerifyContracts validates that all smart contracts in the AC^{2}T are published and correct. For every edge $e = (u, v) \in \mathcal{D}.\mathcal{E}$, VerifyContracts finds a matching smart contract SC_e in the participant evidence. VerifyContracts ensures that SC_e matches its description in the edge e. If any parameter in SC_e does not match its description in e, VerifyContracts fails and returns *false* (Line 22). However, if all smart contracts in the provided list are correct, VerifyContracts returns *true* (Line 20). VerifyContracts ensures that AuthorizeRedeem cannot be executed unless all smart contract in the AC^{2}T are published and correct and hence a commit decision can be reached.

Algorithm 11 Smart contract for permissionless AC^3.

```
class PermissionlessSC extends AtomicSwapSC
 1: procedure CONSTRUCTOR(Address r, SC $SC_w$, Depth d)
 2:     this.rd = this.rf = ($SC_w$, d)
 3:     super(r, this.rd, this.rf) // parent constructor
 4: end procedure
 5: procedure ISREDEEMABLE(Evidence e)
 6:     if e validates that $SC_w$'s state is $RD_{auth}$ and and that $SC_w$'s state update is at
    depth $\geq d$ then
 7:         return true
 8:     end if
 9:     return false
10: end procedure
11: procedure ISREFUNDABLE(Evidence e)
12:     if e validates that $SC_w$'s state is $RF_{auth}$ and that $SC_w$'s state update is at depth
    $\geq d$ then
13:         return true
14:     end if
15:     return false
16: end procedure
```

Algorithm 11 presents a smart contract class inherited from the smart contract template in Algorithm 9 in order to use SC_w's state as redemption and refund commitment scheme secrets. IsRedeemable returns *true* if SC_w's state is RD_{auth} (Line 6), while IsRefundable returns *true* if SC_w's state is RF_{auth} (Line 12). As the witness network is permissionless, forks could possibly happen resulting in *two* concurrent blocks where SC_w's state is RD_{auth} in the first block and SC_w's state is RF_{auth} in the second block. To avoid atomicity violations, participants cannot use a witness network block where SC_w's state is RD_{auth} or RF_{auth} in their smart contract redemption and refund respectively unless this block is buried under at least d blocks in the witness network. As the probability of a fork of depth d (e.g., 6 blocks in the Bitcoin network [23]) is negligible, SC_w's state eventually converges to either RD_{auth} or RF_{auth}.

The following steps summarizes the AC^{3}WN protocol steps to execute the AC^{2}T shown in Figure 5.1:

1. Alice and Bob construct the AC^2T's graph $\mathcal{D}$ and multisign $(\mathcal{D}, t)$ to generate $ms(\mathcal{D})$.
2. Either Alice or Bob registers $ms(\mathcal{D})$ in a smart contract SC_w and publishes SC_w in the witness network setting SC_w's state is P. SC_w follows Algorithm 10.
3. Afterwards, Alice publishes a smart contract SC_1 using Algorithm 11 to the Bitcoin network that states the following:
   - Move X bitcoins from Alice to Bob if Bob provides evidence that SC_w's state is RD_{auth}.
   - Refund X bitcoins from SC_1 to Alice if Alice provides evidence that SC_w's state is RF_{auth}.
4. Concurrently, Bob publishes a smart contract SC_2 to the Ethereum network using Algorithm 11 stating the following:
   - Move Y ethers from Bob to Alice if Alice provides evidence that SC_w's state is RD_{auth}.
   - Refund Y ethers from SC_2 to Bob if Bob provides evidence that SC_w's state is RF_{auth}.
5. After both SC_1 and SC_2 are published, any participant can submit a state change request of SC_w from P to RD_{auth} to the witness network miners. This request

is accompanied by evidence that SC_1 and SC_2 are published in the Bitcoin and the Ethereum blockchains respectively. The witness network miners first verify that SC_w's state is currently P. Then, they verify that both SC_1 and SC_2 are published and correct in their corresponding blockchains. If these verifications succeed, the miners of the witness network record SC_w's state change to RD_{auth} in their current block. Once a block that reflects the state change of SC_w to RD_{auth} is mined and gets buried under d blocks in the witness network, Alice and Bob can use this block as evidence to redeem their assets from SC_2 and SC_1 respectively.

6. If a participant declines to publish a smart contract, the other participant can submit a state change request of SC_w from P to RF_{auth} to the witness network miners. The witness network miners verify that SC_w's state is currently P. If true, miners record SC_w's state change to RF_{auth} in their current block. Once a block that reflects the state change of SC_w to RF_{auth} is mined and gets buried under d blocks in the witness network, Alice and Bob can use this block as evidence to refund their assets from SC_1 and SC_2 respectively.

This protocol uses two blockchain techniques to ensure that SC_w's states RD_{auth} and RF_{auth} are *mutually exclusive*. First, it uses the smart contract programmable logic to ensure that SC_w's state can only be altered from P to RD_{auth} or from P to RF_{auth}. Second, it uses the longest chain fork resolving technique to resolve forks in the witness network blockchain. This ensures that in the rare case of forking where one fork chain has SC_w's state of RD_{auth} and another fork chain has SC_w's state of RF_{auth}, the fork is eventually resolved resulting in either SC_w's state is RD_{auth} or SC_w's state is RF_{auth} but not both.

5.5 AC^{3}WN Analysis

This section analyzes the AC^{3}WN protocol introduced in Section 5.4.2. First, we establish that the proposed protocol ensures atomicity. Then we analyze the scalability of the witness network and how it affects the scalability of the commitment protocol. Finally, we explain how this protocol extends the functionality of previous proposals in [169, 117].

5.5.1 AC^{3}WN: Atomicity Correctness Proof

Lemma 5.5.1 *Assume no forks in the witness network, then the AC^{3}WN protocol is atomic.*

Proof: Assume an AC^{2}T executed by the AC^{3}WN protocol and the atomicity of this transaction is violated. This atomicity violation implies that there exists two smart contract SC_i and SC_j in AC^{2}T where SC_i is redeemed and SC_j is refunded. The redemption of SC_i implies that there exists a block in the witness network where SC_w's state is RD_{auth} while the refund of SC_j implies that there exists a block in the witness network where SC_w's state is RF_{auth}. Since SC_w is programmed to allow only the state transitions either from P to RD_{auth} or from P to RF_{auth}, the two function calls to alter SC_w's state from P to RD_{auth} and from P to RF_{auth} cannot take effect in one block. Miners of the witness network shall accept one and reject the other. Therefore, these two state changes must be recorded in two separate blocks. As there exists no forks in the witness network, one of these two blocks must *happen before* the other. This implies that either SC_w's state is altered from RD_{auth} in one block to RF_{auth} in a following block or altered from RF_{auth} in one block to RD_{auth} in a following block. However, only the state transitions from P to RD_{auth} or from P to RF_{auth} are allowed and no other state transition is permitted leading to a contradiction. ■

Lemma 5.5.2 *Let ϵ be a negligible probability of forks in the permissionless witness network, then AC^3WN protocol is atomic with a probability* $1 - \epsilon$.

Proof: Assume an AC^{2}T executed by the AC^{3}WN protocol and the atomicity of this transaction is violated with a probability $p >>> \epsilon$. This atomicity violation implies that there exists two smart contract SC_i and SC_j in AC^{2}T where SC_i is redeemed and SC_j is refunded. The redemption of SC_i implies that there exists a block in the witness network where SC_w's state is RD_{auth} while the refund of SC_j implies that there exists a block in the witness network where SC_w's state is RF_{auth}. As SC_w's states RD_{auth} and RF_{auth} are conflicting states, this implies that the block where SC_w's state update to RD_{auth} occurs must exist in a fork from the block where SC_w's state update to RF_{auth} occurs. The atomicity violation of the AC^{2}T with a probability p implies that the fork probability in the witness network must be p leading to a contradiction. ■

5.5.2 The Scalability of AC^{3}WN

One important aspect of AC^3 protocols is *scalability*. Does using a permissionless network of witnesses to coordinate AC^{2}Ts limit the scalability of the AC^{3}WN protocol? In this section, we argue that the answer is *no*. To explain this argument, we first develop an understanding of the properties of executing AC^{2}Ts and the role of the witness network in executing AC^{2}Ts.

An AC^{2}T is a distributed transaction that consists of sub-transactions. Each sub-transaction is executed in a blockchain. An AC^3 protocol coordinates the atomic execution of these sub-transactions across several blockchains. An AC^3 protocol must ensure an atomic execution of the distributed transaction. This atomic execution of a distributed transaction requires the ACID [110, 115] execution of every sub-transaction in this distributed transaction in addition to the atomic execution of the distributed transaction itself. The ACID execution of a sub-transaction executed within a single blockchain is guaranteed by the miners of this blockchain. Miners use many techniques including mining, verification, and the miner's rationale to join the longest chain in order to implement ACID executions of transactions within a single blockchain. The atomicity of the distributed transaction is the responsibility of the distributed transaction coordinator. Therefore, the main role of the witness network in the AC^{3}WN protocol is to ensure the atomicity of the AC^{2}T. Since the atomicity coordination of AC^{2}Ts is *embarrassingly parallel*, different witness network can be used to coordinate different AC^{2}Ts.

Assume two concurrent AC^{2}Ts, t_1 and t_2. The atomic execution of t_1 does not require any coordination with the atomic execution of t_2. Each AC^{2}T requires its witness network to ensure that either all sub-transactions in the AC^{2}T are executed or none of them is executed. Therefore, t_1 and t_2 do not have to be coordinated by the same witness network. t_1 can be coordinated by one witness network while t_2 can be coordinated by another witness network. If t_1 and t_2 conflict at the sub-transaction level, this conflict is resolved by the miners of the blockchain where these sub-transactions are executed. Therefore, using a permissionless witness network to coordinate AC^{2}Ts does not limit the scalability of the AC^{3}WN protocol. Different permissionless networks are used to coordinate different AC^{2}Ts. For example, the Bitcoin network can be used to coordinate t_1 while the Ethereum network can be used to coordinate t_2.

5.5.3 Handling Complex AC^{2}T Graphs

One main improvement of the AC^{3}WN protocol over the state-of-the-art AC^3 protocols in [117, 169] is its ability to coordinate the atomic execution of AC^{2}Ts with complex graphs. This improvement is achieved because the AC^{3}WN protocol does not depend on the rational behavior of the participants in the AC^{2}T to ensure atomicity. Instead,

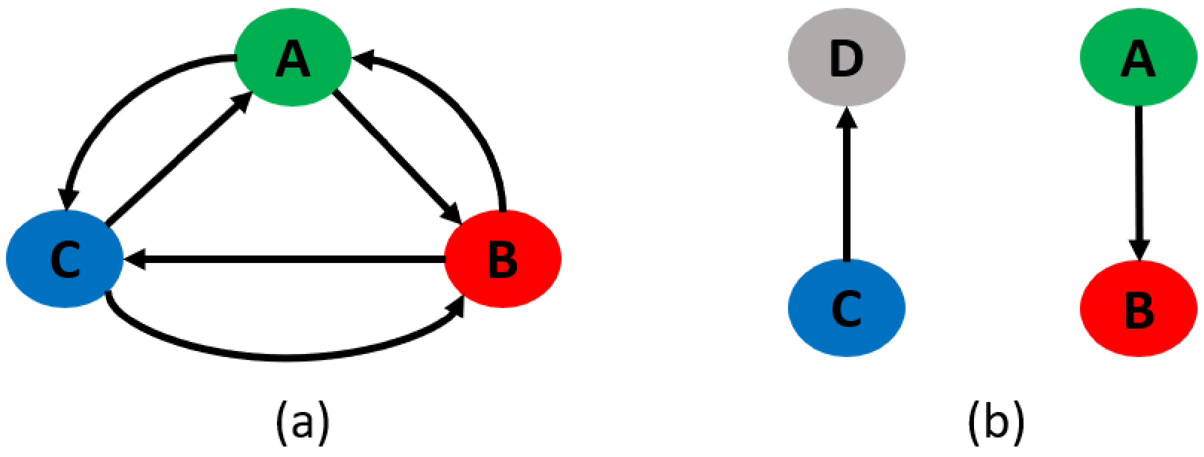


Figure 5.4: Examples of complex graphs handled by the AC^{3}WN protocol: (a) cyclic and (b) disconnected.

the protocol depends on a permissionless network of witnesses to coordinate the atomic execution of AC^{2}Ts. Once the participants agree on the AC^{2}T graph and register it in the smart contract SC_w in the witness network, participants cannot violate atomicity as the commit and the abort decisions are decided by the state of SC_w. The state transitions of SC_w are witnessed and verified by the miners of the witness network. Therefore, the publishing order of the smart contracts in the AC^{2}T cannot result in an advantage to any coalition among the participants. Participants can concurrently publish their smart contracts in the AC^{2}T, both in Figures 5.1 and 5.4, without worrying about the maliciousness of any participant.

Figure 5.4 illustrates two complex graph examples that either cannot be atomically executed by the protocols in [169, 117] or require additional mechanisms and protocol modifications to be atomically executed. These graphs appear in supply-chain applications. Both Nolan's and Herlihy's single leader protocol require the AC^{2}T graph to be acyclic once the leader node is removed. Therefore, both protocols fail to execute the transaction graph shown in Figure 5.4a. Removing any node from the graph in Figure 5.4a still results in a cyclic graph. Herlihy presents a multi-leader protocol in [117] to handle cyclic graphs. However, both Nolan's and Herlihy's protocols fail to handle disconnected graphs similar to the graph shown in Figure 5.4b. On the other hand, the AC^{3}WN protocol ensures the atomic execution of AC^{2}Ts irrespective of the AC^{2}T's graph structure.

5.6 AC^{3}WN Evaluation

This section analytically compares the performance and the overhead of the AC^{3}WN protocol to the state-of-the-art atomic swap protocol presented by Herilhy in [117]. First, we compare the latency of AC^{2}Ts as the diameter of the transaction graph $\mathcal{D}$ increases in Section 5.6.1. Then, the monetary cost overhead of using a permissionless network of witnesses to coordinate the AC^{2}T is analyzed in Section 5.6.2. Afterwards, an analysis on how to choose the witness network is developed in Section 5.6.3. Finally, an analysis of the AC^{2}T throughput as the witness network is chosen from the top-4 permissionless cryptocurrencies, sorted by market cap, is presented in Section 5.6.4.

5.6.1 Latency

The AC^{2}T latency is defined as the difference between the timestamp t_s when an AC^{2}T is started and the timestamp t_c when the AC^{2}T is completed. t_s marks the moment when participants in the AC^{2}T start to agree on the AC^{2}T graph $\mathcal{D}$. t_c marks the completion of all the asset transfers in the AC^{2}T by redeeming all the smart contracts in AC^{2}T.

Let Δ be enough time for any participant to publish a smart contract in any permissionless blockchain, or to change a smart contract state through a function call of this smart contract, and for this change to be publicly recognized [117]. Also, let $Diam(\mathcal{D})$ be the AC^{2}T graph diameter. The $Diam(\mathcal{D})$ is the length of the longest path from any vertex in $\mathcal{D}$ to any other vertex in $\mathcal{D}$ including itself.

The single leader atomic swap protocol presented in [117] has *two* phases: the AC^{2}T smart contract sequential deployment phase and the AC^{2}T smart contract sequential redemption phase. The deployment phase requires the deployment of all smart contracts in the AC^{2}T, N, where exactly $Diam(\mathcal{D}) \leq N$ smart contracts are sequentially deployed resulting in a latency of $\Delta \cdot Diam(\mathcal{D})$. Similarly, the redemption phase requires the redemption of all smart contracts in the AC^{2}T, N, where exactly $Diam(\mathcal{D}) \leq N$ smart contracts are sequentially redeemed resulting in a latency of $\Delta \cdot Diam(\mathcal{D})$. The overall latency of an AC^{2}T that uses this protocol equals to the latency summation of these two phases $2 \cdot \Delta \cdot Diam(\mathcal{D})$. Figure 5.5 visualizes the two phases of the protocol where time advances from left to right. As shown, some smart contracts (e.g., SC_2, SC_3, and SC_4) could be deployed and redeemed in parallel but there are exactly $Diam(\mathcal{D})$ sequentially deployed and $Diam(\mathcal{D})$ sequentially redeemed smart contracts resulting in an overall latency of $2 \cdot \Delta \cdot Diam(\mathcal{D})$. Note that the protocol allows the parallel deployment and redemption of some smart contracts as long as they do not lead to an advantage to either a participant or a coalition in the AC^{2}T.

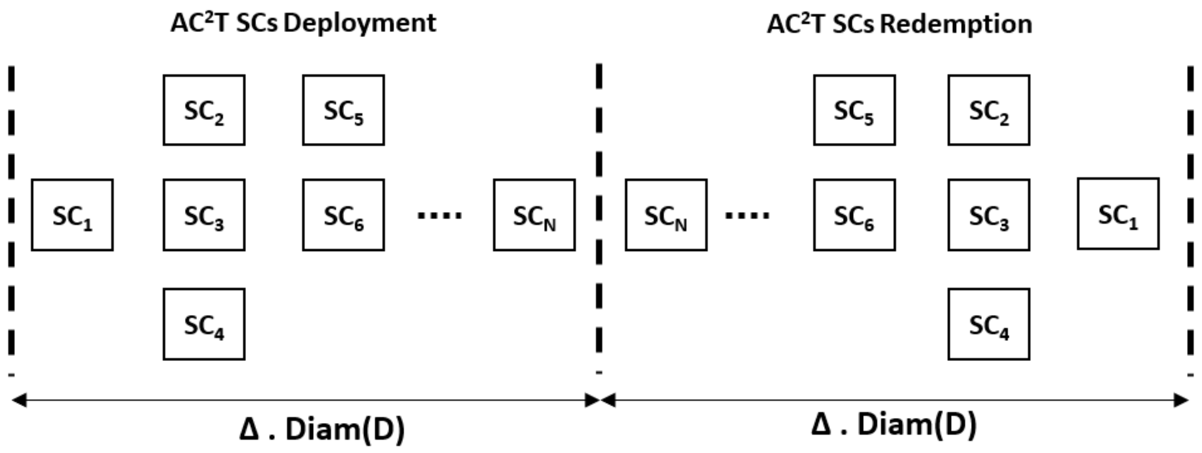


Figure 5.5: The overall transaction latency of $2 \cdot \Delta \cdot Diam(\mathcal{D})$ when the single leader atomic swap protocol in [117] is used.

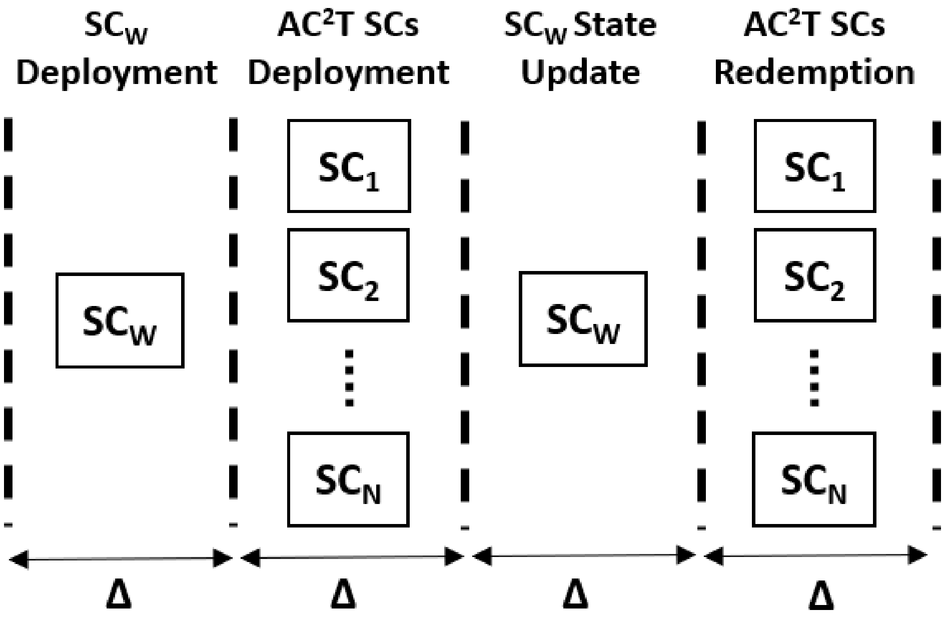


Figure 5.6: The overall transaction latency of $4 \cdot \Delta$ when the AC^{3}WN protocol in Section 5.4.2 is used.

On the other hand, the protocol presented in Section 5.4.2 has *four* phases: the witness network smart contract deployment phase, the AC^{2}T smart contract parallel deployment phase, the witness network smart contract state change phase, and the AC^{2}T smart contract parallel redemption phase. The witness network smart contract deployment requires the deployment of the smart contract SC_w in the witness network resulting in a latency of Δ. The AC^{2}T smart contract parallel deployment requires the parallel deployment of all smart contracts, N, in the AC^{2}T resulting in a latency of Δ. The witness network smart contract state change requires a state change in SC_w either from P to RD_{auth} or from P to RF_{auth} through SC_w's Redeem or Refund function calls resulting in a latency of Δ. Finally, the AC^{2}T smart contract parallel redemption requires the parallel redemption of all smart contracts, N, in the AC^{2}T resulting in a latency of Δ. The overall latency of an AC^{2}T that uses this protocol equals to the latency summation of these four phases $4 \cdot \Delta$. Figure 5.6 visualizes the four phases of the protocol where time advances from left to right. As shown, all smart contracts in the AC^{2}T are parallelly deployed and parallelly redeemed resulting in an overall latency of $4 \cdot \Delta$.

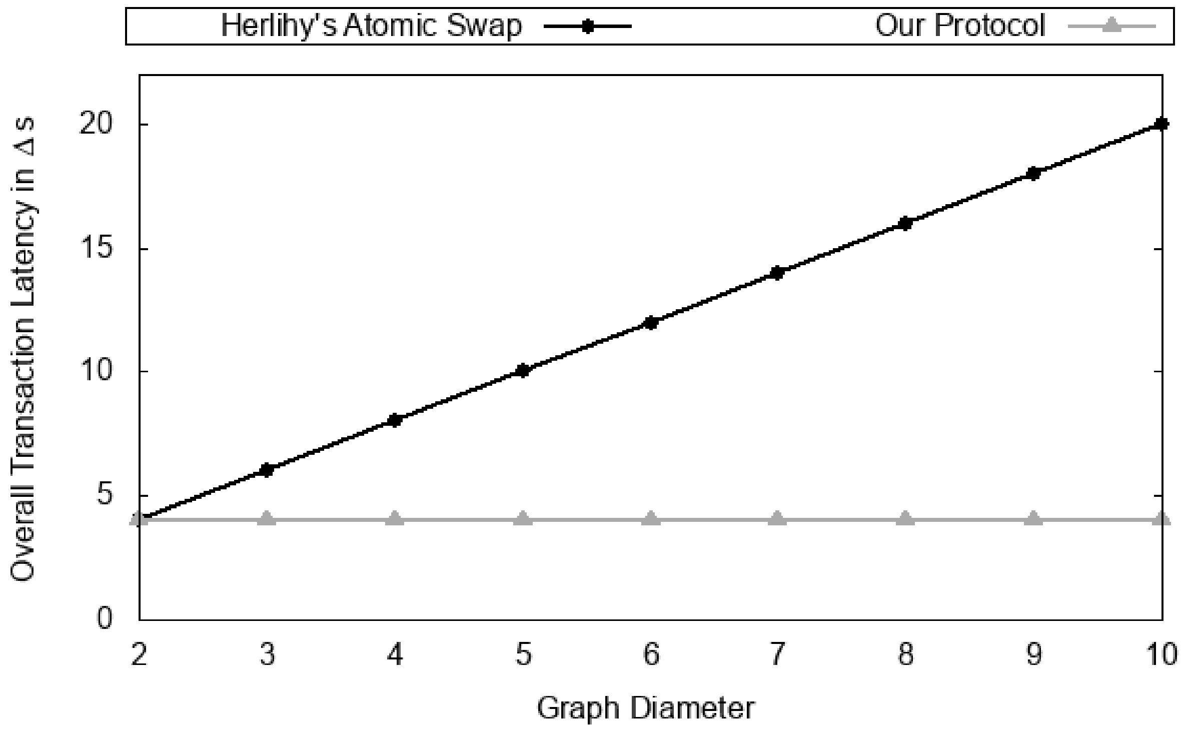


Figure 5.7: The overall AC^{2}T latency in Δs as the graph diameter, $Diam(\mathcal{D})$, increases.

Figure 5.7 compares the overall AC^{2}T latency in Δs resulting from Herlihy's protocol in [117] and our protocol in Section 5.4.2 as the transaction graph diameter, $Diam(\mathcal{D})$ increases. As shown, our protocol achieves a constant latency of $4 \cdot \Delta$ irrespective of the transaction diagram value while Herlihy's protocol achieves a linearly increasingly latency the transaction diagram value increases. Note that the smallest transaction graph consists of *two* nodes and *two* edges and hence the graph diameter in Figure 5.7 starts at 2.

5.6.2 Cost Overhead

This section analyzes the monetary cost overhead of the AC^{3}WN protocol in comparison to Herlihy's atomic swap protocol in [117]. As explained in Section 5.2, miners charge end-users a fee for every smart contract deployment and every smart contract function call that results in a smart contract state change. This fee is necessary to incentives miners to add smart contracts and append smart contract state changes to their mined blocks. As shown in Figures 5.5 and 5.6, both protocols deploy a smart contract for every edge $e \in \mathcal{E}$ where $\mathcal{E}$ is the edge set of the AC^{2}T graph $\mathcal{D}$. This results in the deployment of $N = |\mathcal{E}|$ smart contracts in the smart contract deployment phase of both protocols. In addition, both protocols invoke a redemption or a refund function call for every deployed smart contract in the AC^{2}T resulting in N function calls. However, the AC^{3}WN protocol requires to deploy an additional smart contract SC_w in the witness network in addition to an additional function call to change SC_w's state either from P to RD_{auth} or from P to RF_{auth}. The cost of SC_w deployment and SC_w state transition function call comprises the monetary cost overhead of our protocol. Let f_d be the deployment fee of any smart contract $SC_i \in AC^2T$ and f_{fc} be the function call fee of any smart contract function call. Then, the overall AC^{2}T fee of Herlihy's protocol is $N \cdot (f_d + f_{fc})$ while the overall AC^{2}T fee of the AC^{3}WN protocol is $(N + 1) \cdot (f_d + f_{fc})$. This analysis shows that AC^{3}WN imposes a monetary cost overhead of $\frac{1}{N}$ the transaction fee of Herilhy's protocol assuming equal deployment and functional call fees for all the smart contracts in the AC^{2}T.

But, *How much does it cost in dollars to deploy a smart contract and make a smart contract function call?* The answer is, it depends. Many factors affect a smart contract fee such as the length of the smart contract and the average transaction fee in the smart contract's blockchain [188, 40]. Ryan [188] shows that the cost of deploying a smart contract with a similar logic to SC_w's logic in the Ethereum network costs approximately

\$4 when the ether to USD rate is \$300. Currently, this costs approximately \$2 assuming the current ether to USD rate of \$140.

5.6.3 Choosing the Witness Network

This section develops some insights on how to choose the witness network for an AC^{2}T. This choice has to consider the risk of choosing different permissionless blockchain networks as the witness of an AC^{2}T and the relationship between this risk and the value of the assets exchanged in this AC^{2}T. As the state of the witness smart contract SC_w determines the state of an AC^{2}T, forks in the witness network present a risk to the atomicity of the AC^{2}T. A fork in the witness network where one block has SC_w's state of RD_{auth} and another block has SC_w's state of RF_{auth} might result in an atomicity violation leading to an asset loss of some participants in the AC^{2}T. To overcome possible violation, our AC^{3}WN protocol does not consider a block where SC_w's state is either RD_{auth} or RF_{auth} as a commit or an abort evidence until this block is buried under d blocks in the witness network. This technique of resolving forks by waiting is presented in [164] and used by Pass and Shi in [175] to eliminate uncertainty of recently mined blocks. This fork resolution technique is efficient as that the probability of eliminating a fork within d blocks is sufficiently high.

However, a malicious participant in an AC^{2}T could fork the witness blockchain for d blocks in order to steal the assets of other participants in the AC^{2}T. To execute this attack, a malicious participant rents computing resources to execute a 51% attack on the witness network. The cost of an hour of 51% attack for different cryptocurrency blockchains is presented in [41]. If the cost of running this attack for d blocks is less than the expected gains from running the attack, a malicious participant is incentivized to act maliciously.

To prevent possible maliciousness, the cost of running a 51% attack on the witness network for d blocks must be set to exceed the potential gains of running the attack. Let V_a be the value of the potentially stolen assets if the attack succeeds. Also, let C_h be the hourly cost of a 51% attack on the witness network. Finally, let d_h be the expected number of mined blocks per hour for the witness blockchain (e.g., $d_h = 6$ blocks / hour for the Bitcoin blockchain). The value d must be set to ensure that V_a is less than the cost of running the attack for d blocks $\frac{d \cdot C_h}{d_h}$. Therefore d must be set to achieve the inequality $d > \frac{V_a \cdot d_h}{C_h}$ in order to disincentivize maliciousness. For example, let V_a be \$1M and assume that the Bitcoin network is used to coordinate this transaction. The cost per hour of a 51% attack on the Bitcoin network is approximately $C_h = \$300K$. Therefore, d must be set to be $> \frac{\$1M \cdot 6}{\$300K} = 20$.

5.6.4 Throughput

The throughput of the AC^{2}Ts is the number of transactions per second (tps) that could be processed assuming that every AC^{2}T spans a fixed set of blockchains and is witnessed by a fixed witness blockchain. For an AC^{2}T that spans multiple blockchains, the throughput is bounded by the slowest involved blockchain in the AC^{2}T including the witness network. Let tps_i be the throughput of blockchain i. The throughput of the AC^{2}Ts that span blockchains i, j, .., n and are witnessed by the blockchain w equals to $min(tps_i, tps_j .., tps_n, tps_w)$.

Blockchain	**tps**	**Blockchain**	**tps**
1) Bitcoin	7	3) Litecoin	56
2) Ethereum	25	4) Bitcoin Cash	61

Table 5.1: The throughput in tps of the top-4 permissionless cryptocurrencies sorted by their market cap [170].

Table 5.1 shows the transaction throughput of the top-4 permissionless cryptocurrencies sorted by their market cap. An example AC^{2}T that exchange assets among Ethereum and Litecoin blockchains and are witnessed by the Bitcoin network achieves a throughput of 7. The witness network should be chosen from the set of involved blockchains (Litecoin and Ethereum in this example) to avoid limiting the transaction throughput.

5.7 AC^{3}WN Concluding Remarks

This chapter presents AC^{3}WN, the first decentralized **A**tomic **C**ross-**C**hain **C**ommitment protocol that ensures the all-or-nothing atomicity semantics even in the presence of participant crash failures and network denial of service attacks. Unlike in [169, 117] where the protocol correctness mainly relies on participants rational behaviour, AC^{3}WN separates the coordination of an Atomic Cross-Chain Transaction, AC^{2}T, from its execution. A permissionless open network of witnesses coordinates the AC^{2}T while participants in the AC^{2}T execute sub-transactions in the AC^{2}T. This separation allows AC^{3}WN to ensure atomicity of all the sub-transactions in an AC^{2}T even in the presence of failures. In addition, this separation enables AC^{3}WN to parallelly execute sub-transactions in the AC^{2}T reducing the latency of an AC^{2}T from O($Diam(\mathcal{D})$) in [117], where $Diam(\mathcal{D})$ is the diameter of the AC^{2}T graph $\mathcal{D}$, to $O(1)$ irrespective of the size of the AC^{2}T graph $\mathcal{D}$. Also, this separation allows AC^{3}WN to scale by using different permissionless witness networks to coordinate different AC^{2}Ts. This ensures that using a permissionless network of witnesses for coordination does not introduce any performance bottlenecks. Finally, the AC^{3}WN protocol extends the functionality of the protocol in [117] by supporting AC^{2}Ts with complex graphs (e.g., cyclic and disconnected graphs). AC^{3}WN introduces a slight monetary cost overhead to the participants in the AC^{2}T. This cost equals to the cost of deploying a coordination smart contract in the witness network plus the cost of a function call to the coordination smart contract to decide whether to commit or to abort the AC^{2}T. The smart contract deployment and function call approximately cost \$2 combined per AC^{2}T when the Ethereum network is used to coordinate this AC^{2}T.

Chapter 6

Towards Global Asset Management in Blockchain Systems

6.1 Overview

A blockchain is a distributed data structure for recording transactions maintained by nodes without a central authority [66]. Nodes in a blockchain system agree on their shared states across a large network of *untrusted* participants. Existing blockchain systems can be divided into two main categories: permissionless blockchain systems, e.g., Bitcoin (with PoW-based consensus) [164] and permissioned blockchain systems, e.g., Tendermint (with BFT-type consensus) [136].

Permissionless blockchains, which are mainly devised for cryptocurrency assets, e.g., Bitcoin [164], are public. Any computing node can participate in maintaining the blockchain without obtaining a permission from a centralized authority, hence the name permissionless. In Permissionless blockchains, transactions are used to transfer cryptocurrency assets from one identity to another. In addition, new currency units are generated through mining; once a new block of transactions is added to the blockchain, the miner of the block receives some currency units as the mining reward. The amount of mining reward is specified as part of the blockchain protocol.

Permissionless blockchains are public and computing nodes without a priori known identities can join or leave the blockchain network at any time. On the other hand, a *permissioned* blockchain uses a network of a priori known and identified computing nodes to manage the blockchain. In a permissioned blockchain systems, every node maintains a copy of the blockchain ledger and a consensus protocol is used to ensure that the nodes agree on a unique order in which entries are appended to the blockchain ledger. To reach agreement among the nodes, asynchronous fault-tolerant replication protocols have been used. Nodes in a permissioned blockchain might crash or maliciously behave. Depending on the failure model of nodes, crash fault-tolerant protocols, e.g., Paxos [140], or Byzantine fault-tolerant protocols, e.g., PBFT [67], are used to achieve consensus. The tutorial by C. Mohan [162] provides an overview and discusses many aspects of permissioned blockchains.

The blockchain model is similar to an object-oriented programming language (OOPL). Similar to the primitive data types, user-defined functions, and classes in an OOPL, each blockchain also has primitive data types (e.g., an asset, asset ownership, etc) and primitive functions operating on these primitive data types (e.g., transactions that move currency units from one user identity to another). Classes and complex functionalities are implemented in the blockchain using smart contracts. A ***smart contract***, as exemplified by Ethereum [205], is a computer program that self-executes once it is established and deployed. A smart contract can be seen as a class in an object-oriented programming language where assets are the objects of that class and transactions update the state (ownership) of the objects. The state transformation of a smart contract is made persistent in the blockchain by ensuring that every state change appears as a record in the blockchain.

While permissionless blockchains only support cryptocurrency assets, smart contracts are more generic and can support any type of asset. Indeed, a smart contract, like a class in the object-oriented programming, could potentially have different attributes and functions. Once a smart contract is written, it can be deployed on a blockchain and different transactions can call the functions of the smart contract to change its attributes or even destroy the contract (using a destructor function), making it void.

Deploying general assets (e.g., cars, houses, etc) on the blockchain, in contrast to cryptocurrency assets, gives rise to several challenges. First, ensuring the existence of a registered asset requires some form of *authentication* of the asset. Second, the blockchain system should prevent a malicious end-user from *double spending* the same asset through two different smart contracts either within the same or on different permissionless blockchains. Finally, depending on the asset, the asset transfer should be *legally* allowed by the State law.

To address the aforementioned challenges of *authentication*, *double spending*, and *legality* for complex assets in permissionless blockchains, in this chapter, we propose a *global asset management system* that unifies permissionless and permissioned blockchains. In the proposed system, a governmental permissioned blockchain authenticates the registration of end-user assets through smart contract deployments on a permissionless blockchain. When an end-user requests to register their assets, in order to prevent *double spending*, a governmental office checks if the asset is not already registered as a smart contract in any permissionless blockchain. Next, the governmental office issues an *authenticated* smart contract registering the asset wherein the contract also includes the *legal laws* associated with the asset and deploys the smart contract on the permissionless blockchain. Finally, the end-user will be able to trade the asset in the permissionless blockchain while preserving the law enforcement explicitly specified in the smart contract.

Registering complex assets in permissionless blockchains extends the transaction model of the permissionless blockchains. While permissionless blockchains support intra-chain cryptocurrency trades and cross-chain cryptocurrency trades (with the help of cross-chain swap protocols [117, 169]), the proposed system is able to support any type of transactions in either a single or in multiple chains with any kind of assets.

A key objective of this work is to demonstrate how global assets can be managed in a blockchain system. The main contributions of this work are:

- a global asset management system that unifies permissioned and permissionless blockchains to manage complex asset,
- an extended transaction model that supports varied types of transactions operating on complex assets in multiple blockchains, and finally,
- a thorough analysis of the challenges that arise in designing blockchain-based asset management systems.

The rest of the chapter is organized as follows. Section 6.2 presents the architecture and asset management of permissionless blockchains. Section 6.3 explains how smart contracts are used to extend the functionality of permissionless blockchains. Section 6.4 describes the architecture and asset management of permissioned blockchain. In Section 6.5 permissionless and permissioned blockchains are unified in order to build a novel global asset management system. We discuss the challenges that arise as a result of this unification in Section 6.6. The related work is presented in Section 6.7 and the chapter is concluded in Section 6.8.

6.2 Permissionless Blockchains

Permissionless blockchains are public and therefore, computing nodes, also known as miners, can join or leave the blockchain network without obtaining a permission. Miners maintain a copy of the blockchain ledger and process end-user transactions. Miners and end-users use their public keys as their identities in the blockchain system. Given the open and public model of blockchains, these systems are exemplified by the complete absence of the notion of trust. That is, these blockchains must operate in spite of a complete absence of any trusted entity in the network. Permissionless blockchains are mainly devised for cryptocurrency assets, e.g., Bitcoin [164]. In this section we first explain the architecture of permissionless blockchains and then present the data and transaction models of such systems.

6.2.1 Architecture Overview

A permissionless blockchain system [155] (e.g., Bitcoin, Ethereum) typically consists of three layers: an application layer, a consensus layer, and a storage layer, as illustrated in Figure 6.1.

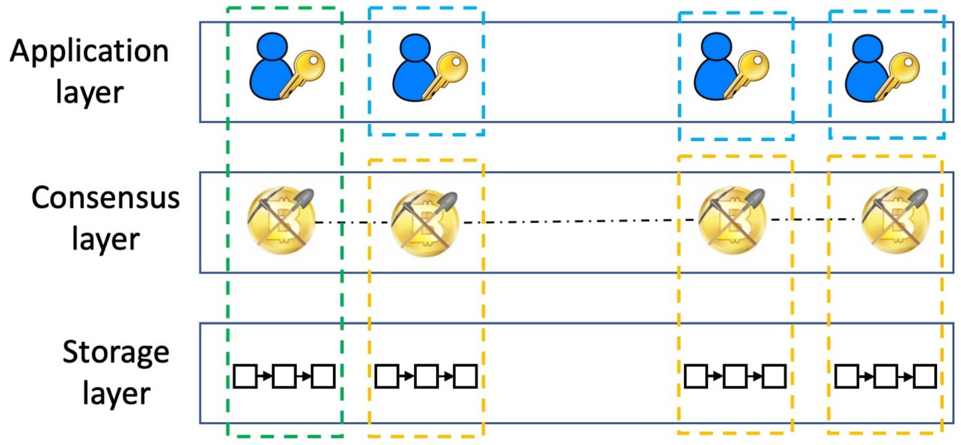


Figure 6.1: Permissionless Blockchain Architecture Overview

The application layer. Transactions are initiated by end-users in the application layer. End-users have identities, defined by their public keys and signatures, generated using their private keys. Digital signatures are the end-users' way to generate transactions. Once transactions are generated, the users multicast their transactions to the mining nodes in the consensus layer through a client library. Transactions are used to transfer assets from one end-user identity to another.

The Consensus Layer. In permissionless blockchains consensus is established through mining. A mining node validates the transactions it receives, puts the valid transactions into a block and try to solve some cryptographic puzzle. The industrious miner who solves the puzzle multicasts the block to all nodes. To make progress, when a miner receives a block of transactions, it first validates the solution to the puzzle and all transactions in the block, appends the block to the blockchain, and then proceeds to mine the next block.

The storage layer. The ledger is a tamper-proof chain of blocks that is maintained by every mining node. The storage layer comprises a decentralized distributed ledger managed by an open network of nodes. Each block of the ledger contains a set of valid transactions that transfer assets among end-users.

Nodes in a permissionless blockchain are either end-users or miners. While end-users have only the application layer, the architecture of miners consist of the consensus and storage layers. Note that a miner can also be an end-user, thus has all three layers.

6.2.2 Asset Management

From a data point of view, assets in a permissionless blockchains can be modeled using data types, i.e., an asset is represented by currency units and its ownership. Transactions, on the other hand, transfer the ownership of assets, i.e., move some currency units from one user identity to another user identity.

The ownership information of assets is stored in the storage layer. The owner of an asset is determined using identities that are implemented using public keys. A coin that is linked to a user's public key is owned by that user.

Transactions transfer the ownership of an asset from one identity to another. A transaction is basically a digital signature. End-users, in the application layer, use their private keys [186] to digitally sign assets linked to their identity to transfer these assets to other identities, identified by their public keys. These digital signatures are submitted to the consensus layer via message passing through a client library. It is the responsibility of the miners to validate that end-users can transact only on their own assets. If an end-user digitally signs an asset that is not owned by this end-user, the resulting transaction is not valid and is rejected by the miners. In addition, miners validate that an asset cannot be spent twice and hence prevent double spending of assets. Using transactions, an asset can be tracked from its registration in the blockchain, the first owner, to its latest owner in the blockchain. Transactions are stored in the blockchain in the storage layer.

Registration and divisibility are two other aspects of asset management. In bitcoin and many other cryptocurrencies, new coins are generated and registered in the blockchain through mining. In fact, once a miner solves the puzzle, it is allowed to generate some amount of coin as a mining reward.

Assets can be split or merged using transactions. Each transaction takes one or more input assets owned by one identity and outputs one or more assets where each output asset is owned by one identity. Indeed, a transaction references previous transaction outputs as new transaction inputs and dedicates all input coin values to new outputs. The summation of a transaction's input assets matches the summation of its output assets assuming that no transaction fees are imposed.

Figure 6.2 shows an example of three transactions A, B, and C in the Bitcoin blockchain. As can be seen, in transaction A, a user (with address $addr_Q$) transfers 0.4 bitcoins to another user $addr_X$. In Transaction B, $addr_P$ splits 1.1 bitcoins to (1) 0.8 to $addr_R$ and (2) 0.3 to $addr_X$. Finally, in Transaction C, the outputs of transactions A and B that

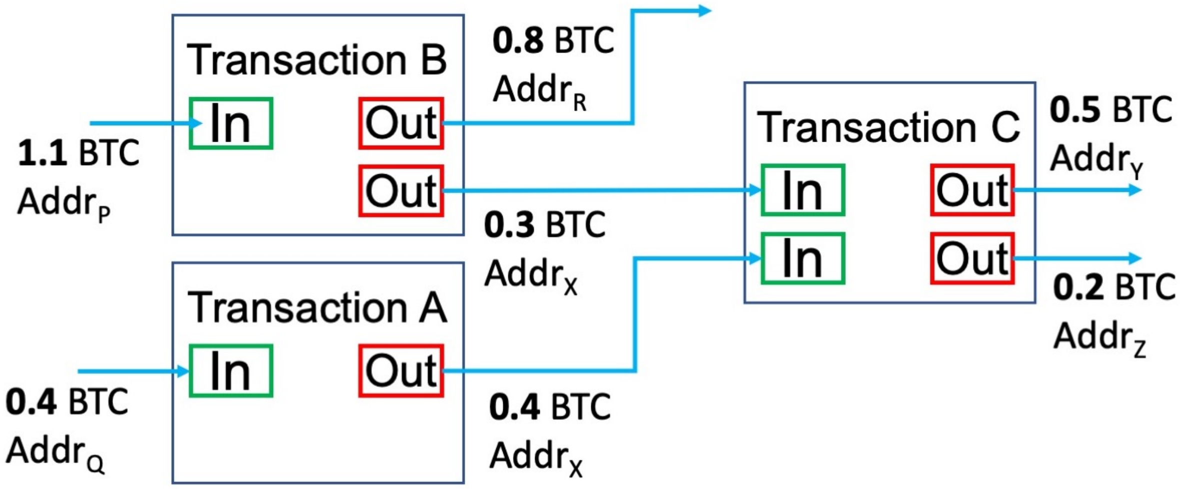


Figure 6.2: Transactions input and output in blockchain

are owned by $addr_X$ (0.4 bitcoins from transaction A and 0.3 bitcoins from transaction B) are merged and then split to 0.5 to $addr_Y$ and 0.2 to $addr_Z$.

In traditional databases, end-user transactions execute arbitrary updates in the storage layer as long as the semantic and the access control rights of a transaction are validated in the application layer. On the other hand, in blockchain systems, this validation is explicitly enforced in the consensus layer and hence end-users, in the application layer, are allowed to transact only on the assets they own in the storage layer. This is in contrast to the database systems model where individual transactions in isolation and in the absence of concurrency are assumed to be *correct*. Indeed, the database concurrency control component provides the guarantee that the interleaved execution of multiple transactions will be equivalent to some serial execution. In the blockchain context, however, the correctness of individual user transactions cannot be assumed due to the absence of a trust model and hence the underlying storage system checks the validity of the user transactions.

Note that this is only feasible due to the restrictive semantics of the currency-based asset model. More complex applications on permissionless blockchain also need to deal with the lack of assumption of the *correct transaction model* that is made in traditional database systems.

6.3 Smart Contracts

Blockchains can be viewed as analogous to object-oriented programming languages. Consider permissionless blockchains where each blockchain consists of primitive data types such as an asset represented by currency units, user identities, user accounts, etc, along with primitive functions that are applicable on these primitive data types such as transactions that move some currency units from one user identity to another user identity. To represent a complex asset, analogous to a complex data type, an end-user writes a **smart contract** [64] that represents this complex asset. A smart contract is a program written in some scripting language (e.g., Solidity for Ethereum smart contracts [38]) that allows general program executions on a blockchain's mining nodes. A smart contract can be thought of as a class in an object-oriented programming language. End-users write the specification of the member variables (the state) and the member functions (the state transitions) of this class in the smart contract code. For example, Alice can write a smart contract that represents her ownership of a car. The member variables of this smart contract could include the car attributes (e.g., make, model, year, the VIN), the car owner (Alice's public key), and the sell price of the car (e.g., 10 bitcoins). The member functions could include a buy function that allows Alice to move the ownership of the car to another end-user if this end-user pays Alice the car price through a buy function call.

After an end-user writes the description of the smart contract class, the end-user deploys the smart contract on a blockchain through a *deployment message* that is sent to the mining nodes in the consensus layer. The deployment message includes the smart contract code. Deploying a smart contract on the blockchain instantiates an object [118, 86] of the smart contract class and stores this object in the blockchain. This object has a state, a constructor that is called when a smart contract is first deployed on the blockchain, and a set of functions that could alter the state of this object. The constructor initializes the object state. To alter the state of the object, end-users call smart contract functions via *function call messages*. End-users send function call messages to the mining nodes accompanied by the function parameters to the blockchain mining nodes. Miners execute the function on the current contract state and record any contract state transitions in their current block in the blockchain. Therefore, a smart contract state might span many blocks after the block where the smart contract is first deployed. The deployment message is a special case of a function call message that includes the smart contract code and results in executing the constructor of this smart contract. End-users pay a fee to the mining nodes for every function call message, including the deployment message, to incentivize the mining nodes to execute this function and record the state transitions of the smart contract object in their current block.

Every function call message, msg, includes some implicit parameters that are passed in the message and are accessible by the function code. These parameters include the sender end-user public key, accessed through $msg.sender$, and an optional asset value, accessed through $msg.val$. This optional asset value allows end-users to use their assets, in currency units, in the smart contract functions. For example, Alice might deploy a smart contract that locks 10 ethers of hers in the contract, passed in the deployment message, and conditionally transfers these 10 ethers to Bob if Bob solves some puzzle that is written in the contract. Another example is the car ownership transfer where Bob passes 10 bitcoins of his in the buy function call of Alice's smart contract in order to buy Alice's car. Note that miners have to verify that end-users who pass an asset value in a smart contract function call must own this asset value and they cannot double spend this asset value in another smart contract function call or another implicit transaction.

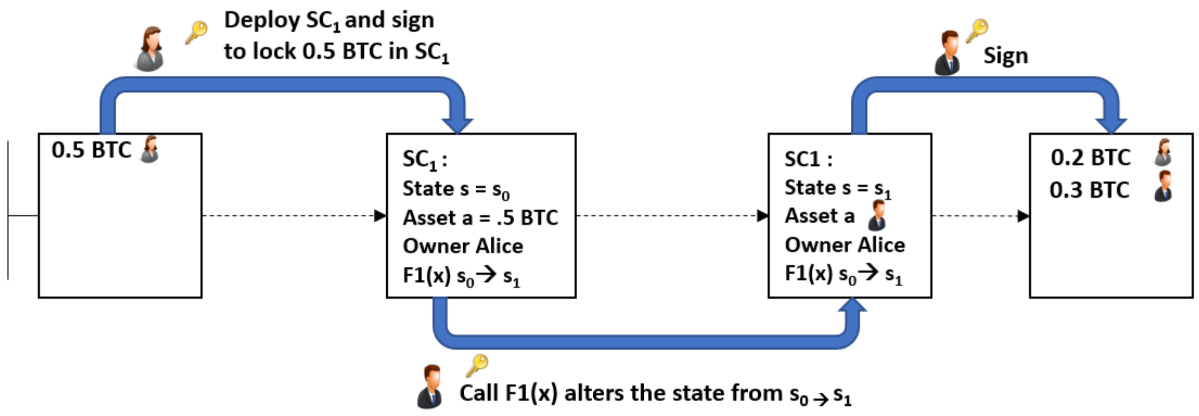

Deploy SC1 and sign to lock 0.5 BTC in SC1
Sign
0.5 BTC
SC1 : State s = s0 Asset a = .5 BTC Owner Alice F1(x) s0 → s1
SC1 : State s = s1 Asset a Owner Alice F1(x) s0 → s1
0.2 BTC
0.3 BTC
Call F1(x) alters the state from s0 → s1

Figure 6.3: Smart contract state can span multiple blocks in the blockchain.

Figure 6.3 illustrates a smart contract example where the smart contract state spans multiple blocks in the blockchain. As shown, Alice deploys smart contract SC_1 on the Bitcoin blockchains. Along with the deployment message, Alice passes her 0.5 bitcoins signed to be locked in SC_1. This locking moves the ownership of the 0.5 bitcoins from Alice to SC_1. SC_1 has a state variable $s = s_0$, an asset a (0.5 BTC), and an owner *Alice*. SC_1 has a function $F1(x)$ that transfers the ownership of the asset a to any caller who provides a valid parameter x according $F1$'s logic. Also, $F1(x)$ transfers SC_1 state variable s from s_0 to s_1. When Bob calls $F1(x)$ providing a valid parameter x, the mining nodes execute this function call and record all the smart contract state transitions in their current block. As shown, after Bob calls $F1(x)$, the contract's variable s is set to s_1 and the asset a, 0.5 bitcoin, is moved to Bob. Bob can spend the transferred asset via transactions in the following blocks. In Figure 6.3, the ownership of the 0.5 bitcoins is moved from Alice to SC_1 through the deployment message, from SC_1 to Bob through the $F1$ function call, and finally is split among Alice (0.2 BTC) and Bob (0.3 BTC) via a Bitcoin transaction.

Algorithm 12 shows an example of a smart contract to register a car as a complex

Algorithm 12 Smart contract that represents a car as a complex asset

```
class CarSmartContract
 1: String make                                  ▷ the make of the car
 2: String model                                 ▷ the model of the car
 3: Integer year                                 ▷ the manufacture year of the car
 4: Double p                                     ▷ the price of the car
 5: Address o                                    ▷ the public key of the owner
 6: procedure CONSTRUCTOR(String make, String model, Integer year, Double p)
 7:     this.make = make
 8:     this.model = model
 9:     this.year = year
10:     this.p = p
11:     this.o = $msg.sender$
12: end procedure
13: procedure BUY(Address curOwner)
14:     requires($msg.val \geq$ this.p and curOwner == this.o)
15:     transfer $msg.val$ to this.o
16:     this.o = $msg.sender$
17: end procedure
18: procedure UPDATEPRICE(Double p)
19:     requires($msg.sender$ == this.o)
20:     this.p = p
21: end procedure
```

asset. The member variable (Lines 1 – 5) represent the attributes of the car. The constructor (Line 6) initializes the car object with the attribute values passed in the deployment message (e.g., make, model, year, and price). In addition, the constructor uses the implicit parameter msg.sender to initialize the car owner (Line 11). The smart contract has two other functions: Buy (Line 13) and UpdatePrice (Line 18). The Buy function allows other end-users to buy the car asset. An end-user who wants to buy the car sends a Buy function call message accompanied by the implicit parameters msg.sender and msg.val in addition to, an explicit parameter curOwner that includes the address of the current car owner. msg.sender determines the identity of the end-user who wants to buy the car and msg.val determines the value in currency units that the end-user wants to pay for the car. curOwner determines the current owner of the asset from the perspective of the function call request. This is necessary to prevent concurrent Buy requests from buying the same asset. Assume *two* concurrent Buy function calls that are submitted with the same curOwner value. If one Buy request succeeds, the owner of the asset will be altered as a result. Therefore, the other Buy request will fail. The Buy function requires msg.val to be greater than or equal to the car price and curOwner to be equal to the current car owner (Line 14). If true, msg.val is transferred to the current owner and the ownership of the car is transferred to msg.sender. However, if the *requires* instruction fails, the function execution is terminated and the transfers do not take place. Finally, the function UpdatePrice allows only the current owner of the car to update its price.

Although smart contracts are powerful tools to represent the attributes and the functionality of complex assets in permissionless blockchain, registering complex assets via smart contract deployments faces several challenges including *the authentication*, *double spending*, and *legality*.

The authentication challenge. “How can end-users *authenticate the registered asset* and ensure its existence?”. For example, if Alice registers her car title in the Bitcoin blockchain, “how could Bob who wants to buy this car authenticate that this car physically exists and that Alice is not maliciously registering a car that does not exist?”.

The double spending challenge. “How can the blockchain system prevent a malicious end-user from *registering the same asset in two smart contracts* within the same permissionless blockchain or in different permissionless blockchains?”. In the previous example, even if Bob could magically authenticate Alice’s car smart contract, ”how could Bob ensure that this is the only smart contract that Alice deployed to register her car in a permissionless blockchain?”.

The legality challenge. “How can end-users ensure that this asset transfer is *legally allowed by State law* where this transfer takes place?”. This challenge addresses the State laws including the taxation law. Transferring the ownership of a car requires the buyer to pay a transfer taxes to the State according to the State law.

Our proposal in Section 6.5 addresses these challenges by unifying both permissioned and permissionless blockchains. This unification allows end-users to use the infrastructure of permissionless blockchains to trade their assets without violating State laws while preventing double spending and trading unauthenticated assets.

6.4 Permissioned Blockchains

In a blockchain, nodes agree on their shared states across a network of participants. Existing blockchain systems can be divided into two main categories of permissionless and permissioned blockchains. While *Permissionless* blockchains are public and any computing node can participate in maintaining the blockchain ledger, *permissioned* blockchain consists of a set of known and identified nodes that do not fully trust each other.

Blockchain was originally devised for Bitcoin cryptocurrency [164], however, recent systems focus on its unique features such as transparency, provenance, fault-tolerant, and authenticity to deploy a wide range of distributed applications such as supply chain management, IoT, and healthcare in a permissioned settings.

6.4.1 Architecture Overview

The architecture of a permissioned blockchain consists of *Application layer*, *Consensus layer*, and *Storage layer*. The application layer of a permissioned blockchain, similar to permissionless blockchains, consists of end-users who submit their transactions to the blockchain through a client library. However the consensus layer which is mainly responsible for ordering and validating the transactions differs from the consensus layer in permissionless blockchains. In fact, since the nodes in a permissioned blockchain are

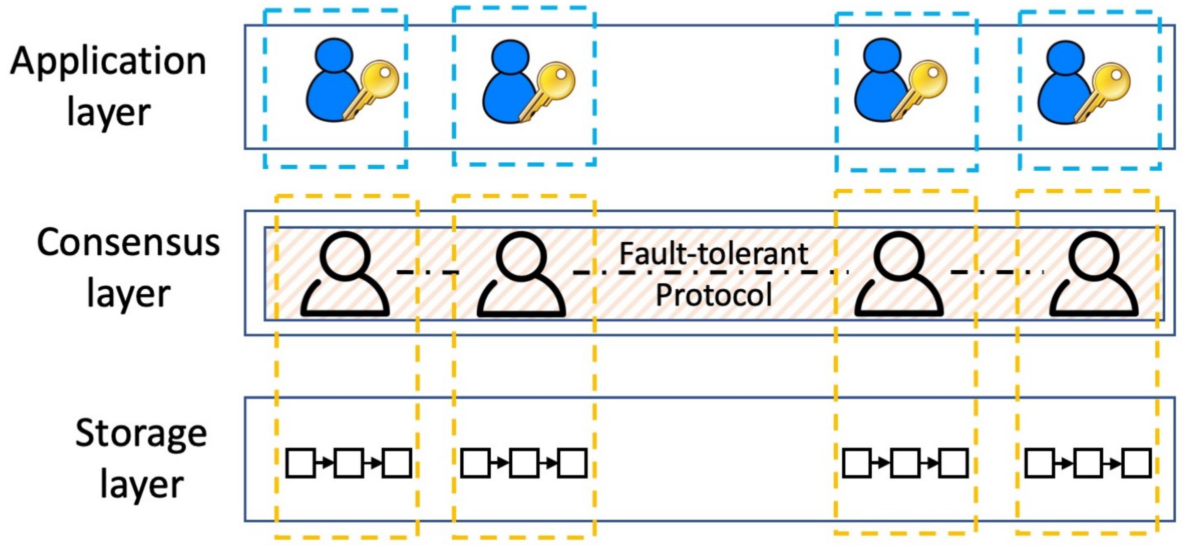


Figure 6.4: Permissioned Blockchain Architecture Overview

known and identified, mining can be replaced with traditional consensus protocols in order to establish a total order on the requests [65]. Finally, the storage layer, similar to permissionless blockchains, consists of a decentralized distributed ledger maintained by every node within the blockchain.

The consensus layer runs a consesus protocol among the computing nodes of the consensus layer. Consensus protocols employ State Machine Replication (SMR) technique to replicates data, e.g. ledger, over nodes. State machine replication is a technique for implementing a fault-tolerant service by replicating servers [139]. In the state machine replication model replicas agree on an ordering of incoming requests.

To establish consensus among the nodes in a permissioned blockchain, asynchronous fault-tolerant protocols can be used. Nodes in a permissioned blockchain might crash or maliciously behave. In a crash failure model, nodes operate at arbitrary speed, may fail by stopping, and may restart, however, they may not collude, lie, or otherwise, attempt to subvert the protocol. Whereas, in a Byzantine failure model, faulty nodes may exhibit arbitrary, potentially malicious, behavior.

Crash fault-tolerant protocols guarantee safety in an asynchronous network using $2f{+}1$ nodes to overcome the simultaneous crash failure of any f nodes while in Byzantine fault-tolerant protocols, $3f{+}1$ nodes are usually needed to provide the safety property in the presence of f malicious nodes.

Permissioned blockchain mainly follow an order-execute paradigm where a set of peers (might be all of them) validates the transactions, agrees on a total order for the transactions, puts them into blocks and multicasts them to all the nodes. Each node then validates the block, executes the transactions using a "smart contract", and updates the ledger.

6.4.2 Data Management

The permissioned blockchain systems are distinguished from the permissionless blockchain systems in one critical way: although there is in general a lack of trust among entities, all entities or components in the system are completely identified. The identified storage nodes in the permissioned blockchains can come together to allow a much more general-purpose data model then that is stipulated in the permissionless system. Thus, Permissioned blockchains can be used for different distributed applications. In the same vein, since end-users of permissioned systems have known identities, we can enforce the *correct transaction computation* assumption from the database systems. This allows the transaction models in permissioned system to be more general than the transaction model in permissionless blockchains where each transaction mainly transfers the ownership of assets, i.e., cryptocurrencies, from one identity to another. In a permissioned blockchain depending on the application, different types of transactions can be defined. For example, a Supply Chain Management includes different processes such as farming, refining, design, manufacturing, packaging, and transportation. As a result, to support a

Supply Chain Management system, the permissioned blockchain should be able to record all transactions within these different processes.

To summarize, permissionless systems are completely open and public and therefore do not have the notion of identity and trust. In effect, these systems have to withstand malicious behavior at all levels: at the level of an end-user, at the level of a network node who is miner, as well as other nodes that try to compromise the sanctity of the system. End-users, consensus nodes, and storage nodes in the permissioned system, on the other hand, all have known identities. The lack of trust is primarily because of two possibilities. If the permissioned system belongs to a single enterprise, the consensus and storage nodes may be stored at different infrastructure providers. The source of maliciousness in this setting may arise if one or more of the infrastructures are compromised. Alternatively, the permissioned system may be a designed to facilitate cooperation among multiple enterprises. The source of maliciousness in this setting may arise due to the competition among these cooperating entities.

6.5 Global Asset Management System

This section proposes a global asset management system that leverages permissioned blockchains to address the *authentication*, the *double spending*, and the *legality* challenges of using smart contract to represent complex assets in permissionless blockchains. Governmental offices deploy their own permissioned blockchains. End-users request from a governmental office to register their assets in a smart contract in some permissionless blockchain. End-users pay a registration fee to the governmental office for this registration. The governmental office checks if this asset has not been previously registered in any permissionless blockchain smart contract. This check is necessary to ensure that end-users cannot *double spend* their assets through several smart contracts. If true, the governmental office issues an *authenticated* smart contract to deploy on the permissionless blockchain included in the registration request. The governmental office encodes the legal laws, including the taxation law, in the terms of the smart contract. Afterwards, the governmental office deploys the smart contract on behalf of the end-user. The smart contract, owned by the governmental office identity, registers an asset, owned by the end-user identity, and allows the end-user to trade the asset in the permissionless blockchain while preserving the legal rights of the governmental office. For example, the California DMV office deploys a car registration permissioned blockchains. When Alice wants to register her car in the Ethereum blockchain, she requests a smart contract registration of her car in the Ethereum blockchain from the DMV office. The DMV office issues this smart contract stating that any transfer of ownership of this car should pay the governmental office some tax percentage, say 10%, from the car price. Alice cannot double spend her car as there exists only one smart contract that represents Alice's car in any permissionless blockchain. Now, if Bob wants to buy Alice's car, Bob first checks that this smart contract is authenticated by the governmental office identity to ensure the authenticity of the car in the smart contract. If true, Bob can buy the car by submitting a Buy function call request to the mining nodes of the permissionless blockchain. This request is accompanied by Bob's currency units that he wants to pay for the car in the implicit parameter $msg.val$. If the Buy function call succeeds, Alice gets paid in currency units, the governmental office gets paid a tax in currency units, and the ownership of the car is transferred to Bob.

This proposal simplifies the process of trading assets by leveraging the permissionless blockchain infrastructure. Once an asset is registered in a smart contract, trading this asset among end-users is as simple as a permissionless blockchain transaction. End-users are motivated to register their assets as this registration offers them an elimination of the bureaucratic process needed to trade their assets. Governmental offices are motivated to participate by running a permissioned blockchain as it offers them automation and transparency.

In Section 6.5.1, we present the architecture overview of the permissioned and permissionless blockchain unification proposal. Then, we explain the transaction model of the registered assets in permissionless blockchain in Section 6.5.2. We present a car registration smart contract example in Section 6.5.3. Finally, we discuss alternative asset management models in Section 6.5.4.

6.5.1 Architecture Overview

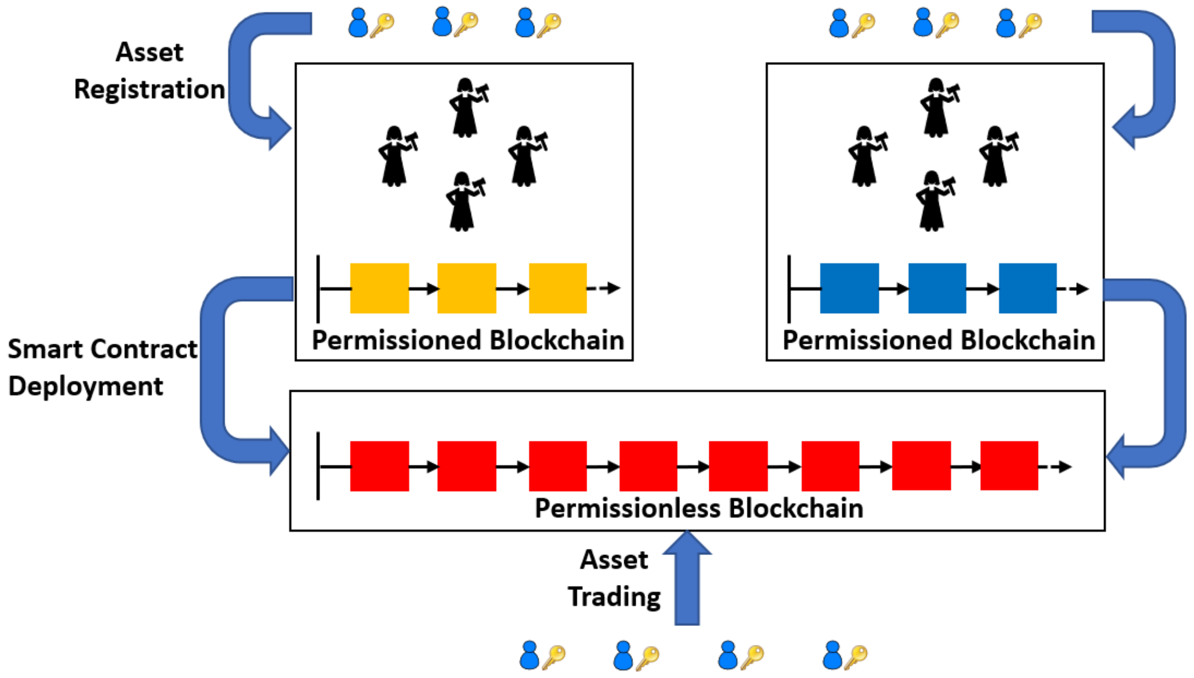


Figure 6.5: Architecture overview of the permissioned and permissionless blockchain unification proposal.

Figure 6.5 illustrates the architecture overview of the permissioned and permissionless blockchain unification proposal. Governmental offices run their trusted asset registration systems. These trusted asset registration systems could be as simple as a database management systems. Governmental offices run permissioned blockchains with a set of trusted governmental officials, called *validators*. Such governmental officials might fail, e.g., the identity of a governmental official gets stolen. Depending on the failure model of the validators, they run a crash fault-tolerant, e.g., Paxos [140] or a Byzantine fault-tolerant, e.g., PBFT [67] consensus protocol to agree on the registered assets.

An end-user sends an *asset registration* request to the permssioned blockchain validators. Validators run a consensus protocol among themselves to ensure that this asset is not previously registered. Once the consensus is achieved, each validator executes the request using some predetermined smart contract. The smart contract generates another smart contract representing the registered asset. To ensure deterministic execution of transactions, as mentioned in Section 6.3, smart contracts are written in scripting languages like Solidity.

Validators then add the asset registration record in their permissioned blockchain. In addition, they authenticate the deployment of the resulted smart contract in a permissionless blockchain. This smart contract is owned by a multi-signature address of the validators. In addition, the asset in the smart contract is owned by the end-user identity. Once the smart contract is deployed on the permissionless blockchain, end-users can trade assets through smart contract function calls.

As shown in Figure 6.5, different governmental offices can use the same permissionless blockchain to deploy their asset registration smart contracts on. Also, a governmental office can use multiple permissionless blockchains to deploy their smart contracts on. For example, both car and house registration offices can use the Ethereum blockchain to register cars and houses. Also, the car registration office can register some cars in the Ethereum blockchain while registering other cars in the Bitcoin blockchain. Once assets are registered in a permissionless blockchains, end-users can transact over these assets as explained next in Section 6.5.2.

6.5.2 Transaction Model

Registering complex assets in permissionless blockchains extends the transaction model of these blockchain. We divide the supported transactions into *four* categories as described below.

Currency units transactions. These transactions are the primitive built-in supported transactions that allow end-users to transfer the ownership of currency units among end-user identities. In addition, these transactions allow end-user to split and merge currency units as explained in Section 6.2.

Complex asset to currency units, of the same blockchain, transactions. These transactions allow end-users to trade complex assets for currency units of the same blockchain where the complex asset is registered. These transactions are allowed through smart contract function calls. Smart contract classes of complex assets include the trading functionalities of these complex assets. For example, an end-user who wants to buy a complex asset calls the Buy function of the smart contract of this complex asset. This Buy function call is accompanied with end-user's currency units. The Buy function transfers the currency units to the current owner of the complex asset and transfers the ownership of the complex asset to the Buy function caller.

Complex asset to currency units, of another blockchain, transactions. These transactions allow end-users to trade complex assets for currency units of a different blockchain from the one where the complex asset is registered. These transactions are enabled by atomic cross-chain swap protocols [117, 169, 212]. Also, these protocols require the smart contracts of complex assets to support the functionality of atomic cross-chain transactions. For example, an atomic cross-chain transaction could allow Alice to sell her car, registered in the Bitcoin blockchain, to Bob who owns ether currency units in the Ethereum blockchain. An atomic cross-chain commitment protocol must guarantee that either both the transfer of Alice's car to Bob in the Bitcoin blockchain and the transfer of Bob's ether to Alice in the Ethereum blockchain take place or none of these two transfers takes place.

Complex asset to complex asset transactions. These transactions allow end-users to swap complex assets within the same permissionless blockchain or across permissionless blockchains. For example, Alice might want to exchange her car, registered in the Bitcoin blockchain, with Bob's boat, registered in the Ethereum blockchain. These transactions use atomic cross-chain swap protocols and require the smart contracts of complex assets to support the functionality of atomic cross-chain transactions.

6.5.3 Car Smart Contract Example

This section presents an authenticated smart contract example that represents a car as a complex asset. This example illustrates the necessary updates to the smart contract presented in Algorithm 12 in order to ensure the *authenticity* and the *legality* of the car registration in a permissionless blockchain. These updates are reflected in Algorithm 13.

The member variable (Lines 1 – 7) represent the attributes of the car. As shown, the smart contract itself is owned by the validators multi-signature address (Line 6). In addition, the car itself, as a complex asset example, is owned by an end-user (Line 5).

The constructor (Line 8) initializes the car object with the attribute values passed in the deployment message (e.g., make, model, year, price, tax percentage, and the owner's public key). In addition, the constructor uses the implicit parameter *msg.sender* to initialize the smart contract owner (Line 15).

The smart contract has two functions to manipulate the car asset: Buy (Line 17) and UpdatePrice (Line 23). In addition, the smart contract has two functions to manipulate the smart contract itself: UpdateContractOwner (Line 27) and DestroyContract (Line 31).

Algorithm 13 Authenticated smart contract that represents a car as a complex asset

```
class CarSmartContract
 1: String make                                         ▷ the make of the car
 2: String model                                        ▷ the model of the car
 3: Integer year                                        ▷ the manufacture year of the car
 4: Double p                                            ▷ the price of the car (currency units)
 5: Address o                                           ▷ the public key of the car owner
 6: Address co  ▷ the contract owner represented by a multisignature address
    of the validators
 7: Double tp                                           ▷ the sales tax percentage
 8: procedure CONSTRUCTOR(String make, String model, Integer year, Double p, Double
    tp, Address o)
 9:     this.make = make
10:     this.model = model
11:     this.year = year
12:     this.p = p
13:     this.o = o
14:     this.tp = tp
15:     this.co = $msg.sender$
16: end procedure
17: procedure BUY(Address curOwner)
18:     requires($msg.val \geq this.p \cdot (1 + \frac{this.tp}{100})$ and curOwner == this.o)
19:     transfer $msg.val \cdot (1 - \frac{this.tp}{100})$ to this.o
20:     transfer $msg.val \cdot \frac{this.tp}{100}$ to this.co
21:     this.o = $msg.sender$
22: end procedure
23: procedure UPDATEPRICE(Double p)
24:     requires($msg.sender$ == this.o)
25:     this.p = p
26: end procedure
27: procedure UPDATECONTRACTOWNER(Address co)
28:     requires(validate-multisig($msg.sender$, this.co))
29:     this.co = co
30: end procedure
31: procedure DESTROYCONTRACT
32:     requires(validate-multisig($msg.sender$, this.co))
33:     destruct-contract()
34: end procedure
```

The Buy function is slightly different from the Buy function of the smart contract in Algorithm 12. The main difference is that the validators of the permissioned blockchain embed the taxation law in the code of the Buy function. When a Buy function call is received by the permissionless blockchain mining nodes, they verify that value of the currency units sent in *msg.val* is greater than or equal to the sum of the car price and the sales tax value of this car. If true, the car price is sent to the current car owner, the tax value is sent to the contract owner (the validators' multi-signature address), and the ownership of the car is transferred to *msg.sender*. The UpdatePrice function is the same as the UpdatePrice function in Algorithm 12.

The UpdateContractOwner function allows the validators to change the ownership of the contract to another multi-signature address. This function is necessary to alter the contract ownership in case a validator's identity is stolen. Validators replace the current multi-signature address that includes a stolen identity by a newly generated multi-signature address that excludes the stolen identity. The DestroyContract function allows the validators to destroy the smart contract object.

6.5.4 Alternative Asset Management Model

The proposed global asset management system leverages a permissioned blockchain only in the registration process of complex assets in a permissionless blockchain. Once an asset is registered, the permissionless blockchain has the only record of the current ownership of the asset in the asset's smart contract object. In addition, the permissionless blockchain is the only marketplace where this asset is traded. Of course, the asset can be traded for other assets and currency units in other permissionless blockchain through atomic swaps. However, the asset object indefinitely remains in the same permissionless blockchain from its registration time until the asset's smart contract object is explicitly destroyed.

An alternative model can unify permissioned and permissionless blockchains as follows. A permissioned blockchain maintains the ownership record of an asset. If the asset owner wants to trade it for some assets or currency units of some permissionless blockchain, the owner requests to register this asset in a trading smart contract in the permissionless blockchain. The permissionless blockchain only acts as the marketplace to trade assets. After registering the asset, end-users can complete the trade through single-chain or cross-chain transactions. Once the trade is completed, the ownership is updated in the permissioned blockchain and the trading smart contract object is destroyed. This model separates the ownership storing platform, the permissioned blockchain, from the trading platform, the permissionless blockchain.

6.6 Challenges

Our proposal of unifying permissioned and permissionless blockchains to create a global asset management system faces many challenges. First, the *scalability* of the global asset management system is bounded by the scalability of the underlying permissionless blockchain. Current permissionless blockchains are not scalable (e.g., Bitcoin blockchain processes 3∼7 transactions per second [155]). As a result, the scalability of the global asset management system could be limited. We address the scalability challenge in Section 6.6.1. The second challenge is *validator identity theft.* If the identity of some validators of the permissioned blockchain are stolen, the stolen identities can be used to destroy currently deployed smart contracts in addition to authenticating smart contracts of assets that do not exist. The problem of validator identity theft is addressed in Section 6.6.2. Finally, we address the *asset registration flexibility* challenge. Our current model allows a complex asset to be registered in only one permissionless blockchain at a time. We discuss open research challenges that arise from allowing a complex asset to be concurrently registered and marketed at several permissionless blockchains in Section 6.6.3.

6.6.1 The Scalability Challenge

The global asset management system requires the governmental offices to register assets in permissionless blockchains through smart contract deployment. Registered assets are traded through smart contract function calls that result in transactions in the underlying permissionless blockchain. Although, the scalability, represented by the number of executed transactions per second (TPS), of every individual permissionless blockchain is limited, the global asset management system can scale. Each governmental office can use multiple permissionless blockchains to register different end-user assets. Therefore, the scalability of the asset management system is not bounded by the scalability of an individual permissionless blockchain. Instead, the TPS of the asset management system can scale up to the aggregated TPS of all the permissionless blockchains used in registering the assets. For example, if the Bitcoin blockchain executes up to 7 TPS and the Ethereum blockchain executes up to 25 TPS, an asset management system that register assets in both Bitcoin and Ethereun blockchains can scale up to 32 TPS. Using additional permissionless networks to register assets increases the overall TPS of the asset management system.

Other permissionless blockchain scaling techniques can be used to scale the global asset management system. One technique is *sharding* [63]. A permissionless blockchain is partitioned into multiple shards and each shard is maintained by some mining nodes. Transactions that span one shard are handled by the mining nodes of this shard. Transactions that span multiple shards are handled by the mining nodes of these multiple shards and coordinated by an atomic swap protocol [169, 117, 21, 212]. Another technique to scale permissionless blockchains is off-chain transactions. Lightning networks [180] can be used to execute complex assets to currency transactions. The question of how to ensure the correctness and tolerate maliciousness in off-chain complex assets to currency transactions remains an open research question.

6.6.2 Validator Identity Theft Challenge

An important challenge of unifying permissioned blockchains and permissionless blockchains is *trust*. Permissionless blockchains by design are trust-free and they only assume that some percentage of the computing power (51% for Bitcoin) or of the stake owners are honest and correct. On the other hand, permissioned blockchains depend on a known set of trusted identities, the validators. If the validator failure model include byzantine failures, typically the number of validators is set to $3f + 1$ where f is the number of validators that can maliciously fail. For example, a permissioned blockchain with four validators can tolerate a malicious failure of one validator. Trusting the validators of the permissioned blockchain is necessary to trust the authentication of smart contract that represents complex assets in the permissionless blockchain. However, the identity theft of more than f validators could result in authenticating the registration of non existing assets and destroying the smart contract objects of existing assets.

To address this challenge, the standard technique of key rotation [159, 129] can be used to limit the damage that results from validator identity theft. As shown in Figure 6.6, the permissioned network divides the timeline into epochs. For every epoch, a fresh set of validator identities is used to authenticate the smart contracts that register assets

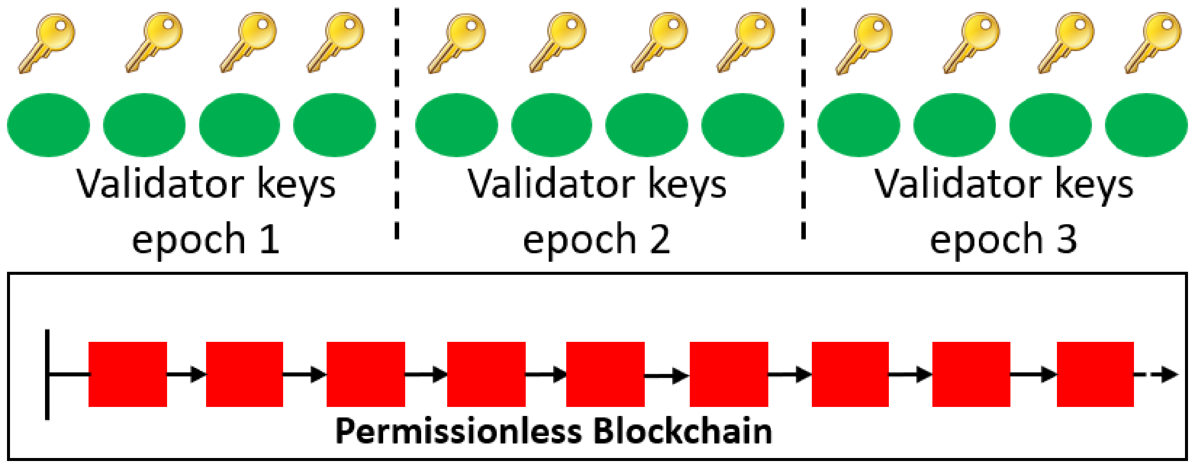


Figure 6.6: Permissioned blockchains use validators key rotation to limit the damage that results from validator identity theft.

in the permissionless blockchain during this epoch. A stolen validator identity set can maliciously register non existing assets only during one epoch. If a validator identity theft is detected within an epoch, the permissioned blockchain can immediately reset the epoch invalidating the stolen validator identity. The question of how to solve the trust problem while achieving authenticity of asset registration remains an open research question.

6.6.3 Asset Registration Flexibility Challenge

The proposed global asset management system requires to limit the number of permissionless blockchain where an asset is registered to one at a time. This requirement is necessary to prevent the *double spending* of the one asset on different blockchains. An asset can be registered on one permissionless blockchain and if the current asset owner wants to change the registration blockchain, an owner has to request a contract cancellation from the validators of the permissioned blockchain. After the contract is cancelled, the owner needs to request the registration of the asset in another permissionless blockchain. Asset owners might want to market their assets on many permissionless blockchains at a time. If this flexibility is allowed, a protocol is required to ensure that once the asset is traded in a smart contract on one blockchain, other smart contracts on other blockchains must be atomically invalidated. This protocol can be thought of as a variation of atomic cross-chain swaps. The design of such protocol and the details of its correctness remain open research questions.

6.7 Related Work

Bitcoin [164] is considered the first successful global scale peer-to-peer cryptocurrency. The Nakamoto consensus protocol used in Bitcoin allows participants to transact with each other without the need for a trusted third party, such as a banks or a credit card company. The ledger that records all the transaction history in traditional trusted banks is replaced by a distributed ledger stored in all the participants in Bitcoin, thus eliminating the need for trusted third parties. Many of the recent works on permissionless blockchains are focused on enhancing one aspect of Bitcoin – the performance limitation. BitcoinNG [94] separates the blocks in the chain into *key-blocks* and *micro-blocks*. Key-blocks are created by solving the proof-of-work challenge and the miner who solved the puzzle becomes the *leader* producing many micro-blocks consisting of transactions. The leader is replaced when another miner mines the next key-block. Thus, by increasing the frequency of micro-blocks produced by a leader, BitcoinNG improves the throughput of Bitcoin. But empowering a single leader to produce micro-blocks entails considerable risks. ByzCoin [131] identifies the benefits of separating the blocks into key and micro blocks, as well as the issues with a single leader. ByzCoin replaces a single leader with a dynamically changing group of *trustees*. Trustees execute PBFT [67] to decide on the next micro-block and use Collective Signing (CoSi) [199] to collectively sign the chosen block. Elastico [152] is another blockchain solution aiming to increase the performance of Bitcoin. The key idea proposed in Elastico is to split all the servers in the system into smaller sized groups called *committees*. Every committee is then assigned with a disjoint set of transactions and the committee members verify those transactions. Each committee executes classical PBFT in order to agree on the set of verified transactions. These transactions are then sent to a *final committee* which is in-charge of aggregating all the transactions produced by different committees into one block and then to broadcast the final block. Thus by allowing different committees to process different *shard* of transactions, Elastico increases the throughput of Bitcoin.

Although the above discussed solutions provide various strategies to increase the performance on Bitcoin, most of the solutions assume a cryptocurrency application. Even if they can be easily extended to include smart contracts, they would still lack in managing global assets. The high churn of participants in a permissionless blockchain network poses impediments is regulating laws associated with global assets.

While Permissionless blockchains are public and anyone can participate without a specific identity, in permissioned blockchains nodes are known and identified. This work uses permissioned blockchains to register global assets and deploy them on permissionless blockchains. Most existing permissioned blockchains follow the order-execute architecture where nodes agree on the order of incoming requests and then execute the requests in the same order. Permissioned blockchains differ mainly in their consensus protocols. Tendermint [136] is different from the original PBFT in two ways, first, only a subset of nodes, called validators, participate in the consensus protocol and second, the leader is changed after the construction of every block (leader rotation). Quorum [70] is an enterprise-focused version of Ethereum [205] developed by JP Morgan. Quorum introduces a consensus protocol based on Raft [173]: a well-known crash fault-tolerant protocol. Chain Core [2], Multichain [113], Hyperledger Iroha [6], and Corda [3] are some other prominent permissioned blockchains that follow order-execute architecture and use varients of Byzantine fault-tolerant protocols.

Fabric [49] introduces the execute-order-validate architecture and leverages parallelism by executing the transactions of different applications simultaneously. Modular design, pluggable fault-tolerant protocol, policy-based endorsement, and non-deterministic execution are some of the main advantages of Fabric. However, it performs poorly on workloads with high-contention, i.e., many *conflicting transactions* in a block. To support conflicting transactions, Parblockchain [46] introduces the order-(parallel)execute architecture where the orderer nodes generate a dependency graph in the ordering phase and transactions are executed in parallel following the generated dependency graph in the execution phase.

Users on the same or different blockchains should be able to initiate transactions in order to exchange assets. Our proposal supports four types of transactions: transactions in currency units, transactions between complex asset and currency units in the same blockchain, transactions between complex asset and currency units in different blockchains, and transactions between two complex assets. Different techniques have been presented to support intra- and cross-chain asset trades. Atomic cross-chain swaps [117] are used for trading assets on two unrelated blockchains. Atomic swaps use hash-lock and time-lock mechanisms to either perform all or none of a cryptographically linked set of transactions. Interledger protocols (ILPV [202]) which are presented by the World Wide Web Consortium (W3C) use a generalization of atomic swaps and enable secure transfers between two blockchain ledgers using escrow transactions. since the redemption of an escrow transaction needs fulfillment of all the terms of an agreement, the transfer is atomic. Lightning network [161][180] also generalizes atomic swap to transfer assets between two different clients via a network of micro-payment channels. Blocknet [83], BTC [61], Xclaim [221], POA Bridge [7] (designed specifically for Ethereum), Wanchain

[8], and Fusion [5] are some other blockchain systems that allow users to transfer assets between two chains.

Hyperledger also addresses atomic cross-chain swap between permissioned blockchains that are deployed on different channels by either assuming the existence of a trusted channel among the participants or using an atomic commit protocol [50][49].

Using sidechain is proposed in [52] to transfer assets from a main blockchain to the sidechain(s) and execute some transactions in the sidechain(s) in order to reduce confirmation time and transaction cost, and support more functionality. Liquid [88], Plasma [179], Sidechains [98], and RSK [142] are some other blockchain systems that use sidechains. In Polkadot [206] and Cosmos [137] also assets can be exchanged using a main chain and a set of (side) blockchains. Both Polkadot and Cosmos rely on byzantine consensus protocol in both sender and receiver sides.

To support global assets in blockchains, using tokens which are backed by external assets, called asset-backed tokens, is proposed [120]. Tokenization is the process of representing the ownership of real world assets digitally on a blockchain. While the main purpose of tokenization is to use tokens as assets (investment instrument) and split it into smaller pieces, this work mainly focuses on how to authenticate an asset as being legitimate so that it can be transacted in a marketplace (i.e., transfer of ownership). In addition, the tokenization of the assets on the blockchain is being done by a known entity (highly centralized) whereas in our proposal the centralized entity is replaced by a governmental permissioned blockchain which first, puts the responsibility of forcing the law on the government, and second, ensures that the centralized entities do not monopolize the tokenization of assets on the blockchain.

6.8 Conclusion

This chapter proposes a global asset management system that leverages both permissioned and permissionless blockchains. Governmental offices maintain trusted permissioned blockchains. Permissioned blockchains *authenticate* the registration of end-user assets in permissionless blockchains through smart contracts. In addition, permissioned blockchains prevent the *double spending* of assets by ensuring that every asset can be registered in only one authenticated smart contract in one permissionless blockchain. Finally, the permissioned blockchain ensure the *legality* of trading the assets by encoding the laws (e.g., taxation law) in the smart contract code. Permissionless blockchains are marketplaces to trade the registered assets. Registered assets can be traded for currency units or other assets on the same permissionless blockchain or on other permissionless blockchain. This extended transaction model is enabled through single-chain and cross-chain transactions.

Chapter 7

Transactional Smart Contracts in Blockchain Systems

7.1 Introduction

Executing concurrent operations has been a long-term challenge in the design of large software systems. Without careful usage of synchronization primitives [87], the concurrent execution of multiple procedures that access shared variables can easily result in anomalous executions. Instead of using synchronization primitives, that a programmer must carefully program, database systems introduced the elegant declarative notion of *transactions* [110]. Programs that may be executed concurrently are each executed as a transaction, and the database management system ensures that transaction execution is isolated from each other and that the concurrent and interleaved execution of multiple transactions is serializable, i.e., equivalent to a serial execution [57].

Recent interest in blockchains has resulted in its rapid usage in diverse applications, and its evolution to support complex concurrent executions. The original blockchain, as proposed in Bitcoin[164], involved simple *transactions*, that transfer some bitcoins from one end-user (typically Alice) to another end-user (typically Bob). The original bitcoin blockchain can be easily modelled as an abstract data type representing a linked list of blocks of transactions. The accessed data is the cryptocurrency, bitcoins, and transactions transfer part of the remaining, unused assets of Alice to Bob, while keeping the rest with Alice (hence the term Unspent Transaction Output, UTXO to refer to the assets belonging to a client in Bitcoin). A *miner* adds a transaction to a block if the assets consumed in the transaction are not double spent in the same block and if the miner can validate that the end-user does actually have these assets, i.e., the UTXO actually belongs to the end-user issuing the transaction. Finally, a miner adds a block to the blockchain if it solves the Proof of Work (PoW) puzzle [164].

Ethereum [205] reintroduced the notion of *smart contracts* [200] to blockchains. Smart contracts extend the simple abstract data type notion of blockchain transactions to include complex data type classes with end-user defined variables and functions. When an end-user deploys a smart contract in a blockchain, this deployment results in instantiating an object instance of the smart contract class in the blockchain [118, 86]. The object state is initially stored in the block where the object is instantiated. End-users can issue a smart contract function call by sending function call requests to the miners of a blockchain. These function calls are transactions that are sent to the *address* of the smart contract object. Miners execute these transactions and record object state changes in their currently mined block. Therefore, the state of a smart contract object could span one or more blocks of a blockchain.

Smart contracts now have their own variables and multiple functions that may be executed by different end-users results in transactions which might be incorporated in different blocks by different miners. This clearly results in complex concurrency challenges which need to be handled by smart contract developers. Distributed database literature [79, 195] has shown that putting the burden of implementing transaction logic in the application layer is problematic. This is no simple task and serious smart contract concurrency bugs have been highlighted in the blockchain literature [132, 151, 193, 86]. In fact, from a financial point-of-view, two such famous anomalies in the context of blockchains, TheDAO [4, 62] and the BlockKing [9] have resulted in a loss of tens of millions of investors' dollars [151].

This work advocates leveraging the traditional transactional approach to address the concurrency violations in the context of smart contract executions in large scale blockchain systems. In particular, we propose Transactional Smart Contracts (TXSC) as a framework that allows developers to write smart contracts with correct transaction isolation semantics. Unlike previous works [132, 151, 193] that propose smart contract analysis tools to detect concurrency bugs in smart contracts, TXSC aims to free smart contract developers from the burden of implementing correct concurrency control semantics for each smart contract. Instead, developers can focus on the smart contract application semantics and leave the concurrency semantics to TXSC.

Concurrency control problems arise in two general contexts during smart contract function execution depending on whether the application semantic functionality is implemented by a single or multiple functions. In a **single function**, each function in a smart contract is executed correctly (and in isolation) as a miner validates its execution. However, the state of the data in the blockchain is visible and can be read all the time by any end-user. An end-user might take action based on a value read, but due to the concurrent execution of smart contract functions, such a read value might be stale when the function is executed. TXSC needs to ensure that the attribute values observed by an end-user, where these attributes are in the read set of a function, are still valid when the function is executed. Alternatively, the semantic functionality might be executed by **multiple functions** in the same or even different smart contracts on potentially different blockchains. These functions might invoke each other in an asynchronous manner. In particular, a function, before termination may call another function to perform a specific task, which in turn calls a third function, and so on. This arises due to smart contracts in a single blockchain like the puzzle example in [151] or across multiple chains [9, 62] that requires atomic execution across blockchains [169, 117, 212]. In this case, different invocations of the function might be interleaved resulting in incorrect executions due to the lack of isolation.

In this chapter, we propose the Transactional Smart Contracts paradigm to solve these concurrency problems. In particular,

1. This work models smart contract concurrency anomalies as transaction isolation problems. Examples illustrate how different smart contract concurrency anomalies can be mapped to the problem of transaction isolation of either single domain or distributed cross-domain transactions.

2. TXSC is the first framework to provide smart contract developers with transactional primitives *start transaction* and *end transaction*. TXSC takes a smart contract that contains these primitives as an input and translates it to a transactionally correct smart contract using the smart contract native language.

The rest of the chapter is organized as follows. We start with two examples to illustrate the types of concurrency anomalies that can arise in the context of smart contracts in Section 7.2. Data and transaction models are presented in Section 7.3. Section 7.4 explains our solution and presents TXSC and the chapter is concluded in Section 7.5.

7.2 Concurrency Anomalies in Smart Contracts

Most of the smart contract anomalies identified in prior work [132, 151, 193, 86] are rooted to faulty transaction isolation semantics implemented by the smart contract developers. These anomalies can be classified into *two* categories: 1) faulty transaction isolation semantics among transactions that span a single administrative domain (or one blockchain) and 2) faulty transaction isolation semantics among distributed transactions that span several administrative domains (more than one blockchain or one blockchain and services outside the domain of this blockchain). We explain the two categories using the following two examples from [151] and [193]. For consistency with the original blockchain terminology, in this section, we refer to a function call request as a transaction (later we will change this).

The puzzle example. This example illustrates the first category of smart contract concurrency anomalies. In this example, an end-user, *the challenger*, deploys a smart contract that pays another end-user, *the solver*, a reward if the solver's submitted puzzle solution is correct. Algorithm 14 shows the puzzle smart contract pseudocode. As shown, the smart contract has three functions: a `Constructor` (Line 6), `UpdateReward` (Line 12), and `SubmitSolution` (Line 19) functions. The `Constructor` is executed by the contract owner, the challenger, to initialize the smart contract object. `UpdateReward` can be executed only by the challenger to update the reward value of the puzzle. Furthermore, `UpdateReward` can only be executed if the puzzle has not been solved yet (Line 14) and `UpdateReward` sends the old reward value to the challenger and updates the reward value with the new value sent by the challenger (Line 16). `SubmitSolution` (Line 19) allows any solver to submit a solution to the puzzle only if the puzzle has not been solved yet. If the submitted solution is correct (Line 21), the reward goes to the solver, the puzzle's solution is updated, and the puzzle is marked as solved.

Now, assume Alice is a challenger who posts a puzzle that follows the smart contract description in Algorithm 14 in the Ethereum network and she sets the reward value r to $r = 2$ ethers, the currency of the Ethereum network. Bob, a solver, reads the reward value $r = 2$ ethers, solves the puzzles, and submits the solution to the smart contract through a transaction TX_1. Bob assumes to receive a puzzle reward of 2 ethers if his solution is correct. Concurrently, Alice might, benignly or maliciously, schedule a transaction TX_2 that updates the reward of the puzzle to a smaller value than the current reward e.g., $r = 0$. If TX_2 is executed first, r would be updated to its new value 0. While updating the reward value should result in aborting TX_1 as the value of r read by TX_1 is stale, the smart contract code in Algorithm 14 would allow TX_1 to execute. This results in Alice receiving a solution to her puzzle while Bob gets a reward of 0 ethers. As both TX_1 and TX_2 access an object that spans only one blockchain, the Ethereum network, this concurrency anomaly falls into the first category of the two aforementioned categories.

The BlockKing [193, 9] example. This example demonstrates the second category of smart contract concurrency anomalies where end-user distributed transactions span several administrative domains (objects of one or more blockchains in addition to asynchronous calls to external services). Algorithm 15 shows code snippets from the original 366 lines of code of the BlockKing smart contract [9] where concurrency anomalies occur. The BlockKing smart contract works as follows. At any moment in time, there exists one block king, initially, the contract owner. Users send money to the contract via the `Enter` function (Line 4) as bids to become the next block king. The `Enter` function stores the address of the caller, the current block number, and the caller's bid value in the attributes `warrior`, `warriorBlock`, and `warriorGold` respectively. Then, the `Enter` function calls an external random number generator to generate a random number between 1-9 and if the returned number equals to the first digit of the block number stored in the `warriorBlock` attribute, the caller of the `Enter` function becomes the new block king. A block king gets a percentage of the bid money of every call to the `Enter` function and the contract owner gets the remaining percentage of this bid money. Notice that the random number generator triggers an asynchronous callback function (Line 10) where the returned random number is checked against the block number in the `warriorBlock`

Algorithm 14 Puzzle smart contract example in [151]

```
class Puzzle
 1: address public owner                          ▷ contract owner
 2: bool public solved                            ▷ true if the puzzle is solved
 3: uint public reward                            ▷ puzzle solving reward
 4: bytes32 public diff                           ▷ puzzle difficulty
 5: byte32 public solution                        ▷ puzzle solution if found
 6: procedure Constructor
 7:     this.owner = msg.sender
 8:     this.reward = msg.value
 9:     this.solved = false
10:     this.diff = bytes32(msg.data)             ▷ set difficulty
11: end procedure
12: procedure UpdateReward
13:     requires($msg.sender == this.owner$)
14:     if ! solved then
15:         transfer reward to owner
16:         reward = msg.value
17:     end if
18: end procedure
19: procedure SubmitSolution
20:     if ! solved then
21:         if sha256(msg.data) ¡ diff then
22:             transfer reward to msg.sender
23:             solution = msg.data
24:             solved = true
25:         end if
26:     end if
27: end procedure
```

attribute. If the returned random number matches the first digit of the block number in the `warriorBlock`, the current warrior becomes the new block king.

If calls to the `Enter` function are blocking; meaning that at most one call to the `Enter` function is allowed until its callback is completed, the smart contract in Algorithm 15 would not have any concurrency anomalies. However, the smart contract in Algorithm 15 is non-blocking. This non-blocking behavior allows many concurrent calls to the `Enter` function to take place. If multiple transactions are concurrently sent to the `Enter` func-

Algorithm 15 Snippets from the BlockKing contract [9]

```
class BlockKing
address public king, warrior
uint public kingBlock, warriorBlock
uint public warriorGold, randomNumber
procedure ENTER
    ...                                                  ▷ check if minimum bet is sent
    warrior = msg.sender, warriorGold = msg.value
    warriorBlock = block.number
    byte32 myid = oraclize_query(0, ”WolframAlpha”, ”random number between 1
and 9”)
end procedure
procedure _CALLBACK(byte32 myid, string result)
    requires(msg.sender == oraclize_cbAddress())
    randomNumber = uint(bytes(result)[0]) - 48;
    if singleDigitBlock == randomNumber then
        ...                                              ▷ update reward
        king = warrior, kingBlock = warriorBlock
    end if
end procedure
```

tion, each transaction would replace the values of the `warrior`, the `warriorBlock`, and the `warriorGold` attributes of all the previous incomplete transactions. This leads to an advantage to the latest caller who sends a transactions to the `Enter` function before all previous callbacks occur. Every trigger to the callback function gives the latest caller a chance to become the new block king while previous callers have no chance to become the new block king. We illustrate this transaction isolation anomaly using the following example. Assume Alice, Bob, and Carol concurrently want to become the next block king. They send three transactions (corresponding to three `Enter` function calls) TX_1, TX_2, and TX_3 accompanied by their bids to the *enter* function respectively. TX_1 updates the warrior attributes to Alice's attributes sent along with TX_1 then, calls the external random number generator. Before TX_1's callback is triggered, TX_2 replaces the warrior attributes with Bob's attributes sent with TX_2 and similarly, TX_3 replaces the warrior attributes with Carol's attributes. When the callbacks of TX_1, TX_2, and TX_3 are triggered, which possibly could take place in another block in the BlockKing blockchain, the three callbacks use the warrior attribute values of Carol to decide if she could be the next block king or not. Carol gets 3 chances to become the block king while Alice and Bob have no chance. This concurrency violation occurs as transactions TX_1, TX_2, and TX_3 are not being executed in isolation.

Transactions in the first category can be atomically executed in one shot within one block of its smart contract blockchain. On the other hand, distributed transactions could span multiple blocks in one or more blockchains and hence ensuring their atomicity while executing them in isolation is significantly more complicated than executing transactions in the first category in isolation.

7.3 Data and Transaction Models

An open permissionless blockchain [155] comprises an application layer and a storage layer. Clients in the application layer have public identities represented by their public keys and private signatures generated using their private keys. Clients send signed transactions to the storage layer in order to transfer assets from one client to another. The storage layer consists of mining or computing nodes, *miners*, and each miner manages a copy of the blockchain. Transactions, in the storage layer, are grouped into blocks and each block is hash chained to the previous block; hence the name *blockchain*. When a mining node receives a transaction, it verifies the transaction and adds it to its current block, *only if the transaction is valid*. Mining nodes run a consensus algorithm or in a permissionless blockchain Proof of Work (PoW) to reach consensus on the next block to be added to the blockchain.

Smart contracts are analogous to classes [118, 86, 215] in Object Oriented Programming Languages (OOPL) and are used by clients to implement complex data types. Clients deploy smart contracts to a blockchain by sending a deployment message to miners of this blockchain. As a result, a miner instantiates an object of the smart contract class and stores this object in the current block in the blockchain. Smart contract objects have attributes that capture their state. Once a smart contract object is instantiated in a blockchain, the state of this object, as part of the blockchain, is made public and can be **externally read by any client at any moment**. In addition, smart contract objects have functions that define the possible state transitions of these objects. Since an object state is public, smart contract read-only functions are pointless. Therefore, it is safe to assume that any smart contract function call has to update at least one attribute of the smart contract object [209]. A smart contract object has an address in the blockchain. When a client wants to issue a smart contract function call, the client sends a function call request to the miners of the blockchain where the smart contract is deployed. This function call request is directed to the address of the smart contract object. Miners use the smart contract address to locate the smart contract object (state and code). This function call is accompanied by some implicit parameters like *msg.sender*, the address of the client who sent the transaction, *msg.val*, the value of the money sent along with the transaction, and *msg.data*, any data that needs to be sent along with the transaction. In addition, function calls could be accompanied by some function explicit parameters.

We follow the Ethereum [205] smart contract execution model. Each function call is accompanied by some *gas* value. The *gas* value represents the amount of money a client is willing to pay to incentivize miners to execute the function call. Miners charge some *gas* for every executed line of code in the called function. A miner stores any intermediate results of a function call in their local storage. If the function call completes before the function call runs out of *gas*, the intermediate results are finalized and included in the miner's current block. However, if a function call runs out of *gas* before the function call is completed, intermediate results are deleted and the smart contract object state does not change. Either way, the miner includes a transaction that pays the miner the amount of *gas* spent during the execution of the function call in its current block. Smart contract function calls are atomic meaning that each function call either terminates after it successfully updates the object state in the blockchain or rolls back to the object state before the call occurs. Concurrent function calls are sequentially executed one after the other without any interruption [193]. In blockchain terminology, a function call request is usually referred to as a transaction. Yet, a function call might not ensure the ACID [57] properties of transactions in traditional databases.

In traditional DBMS, a client transaction starts when a client calls the *start (begin) transaction* command. Afterwards, a transaction reads and updates some data values followed by an *end (commit) transaction* command. The role of the DBMS is to ensure the ACID properties of a client transaction from the moment the transaction begins till the moment the transaction ends (whether the transaction commits or aborts).

In permissionless blockchains, miners have no way to learn the details of all client activities before calling the smart contract functions, e.g., when the client activities start and what values were read before a function call request is sent to the miners. Even when each function call is executed in isolation from concurrent function calls, transaction isolation concurrency violation still occur as shown in Algorithms 14 and 15 as a result of poor client transaction isolation, network asynchrony, and smart contract asynchronous callbacks. We consider a *client transaction span* to include all the read operations that took place before the client sends a function call, the function execution caused by the function call, and any callbacks that are triggered as a result of this function call. The goal of this work is to ensure the ACID properties of client transactions from the time a client starts a transaction till the end of the function call that terminates this transaction.

7.4 Transactional Smart Contracts

Algorithm 16 A smart contract example that uses TXSC

```
class SmartContract
 1: procedure F1
 2:     start transaction
 3:     f1's logic
 4:     end transaction
 5: end procedure
 6: procedure F2
 7:     start transaction
 8:     f2's logic
 9:     end transaction
10: end procedure
```

This section presents TXSC, a framework that allows smart contract developers to write smart contracts with correct client transaction isolation semantics. The goal of TXSC is to provide developers with the primitives *start transaction* and *end transaction.* We call each function surrounded by these primitives, a **transactional** function. TXSC ensures that calls to transactional functions are executed in isolation from any concurrent function calls to the same function or any other function in the smart contract even in the presence of network asynchrony. Algorithm 16 illustrates an example smart contract written using TXSC. This smart contract has two functions F1 and F2 and both functions are transactional functions.

The ACID execution of a client transaction requires *atomic*, *consistent*, *isolated*, and *durable* execution of this client transaction. If the semantics of every smart contract function is correct, function calls should transfer the smart contract object from one consist state to another. Therefore, *consistency* is the responsibility of the smart contract developer. *Durability* of a function call is guaranteed through the blockchain protocol. Function calls that complete execution and are included in a mined block are durable assuming this block gets enough confirmations [23]. Since confirmed blocks are replicated to most of the mining nodes, these blocks are durable even in the presence of failures of many mining nodes. This leaves the responsibility of ensuring atomicity and isolation of client transactions on TXSC.

Isolation: Since smart contract developers have no way to detect which attribute values have been read by the client before a function call request is sent to miners, a smart contract developer has to insert checks at the beginning of every smart contract function call (similar to optimistic concurrency control [135]) to ensure that any data attribute value read by the client and is in the read set of the function call matches its current value in the blockchain. The read set of a smart contract function is the set of attributes that a function reads during its execution. We assume that the outcome of each function is **invariant** to any attribute outside the read set of this function. To ensure serializability [57] of client transactions, the client has to send her observed attribute values of the read set of the function along with the function call. The smart contract has to ensure that the received attribute values are up-to-data and they match the current values of all the attributes in the function read set before executing the function call. Otherwise, the function call has to abort. A function call and all its asynchronous callbacks must be executed in isolation from concurrent function calls and callbacks.

Atomicity: The smart contract code has to guarantee that a function call and all its asynchronous callbacks are atomic. This means that updates that result from a function call and all its asynchronous callbacks should either all take place or none of them do.

TXSC automatically adds transaction isolation checks at the beginning of every transactional function to ensure an isolated execution of every call to any transactional function. TXSC handles the atomicity of single domain transactional functions differently from cross-domain distributed transactional functions as follows.

7.4.1 Single Domain Transactional Functions

A Single Domain Transactional Function (SDTF for short) is a function that reads and updates one or more smart contract objects stored under a single administrative domain or a single blockchain. SDTFs do not access external services or objects outside the domain of their blockchain. As a result, SDTF calls do not trigger any asynchronous callbacks. Any transactional function that accesses external services, blockchains, or trigger callbacks is classified as cross-domain distributed transactional function.

Since all the objects accessed by SDTF calls are stored in a miner's copy of the blockchain and since SDTFs do not trigger asynchronous callbacks, a SDTF call can atomically be executed in one shot. Therefore, the atomicity of a client transaction that calls a SDTF is guaranteed by the smart contract execution model. To ensure a seriablizable execution of a SDTF, the function code has to only ensure the freshness of the read set of this function. TXSC scans every SDTF in a smart contract to determine the object's attributes in the read set of this SDTF. Then, TXSC adds checks at the beginning of the SDTF to ensure that the attribute values observed by the client at the time when the transaction started are equivalent to attribute values when the function call is received by miners.

Recall the puzzle example in Algorithm 14. Both UpdateReward and SubmitSolution are single domain function calls. To convert UpdateReward to a SDTF, TXSC adds a requirement that every function call to the UpdateReward function must be accompanied by the client observed value of the attribute *solved*, in the function read set, in its implicit parameter *msg.data*. Then, TXSC adds a requirement check $solved == msg.data.solved$. If the *solvd* attribute value in a client's UpdateReward function call is stale, the call must abort and the smart contract object state remains unchanged. However, if the *solved* attribute is up-to-date and the function call is also accompanied by sufficient *gas*, the call can be atomically executed and as a result, the reward value is updated.

For the SubmitSolution function call, TXSC adds the requirement checks $solved == msg.data.solved$ and $reward == msg.data.reward$. Recall the concurrency violation of the puzzle smart contract in Section 7.2. When Bob sends his solution to the SubmitSolution function, Bob would send the attribute values $solved = false$ and $reward = 2ethers$ in the $msg.data$ parameter of his function call. When Bob's request is received by a miner, there are two possible outcomes: 1) the function call gets executed only if the current reward value equals to 2 ethers and the puzzle is not solved and 2) the function call aborts if the reward value has been updated in between the time when Bob's transaction started and the time when his function call is received by a miner. Both outcomes do not violate the serializability guarantee.

7.4.2 Cross-Domain Transactional Functions

A Cross-Domain Distributed Transactional Function (CDTF for short) is a function that reads and updates one or more smart contract objects stored under multiple administrative domains or multiple blockchains. In addition, CDTFs can access external services or objects outside the domain of their blockchain. Also, CDTFs may trigger asynchronous callbacks. As a result, updates made by a CDTF can span more than one block of the blockchain. Recall the BlockKing smart contract in Algorithm 15. Each function call first updates the warrior, warriorBlock, and warriorGold in some block and might update the BlockKing in another block when the callback function is trigger. Allowing a CDTF call to update the state of a smart contract object in several blockchain blocks is problematic. If the updates in the first block gets committed in a mined block, committed updates cannot be rolled back even if updates in the following blocks fail due to an exception or that the call runs out of gas. We first explain isolation and atomicity challenges of CDTFs. Afterwards, we explain how TXSC handles CDTFs.

Isolation: A CDTF has an entry point that comprises a function call in an object in some blockchain. This function call might trigger other function calls of objects stored in different blockchains. Since a CDTF can span multiple blockchain objects, sending the read set of all the accessed objects at the entry point (the first function call) is not sufficient to guarantee transaction isolation as in SDFT. Since all subsequent function calls to other objects are trigger over an asynchronous network, the state of these subsequent objects might change in the time between the entry point and the point when the subsequent call is received by the miners of the blockchain where these objects are stored. Even if the read set is carried on with every subsequent call, a stale attribute in the read set might result in aborting a subsequent call. However, the first call might have been committed leading to a violation to atomicity.

Atomicity: Guaranteeing the atomicity of CDTF calls is significantly more complicated than SDFTs. First, atomicity could be violated if one of the subsequent calls to functions in other blockchains runs out of *gas*. Second, if an external service (e.g., the random number generator in the BlockKing example) crashes for a long time or if the message from this external services that triggers the callback function is lost, atomicity can be violated resulting in an inconsistent state (some updates occur in one block but the callback is never triggered to complete the execution of the function call).

This chapter presents a high level solution that guarantees both the isolation and atomicity of CDTFs. The atomic and isolated execution of a CDTF that spans multiple blockchains can be mapped to the problem of atomic cross-chain transaction processing. Atomic cross-chain commitment protocols have been introduced in [169, 119, 117, 212]. First, the solution requires to lock all the object attributes in both the read set and the write set of *all* the functions in a CDTF before calling the entry point. This locking guarantees the isolation of a CDTF from all concurrent function calls to any of the functions that can update either the read set or the write set of a CDTF. However, as shown in [212], using timelocks as proposed in [169, 117] can lead to atomicity violations. The AC^3WN [212] and the CBC [119] protocols that use an additional blockchain as a lock manger are possible solutions to manage the locking of object attributes across blockchains. After all the object attributes are locked, a caller can send a function call request to the entry point accompanied by evidence that all the object attributes in both the read set and the write set of this function call and all subsequent function calls are locked. Object attributes are unlocked only when the function call that accesses them and its corresponding callbacks, if any, terminate. Recall the BlockKing concurrency anomaly in Section 7.2. Alice's call locks the accessed attributes before calling the `Enter` function. This prevents other callers, Bob and Carol, from issuing concurrent function calls to the `Enter` function. Second, economic incentives should be used to enforce callers to accompany function calls with enough *gas*. At the entry point, a caller locks some money in the contract that gets refunded to the caller only if all her function calls terminate. If any function call runs out of gas, the caller loses her locked money to the contract owner who can complete the call and gets the locked objects unlocked. Finally, redo logs can be used to overcome the atomicity violations in the presence of external service crashes. In the BlockKing example, the smart contract object should have an "after-image" attribute corresponding to every attribute in the object. The `Enter` function should update the after-images of warrior, warriorBlock, and warriorGold attributes. When the `callback` is triggered, only then, the after-image attributes can be copied to the actual attributes of the object. This guarantees that even if the external service crashes or the callback trigger is lost, the object is in consistent state.

7.5 TXSC Concluding Remarks

This chapter presents TXSC, a framework that allows developers to write smart contracts with correct transactional semantics. We showed that TXSC can help developers solve isolation anomalies of both single domain and cross-domain distributed transactional functions.

Part III

Conclusion

Chapter 8

Concluding Remarks and Future Directions

It is an exciting time for Global Scale Data Management (GSDM) systems and distributed computing researchers. On one hand, social networks, web, and mobile applications strive to efficiently serve billions of end-users across the globe. These systems are supported by several cloud service providers backed by hundreds of core data-centers and thousands of edge data-centers. The wide usage of current social networks, web, and mobile applications, the outreach of mobile and internet technology to billions of users around the globe, and the high competition among cloud service providers to build more data-centers suggest that building GSDM systems backed by cloud infrastructure is a continuing trend. On another hand, the wide adoption of permissionless open blockchain networks by both industry (e.g., Bitcoin, Ethereum, etc) and academia (e.g., Bzycoin, Elastico, BitcoinNG, Algorand, etc) and the rapid and extensive investment in both permissionless and permissioned blockchains suggest that blockchain technology is here to stay.

In this dissertation, we aspire to provide efficient and decentralized solutions that address several challenges to GSDM systems in both controlled cloud and open blockchain environments. We foresee that the value of our proposed solutions would increase over time as more data-centers are built and as the number of blockchain systems increases.

Controlled cloud environments: as the competition among cloud providers intensifies, many core and edge data-centers will be built. Having hundreds of core data-centers and thousands of edge data-centers built by several cloud providers would make the problem of optimal data replica placement extremely hard on system administrators. This would make GPlacer a necessity for GSDM system to optimize the placement of data replicas among the available data-centers. CoT has several applications in data processing and data retrieval at the edge. As the adoption of Internet-of-Things (IoT) devices increases with many sensors and devices that have limited capabilities at the edge, the need for an elastic cache and a replacement policy that minimizes the needed cache sizes will rise. CoT would be of significant importance for real-time data analytics at the edge where CoT's replacement policy caches the heavy hitter data values at the edge closer to where data processing takes place. The need to build more user-centric systems like Aegis that enable social networks and web users to have fine-control over their personal data privacy would increase. Since online surveillance and tracking are on rise and social network negative effects on society and democracy are increasing, systems like Aegis would help social network and web end-users protect their privacy against several personal data privacy threats.

Open blockchain environments: blockchain technology enables business opportunities among end-users that do not trust each other by leveraging untrusted infrastructure. To envision open blockchains as the next generation public cloud, many primitives and abstractions need to be developed to enable this vision. AC^{3}WN, TXSC, and global asset management system are among these necessary primitives and abstractions. Currently, there are thousands of permissionless blockchains and millions of end-users who own the cryptocurrencies of these blockchains. The need for an atomic cross-chain commitment protocol that does not rely on a trusted coordinator or an exchange is on rise. AC^{3}WN is the first atomic commitment protocol across permissionless blockchains that ensures safety of cross-chain transactions. The value of AC^{3}WN would increase as more end-users and businesses adopt permissionless blockchains. Also, AC^{3}WN is a necessary primitive to enable a generalized transaction model in global asset management systems. A future that has generic assets such as cars and houses stored in permissionless blockchains requires AC^{3}WN to extend the transaction model to support generic asset transactions over several blockchains. Registering generic assets in permissionless blockchains requires a unification of permissioned and permissionless blockchains. Permissioned blockchains are used for the asset verification and permissionless blockchains are used as the assets marketplace. This cannot be achieved without developing primitives that allow smart contract developers to write smart contracts with correct ACID transaction semantics. We believe that TXSC is a step towards achieving this goal.

8.1 Future Directions

The section summarizes some of the future directions that, we believe, are of extreme importance for GSDM in both controlled cloud environments and permissionless blockchain environments.

8.1.1 Data Privacy in Cloud and Blockchain Environments

One of the main challenges that faces both controlled cloud environments and permissionless blockchain environments is data privacy. Data privacy and security are the main hurdles for cloud adoption by enterprises [73]. Several works in Private Information Retrieval (PIR) [184] and Oblivious RAM (ORAM) [106] enable private retrieval and update accesses on outsourced data to the cloud. However, many of these technologies introduce significant performance overheads that make them impractical for GSDM systems. Also, many of these solutions consider only sequential access to the outsourced data and assume reliable system components that do not fail. All these assumptions make the adoption of privacy techniques by cloud-based GSDM systems nonviable. During the development of this thesis, we contributed in TaoStore [189], an ORAM based technique that allows concurrent and asynchronous accesses to the outsourced data while achieving the same privacy guarantees of sequential ORAM proposals. TaoStore is a step towards building efficient ORAM techniques for GSDM systems. However, TaoStore assumes reliable infrastructure. Since cloud infrastructures run on commodity machines where failures are the norm, assuming reliable infrastructure puts the availability of cloud-based GSDM systems at risk. Therefore, addressing the fault-tolerance aspect of these privacy techniques is of extreme importance to their adoption in GSDM systems. We show in [219] that applying traditional replication mechanisms to address fault tolerance in ORAM systems is not a simple task. Since privacy and performance are in tension, several of the efficient replication mechanisms can not be directly leveraged to achieve fault tolerance of ORAM systems without violating their privacy guarantees. Therefore, novel replication techniques need to be developed to achieve efficient fault tolerance of ORAM techniques without imposing additional overheads, a concern that pushes GSDM systems back from using ORAM techniques.

Permissionless blockchains such as Bitcoin and Ethereum are built on transparency. All end-user transactions are publicly stored in the blockchain including the transaction source, destination, and value. Although end-users use pseudonyms to hide their identities and protect the privacy of their spending activities, an increasing body of research [183, 55, 187] has shown that using information of transactions and the transaction graph in the blockchain can result in revealing the identities of transaction identities. This motivated the development of several protocols e.g., [191] to use cryptographic primitives such as zero knowledge proofs to protect the privacy of end-user transactions while guaranteeing the verifiability of transparent blockchains. We believe that such techniques must be extended to protect the privacy of smart contract users, publishers, and function call invocations. Also, several efficient privacy techniques need to be developed to facilitate private data sharing in blockchain systems. This could enable the usage of permissionless blockchains as a marketplace for data sharing where data is privately stored and accessed in the blockchain. Data accesses are granted through smart contract function calls and payments are facilitated through the native cryptocurrency of the blockchain.

8.1.2 Blockchain as the New Public Cloud and Scalability

Permissionless blockchains promise a novel compute paradigm. End-users who do not trust each other can still leverage untrusted infrastructures to do business transactions. This promise enables several business opportunities that do not exist in trusted environments. However, permissionless blockchains such as Bitcoin and Ethereum execute tens of transactions per second [82, 156] whereas trusted, centralized systems such as Visa execute tens of thousands of transactions per second [82]. This scalability limitation represents a hurdle in envisioning permissionless blockchain infrastructures as the new public cloud. Until scalability of permissionless blockchains is achieved, their main use-case remains managing cryptocurrency transactions. One direction to scale permissionless blockchains is sharding. The mining network could be sharded where each shard manages transactions of some end-user partition (horizontal sharding) or transactions over some cryptocurrency unit partition (vertical sharding). Both horizontal and vertical sharding require the abstraction of the atomic cross shard commitment protocol (e.g., AC^{3}WN) to enable transactions that span multiple blockchain shards. AC^{3}WN is an important step towards scaling permissionless blockchains.

Another challenge that faces leveraging permissionless blockchains as the new public cloud is the lack of the right abstractions in blockchain systems. Many abstractions (e.g., concurrency, transaction, atomic commitment, replication, etc) have been extensively studied in trusted environments. However, since the field of blockchain research is young, many of these abstractions are absent in untrusted environments. TXSC is one necessary primitive for permissionless blockchain systems. However, there are several other abstractions that need to be rethought and developed before permissionless blockchains could be used as the new public cloud.

Bibliography

[1] Announcing amazon elastic compute cloud (amazon ec2) - beta. https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/. 2006.

[2] Chain. http://chain.com.

[3] Corda. https://github.com/corda/corda.

[4] The dao (organization). https://en.wikipedia.org/wiki/The_DAO_(organization)/.

[5] Fusion whitepaper: An inclusive cryptofinance platform based on blockchain. https://docs.wixstatic.com/ugd/ 76b9ac_be5c61ff0e3048b3a21456223d542687.pdf.

[6] Hyperledger iroha. https://github.com/hyperledger/iroha.

[7] Poa bridge. https://github.com/poanetwork/token-bridge.

[8] Wanchain: Building super financial markets for the new digital economy. https://www.wanchain.org/files/Wanchain-Whitepaper-EN-version.pdf.

[9] (2016). Blockking contract. https://etherscan.io/address/0x3ad14db4e5a658d8d20f8836deabe9d5286f79e1.

[10] (2016). Db-risk: The game of global database placement [live demo]. http://cs.ucsb.edu/~victorzakhary/demo/demo.html/.

[11] (2016). Facebook, instagram, and twitter provided data access for a surveillance product marketed to target activists of color. https://www.aclunc.org/blog/facebook-instagram-and-twitter-provided-data-access-surveillance-product-marketed-target.

[12] (2016). General data protection regulation. https://gdpr-info.eu/.

[13] (2017). Datacenter map. http://www.datacentermap.com/datacenters.html/.

[14] (2017). GLPK: GNU Linear Programming Kit. https://www.gnu.org/software/glpk/.

[15] (2017). Hbase. https://hbase.apache.org/.

[16] (2017). Russia-linked facebook ads reportedly aimed for swing states. http://fortune.com/2017/10/04/trump-russia-facebook-ads-michigan-wisconsin-swing/.

[17] (2017). Russian-linked facebook ads targeted michigan and wisconsin. https://www.cnn.com/2017/10/03/politics/russian-facebook-ads-michigan-wisconsin/index.html.

[18] (2017). Tor Browser. https://www.torproject.org/.

[19] (2018). Aclu and workers take on facebook for gender discrimination in job ads. https://www.aclu.org/news/aclu-and-workers-take-facebook-gender-discrimination-job-ads.

[20] (2018). Amazon elasticache in-memory data store and cache. https://aws.amazon.com/elasticache/.

[21] (2018). Atomic cross-chain trading. https://en.bitcoin.it/wiki/Atomic_cross-chain_trading.

[22] (2018). Azure redis cache. https://azure.microsoft.com/en-us/services/cache/.

[23] (2018). Bitcoin confirmations. https://www.buybitcoinworldwide.com/confirmations/.

[24] (2018). California consumer privacy act. https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375.

[25] (2018). Cambridge analytica data breach. http://www.businessinsider.com/cambridge-analytica-data-breach-harvested-private-facebook-messages-2018-4.

[26] (2018). Coinbase. https://coinbase.com.

[27] (2018a). Distribution of twitter users worldwide as of october 2018, by gender. https://www.statista.com/statistics/828092/distribution-of-users-on-twitter-worldwide-gender/.

[28] (2018b). Docs – twitter developers. https://developer.twitter.com/en/docs.html.

[29] (2018). Facebook company info. http://newsroom.fb.com/company-info/.

[30] (2018). Facebook pixel. https://www.facebook.com/business/learn/facebook-ads-pixel.

[31] (2018). Facebook will remove 5,000 ad targeting categories to prevent discrimination. https://www.theverge.com/2018/8/21/17764480/facebook-ad-targeting-options-removal-housing-racial-discrimination.

[32] (2018). Memcached. a distributed memory object caching system. https://memcached.org/.

[33] (2018). Monthly active instagram users. https://www.statista.com/statistics/253577/number-of-monthly-active-instagram-users/.

[34] (2018). Other data breaches like cambridge analytica. https://www.cnbc.com/2018/04/08/cubeyou-cambridge-like-app-collected-data-on-millions-from-facebook.html.

[35] (2018). Redis. http://redis.io/.

[36] (2018). Robinhood. https://robinhood.com/.

[37] (2018). A simple, asynchronous, single-threaded memcached client written in java. http://code.google.com/p/spymemcached/.

[38] (2018). Solidity — solidity 0.5.5 documentation. https://solidity.readthedocs.io/en/v0.5.5/.

[39] (2018c). Twitter: number of active users 2010-2018. https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/.

[40] (2019). Avg. transaction fee historical chart. https://bitinfocharts.com/comparison/transactionfees-btc-eth.html.

[41] (2019). Cost of a 51% attack for different cryptocurrencies. https://www.crypto51.app/.

[42] Adya, A., Dunagan, J., and Wolman, A. (2010). Centrifuge: Integrated lease management and partitioning for cloud services. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 1–1. USENIX Association.

[43] Adya, A., Myers, D., Howell, J., Elson, J., Meek, C., Khemani, V., Fulger, S., Gu, P., Bhuvanagiri, L., Hunter, J., et al. (2016). Slicer: Auto-sharding for datacenter applications. In *OSDI*, pages 739–753.

[44] Agarwal, S., Dunagan, J., Jain, N., Saroiu, S., Wolman, A., and Bhogan, H. (2010). Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, volume 10, pages 28–0.

[45] Ahmad, W. U., Chang, K.-W., and Wang, H. (2018). Intent-aware query obfuscation for privacy protection in personalized web search. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 285–294. ACM.

[46] Amiri, M. J., Agrawal, D., and El Abbadi, A. (2019). Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

[47] Anderson, M. (2016). Researchers take author recognition to neural networks. `https://thestack.com/world/2016/06/07/researchers-take-author-recognition-to-neural-networks/`.

[48] Andrés, M. E., Bordenabe, N. E., Chatzikokolakis, K., and Palamidessi, C. (2013). Geo-indistinguishability: Differential privacy for location-based systems. In *Proc. of ACM SIGSAC*, CCS '13, pages 901–914. ACM.

[49] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al. (2018a). Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM.

[50] Androulaki, E., Cachin, C., De Caro, A., and Kokoris-Kogias, E. (2018b). Channels: Horizontal scaling and confidentiality on permissioned blockchains. In *European Symposium on Research in Computer Security*, pages 111–131. Springer.

[51] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. (2012). Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM.

[52] Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timón, J., and Wuille, P. (2014). Enabling blockchain innovations with pegged sidechains.

[53] Bacon, D. F., Bales, N., Bruno, N., Cooper, B. F., Dickinson, A., Fikes, A., Fraser, C., Gubarev, A., Joshi, M., Kogan, E., Lloyd, A., Melnik, S., Rao, R., Shue, D., Taylor, C., van der Holst, M., and Woodford, D. (2017). Spanner: Becoming a sql system. In *Proc. SIGMOD 2017*, pages 331–343.

[54] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. (2011). Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234.

[55] Barber, S., Boyen, X., Shi, E., and Uzun, E. (2012). Bitter to better—how to make bitcoin a better currency. In *International Conference on Financial Cryptography and Data Security*, pages 399–414. Springer.

[56] Beckmann, N., Chen, H., and Cidon, A. (2018). LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, Renton, WA. USENIX Association.

[57] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). Concurrency control and recovery in database systems.

[58] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. (1999). Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE.

[59] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., et al. (2013). Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60.

[60] Brown, J. W., Ohrimenko, O., and Tamassia, R. (2013). Haze: Privacy-preserving real-time traffic statistics. In *Proc. of ACM SIGSPATIAL*, pages 540–543. ACM.

[61] Buterin, V. (2016a). Chain interoperability. *R3 Research Paper.*

[62] Buterin, V. (2016b). Critical update re: Dao vulnerability. `https://ethereum.github.io/blog/2016/06/17/critical-update-re-dao-vulnerability/`.

[63] Buterin, V. (2018). On sharding blockchains. `https://github.com/ethereum/wiki/wiki/Sharding-FAQs`.

[64] Buterin, V. et al. (2014). A next-generation smart contract and decentralized application platform. *white paper.*

[65] Cachin, C. (2016). Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, volume 310.

[66] Cachin, C. and Vukolić, M. (2017). Blockchain consensus protocols in the wild. In *31 International Symposium on Distributed Computing, DISC*, pages 1–16.

[67] Castro, M., Liskov, B., et al. (1999). Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186.

[68] Chandra, S., Khan, L., and Muhaya, F. B. (2011). Estimating twitter user location using social interactions–a content based approach. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, pages 838–843. IEEE.

[69] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4.

[70] Chase, J. M. (2016). Quorum white paper.

[71] Cheng, Y., Gupta, A., and Butt, A. R. (2015). An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, page 4. ACM.

[72] Cheng, Z., Caverlee, J., and Lee, K. (2010). You are where you tweet: a content-based approach to geo-locating twitter users. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 759–768. ACM.

[73] Chow, R., Golle, P., Jakobsson, M., Shi, E., Staddon, J., Masuoka, R., and Molina, J. (2009). Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 85–90. ACM.

[74] Cidon, A., Eisenman, A., Alizadeh, M., and Katti, S. (2015). Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*.

[75] Cidon, A., Eisenman, A., Alizadeh, M., and Katti, S. (2016). Cliffhanger: Scaling performance cliffs in web memory caches. In *NSDI*, pages 379–392.

[76] Cohen, E. and Strauss, M. (2003). Maintaining time-decaying stream aggregates. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 223–233. ACM.

[77] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. (2008). Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288.

[78] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM.

[79] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. (2013). Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8.

[80] Cormode, G., Korn, F., and Tirthapura, S. (2008). Exponentially decayed aggregates on data streams. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1379–1381. IEEE.

[81] Cormode, G., Shkapenyuk, V., Srivastava, D., and Xu, B. (2009). Forward decay: A practical time decay model for streaming systems. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 138–149. IEEE.

[82] Croman, K., Decker, C., Eyal, I., Gencer, A. E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., et al. (2016). On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer.

[83] Culwick, A. and Metcalf, D. (2018). The blocknet design specification.

[84] Das, S., Nishimura, S., Agrawal, D., and El Abbadi, A. (2011). Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment*, 4(8):494–505.

[85] Dasgupta, A., Kumar, R., and Sarlós, T. (2017). Caching with dual costs. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 643–652. International World Wide Web Conferences Steering Committee.

[86] Dickerson, T., Gazzillo, P., Herlihy, M., and Koskinen, E. (2017). Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 303–312. ACM.

[87] Dijkstra, E. W. (1968). Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer.

[88] Dilley, J., Poelstra, A., Wilkins, J., Piekarska, M., Gorlick, B., and Friedenbach, M. (2016). Strong federations: An interoperable blockchain solution to centralized third-party risks. *arXiv preprint arXiv:1612.05491.*

[89] Ding, S. H., Fung, B. C., Iqbal, F., and Cheung, W. K. (2017). Learning stylometric representations for authorship analysis. *IEEE Transactions on Cybernetics.*

[90] Dwork, C. (2006). *Automata, Languages and Programming: ICALP 2006, Proceedings, Part II*, chapter Differential Privacy, pages 1–12. Springer Berlin Heidelberg.

[91] Dwork, C. (2008). Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*, pages 1–19. Springer.

[92] Dwork, C. (2011). Differential privacy. In *Encyclopedia of Cryptography and Security*, pages 338–340. Springer.

[93] Dwork, C., Naor, M., Pitassi, T., Rothblum, G. N., and Yekhanin, S. (2010). Pan-private streaming algorithms. In *Proceedings of ICS*.

[94] Eyal, I., Gencer, A. E., Sirer, E. G., and Van Renesse, R. (2016). Bitcoin-ng: A scalable blockchain protocol. In *NSDI*, pages 45–59.

[95] Fan, B., Andersen, D. G., and Kaminsky, M. (2013). Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384.

[96] Fan, B., Lim, H., Andersen, D. G., and Kaminsky, M. (2011). Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 23. ACM.

[97] Fitzpatrick, K. (2018). How did social media become dangerous for democracy? `https://news.utexas.edu/2018/11/05/how-did-social-media-become-dangerous-for-democracy/`.

[98] Garoffolo, A. and Viglione, R. (2018). Sidechains: Decoupled consensus between chains. *arXiv preprint arXiv:1812.05441*.

[99] Gavrielatos, V., Katsarakis, A., Joshi, A., Oswald, N., Grot, B., and Nagarajan, V. (2018). Scale-out ccNUMA: exploiting skew with strongly consistent caching. In *Proceedings of the Thirteenth EuroSys Conference*, page 21. ACM.

[100] Georgiou, T., El Abbadi, A., and Yan, X. (2017a). Extracting topics with focused communities for social content recommendation. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing.*

[101] Georgiou, T., El Abbadi, A., and Yan, X. (2017b). Privacy-preserving community-aware trending topic detection in online social media. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 205–224. Springer.

[102] Ghandeharizadeh, S., Almaymoni, M., and Huang, H. (2019). Rejig: a scalable online algorithm for cache server configuration changes. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLII*, pages 111–134. Springer.

[103] Ghandeharizadeh, S. and Nguyen, H. (2019). Design, implementation, and evaluation of write-back policy with cache augmented data stores. *Proceedings of the VLDB Endowment*, 12(8):836–849.

[104] Ghufran, M., Quercini, G., and Bennacer, N. (2015). Toponym disambiguation in online social network profiles. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 6. ACM.

[105] Goldreich, O. (2001). *Foundations of cryptography: volume 1, basic tools.* Cambridge University Press, Cambridge.

[106] Goldreich, O. and Ostrovsky, R. (1996). Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473.

[107] Gong, X., Chen, X., Xing, K., Shin, D.-H., Zhang, M., and Zhang, J. (2015). Personalized location privacy in mobile networks: A social group utility approach. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 1008–1016. IEEE.

[108] Grant, T. (2008). How text-messaging slips can help catch murderers. `https://www.independent.co.uk/voices/commentators/dr-tim-grant-how-text-messaging-slips-can-help-catch-murderers-923503.html`.

[109] Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

[110] Gray, J. et al. (1981). The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154. Citeseer.

[111] Gray, J. and Lamport, L. (2006). Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160.

[112] Gray, J. N. (1978). Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer.

[113] Greenspan, G. (2015). Multichain private blockchain-white paper. *URl: http://www. multichain. com/download/MultiChain-White-Paper. pdf*.

[114] Guo, L., Tan, E., Chen, S., Xiao, Z., and Zhang, X. (2008). The stretched exponential distribution of internet media access patterns. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 283–294. ACM.

[115] Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317.

[116] Heatherly, R., Kantarcioglu, M., and Thuraisingham, B. (2013). Preventing private information inference attacks on social networks. *IEEE TKDE*, 25(8):1849–1862.

[117] Herlihy, M. (2018). Atomic cross-chain swaps. *arXiv preprint arXiv:1801.09515*.

[118] Herlihy, M. (2019). Blockchains from a distributed computing perspective. *Communications of the ACM*, 62(2):78–85.

[119] Herlihy, M., Liskov, B., and Shrira, L. (2019). Cross-chain deals and adversarial commerce. *arXiv preprint arXiv:1905.09743*.

[120] Hill, E. What is an asset-backed token? a complete guide to security token assets. `https://medium.com/ico-launch-malta/what-is-an-asset-backed-token-a-complete-guide-to-security-token-assets-f7a0f111d443`.

[121] Hong, Y.-J. and Thottethodi, M. (2013). Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 13. ACM.

[122] Huang, Q., Gudmundsdottir, H., Vigfusson, Y., Freedman, D. A., Birman, K., and van Renesse, R. (2014). Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 8. ACM.

[123] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9.

[124] Hwang, J. and Wood, T. (2013). Adaptive performance-aware distributed memory caching. In *ICAC*, volume 13, pages 33–43.

[125] Jiang, S. and Zhang, X. (2002). Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42.

[126] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. (2017). Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136. ACM.

[127] Johnson, T. and Shasha, D. (1994). X3: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th VLDB Conference*.

[128] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM.

[129] Katar, S., Yonge, L. W., and Krishnam, M. (2015). Network encryption key rotation. US Patent 8,989,379.

[130] Kemme, B. and Alonso, G. (2000). Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB*, pages 134–143. Citeseer.

[131] Kogias, E. K., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., and Ford, B. (2016). Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296.

[132] Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., and Saxena, P. (2018). Exploiting the laws of order in smart contracts. *arXiv preprint arXiv:1810.11605*.

[133] Kosinski, M., Stillwell, D., and Graepel, T. (2013). Private traits and attributes are predictable from digital records of human behavior. *Proceedings of the National Academy of Sciences*, 110(15):5802–5805.

[134] Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. (2013). Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM.

[135] Kung, H.-T. and Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226.

[136] Kwon, J. (2014). Tendermint: Consensus without mining. *Draft v. 0.6, fall.*

[137] Kwon, J. and Buchman, E. (2016). Cosmos: A network of distributed ledgers. *URL https://cosmos. network/whitepaper.*

[138] Lakshman, A. and Malik, P. (2009). Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 5–5. ACM.

[139] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

[140] Lamport, L. et al. (2001). Paxos made simple. *ACM Sigact News*, 32(4):18–25.

[141] Lee, D., Choi, J., Kim, J.-H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. (2001). Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361.

[142] Lerner, S. D. (2015). Rootstock: Bitcoin powered smart contracts.

[143] Li, N., Li, T., and Venkatasubramanian, S. (2007). t-closeness: Privacy beyond k-anonymity and l-diversity. In *IEEE ICDE*, pages 106–115.

[144] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. (2014). Mica: a holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444.

[145] Lin, Q., Chang, P., Chen, G., Ooi, B. C., Tan, K.-L., and Wang, Z. (2016). Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1659–1674. ACM.

[146] Liu, G. and Shen, H. (2016). Minimum-cost cloud storage service across multiple cloud providers. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 129–138. IEEE.

[147] Lomet, D. (2018a). Caching data stores: High performance at low cost. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1661–1661. IEEE.

[148] Lomet, D. (2018b). Cost/performance in modern data stores: how data caching systems succeed. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, page 9. ACM.

[149] Lomet, D. B. (2019). Data caching systems win the cost/performance game. *IEEE Data Eng. Bull.*, 42(1):3–5.

[150] Lu, H., Hodsdon, C., Ngo, K., Mu, S., and Lloyd, W. (2016). The snow theorem and latency-optimal read-only transactions. In *OSDI*, pages 135–150.

[151] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016a). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269. ACM.

[152] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., and Saxena, P. (2016b). A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM.

[153] Machanavajjhala, A., Kifer, D., Gehrke, J., and Venkitasubramaniam, M. (2007). L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1).

[154] Mahmoud, H., Nawab, F., Pucher, A., Agrawal, D., and El Abbadi, A. (2013). Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, 6(9):661–672.

[155] Maiyya, S., Zakhary, V., Agrawal, D., and El Abbadi, A. (2018). Database and distributed computing fundamentals for scalable, fault-tolerant, and consistent maintenance of blockchains. *Proceedings of the VLDB Endowment*, 11(12):2098–2101.

[156] Maiyya, S., Zakhary, V., Amiri, M. J., Agrawal, D., and El Abbadi, A. (2019). Database and distributed computing foundations of blockchains. In *Proceedings of the 2019 International Conference on Management of Data*, pages 2036–2041. ACM.

[157] Megiddo, N. and Modha, D. S. (2003). Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130.

[158] Metwally, A., Agrawal, D., and El Abbadi, A. (2005). Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer.

[159] Metzger, B., Mauldin, S., Sandell, B., and Chang, J. (2007). Key rotation. US Patent App. 11/236,046.

[160] Micali, S. (2016). Algorand: the efficient and democratic ledger. *CoRR, abs/1607.01341*.

[161] Miller, A., Bentov, I., Kumaresan, R., Cordi, C., and McCorry, P. (2017). Sprites and state channels: Payment networks that go faster than lightning. *arXiv preprint arXiv:1702.05812*.

[162] Mohan, C. (2017). Tutorial: blockchains and databases. *Proceedings of the VLDB Endowment*, 10(12):2000–2001.

[163] Mokbel, M. F., Chow, C.-Y., and Aref, W. G. (2006). The new casper: Query processing for location services without compromising privacy. In *Proc. of VLDB*, pages 763–774.

[164] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

[165] Nawab, F., Agrawal, D., and El Abbadi, A. (2013). Message futures: Fast commitment of transactions in multi-datacenter environments. In *CIDR*.

[166] Nawab, F., Arora, V., Agrawal, D., and El Abbadi, A. (2015). Minimizing commit latency of transactions in geo-replicated data stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1279–1294. ACM.

[167] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al. (2013). Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398.

[168] Nissenbaum, H. (2004). Privacy as contextual integrity. *Wash. L. Rev.*, 79:119.

[169] Nolan, T. (2013). Alt chains and atomic transfers. `https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949`.

[170] O'Keeffe, D. (2018). Understanding cryptocurrency transaction speeds. `https://medium.com/coinmonks/understanding-cryptocurrency-transaction-speeds-f9731fd93cb3`.

[171] O'neil, E. J., O'neil, P. E., and Weikum, G. (1993). The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306.

[172] O'neil, E. J., O'Neil, P. E., and Weikum, G. (1999). An optimality proof of the lru-k page replacement algorithm. *Journal of the ACM (JACM)*, 46(1):92–112.

[173] Ongaro, D. and Ousterhout, J. K. (2014). In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319.

[174] Pagh, R. and Rodler, F. F. (2004). Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144.

[175] Pass, R. and Shi, E. (2017). Hybrid consensus: Efficient consensus in the permissionless model. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[176] Patterson, S. et al. (2012). Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*.

[177] Paul, M. J. and Dredze, M. (2011). You are what you tweet: Analyzing twitter for public health. *Icwsm*, 20:265–272.

[178] Ping, F., Hwang, J.-H., Li, X., McConnell, C., and Vabbalareddy, R. (2011). Wide area placement of data replicas for fast and highly available data access. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*, pages 1–8. ACM.

[179] Poon, J. and Buterin, V. (2017). Plasma: Scalable autonomous smart contracts. *White paper*.

[180] Poon, J. and Dryja, T. (2016). The bitcoin lightning network: Scalable off-chain instant payments.

[181] Qureshi, M. K., Gurumurthi, S., and Rajendran, B. (2011). Phase change memory: From devices to systems. *Synthesis Lectures on Computer Architecture*, 6(4):1–134.

[182] Regalado, A. Who coined 'cloud computing'? `https://www.technologyreview.com/s/425970/who-coined-cloud-computing/`. 2011.

[183] Reid, F. and Harrigan, M. (2013). An analysis of anonymity in the bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer.

[184] Rivest, R. L., Adleman, L., Dertouzos, M. L., et al. (1978a). On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180.

[185] Rivest, R. L. et al. (1998). Chaffing and winnowing: Confidentiality without encryption. *CryptoBytes (RSA laboratories)*, 4(1):12–17.

[186] Rivest, R. L., Shamir, A., and Adleman, L. (1978b). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

[187] Ron, D. and Shamir, A. (2013). Quantitative analysis of the full bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security*, pages 6–24. Springer.

[188] Ryan, D. (2017). Costs of a real world ethereum contract. `https://hackernoon.com/costs-of-a-real-world-ethereum-contract-2033511b3214`.

[189] Sahin, C., Zakhary, V., El Abbadi, A., Lin, H., and Tessaro, S. (2016). Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 198–217. IEEE.

[190] Samarati, P. (2001). Protecting respondents identities in microdata release. *IEEE TKDE*, 13(6):1010–1027.

[191] Sasson, E. B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. (2014). Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE.

[192] Schwartz, H. A., Eichstaedt, J. C., Kern, M. L., Dziurzynski, L., Ramones, S. M., Agrawal, M., Shah, A., Kosinski, M., Stillwell, D., Seligman, M. E., et al. (2013). Personality, gender, and age in the language of social media: The open-vocabulary approach. *PloS one*, 8(9):e73791.

[193] Sergey, I. and Hobor, A. (2017). A concurrent perspective on smart contracts. In *International Conference on Financial Cryptography and Data Security*, pages 478–493. Springer.

[194] Sharov, A., Shraer, A., Merchant, A., and Stokely, M. (2015). Take me to your leader!: online optimization of distributed storage configurations. *Proceedings of the VLDB Endowment*, 8(12):1490–1501.

[195] Shute, J., Oancea, M., Ellner, S., Handy, B., Rollins, E., Samwel, B., Vingralek, R., Whipkey, C., Chen, X., Jegerlehner, B., et al. (2012). F1: the fault-tolerant distributed rdbms supporting google's ad business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 777–778. ACM.

[196] Stonebraker, M. (2010). Why enterprises are uninterested in nosql. http://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext/.

[197] Sweeney, L. (2002a). Achieving k-anonymity privacy protection using generalization and suppression. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):571–588.

[198] Sweeney, L. (2002b). K-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570.

[199] Syta, E., Tamas, I., Visher, D., Wolinsky, D. I., Jovanovic, P., Gasser, L., Gailly, N., Khoffi, I., and Ford, B. (2016). Keeping authorities" honest or bust" with decentralized witness cosigning. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 526–545. Ieee.

[200] Szabo, N. (1997). Formalizing and securing relationships on public networks. *First Monday*, 2(9).

[201] Taft, R., Mansour, E., Serafini, M., Duggan, J., Elmore, A. J., Aboulnaga, A., Pavlo, A., and Stonebraker, M. (2014). E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256.

[202] Thomas, S. and Schwartz, E. (2015). A protocol for interledger payments.

[203] Thomson, A. et al. (2012). Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*.

[204] Viejo, A., SáNchez, D., and Castellà-Roca, J. (2012). Preventing automatic user profiling in web 2.0 applications. *Knowledge-Based Systems*, 36:191–205.

[205] Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32.

[206] Wood, G. (2016). Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*.

[207] Wu, C., Sreekanti, V., and Hellerstein, J. M. (2019). Autoscaling tiered cloud storage in anna. *Proceedings of the VLDB Endowment*, 12(6):624–638.

[208] Wu, Z., Butkiewicz, M., Perkins, D., Katz-Bassett, E., and Madhyastha, H. V. (2013). SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308. ACM.

[209] Wüst, K., Matetic, S., Egli, S., Kostiainen, K., and Capkun, S. Ace: Asynchronous and concurrent execution of complex smart contracts.

[210] Yakout, M., Ouzzani, M., Elmeleegy, H., Talukder, N., and Elmagarmid, A. K. (2010). Privometer: Privacy protection in social networks. *IEEE ICDEW*, 00:266–269.

[211] Zakhary, V., Agrawal, D., and El Abbadi, A. (2017a). Caching at the web scale. *Proceedings of the VLDB Endowment*, 10(12):2002–2005.

[212] Zakhary, V., Agrawal, D., and El Abbadi, A. (2019a). Atomic commitment across blockchains. *arXiv preprint arXiv:1905.02847*.

[213] Zakhary, V., Agrawal, D., and El Abbadi, A. (2019b). CoT: Decentralized elastic caches for cloud environments. *preprint*.

[214] Zakhary, V., Agrawal, D., and El Abbadi, A. (2019c). Transactional smart contracts in blockchain systems. *arXiv preprint arXiv:1909.06494*.

[215] Zakhary, V., Amiri, M. J., Maiyya, S., Agrawal, D., and El Abbadi, A. (2019d). Towards global asset management in blockchain systems. *arXiv preprint arXiv:1905.09359*.

[216] Zakhary, V., Gupta, I., Tang, R., and El Abbadi, A. (2019e). Multifaceted privacy: How to express your online persona without revealing your sensitive attributes. *arXiv preprint arXiv:1905.09945*.

[217] Zakhary, V., Nawab, F., Agrawal, D., and El Abbadi, A. (2016). Db-risk: The game of global database placement. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2185–2188. ACM.

[218] Zakhary, V., Nawab, F., Agrawal, D., and El Abbadi, A. (2018a). Global-scale placement of transactional data stores. In *EDBT*, pages 385–396.

[219] Zakhary, V., Sahin, C., El Abbadi, A., Lin, H., and Tessaro, S. (2018b). Pharos: Privacy hazards of replicating oram stores. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018*.

[220] Zakhary, V., Sahin, C., Georgiou, T., and El Abbadi, A. (2017b). Locborg: Hiding social media user location while maintaining online persona. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 12. ACM.

[221] Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., and Knottenbelt, W. J. (2019). Xclaim: A framework for blockchain interoperability.

[222] Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., and Zhang, M. (2015). In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948.

[223] Zhang, Y., Humbert, M., Rahman, T., Li, C.-T., Pang, J., and Backes, M. (2018a). Tagvisor: A privacy advisor for sharing hashtags. *arXiv preprint arXiv:1802.04122*.

[224] Zhang, Y., Jiang, J., Xu, K., Nie, X., Reed, M. J., Wang, H., Yao, G., Zhang, M., and Chen, K. (2018b). Bds: a centralized near-optimal overlay network for inter-datacenter data replication. In *Proceedings of the Thirteenth EuroSys Conference*, page 10. ACM.

[225] Zheleva, E. and Getoor, L. (2009). To join or not to join: the illusion of privacy in social networks with mixed public and private user profiles. In *Proc. of WWW*, pages 531–540. ACM.

[226] Zhou, Y., Philbin, J., and Li, K. (2001). The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104.