EPFL

Thèse n° 10 241

Efficient Protocols for Enforcing Causal Consistency in Geo-Replicated Key-Value Data Stores

Présentée le 15 juin 2020

à la Faculté informatique et communications
Laboratoire de systèmes d'exploitation
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Kristina SPIROVSKA

Acceptée sur proposition du jury

Prof. M. Odersky, président du jury
Prof. W. Zwaenepoel, directeur de thèse
Prof. F. Pedone, rapporteur
Prof. N. Preguiça, rapporteur
Prof. K. Aberer, rapporteur

École
polytechnique
fédérale
de Lausanne

2020

"The years of anxious searching in the dark, with their intense longing,
their alternations of confidence and exhaustion, and final emergence into light-
only those who have experienced it can understand that."
— Albert Einstein

To my dearest husband, loving parents and wonderful sister,
who supported me all the way. . .

Acknowledgements

As I find myself taking the last steps of my PhD journey, I can not go any further without recognizing the people who were there with me and for me all the way. Their unconditional support and faith in me, helped me find the strength that I needed to prevail and succeed in this challenging quest for knowledge. For that, I am forever in their debt.

First and foremost, I would like to express my profound gratitude to my thesis advisor, Prof. Willy Zwaenepoel, for without his support and guidance, this thesis would have been nothing but an unfulfilled dream. His constructive criticism and invaluable advices helped me construct the foundations of what is necessary to be an outstanding researcher. He taught me how to approach challenging problems by taking small steps and analyzing the results before going any further. He showed me the best way to write papers and taught me the real value of concise and creative presentations. Not only that Prof. Willy shaped my development as a researcher, he also influenced my advancement as a person. Prof. Willy gave me the opportunity to do two internships in industry-leading companies abroad, giving me the chance to experience the process of solving real world problems. I must say that it has been a rare honour and a real privilege to work and learn under the guidance of such a distinguished and prestigious mentor.

I would like to thank the members of the jury, Prof. Karl Aberer, Prof. Fernando Pedone and Prof. Nuno Preguiça, for their work in reviewing this thesis and for their insightful comments, questions and invaluable feedback. I would also like to thank Prof. Martin Odersky for presiding over my private thesis defense.

I am truly grateful to Dr. Diego Didona, who I have been working with closely during the past 5 years. He helped me discover my potential and find my strength as a researcher. Diego was always there when I needed someone to discuss ideas and brainstorm about possible solutions.

I would also like to express my gratitude to Scott Hahn, Vishakha Gupta, Luis Remis, Karan Gupta, Mayur Sadavarte, Yanmin Tao and all my other colleagues from IntelLabs and Nutanix. Thank you all for making me part from your teams and for giving me a tremendous opportunity to enhance both my researching and software engineering skills from a practical perspective.

The research in this dissertation has been supported by The Swiss National Science Foundation through Grant No. 166306. I would like to express my gratitude to Amazon for providing a research grant through AWS Cloud Credits.

I would like to express my special gratitude to Ms. Madeleine Robert, who always managed to find a smart solution for even the most challenging administrative problems that I faced.

I was very fortunate to be part of the LABOS family, whose members made my office days an interesting part of my PhD journey. Thank you Jasmina, Maria Fernanda, Pamela, Oana, Florin, Laurent, Calin, Baptiste. You were the people that listen to me and supported me whether I had problem to solve or a good news to share. You were amazing colleagues and you are even better friends. I would also like to thank Dr. Laurent Bindschaedler for reviewing and correcting the French version of the abstract.

My PhD experience would have been much hollower without the support of the new friends that I met in Switzerland. We have shared a lot of happy memories during parties, dinners, board game nights etc. Among them are Maja, Marjan, Ana, Vase, Gorica, Emil, Igor, Beti, Oli, Isabel, Darko, Elena, Mladen, Mia, Miranda, Mira, Danica, Georgia, Eleni, and many others. Thank you all for making my stay in Switzerland feel more like home!

I am also thankful to Ilinka, Vesna, Aneta, Maja, Tanja, my childhood and university friends. Even though they are spread all over the globe, we will always remain close friends.

I would like to express a special gratitude to my in-laws, Anica, Borche, Marina and Laze, for their love and support to achieve my goals throughout my PhD journey.

I will be forever thankful to my loving parents, Julijana and Slobodan, whose unconditional love and support made me who I am today. They have such passion and enthusiasm towards knowledge, which had inspired me to seek it, too. They taught me how to dream and they supported me to follow my dreams, even though my dreams took me far away from home.

I am extremely thankful to my loving sister, Katerina, for always believing in me. She is always here for me, cheering me, supporting me and boosting my confidence.

I cannot find enough words to express my eternal gratitude to my husband, Nikolche. His infinite love and support give me strength to overcome even the hardest challenges that life throws at me. Being an outstanding Computer Science expert himself, Nikolche also provides valuable advice whenever I am stuck on an engineering problem. I enjoy our frequent technical discussions, which prepare me for the technical challenges that I face throughout my career. My dearest, thank you for everything!

Lausanne, 5 June 2020 Kristina

Abstract

Modern large-scale data platforms manage colossal amount of data, generated by the ever-increasing number of concurrent users. Geo-replicated and sharded key-value data stores play a central role when building such platforms. As the strongest consistency model proven not to compromise availability, causal consistency (CC) is perfectly positioned to address the needs of such large-scale systems, while preventing some ordering anomalies. Transactional Causal Consistency (TCC) augments CC by providing richer transactional semantics, simplifying the development of distributed applications. However, achieving CC/TCC in an efficient manner is very challenging.

In this thesis we introduce several protocols and designs for high performance causally consistent geo-replicated key-value data stores in several different settings.

First, we present a new approach to implementing CC in geo-replicated data stores, called Optimistic Causal Consistency (OCC). By introducing a technique that we call client-assisted lazy dependency resolution, OCC makes it possible for updates replicated to a remote data center to become visible immediately, without checking if their causal dependencies have been received. We further propose a recovery mechanism that allows an OCC system to fall back on a pessimistic protocol to continue operating during network partitions. We show that OCC improves data freshness, while offering performance that is comparable or better than its pessimistic counterpart.

Next, we address the problem of providing low-latency TCC reads under full replication. We present Wren, the first TCC system that at the same time achieves low latency by implementing nonblocking read operations and efficiently scales out by sharding. Wren introduces new protocols for transaction execution, dependency tracking and stabilization. The transaction protocol supports nonblocking reads by providing a transaction snapshot as a union of a fresh causal snapshot installed by every partition in the local data center and a client-side cache for writes that are not yet included in the snapshot. The dependency tracking and stabilization protocols require only two scalar timestamps, resulting in efficient resource utilization and providing scalability in terms of shards and replication sites, under a full replication setting. In return for these benefits, Wren slightly increases the visibility latency of updates.

Finally, we present PaRiS, the first TCC system that supports partial replication and provides low latency by implementing non-blocking parallel read operations. PaRiS relies on a novel protocol to track dependencies, called Universal Stable Time (UST). By means of a lightweight background gossip process, UST identifies a snapshot of the data that has been installed by every data center in the system. PaRiS equips clients with a private cache, in which they store their own updates that are not yet reflected in the snapshot. The combination of the UST-defined snapshot with client-side cache enables interactive transactions that can consistently read from any replication site without blocking. Moreover, PaRiS requires only one timestamp to track dependencies and define transactional snapshots, thereby achieving resource efficiency and scalability in terms of shards and replication sites, in a partial replication setting.

Key words: *causal consistency, optimistic causal consistency, transactional causal consistency, geo-replication, key-value stores, low latency, data freshness, nonblocking transactional reads, partial replication*

Résumé

Les plateformes de données modernes à grande échelle gèrent une quantité colossale de données, générées par un nombre toujours croissant d'utilisateurs simultanés. Les magasins de données au format clé-valeur géographiquement répliqués et partagés jouent un rôle central lors de la conception de telles plateformes. La cohérence causale (CC), le plus puissant modèle de cohérence ne compromettant pas la disponibilité, est parfaitement positionnée pour répondre aux besoins de ces systèmes à grande échelle. La cohérence causale transactionnelle (TCC) augmente la CC en fournissant une sémantique transactionnelle plus riche. Cela simplifie le développement d'applications distribuées. Cependant, réaliser une CC/TCC de manière efficace est une tâche ardue.

Dans cette thèse, nous introduisons plusieurs protocoles et modèles pour des magasins de données en format clé-valeur géographiquement répliqués causalement cohérents de haute performance. Premièrement, nous présentons une nouvelle approche pour implémenter la CC dans des magasins de données géographiquement répliquées, appelée Optimistic Causal Consistency (OCC). En introduisant une technique que nous appelons la résolution paresseuse des dépendances assistée par le client, OCC permet aux mises à jour répliquées vers un centre de données distant de devenir immédiatement visibles, sans vérifier si leurs dépendances causales ont été reçues. Cela améliore la fraîcheur des données, tout en offrant des performances comparables à celles de son homologue pessimiste.

Ensuite, nous abordons le problème des lectures avec TCC à faible latence en présence de réplication complète. Nous présentons Wren, le premier système à TCC qui, en même temps, atteint une faible latence en utilisant des opérations de lecture non bloquantes et évolue efficacement par partitionnement. Wren introduit de nouveaux protocoles pour l'exécution des transactions, le suivi des dépendances et la stabilisation. Le protocole de transaction prend en charge les lectures non bloquantes en fournissant un instantané (snapshot) de transaction en tant qu'union d'un nouvel instantané causal installé par chaque partition du centre de données local et un cache côté client pour les écritures qui ne sont pas encore incluses dans l'instantané. Les protocoles de suivi et de stabilisation des dépendances ne nécessitent que deux horodatages scalaires, ce qui permet une utilisation efficace des ressources et une bonne capacité de passage à l'échelle.

Enfin, nous présentons PaRiS, le premier système à TCC qui prend en charge la réplication partielle et fournit une faible latence en mettant en œuvre des opérations de lecture parallèles et non bloquantes. PaRiS s'appuie sur un nouveau protocole pour suivre les dépendances, appelé Universal Stable Time (UST). UST identifie un instantané des données qui ont été installées par chaque centre de données. Les clients utilisent un cache privé pour stocker leurs propres mises à jour qui ne sont pas encore reflétées dans l'instantané. La combinaison de l'instantané défini par UST avec le cache permet des transactions interactives qui peuvent constamment lire à partir de n'importe quel site de réplication sans blocage. De plus, PaRiS ne nécessite qu'un seul horodatage pour suivre les dépendances et définir des instantanés transactionnels, obtenant ainsi une grande efficacité en utilisation des ressources et une bonne capacité de passage à l'échelle.

Mots clefs : *cohérence causale, cohérence causale optimiste, cohérence causale transactionnelle, géo-réplication, magasins de valeurs-clés, faible latence, fraîcheur des données, lectures transactionnelles non bloquantes, réplication partielle*

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Since the dawn of computing, we are witnessing an exponential growth of data. In 2017 alone, a report from IBM Marketing Cloud [30], points out that 90 percent of the world's data, at that time, has been created in the past 12 months. Data has taken a central role in our everyday lives, to the extent that it has even been considered as the currency of the future [8]. This has driven the emergence of large-scale distributed systems, a critical infrastructure component for storing, processing, managing and analyzing immense amounts of data in many of today's popular Internet services. Apart from the colossal amount of data, these systems also have to support the ever-growing number of concurrent users which are interacting with them.

Interestingly, not just the growth of data and number of users, but also the growth of the users' expectations shaped the design goals of today's modern large-scale distributed systems. Nowadays, low latency is not just a nice-to-have property of a system, but a revenue-defining one. For instance, many of the most popular online platforms have reported that slower response times negatively impacted their business [39, 91]. Around a decade ago, Amazon reported their finding that every additional 100 ms of latency cost them 1% in sales [6]. A study made by Tabb Group showed that brokers could lose as much as $4 million in revenues per millisecond if their electronic trading platform is only 5ms behind the competition [101]. Google reported that an extra 0.5 seconds in search page generation time dropped traffic by 20% [75]. Time is money, and every millisecond counts.

Over the years, geo-replication has become the *de facto* standard for storage systems underlying large-scale web applications. By keeping a copy of the data in a data center closer to the users, geo-replication provides high data locality that results in reduced access latencies. Additionally, geo-replication enables these systems to tolerate network partitions and data center failures. Sharding enables horizontal scalability, by slicing the dataset in disjoint partitions, each of which can be assigned to a different server.

The self-imposed requirements, such as scalability and low latency, have became crucial design considerations when building high-performance large-scale distributed systems. Furthermore, the introduction of Brewer's CAP Theorem created a shift in their design space from the traditional architectures, based on relational databases, towards a different data management paradigm, known as NoSQL databases. The NoSQL transition led to the popularization and widespread adoption of the loosely consistent distributed key-value stores, such as Amazon's Dynamo, Yahoo's PNUTS, Google's BigTable, which serve as the backbone of many large scale platforms [33, 27, 36, 98].

Enforcing the necessary consistency guarantees in an efficient manner is one of the core challenges software engineers face when building a distributed system. Since distributed systems are much more difficult to design, operate, understand, test and debug than a single-machine system, the attempts to enforce the desired consistency guarantees often lead to bugs and incorrect behaviour. One example is the famous DAO hack [102], where attackers exploited a concurrency bug on a popular blockchain smart contract to steal fifty million dollars. The central part in enforcing consistency guarantees is based on the theory of event ordering [66]. This also defines the performance of the system. Moreover, the principal difference between strong and weak consistency models is whether global ordering of events is required. On the one hand, strongly consistent systems enforce global order of events, offering the strongest consistency guarantees. That requires coordination between all the constituents of the distributed system, and results in poor performance. Weakly consistent systems, on the other hand, favour performance over consistency by only partially ordering events, which results in one of the many weak consistency guarantees such as eventual consistency.

Causal consistency is an attractive consistency model for geo-replicated data stores because it hits a sweet spot in the performance vs. consistency guarantees space. Nowadays there has been an increased interest in causal consistency in the academic society. Over the past decade, causal consistency appeared as an important topic in the research community, with many publications at top-tier conferences [114, 69, 70, 5, 12, 43, 42, 90, 72, 38, 4, 105, 76, 22, 46, 74, 111, 55, 55, 45, 49]. It has also sparkled a genuine interest from the industry as well [106].

In this dissertation, we study the problem of designing efficient protocols for enforcing causal consistency in geo-replicated partitioned key-value datastores. In the rest of this chapter we present our motivation. Then, we describe several open challenges faced by designers of causally consistent geo-replicated systems and we discuss how we address those challenges by discussing the contributions of this thesis.

1.1 Motivation

Maximizing data freshness. Current state-of-the-art causally consistent systems [69, 70, 43, 92] use different approaches in order to implement causal consistency. In general, these approaches share a common mechanism: they make an update visible in a data center only when all of the causal dependencies of the update are known to be replicated in that data center. Although this allows them to tolerate network partitions, it comes at the price of reduced data freshness due to the fact that such updates are usually stale. Furthermore, to track the delivery of dependencies, these systems have to implement dependency checking [69, 70] or stabilization protocols [42, 4], which results in computational and communication overhead.

Recent research [110, 15, 71] reports that updates, most of the time, are replicated in a naturally consistent order, after their dependencies have already being replicated. Additionally, the works of [15, 93, 23, 67] show that network partitions, especially full DC failures, can be considered relatively rare events. This inspired us to question whether existing protocols are overly pessimistic for modern data deployments. An important requirement for many online information systems, such as e-commerce systems, social networks, trading systems, etc. is to serve fresh data to the users. This was our main motivation behind the work presented in Chapter 3, where we present optimistic causal consistency and we discuss the benefits and trade-offs that it introduces.

Low latency. Latency has a significant impact on user experience [54]. The longer users wait for data, the worse the user experience becomes. For instance, a long response time of a web application drastically decreases its interactivity and usability. This leads to users feeling frustrated and dissatisfied of the application, which is often correlated with lost of revenue. There are numerous examples of how latency influences revenue [39, 91, 6, 101, 75]. For instance, the BBC reported that for every additional second their site took to load, they experienced an additional 10% churning users [29]. DoubleClick by Google found that if the loading of a page took longer than 3 seconds, 53% of mobile site visits were abandoned [103]. There are even psychological studies that explore how slow response times make users experience frustration [54]. Contrariwise, there is also evidence that high-performing web sites engage and retain users better than low-performing ones. For instance, Pinterest found that when they reduced perceived wait times by 40%, their web traffic and sign-ups increased by 15% [40]. When COOK reduced average page load time by 850 milliseconds, that increased conversions by 7%, decreased bounce rates by 7%, and increased pages per session by 10% [31].

Indeed, latency is a crucial property of modern systems, that influences the fate of the system. We found it as key motivation behind the work in Chapter 4 and Chapter 5.

Rich transactional semantics. The CAP theorem inspired the emergence and widespread adoption of key-value data store systems, such as Amazon's Dynamo, Yahoo's PNUTS, Google's

BigTable. At that time, applications' logic was simpler. Designs that supported only single-operation and single-item transactions were enough to satisfy the need of the users. As applications became more complex, richer semantics were desired, such as providing multiple operations that access numerous data items distributed across different servers. This requires the implementation of distributed transactions, an abstraction that combines a group of low-level storage operations, such as reads and writes, into a higher-level construct.

Distributed transactions are very desirable and powerful apparatus because they hide the complexity of how data is distributed from the application. However, they frequently induce blocking scenarios and precipitate communication overheads straightforwardly leading to increased application latencies. This motivated us to explore whether we can make distributed transactions more efficient by eliminating the blocking when reading data, which is typically caused by the protocols that enforce causal consistency. Chapter 4 describes our contributions in this context.

Scalability. The ability to scale horizontally, or to scale out, is of great importance for today's large-scale systems. Indeed, systems that require the whole dataset to fit on one server are not practical anymore. Horizontal scalability enables systems to split their data into disjoint partitions that are spread across many servers. This enables systems to contend with the trend of ever-growing data and to support the thousands of users that are accessing the data simultaneously.

Partial replication. In geo-replicated environments, partial replication is an effective technique to reduce storage requirements and replication costs. In partial replication, each data center stores only a subset of the data, as opposed to the full replication model, where each data center stores the whole dataset. Hence, the system can scale to a higher number of partitions with respect to a full replication approach, and updates performed in one DC are propagated to fewer replicas. Furthermore, in systems with partial replication it is not required all data centers to have the same capacity. This makes it easier to add a new data center to an existing system.

Remarkably, most of the protocols for implementing causal consistency have been designed for a full replication system model, where each data center stores a full copy of the whole dataset. In Chapter 5 we discuss the challenges of achieving partial replication together with causal consistency.

The search for efficient dependency tracking protocols. When designing causally consistent systems, one important design choice is the mechanism for tracking causal dependencies. State-of-the-art causally consistent systems employ different mechanisms for dependency tracking: sequencers [22], explicit dependency checking [69, 70], dependency vectors [49], single timestamp [42], dependency matrices [43]. Do these methods cover all the possible ways to track causality? Are there some better suited mechanisms that are more effective? What are the novel trade-offs with respect to dependency metadata size, resource efficiency, data freshness and performance? We cover these questions in Chapter 4 and Chapter 5.

1.2 Challenges

Maximizing data freshness in a causally consistent system. In order to maximize data freshness, a server at a particular data center has to return the latest available version of every item in that data center. This is at odds with most causally consistent systems, which enforce causal consistency guarantees by making updates visible in a data center only when all of the causal dependencies of those updates are known to be replicated in that data center. We refer to such updates whose causal dependencies are known to have already been received, as stable updates. This approach allows them to tolerate network partitions due to the fact that a data store partition does not have to wait for the receipt of a version from a remote DC. Nonetheless, this approach is likely to return stale items to the clients, regardless whether fresher versions have already been received.

To achieve maximal data freshness, a system has to expose items that are not stable. Enforcing causal consistency while exposing items whose causal dependencies are not received yet can impact the performance or availability of such system. In Chapter 3 we address this challenge and succeed to achieve maximal data freshness while implementing causal consistency with no performance penalty. We do this by shifting a part of the dependency tracking mechanism towards the client, using a technique we call client-assisted lazy dependency resolution.

Achieving causally consistent always-available interactive multi-partition transactions. Enforcing causal consistency while offering always-available interactive multi-partition transactions is a challenging problem [72]. One of the reasons is that the partitions of a distributed system do not progress at the same pace. Current designs that implement transactional causal consistency either block reads until a required snapshot has been installed [4], or they avoid this issue by not supporting sharding [114]. The former approach incurs additional latency of read operations, while the latter approach sacrifices scalability. In Chapter 4 and 5 we address this challenge for both full and partial replication.

Achieving low-latency causally consistent transactional reads. Achieving low-latency causally consistent transactional reads is challenging, especially when implementing interactive read-write transactions. A system that supports interactive read-write transactions, in the same time has to preserve the consistency of the reads and the atomicity of multi-item (and hence multi-partition) writes. Transactional multi-item reads may be executed as parallel individual reads on multiple partitions. That exacerbates the problem, because a read operation that is taking place on one node is unaware what value the read of the other node will return. We address these challenges in 4 and 5 for both full and partial replication.

Efficient dependency tracking. State-of-the-art systems use different methods to track causal dependencies: dependency graphs[69], version vectors [49], dependency matrices [43] or single scalar timestamp [42] . Our research interest is mostly focused on the methods that track causality by timestamps.

The greatest benefit of using version vectors is that they allow to track updates' visibility at finer granularity, due to the fact that they store one timestamp per data center. The drawback of this approach is that this is not a scalable representation because the metadata size grows linearly with the number of data centers. This creates storage and communication overheads, because each update is stored and replicated with its corresponding version vector. The scalability issue is furthermore exacerbated with the dependency matrices. Single scalar, on the contrary, is the most scalable representation used to track dependencies. However, it compresses all dependencies in one single timestamp, resulting in the worst visibility latency determined by the slowest connection (the connection with the furthest data center). This greatly, and often unnecessary, influences the visibility of local updates.

The natural question is how can we achieve scalable representation of dependencies and still keep most of the updates visible? This challenge is addressed in Chapter 4. By using only two scalars to track dependencies, we achieve best of both worlds: a scalable representation that decouples the local from the remote dependencies and enables greater visibility.

Implementing partial replication with transactional causal consistency. Transactional causal consistency must guarantee enforcement of causal consistency, as well as atomicity of multi-item writes. In a system with full replication, the task of enforcing these guarantee is eased by the fact that each data center stores all dependencies of any update, and the fact that every transaction is performed within a single data center. Hence, some sort of communication among partitions in the *same* data center is enough to ensure transactional causal consistency.

However, partial replication imposes a new set of challenges when implementing transactional causal consistency. In a system with partial replication, a read operation can target any replica of the target key. Hence, causal consistency and atomicity have to be preserved despite the fact that different transactions that are targeting the same set of keys can hit different replicas of those keys. The complexity of the problem is further exacerbated by the fact that different replicas of a version of an item may be in different data centers that store different sub-sets of the dependencies of that version of the item. We address these challenges in Chapter 5.

1.3 Contributions and thesis statement

We address the presented challenges and contribute the following:

Optimistic Causal Consistency. We present a new approach to implementing causal consistency in geo-replicated data stores, which we call Optimistic Causal Consistency (OCC). The optimism in our approach lies in that the updates replicated to a data center are made visible in a remote data center immediately, without checking if their causal dependencies have been received. Servers perform the dependency check needed to enforce causal consistency only upon serving a client operation, rather than on the receipt of a replicated data item as in existing systems. To address the challenge of enforcing causal consistency while exposing unstable item versions in a cost-effective way, we introduce the *client-assisted lazy dependency resolution protocol*. The key idea behind the client-assisted lazy dependency resolution protocol is to relieve the data store from the responsibility of tracking the delivery of updates by shifting it to the client.

We explore a novel trade-off in the landscape of CC protocols. OCC's potentially blocking behavior makes it vulnerable to network partitions. Because network partitions are rare in practice, however, OCC chooses to trade availability to maximize data freshness and reduce the communication overhead. We further propose *highly available OCC*, a recovery mechanism that allows an OCC system to fall back on a pessimistic protocol to continue operating even during network partitions.

POCC is an implementation of OCC based on physical clocks. We describe POCC's protocol and we show an informal proof of its correctness. We evaluate POCC and we show that OCC improves data freshness, while offering comparable or better performance than its pessimistic counterpart.

Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store. We present Wren, the first transactional causally consistent system that at the same time i) implements nonblocking read operations, thereby achieving low latency, and ii) allows an application to efficiently scale out within a replication site by sharding.

We address the challenges about achieving causally consistent always-available interactive multi-partition transactions and low-latency transactional reads in Wren by implementing a novel transaction protocol called CANToR (Client-Assisted Nonblocking Transactional Reads). In CANToR, the snapshot of the data store visible to a transaction is defined as the union of two components: i) a fresh causal snapshot that has been installed by *every* partition within the DC; and ii) a per-client cache, which stores the updates performed by the client that are not yet reflected in said snapshot.

We address the efficient dependency tracking challenge in Wren by introducing a new dependency tracking protocol Binary Dependency Time (BDT) and a new stabilization protocol Binary Stable Time (BiST). Regardless of the number of partitions and DCs, these two protocols assign only two scalar timestamps to updates and snapshots, corresponding to dependencies on local and remote items. These protocols provide high resource efficiency and scalability, and preserve availability.

In return for these benefits, Wren slightly increases the visibility latency of updates.

We evaluate Wren on an AWS deployment using up to 5 replication sites and 16 partitions per site. We show that Wren delivers up to 1.4x higher throughput and up to 3.6x lower latency when compared to the state-of-the-art design. The choice of an older snapshot increases local update visibility latency by a few milliseconds. The use of only two timestamps to track causality increases remote update visibility latency by less than 15%.

Causally Consistent Transactions with Non-blocking Reads and Partial Replication. We present PaRiS, the first transactional causally consistent system that supports partial replication and implements non-blocking parallel read operations. The latter reduce read latency which is of paramount importance for the performance of read-intensive applications. We present a novel protocol to track dependencies, called Universal Stable Time (UST). By means of a lightweight background gossip process, UST identifies a snapshot of the data that has been installed by every data center (DC) in the system. Hence, transactions can consistently read from such a snapshot on any server in any replication site without having to block. Moreover, PaRiS requires only one timestamp to track dependencies and define transactional snapshots, thereby achieving resource efficiency and scalability.

We evaluate PaRiS on an AWS deployment composed of up to 10 replication sites. We demonstrate a performance gain of non-blocking reads vs. a blocking alternative (up to 1.47x higher throughput with 5.91x lower latency for read-dominated workloads and up to 1.46x higher throughput with 20.56x lower latency for write-heavy workloads). We also show that the throughput penalty incurred to implement causal consistency, compared to variant without the causal consistency guarantees, is as low as 20% for read-heavy workloads and 37% for write-heavy workloads. We furthermore show that PaRiS scales well with the number of DCs and partitions, while being able to handle larger data-sets than existing solutions that assume full replication.

1.3.1 Thesis statements

In light of these contributions, this thesis makes the following statements:

Maximal data freshness in a causally consistent system can be achieved by utilizing the client-assisted lazy dependency resolution protocol. This protocol absolves the data store of the burden of running distributed protocols to track the delivery of causal dependencies, by empowering the client with the ability to track whether a read operation can be executed in a causally consistent manner.

An efficient way of implementing causally consistent interactive transactions with nonblocking *reads in full geo-replication is by providing a transaction with a snapshot that is the union of a fresh causal snapshot S installed by every partition in the local data center and a client-side cache for writes that are not yet included in S.*

A scalable dependency tracking mechanism, that in the same time minimizes visibility latency, can be achieved by using two timestamps that decouple local from remote items. One timestamp tracks the dependencies on local items and the other summarizes the dependencies on remote items.

An efficient way of implementing causally consistent interactive transactions with nonblocking reads in partial geo-replication is by providing a transaction with a snapshot that is the union of a snapshot S that has been installed by every data center in the system and a client-side cache for writes that are not yet included in S.

1.4 Publications

The results described in Section 1.3 have been previously published:

1. K. Spirovska; D. Didona; W. Zwaenepoel: Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. 2017. *ICDCS'17: 37th IEEE International Conference on Distributed Computing Systems*, Atlanta, GA, USA, June 5-8, 2017 (Chapter 3)

2. K. Spirovska; D. Didona; W. Zwaenepoel: Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store. 2018. *DSN'18: 48th International Conference on Dependable Systems and Networks*, Luxembourg, June 25-28, 2018 (Chapter 4) **Best paper award**

3. K. Spirovska; D. Didona; W. Zwaenepoel: PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication. 2019. *ICDCS'19: 39th IEEE International Conference on Distributed Computing Systems*, Dallas, Texas, July 7-9, 2019 (Chapter 5)

The work presented in Chapter 3 is under submission:

- K. Spirovska; D. Didona; W. Zwaenepoel: Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. 2020. *TPDS journal: IEEE Transactions on Parallel and Distributed Systems*, (Chapter 3)

1.5 Thesis outline

Chapter 2 provides the background on the principles of geo-distributed and partitioned key-value data stores and casual consistency.

Chapter 3 introduces Optimistic Causal Consistency (OCC). We propose a recovery mechanism that allows an OCC system to fall back on a pessimistic protocol to continue operating even during network partitions. Then, we present POCC an implementation of OCC based on physical clocks and we informally show its correctness. Finally, we evaluate POCC and we show that it improves data freshness, while offering comparable or better performance than its pessimistic counterpart.

Chapter 4 presents Wren, the first transactional causally consistent system that at the same time provides low latency by implementing nonblocking read operations, and allows an application to efficiently scale out within a replication site by sharding. We introduce new protocols for transaction execution, dependency tracking and stabilization. We evaluate Wren on an AWS deployment using up to 5 replication sites and 16 partitions per site. We show that Wren delivers up to 1.4x higher throughput and up to 3.6x lower latency when compared to the state-of-the-art design.

Chapter 5 presents PaRiS (short from **Pa**rtial **R**epl**i**cation **S**ystem), the first transactional causally consistent system that supports partial replication and implements non-blocking parallel read operations. We describe the challenges of partial replication and how we overcome them. We introduce a novel protocol to track dependencies, called Universal Stable Time (UST). We describe the new protocol for transaction execution under partial replication. We evaluate PaRiS on an AWS deployment composed of up to 10 replication sites. We demonstrate the performance gain of non-blocking reads vs. a blocking alternative. We furthermore show that PaRiS scales well with the number of DCs and partitions, while being able to handle larger data-sets than existing solutions that assume full replication.

Chapter 6 presents systems and papers related to the work presented in this thesis.

Finally, in **Chapter 7** we draw the conclusions of this thesis and highlight the directions for future work.

2 Background

In this chapter we describe the context of this work, such as: geo-replicated and partitioned key-value datastores, the CAP theorem, consistency models and causal consistency.

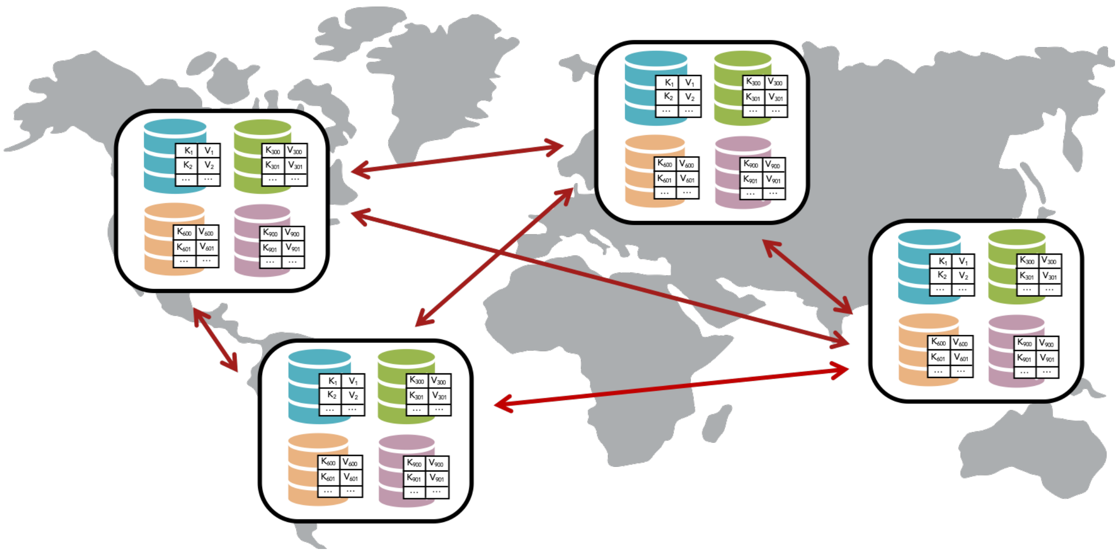


Figure 2.1 – A key-value store deployed in multiple data centers worldwide. The data in a DC is partitioned into several partitions, which are being replicated at all DCs.

2.1 Introduction

The beauty and power of a key-value store is its simplicity. Key-value stores constrain the granularity of data to a single key-value pair, allowing them to be scaled to thousands of servers. There are no complex schemas and data relationships, so they offer fast response times. However, although they have simple semantics, key-value stores are not simple. The users of these systems expect to quickly and reliably access their data from anywhere and at anytime. Geo-replication and sharding have become fundamental features of high performance and reliable key-value datastores.

Figure 2.1 shows a typical deployment of a key-value store. To enable availability and fault tolerance, key-value data stores are geo-replicated. Geo-replication allows keeping a copy of the data in a data center (DC) closer to the users, thus reducing data access latencies. Sharding enables horizontal scalability, by slicing the dataset in disjoint partitions, each of which can be assigned to a different server. Geo-replication and sharding create a challenging, and even hostile, environment for preservation of the consistency guarantees of data. In a distributed environment, unavoidably, servers do not progress at the same pace. Furthermore, if there are multiple copies of the data on different servers, they may be inconsistent with each other, and an application that is not designed to handle such inconsistencies may produce incorrect results. Geo-replication exacerbates this problem even further. The choice of the consistency model becomes one particularly important design choice that software developers need to make.

2.2 CAP theorem

An ideal distributed system should act as infinitely scalable and fault-tolerant version of a centralized system. Ultimately, geo-replication and sharding should be used to improve availability by disguising failures, reduce latency and provide scalability. Moreover, an ideal distributed system should conserve the simplicity of use of the centralized system by keeping its consistency guarantees and providing the illusion of linearizable access. However, such ideal distributed system is not possible to construct.

The CAP Theorem [24] states that in any networked shared-data system there is a fundamental trade-off between availability, strong consistency and partition tolerance. The problem is that each of these properties is highly desirable. Availability ensures that clients will always get responses to their requests. Strong consistency makes it easier for the software developers to reason about the ordering of data operations, thereby reducing the likelihood of data loss due to race conditions. Partition tolerance guarantees that the system can continue operating properly even when data centers cannot communicate with one another. If there is no network partition, a system can be both consistent and available. However, in a distributed system, network partitions are inevitable, so every distributed system has to be designed to tolerate network partitions (P). When a network partition occurs, the system must choose between either strong consistency (C) or availability (A). When designing distributed systems, this choice has pushed software engineers in two major directions: CP systems and AP systems.

CP systems. These systems preserve the strong consistency of the system. Namely, by synchronizing replicas to process events in the same sequential order, a CP system acts as a single-machine system. This conceals the complex nature of the data replication process. However, because it requires all the replicas to agree on the order of events, it significantly increases the latency of the data operations. Moreover, in the event of a network partition, a CP system will block, rather than return inconsistent data or process updates that might be conflicting.

AP systems. AP systems sacrifice consistency to achieve high availability. When a network partition occurs, the nodes will continue to serve requests. However, updates may not be replicated across all replicas. In such case, during a read the system will return the most recent version of the data that is present on a node, which could be stale. Typically, AP systems also accept updates in the event of a network partition, and they process them immediately, at least partially. As soon as the network partition is resolved, the updates are being propagated to other corresponding nodes that share the data being updated.

In this dissertation we focus mainly on AP system designs.

2.3 Consistency models

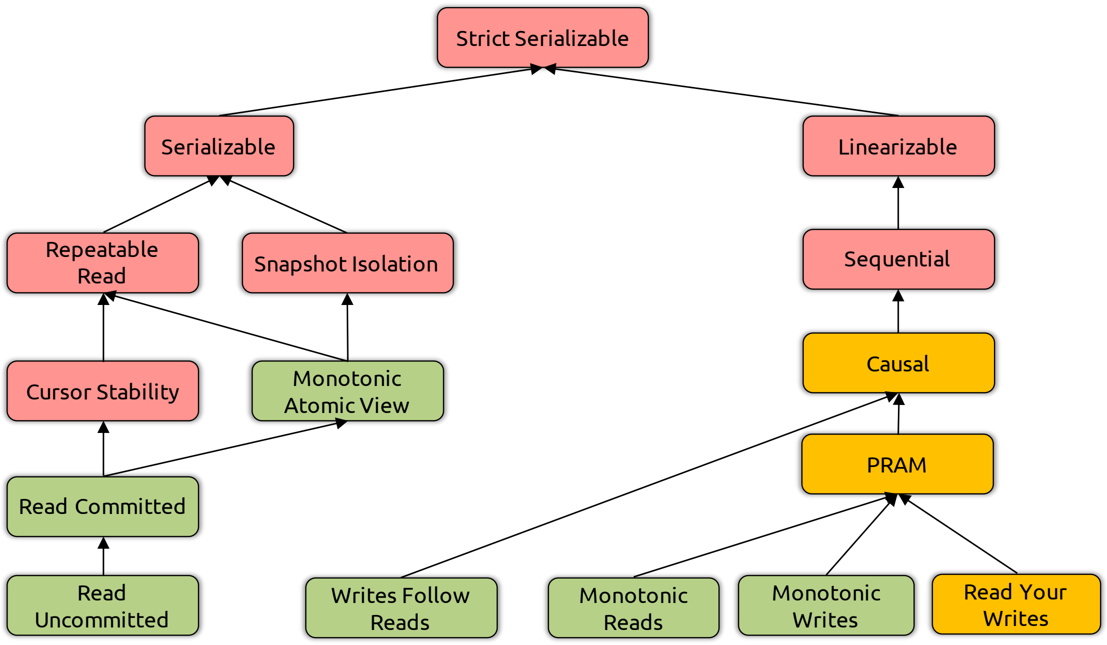


Figure 2.2 – An illustration of the adapted hierarchy of the consistency models based on the taxonomy presented by [13], [108] and [57]. Unavailable models are shown in red, sticky available are shown in yellow and highly available are shown in green.

Informally, a *data consistency model* can be viewed as a contract between a software system and a software developer who uses it. A system is said to support a certain data consistency model if its operations respect the rules defined by the model. When we are dealing with a centralized system, which is running a single process, it is fairly simple to provide the strongest form of consistency. The events in such system, by default, are executed in a sequential order. However, when the system is distributed, and is executing a large number of processes, guaranteeing the ordering of any two events at different processes is challenging. This has led to the design of systems that offer different flavours of consistency guarantees.

Over the years, the meaning of the term *consistency model* was constantly evolving. In the literature, there are many papers that group, order and categorize the consistency models. Figure 2.2 shows an adaptation of the hierarchy of the consistency models based on the taxonomy presented by [13] and [108].

Linearizability. At one end of the data consistency spectrum, linearizability [52] provides strict guaranties about the effective order of potentially concurrent data operations, making it easier for the software developers to reason about concurrency. A system that provides linearizability behaves as if all operations are executed atomically on a single copy of the data. However, the cost of implementing total order of data operations is very high, because it incurs high latency and does not tolerate network partitions [24].

Eventual consistency. At the other end, eventual consistency provides excellent performance and partition tolerance [109], but it is hard to program against. Eventual consistency guarantees that after the last update is made to a given data item, all read operations for that item, in every replica, will eventually return the last updated value. However, it does not guarantee any ordering on operations on different data items in any way, which could cause the system to expose various anomalies and inconsistencies to users. Consider the social network scenario shown on Figure 2.3. Alice posts a message saying that she will miss the train, and afterwards posts a comment on her previous post saying that she was lucky enough to catch the train. In the end, her friend Bob, expressed that he is happy that Alice caught the train. However, Eve, who is connected from a geographically different data center, sees the comments in the wrong order, and gets the impression that Bob is happy that Alice is going to miss the train. This is known as the *comment reordering anomaly*, which is only one of the possible anomalies that eventually consistent systems allow to happen.

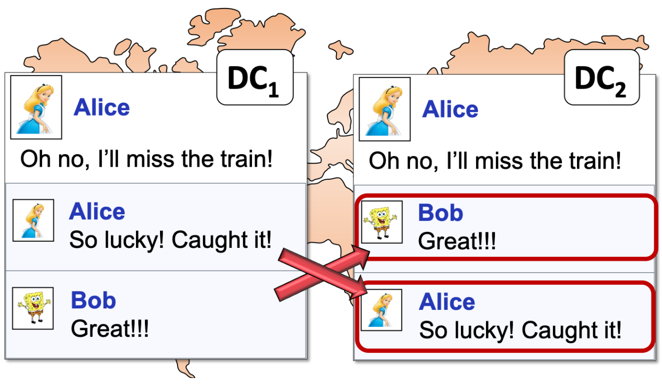


Figure 2.3 – A social networks example of the comment reordering anomaly in an eventually consistent system.

2.4 Causal consistency

2.4.1 Why causal consistency?

Causal consistency [2] lies between the two endpoints of the consistency spectrum. It represents an attractive model for building geo-replicated data stores because it hits a sweet spot in the ease of programming vs. performance trade-off for this kind of systems [69, 70, 43, 5, 12, 42, 114, 38, 96, 72, 4]. On the one hand, it avoids the performance penalty and the scalability bottleneck of strong consistency. On the other hand, it is easy to reason about, and it avoids some anomalies that are otherwise allowed under eventual consistency.

2.4.2 Definition

Causal consistency requires that servers of a system return values that are consistent with the order defined by the *causality* relationship. Causality is a happens-before relationship between two events [66, 2]. For two operations a, b, we say that b causally depends on a, and write $a \rightsquigarrow b$, if and only if at least one of the following conditions holds:

- **Thread-of-execution**: a and b are operations in a single thread of execution, and a happens before b.
- **Reads-from**: a is a write operation, b is a read operation, and b reads a value that has been written by a.
- **Transitivity**: an operation x exists, such that $a \rightsquigarrow x$ and $x \rightsquigarrow b$.

Definition 1. *(Causal consistency) A system is causally consistent if the clients of that system see events in an order consistent with the "happens-before" relation.*

Causal consistency satisfies the following four session guarantees: read-your-writes, monotonic reads, monotonic writes, and writes-follow-reads [100] (Figure 2.2). The read-your-writes guarantee ensures that a read performed by a client cannot return a value that is older than a value written earlier by the same client. The monotonic reads guarantee ensures that a read made by a client includes at least the effects of the writes already observed by a previous read made by that client. The monotonic writes guarantee ensures that a write made by a client only takes effect after all previous writes made by that client. The writes-follow-reads guarantee ensures that writes made by a client take effect only after the writes whose effects were observed by reads made by that client. If a system ensures these four session guarantees, it also ensures causal consistency.

2.4.3 Convergent conflict handling

Two operations a, b are concurrent if neither $a \rightsquigarrow b$ nor $b \rightsquigarrow a$. If both a and b are concurrent write operations on the same key, they *conflict*.

Namely, their result can be propagated to replicas in different orders, potentially leading to replicas to diverge forever. To enforce that different replicas converge to the same value for any key, a protocol must implement a convergent conflict handling procedure. Such procedure must implement a commutative and associative function, such that replicas can manage update replication messages in the order they receive them and converge to the same state.

The most popular convergent conflict handling function is the last-writer-wins (LWW) rule (Thomas' write rule [104]). LWW rule states that in presence of two conflicting updates, one of them is deterministically decided to having occurred later than the other by using a unique total-order identifier. This function is applied independently at every replica, without any communication with the other replicas.

Other popular conflict resolution technique is by using Conflict-Free Replicated Data Types (CRDTs). CRDT [88] represents an abstract data type, such as register counter, set, table or map, that is designed to be replicated at multiple servers. The replicas can be modified without the need of coordination with other replicas. Moreover, if two replicas have received the same updates, they will reach the same state by applying mathematical rules that guarantee state convergence.

The implementation of the systems presented in this dissertation uses, for simplicity, the last-writer-wins rule to achieve convergent conflict handling. However, they can be extended to support CRDTs.

2.4.4 Transactional causal consistency

Over the past decade, there has been an increased interest in transactional causally consistent systems [69, 70, 4] Indeed, in practise, software developers value the stronger semantics that transactions can offer, because it preserves data integrity and prevents additional anomalies. Consider the "leaked photo anomaly" scenario shown on Figure 2.4. Initially, Alice has given Bob permission to access her photo album. However, after taking a controversial photo at a party, she decides to revoke Bob's permissions from her album, and then she adds the photo that she does not want Bob to see. Concurrently, Bob wants to view Alice's album, and in this scenario, he ends up seeing the controversial photo that Alice wanted to hide from him. The problem is that the system has read the old permissions and then the new version of the album, which should not have been allowed to happen if both the permissions and the album were read in a single transaction.

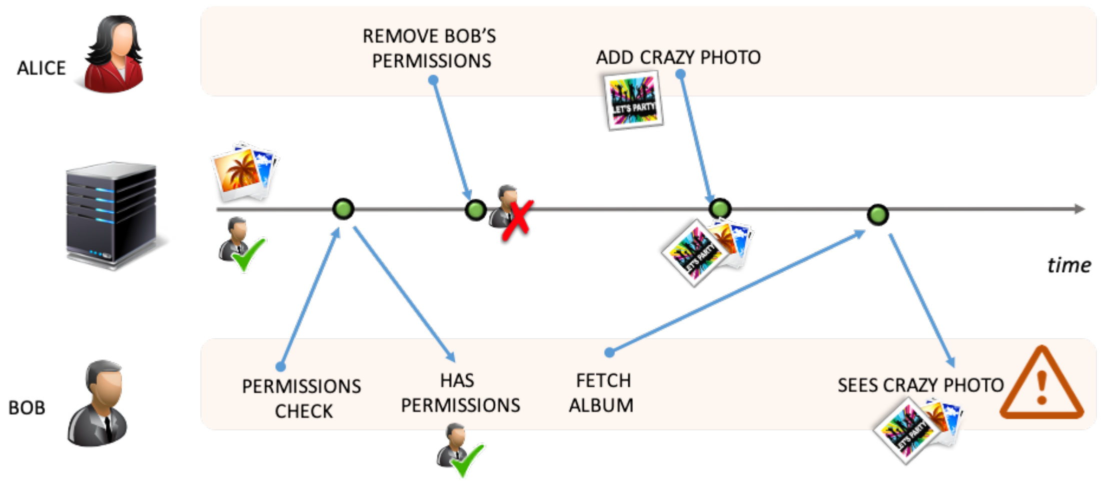


Figure 2.4 – Leaked photo anomaly.

Semantics. TCC extends CC by means of interactive read-write transactions in which clients can issue several operations within a transaction, each reading or writing potentially multiple items [4]. TCC provides a more powerful semantics than one-shot read-only or write-only transactions provided by earlier CC systems [72, 69, 42, 70]. It enforces the following two properties:

1. Transactions read from a causal snapshot. A causal snapshot is a set of item versions such that all causal dependencies of those versions are also included in the snapshot. For any two items, x and y, if $X \rightsquigarrow Y$ and both X and Y belong to the same causal snapshot, then there is no X', such that $X \rightsquigarrow X' \rightsquigarrow Y$.

Transactional reads from a causal snapshot avoid undesirable anomalies that can arise by issuing multiple individual read operations. For example, they prevent the previously described anomaly shown in Figure 2.5. Figure 2.5 shows the correct execution of same scenario with the help of transactions.

Consecutive one-shot read-only transactions, separated by writes, read from different snapshots. TCC, instead, ensures that all reads within an interactive transaction are served from the same causal snapshot, supplemented with the transaction's own writes, making it easier to write and reason about applications.

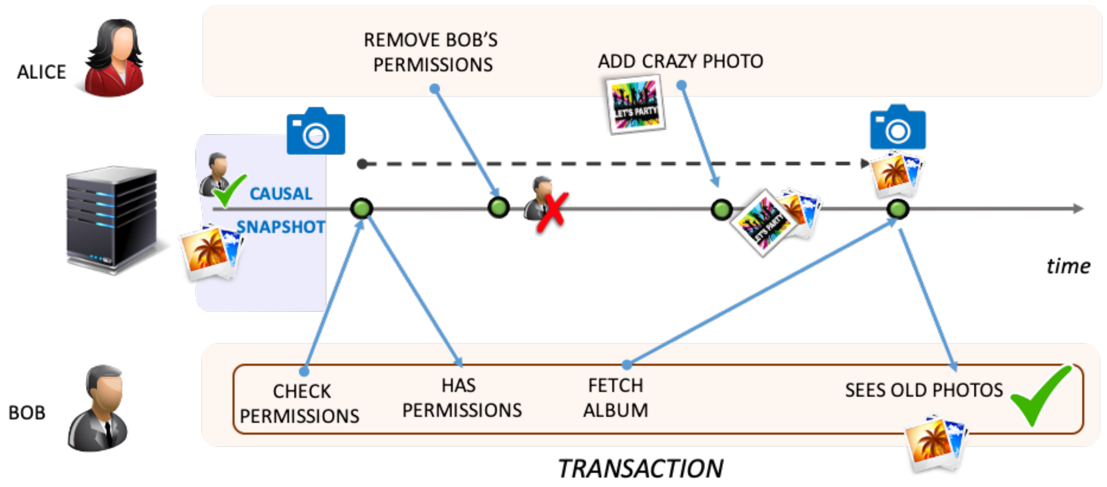


Figure 2.5 – Causal snapshot prevents the leaked photo anomaly.

2. *Updates are atomic.* Either all items written by a transaction are visible to other transactions, or none is. If a transaction writes X and Y, then any snapshot visible to other transactions either includes both X and Y or none of them.

Atomic updates increase the expressive power of applications, e.g., they make it easier to maintain symmetric relationships among entities within an application. For example, in a social network, if person A becomes friend with person B, then B simultaneously becomes friend with A. By putting both updates inside a transaction, both or none of the friendship relations are visible to other transactions [70].

2.4.5 Partial replication

The current trend of exponential data growth, together with the recent proliferation of mobile applications, has shifted the system designers' attention towards geo-replicated systems that support partial replication. In a partial replication setting, each data center stores a subset of the partitions, as opposed to the full replication setting, where each data center stores the whole dataset. Figure 2.6 shows a typical deployment of partially replicated key-value store.

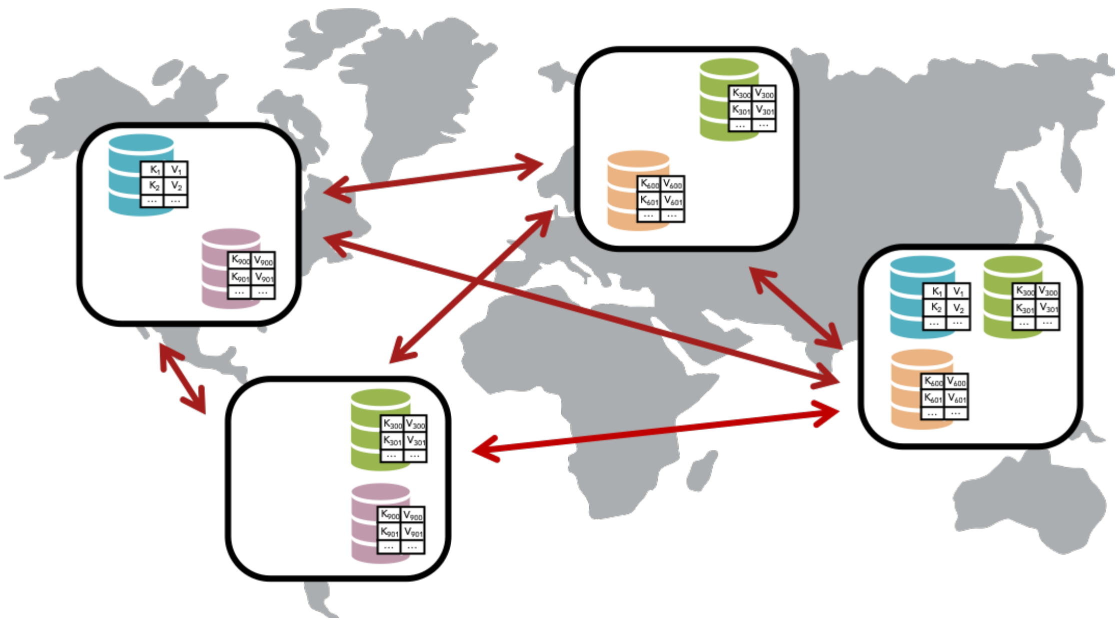


Figure 2.6 – Partially replicated key-value store deployed in multiple data centers worldwide.

Partial replication offers several benefits over full replication. The fact that every data center typically stores only a subset of all partitions in the system, enables the system to scale to a higher number of partitions. Partial replication is an effective way to reduce storage requirements and replication costs, enabling higher storage capacity.

3 Optimistic Causal Consistency

3.1 Introduction

Existing systems employ different techniques to achieve causal consistency, but they all share the same key mechanism. When serving a read operation, a server within a DC returns the most recent version of an item whose causal dependencies are known to have been already replicated in that DC. This invariant allows such data stores to tolerate network partitions and DC failures. It demands, however, the implementation of dependency checking [69, 70] or stabilization protocols [42, 4] to track the delivery of dependencies. Not only do these protocols result in computational and communication overhead, but they also delay the visibility of new versions of data items, increasing the staleness of the data returned to clients. Besides increased data visibility latency, these dependency checking and stabilization protocols can cause even more severe and covert issues in the case when one component of the system is slower or lagging behind the others. This problem is relevant and not infrequent in practice, since it has been pointed out as one of the barriers of not choosing a causally consistent data model for industry deployment [3].

Optimistic Causal Consistency. In this chapter we argue that existing protocols are too *pessimistic* for modern data center deployments. To tackle the aforementioned issues that affect such pessimistic systems, we propose **O**ptimistic **C**ausal **C**onsistency (OCC). According to OCC a server always returns the most recent available version of an item. Potentially unresolved dependencies are detected by the server upon serving a read operation by means of cheap dependency meta-data supplied by the client, without the need for synchronization with other servers. When a potential unresolved dependency is detected, a server waits to receive it. To accomplish this, we introduce CALDeR (**C**lient-**A**ssisted **L**azy **De**pendency **R**esolution) protocol, a novel dependency resolution protocol that relieves the data store from the responsibility of tracking the delivery of updates by shifting it to the client. CALDeR eliminates the need of synchronization among the partitions, which implicitly eradicates the problem caused by a slower partitions that are lagging behind.

The main goal of OCC is to maximize data freshness. Data freshness represents a desirable property for all types of systems. Until recently, it had been omitted from the primary design goals of causally consistent systems, since it makes enforcing causal consistency a challenge. However, with the proliferated development of real-time systems, IoT devices, high-frequency trading systems, collaborative systems and massively multi-player online games, data freshness becomes a critical design goal [28, 59, 58]. These types of systems would benefit greatly from a data store that offers maximum data freshness.

Contributions. In this chapter, we make the following contributions.

1. We introduce Optimistic Causal Consistency, discussing its benefits and inherent performance vs. availability trade-offs.

2. We present the design and implementation of POCC (**O**ptimistic **C**ausal **C**onsistency with **P**hysical clocks), a causally consistent protocol that implements OCC. POCC relies on physical clocks and vector dependency clocks to succinctly keep track of causal dependencies.

3. We discuss the trade-off between data freshness, performance and safety of OCC. Furthermore, we discuss a highly available design of POCC that can fall back on a pessimistic approach to remain available during network partitions.

4. We assess the benefits brought by OCC by comparing POCC with a (pessimistic) state-of-the-art design to causal consistency based on physical vector clocks. We run heterogeneous workloads on a large scale infrastructure on Amazon, comprising up to 96 servers scattered across 3 geo-distributed sites. Our results indicate that optimistic causal consistency is effective in minimizing the staleness of data returned to client and delivers performance that is competitive or better than its pessimistic counterpart in a wide range of realistic workloads.

The remainder of the chapter is structured as follows. Section 3.2 describes the motivation behind OCC. Section 3.3 describes causal consistency and the target system model. Section 3.4 presents the approach underlying OCC. Section 3.5 describes the protocols of POCC. Section 3.6 discusses the correctness of POCC. Section 3.7 introduces the highly available OCC. Section 3.8 provides the evaluation of POCC. Section 3.9 concludes the chapter.

3.2 Motivation

Geo-replicated systems that enforce causal consistency using a pessimistic approach can suffer from performance degradation and increased data visibility latency under unexpectedly high load and when co-located tenants with high resource demands. Under this scenario, the dependency tracking and stabilization protocols, although seemingly inexpensive to run, become the main bottleneck of the system.

As an example, consider an event ticketing system designed to serve millions of events and event guests. The system is partitioned for scalability, and it has multi-master geo-replication between two data centers, for fault tolerance. A very popular artist, with millions of fans, uses this system to sell tickets for a special concert and announces that the sale of the tickets will start on a particular date and time. The data for this particular concert is stored on partition p. As it usually happens with popular events, the tickets are sold out within the first few minutes after the sale has started. Building such system with eventual consistency might be possible, but identifying and addressing the possible anomalies beforehand will be hard. Hence, a stronger consistency model is needed for this kind of system. On the other side, strong consistency models would be prohibitively expensive, performance-wise. With these objectives on mind, causal consistency would be a reasonable choice.

The process of having millions of concurrent users, requesting a ticket at nearly the same time, will result in high load on partition p, causing it to lag behind the other partitions of the system. An update u can be made visible, only after all of u's causal dependencies have already been made visible, which requires maintenance of a per-key chain of versions. To enforce causal consistency, partitions have to synchronize in order to track the delivery of updates, so the slowest partition, in this case p, will stall the progress of the whole system. The number of invisible updates in the version chains will increase, not just on p, but in all partitions of the system. This will cause performance degradation in all partitions due to the large version chains, in the worst case leading to memory exhaustion. It will affect all events hosted on the ticketing system, despite the fact that they are totally unrelated, potentially creating a great number of unhappy users.

Another related issue is the increase of the data visibility latency. Proceeding with the same example, assume that the event ticketing system is deployed on AWS, in the North Virginia (us-east-1) and Ohio (us-east-2) regions. The average one-way trip latency between these regions is ~5 ms [1]. A reasonable frequency of running stabilization protocols in existing systems is to run it every 5-10 ms [42, 38, 96]. However, if we consider that the nodes belonging to one data center can experience clock skew in the range of 1-2 ms, as well as the communication overhead, it would take up to 15 ms to calculate the new stable time and promote invisible updates to visible. This will cause data visibility latency of up to 20 ms (5 ms for geo-replication + 15 ms for stabilization). Under normal conditions, the client could potentially pay 4x increase in data visibility latency. The data visibility latency could drastically increase even further if **any** partition of the data store is facing a heavy load, or if it is co-located with a tenant with high resource demands, which is not unusual in cloud deployments.

Running highly available multi-master data store implies having inconsistencies, e.g. lost updates. Considering the high probability of write conflicts in frequently updated items of the data store, every additional millisecond of visibility latency increases the chances for inconsistencies and multi-master conflicts. Relating to our example, having greater visibility latency when calculating the number of remaining concert tickets could cause higher number of overbooked tickets. For simplicity we assume that the system initiates ticket reservations in each data center at constant speed, hence a 4x increase of visibility latency could result in 4x more overbooked tickets. On the contrary, if the system was able to provide fresher data, the number of such inconsistencies would have been at the minimum that is achievable without losing availability or performance.

These problems served us as motivation to search for a different approach to implement causal consistency in a way that is more resilient to load spikes, co-location with tenants with high resource demands, and at the same time to provide higher data freshness.

The effectiveness of our optimistic approach stems from two main insights. First, recent works have revealed that updates replication in data stores exhibits a *naturally consistent order*. Namely, a data item is typically replicated (and accessed) after its dependencies have already been propagated [110, 15, 71]. Hence, the most recent version of a data item could typically be returned without violating consistency [48]. OCC leverages this insight by having servers waiting to receive missing dependencies when serving a client's request, rather than relying on expensive dependency checking and stabilization protocols. Servers rarely incur such waiting overhead, because of the naturally consistent order of updates. Hence, OCC maximizes the freshness of data returned to clients and improves resource efficiency with negligible impact on performance.

Second, network partitions can be considered relatively rare events (and complete DC failures even more rare) [15, 93, 23, 67]. OCC leverages this insight by avoiding (most of) the overhead associated with network partition tolerance during normal operative conditions, and by incurring it only when a network partition is actually occurring. To this end, OCC entails the infrequent –hence cheap– execution of a stabilization protocol to allow a system to fall-back to operate according to a pessimistic scheme in presence of a network partition. This design contrasts with existing ones, which incur frequent dependency checking overhead and expose stale data items to clients even under normal operational conditions.

3.3 System model

3.3.1 System model

We assume a distributed key-value store that manages a large set of data items. The data-set is split into N partitions and each key is deterministically assigned to a single partition according to a hash function. Each partition is replicated at M different sites, each corresponding to a different data center. Hence, a full copy of the data is stored at each data center.

We assume a multiversion data store. An update operation creates a new version of an item. We say that one item version is fresher than another, if it comes after it with respect to the causally consistent order. In addition to the actual value of the key, each version also stores some metadata, in order to track causality. The system periodically garbage-collects old versions of items. We further assume servers in the system can communicate through point to point lossless FIFO channels.

Unless stated otherwise, we use lower case letters, e.g., x, to refer to a key and the corresponding capital letter, e.g., X to refer to a version of the key. A version X of a data item x is causally dependent on a version Y of data item y if the write of X causally depends on the write of Y.

We define an item X as *stable* in a DC if all the dependencies of X have already been replicated in such DC.

The system provides the following operations to the clients:

- PUT(key, val): A PUT operation assigns value val to an item identified by key. If item key does not exist, the system creates a new item with initial value val. Else, a new version storing val is created.

- val ← GET(key): A GET operation returns the value of the item identified by key. A GET operation is such that its return value does not break causal consistency as explained in the following. Assume X is a version of data item x, Y is a version of data item y and $X \rightsquigarrow Y$. Suppose that a client c issues a GET(y) operation, receiving Y as result. Then, any subsequent GET(x) operation issued by c must return either X or a version X' such that $X' \not\rightsquigarrow X$.

- ⟨vals⟩ ← RO-TX⟨$keys$⟩: This operation provides a causally consistent read-only transaction [69, 70], that enables the client to read a set of items corresponding to the input set of keys. Namely, assume X and Y are two versions of items x and y, respectively. If a read-only transaction returns X and Y, and $X \rightsquigarrow Y$, then there does not exist another version of x, X', such that $X \rightsquigarrow X' \rightsquigarrow Y$.

At the beginning of a session, a client c connects to a server p in the closest data center according to some load balancing scheme. c does not issue the next operation until it receives the reply to the current one. Operations on data items not stored by p are transparently forwarded to the server(s) responsible for such data items, and the result is relayed back to c by p.

Availability. We use the term availability to indicate that a client operation never blocks as the result of a network partition between DCs [24].

3.4 Optimistic Causal Consistency

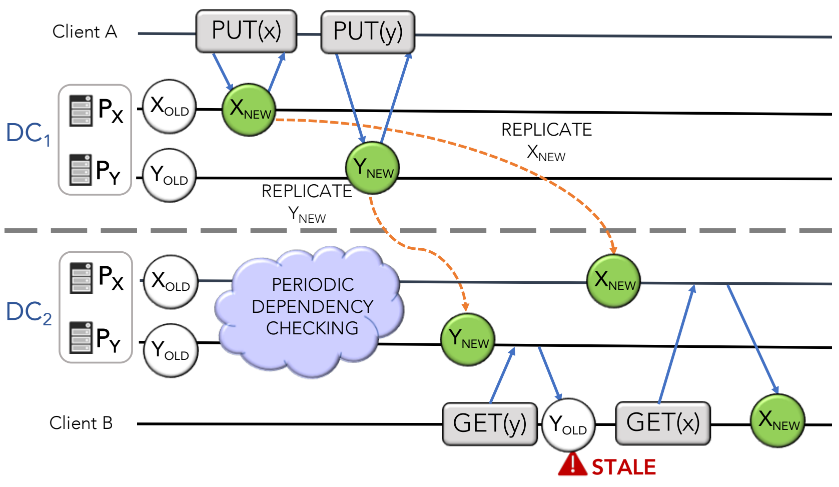
In this section, we describe the design of OCC. We first contrast the optimistic approach of OCC with the pessimistic one of existing systems. Then, we explore the trade-off between data freshness, performance and availability. Next, we describe how OCC enforces causal consistency efficiently by shifting the burden of dependency checking to the clients using the CALDeR protocol. We also describe how OCC improves data freshness. Then, we provide a discussion that highlights the rationale behind the design of OCC. Finally, we describe how OCC can be augmented to preserve availability during network partitions.

3.4.1 The design of OCC

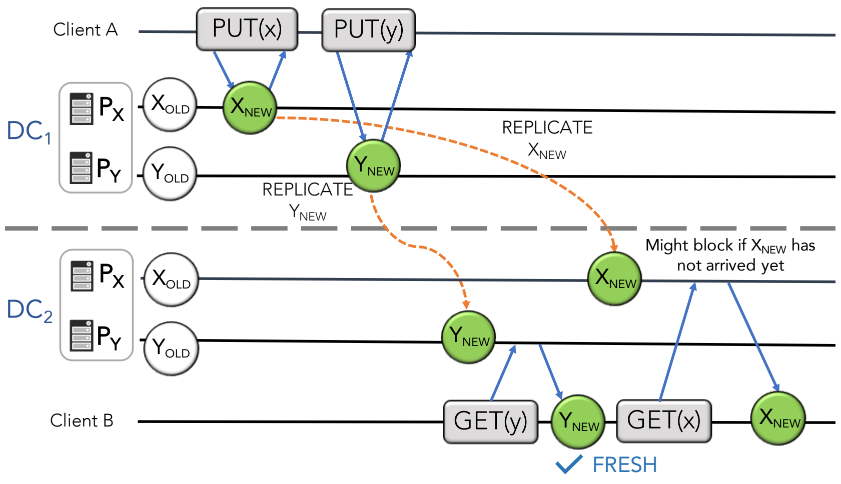
We describe OCC by means of an example, depicted in Figure 3.1, in which we contrast it with existing pessimistic approaches. In our example, the initial versions of x and y are, respectively, X_{OLD} and Y_{OLD}, and are stored by partitions p_x and p_y. Client A in DC$_1$ creates X_{NEW} and Y_{NEW} such that $X_{NEW} \rightsquigarrow Y_{NEW}$. By network asynchrony, however, Y_{NEW} is received in DC$_2$ before X_{NEW}. After Y_{NEW} is received in DC$_2$, client B in DC$_2$ issues a read operation on y and a subsequent read operation on x.

By causal consistency, if client B sees Y_{NEW}, then the following read on x must return X_{NEW}. Enforcing this behavior is nontrivial because DC$_2$ has received Y_{NEW} but not its causal dependency. Hence, client B is at risk of observing Y_{NEW} and X_{OLD}, which violates causal consistency.

Existing pessimistic approaches. State of the art systems use different methods to tackle this issue. The key common mechanism, however, is that a replicated version is made visible to clients within a DC only when all of the item's dependencies have been received in that DC. With reference to our example, pessimistic approaches enforce that Y_{NEW} can be read by client B in DC$_2$ only if also X_{NEW} has been received and can be read in DC$_2$. This behavior is depicted in Figure 3.1a.



(a) Pessimistic case



(b) Optimistic case

Figure 3.1 – In systems with pessimistic behaviour (a), a stale version of a y, Y_{OLD}, is returned because X_{NEW} on which Y_{NEW} depends has not been yet replicated in DC$_2$. OCC (b), on the other hand, maximizes data freshness by returning the freshest version of y, Y_{NEW}. The trade-off is that OCC would have to block if client B requested X_{NEW} before it had been replicated to DC$_2$.

On the one hand, this approach allows a system to tolerate network partitions. A data store partition can always return a causally consistent version to a client without waiting for the receipt of a version from a remote DC. On the other hand, this approach is prone to return stale items to the clients, even though fresher versions have been received. In Figure 3.1a, p$_y$ in DC$_2$ returns Y_{OLD} to client B, despite the fact that Y_{NEW} has been received already. In addition, pessimistic approaches incur two sources of overhead. First, they need to implement a distributed dependency tracking protocol, to communicate which items have been received in a DC, and hence to determine which versions can be made visible. Such protocol is implemented as a background process, hence it is not on the critical path of serving a client request, but it consumes CPU and network bandwidth. Second, when serving a client request, a partition needs to find the freshest visible version of the requested key. This requires traversing a number of versions of the key received so far by the partition. Performing such traversal directly impacts the latency of the read operation.

Optimistic approach. OCC departs from conventional approaches in that it allows a partition in a DC to return a version of an item to a client despite the fact that the version's dependencies have not been received in the DC. The optimism of the approach lies in assuming that the dependencies of such item will be received in the DC by the time the client wants to access them in following operations.

Figure 3.1b shows how OCC behaves in our previous example. p_y in DC_2 returns Y_{NEW} to client B despite X_{NEW} not having been received by DC_2 yet. Then, by the time client B reads x, X_{NEW} has been received in DC_2 and can be returned, preserving causal consistency.

The clearest benefit of OCC is that it increases the freshness of the data returned to clients. Not only does this improve the user experience, but it also has a positive effect on performance, because it avoids traversing the dependency list to find the right stable version to return. In addition, OCC does not require any communication among partitions to determine the set of items that can be made visible within the DC, which further improves resource efficiency.

Performance. From a performance point of view, OCC is a more desirable design choice. First, OCC does not impose the overhead of constantly running protocols for tracking the delivery of dependencies. OCC relieves the data store of this responsibility and pushes it towards the client, by storing inexpensive meta-data for detection of missing dependencies in the client. Second, OCC improves the response time by not having to traverse the item version chain. Considering these benefits in performance, we believe that for applications where the data store resource efficiency is important, OCC would be reasonable, if not favorable design choice.

Trade-offs

OCC's potentially blocking behavior makes it vulnerable to network partitions, since a read operation of a client can block. With reference to our example illustrated on Figure 3.1b, this happens if client B reads x before X_{NEW} is received by DC_2. In that case, p_x in DC_2 needs to wait for the receipt of X_{NEW} in order to preserve causal consistency. Furthermore, if X_{NEW} is prevented from being replicated to DC_2 by a network partition. Since client B has already established a dependency on X_{NEW} by reading Y_{NEW}, p_x from DC_2 must block until the network partition heals to receive X_{NEW} and, thus, to preserve causal consistency.

In such case, the client will be given the flexibility to choose between three options. First, it can wait for the network partition to heal. Next, it can choose to reconnect to another data center. This might or might not solve the problem, because that data center could be missing X_{NEW}, too. Finally, it could continue with running eventual consistency until the problem is resolved, in which time frame it might break the causal consistency guarantees. Namely, OCC offers the client a choice between availability and consistency, depending on the use case. We further address the limitations of this approach by presenting a highly available OCC design, described at a later point.

Existing pessimistic causally consistent protocols do seamlessly tolerate network partitions, and full DC failures (which can be modeled as unhealed network partitions). OCC trades this ability for a higher freshness in data returned to the client and greater resource efficiency.

3.4.2 Client-assisted lazy dependency resolution

The challenge in OCC is to enforce causal consistency while exposing unstable item versions in a cost-effective way, that will, simultaneously, minimize both the meta-data overhead and the read response time. OCC achieves this by means of the CALDeR protocol. The main idea behind the CALDeR protocol is to empower the client with the ability to track whether a read operation is safe in a lightweight manner, and, hence, relieve the data store from the burden of running distributed protocols to track the delivery of dependencies.

Clients store information about the causal dependencies established when performing reads in the form of *client-side dependency meta-data*. If c reads Y and there exists X such that $X \rightsquigarrow Y$, c records that it has established a dependency towards X and Y. The client-side dependency meta-data is supplied by c when it performs a later operation.

For a read operation (GET or RO-TX), client-side dependency meta-data is needed to allow a server p to determine whether its own state is consistent with c's history. Referring to the previous example, if c wants to read X after it has read Y, the dependency meta-data provided by c, together with some state information that p locally stores, allows p to check whether it has already received X or not. If p has already received X, then the freshest local version of x that p stores is compatible with c's history, and it is returned to c. If p has not received X yet, p must receive it before serving c's request. In this case, p simply stalls c's request until it receives X.

For a PUT operation, the client-side dependency meta-data is needed to know what the newly created item depends on. Moreover, such information might also be needed to enforce state convergence. Some convergent conflict handling schemes, in fact, require that upon writing Y on server p, p must have already delivered all the versions of y on which Y causally depends [69, 43, 42]. In this case, in OCC, p must wait, if necessary, until it has received the freshest version of y on which the issuing client depends.

3.4.3 Discussion

OCC offers a different way to implement causally consistent systems by offering higher data freshness to clients and higher resource efficiency than the existing state-of-the art systems. The trade-off is that it is potentially blocking. The effectiveness of OCC relies on the insight that, typically, blocking is a rare event because updates are propagated in order of creation. Therefore, a server typically already stores a version of a key that is causally consistent with a client's history. These claims are backed by empirical evidence gathered on popular commercial cloud data stores [110, 48, 7], a massive scale deployment such as Facebook's [71] and are also confirmed by analytical models [15]. Furthermore, we support this claim that OCC has low blocking probability, with our experimental results in Section 3.8.

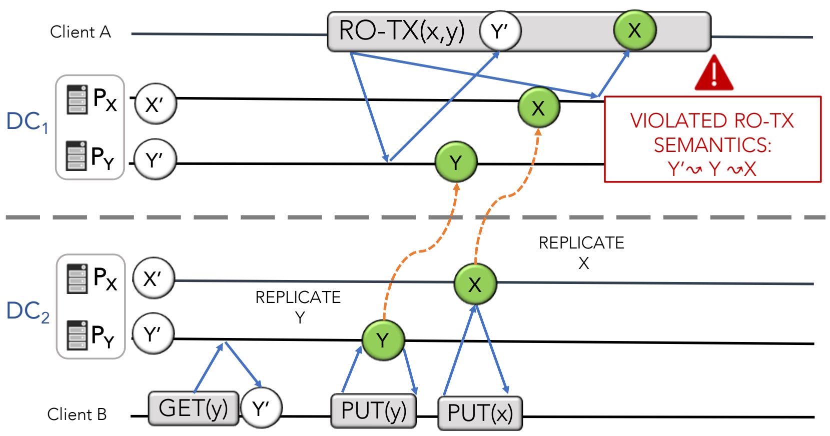
3.5 Implementation

In this section, first, we give a higher level overview of how OCC improves data freshness by implementing a different protocols for the GET operation and the RO-TX. Then, we present POCC, a scalable implementation of OCC with physical clocks.

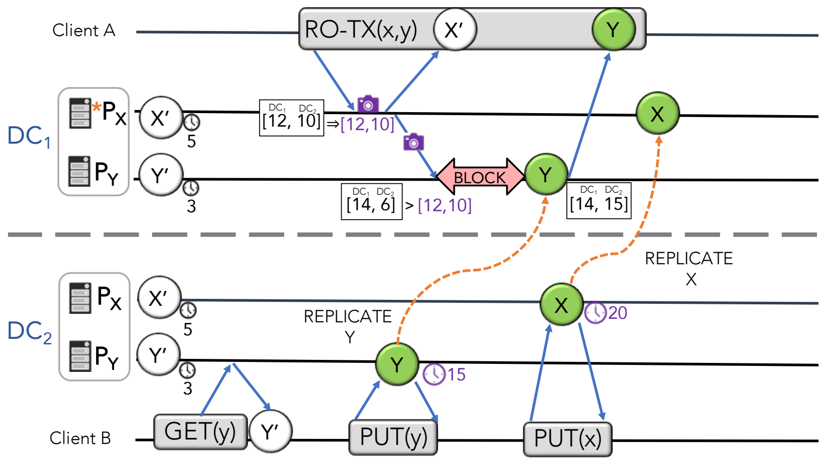
3.5.1 Overview

The primary design goal of OCC is improving the freshness of data returned to the client. Doing so while preserving causal consistency requires the implementation of different protocols depending on whether the client performs a GET operation or a RO-TX. In the remainder of this subsection, we will show how OCC improves data freshness in both cases, when serving a GET operation and when executing a RO-TX.

OCC can be implemented with any dependency tracking mechanism that has been proposed in literature, e.g., dependency lists [69], dependency matrices [43], Hybrid Logical Physical Clocks (HLC) [61], physical scalar clocks [42], physical vector clocks [4]. Because the client has to supply dependency information upon each operation, however, it is paramount to encode dependencies in the most succinct fashion possible. As we shall show in Section 3.5.2, POCC, our implementation of OCC, meets this requirement by relying on physical vector clocks, which can keep track of dependencies with a overhead that is only linear with the number of data centers in the system.



(a) Naive OCC RO-TX implementation



(b) Correct OCC RO-TX implementation

Figure 3.2 – Naive (incorrect) OCC implementation: naively maximizing data freshness in RO-TX by always returning the freshest present item versions could result in violation of the RO-TX semantics(a). Correct OCC implementation (b).

GET operations

When a server p receives a GET(x) request from a client c, p returns the freshest version that it stores, provided that such version is fresh enough and it does not violate causal consistency.

Assume that c has established a dependency towards X_{NEW}, e.g., by reading Y_{NEW}, with $X_{NEW} \rightsquigarrow Y_{NEW}$. If the freshest version of x stored by p is X^*, an even fresher version than X_{NEW}, then p can return X^* to c. If X_{NEW}, instead, is fresher than X^*, then p must block the client operation until p receives X_{NEW}.

Read-only transactions

Implementing RO-TX in an optimistic fashion is more challenging than GET operations. This is because all the reads that are performed within a RO-TX must be causally consistent with the history of the client and also among themselves. The complexity of RO-TX is further exacerbated by the fact that individual read operations take place in parallel on different partitions. Hence, each partition involved in a RO-TX does not know which keys and which versions are returned by other partitions involved in the same RO-TX. For these reasons, a partition that processes a RO-TX cannot simply return the freshest version of a requested key. Doing so, in fact, may lead to consistency violation, as we show with an example, which we depict in Figure 3.2.

In DC$_2$, client B does a series of operations by which it creates the following causal relation between different versions of items x and y: $Y' \rightsquigarrow Y \rightsquigarrow X$. In DC$_1$, client A, which has not established any dependencies so far, does a RO-TX involving keys x and y. When the read request for y gets to p_y, the freshest version of y present in p_y is Y'. When p_x gets the read request for x, the freshest present version of x is X.

If the partitions in DC$_1$ served the reads within client A's RO-TX in the same fashion as two individual GET operations, then the RO-TX would return Y' and X. Such return values are not causally consistent, because there is Y such that $Y' \rightsquigarrow Y \rightsquigarrow X$. To avoid this erroneous behaviour, and at the same time to improve data freshness, OCC leverages the concept of causal snapshot. A causal snapshot is a set of item versions such that all causal dependencies of those versions are also included in the snapshot. Upon starting, a RO-TX is assigned a causal snapshot, and, for each key in the RO-TX, each partition returns the freshest version of that key that belongs to the snapshot.

In OCC, the boundaries of a causal snapshot are defined on the basis of the items currently received by servers in the local data center when the transaction is issued. In existing systems, instead, such boundaries are determined by the set of items that are stable at the time the transaction is issued.

Figure 3.2b shows how OCC implements RO-TX in the scenario represented on Figure 3.2a. When c_A in DC_1 starts the RO-TX to read x and y, it first contacts the transaction coordinator, p_x, that orchestrates the RO-TX transaction and defines the boundaries of the RO-TX transaction snapshot. Since c_A has not established any dependencies yet, the snapshot is determined by the latest received items by p_x, or in their absence, the latest received heartbeat messages. In this example, the snapshot is encoded with the dependency vector [12, 10], which indicates that the snapshot includes all items that originated from DC_1 with timestamp lower or equal to 12 and all items that originated from DC_2 with timestamp lower or equal to 10. Next, p_x forwards the read request for y, together with the transaction snapshot, to p_y and returns X' to the c_A as X' is the latest received version of x that belongs to the snapshot. When p_y receives the read request, it has to wait until the latest item received from DC_2, which currently has the value 6, surpasses the snapshot value 10. After receiving Y, which is the latest value replicated from DC_2, p_y returns Y, as the freshest value of item y that belongs to the transaction snapshot.

3.5.2 POCC: a scalable implementation of OCC

This section presents the design of POCC, a system that implements OCC. In this chapter, we do not present the implementation of the protocols needed to recover from network partitions or full DC failures, but we only focus on the protocol run by POCC during normal operational behavior.

In POCC each server is equipped with a physical clock, which provides monotonically increasing timestamps. We assume that such clocks are loosely synchronized by a time synchronization protocol, such as NTP [82]. The correctness of our protocol does not depend on the synchronization precision.

In POCC each update u is assigned a physical clock timestamp that represents the time at which the corresponding item has been created. u is also assigned a dependency vector with one entry per DC. Because dependencies are tracked at the granularity of the DC, this vector tracks *potential* dependencies. Namely, if the i-th entry of the vector is t, the update is marked as *potentially* dependent on *every* item originated from the i-th DC with timestamp lower than t. Dependency vectors represent a trade-off between meta-data efficiency and dependency tracking granularity. On the one hand, they allow POCC to succinctly track dependencies. On the other hand they might cause a client's request to be (uselessly) stalled because of a *potentially* unresolved dependency that does not correspond to any real dependency.

We now describe in more detail the protocol the meta-data stored and the protocols implemented by clients and servers in POCC.

3.5.3 Meta-data

Item. An item version d is represented as a tuple $\langle k, v, sr, ut, dv \rangle$. k is the unique id that identifies the item of which d is a version. v is the value of the item. sr is the source replica of d, namely the id of the DC in which d has been created by means of a PUT operation. ut is the update time, i.e., the physical timestamp of the item, computed as the creation time of the item at its source replica. dv represents a dependency vector and consists of M entries. $dv[i]$ is the update time of the item d' with the highest timestamp such that $i)$ d' has been originated at the $i-$th DC and $ii)$ d potentially depends on d'.

Symbol	Definition
N	Number of partitions
M	Number of replicas per partition
p_n^m	The $m-$th replica of the $n-$th partition
$Clock_n^m$	Physical clock time at p_n^m
VV_n^m	Version vector of p_n^m
DV_c	Client dependency vector
d	A tuple $\langle k, v, sr, ut, dv \rangle$
k	A key
v	A value
sr	Source replica id
ut	Update time
dv	Dependency vector

Table 3.1 – Symbols used in POCC's protocol description and corresponding meaning.

Client. A client c maintains during its session a dependency vector DV_c consisting of M entries, where M is the number of DCs. $DV_c[i]$ is the update time of the item d with the highest timestamp such that $i)$ d has been originated in the $i-$th DC and $ii)$ c has established a potential dependency towards d. This vector is stored together with any item c writes.

Server. A server p_n^m is identified by the partition id (n) and the DC id (m). In our description, m is the local DC and n is the local partition in the local DC of the server. p_n^m maintains a version vector VV_n^m [85, 2] consisting of m physical timestamps corresponding to updates seen by p_n^m. $VV_n^m[m]$ is the highest update timestamp of any update originated from p_n^m. Similarly, p_n^m has received all updates with timestamp up to $VV_n^m[i](i \neq m)$ from p_n^i.

Each server has access to monotonically increasing clock, $Clock_n^m$. Clocks are loosely synchronized by a time synchronization protocol such as NTP [82].

3.5.4 Operations

We now describe how servers running POCC support GET, PUT and RO-TX operations issued by clients, replicate updates and exchange heartbeats to stay synchronized. Algorithms 4 and 2 show the pseudo-code of the protocol running at the client and server side, respectively. Algorithm 3 shows the pseudo-code for the creation and replication of updates and heartbeats at the server.

Algorithm 1 POCC: operations at client c

```
 1: function GET(key $k$)
 2:     send ⟨GETReq $k, DV_c$⟩ to $p_n^m$
 3:     receive ⟨GETReply $v, ut, DV, sr$⟩ from $p_n^m$
 4:     $DV_c \leftarrow max\{DV_c, DV\}$
 5:     $DV_c[sr] \leftarrow max\{DV_c[sr], ut\}$          ▷ Update client's dependencies
 6:     return v
 7: end function

 8: function PUT(key $k$, value $v$)
 9:     send ⟨PUTReq $k, v, DV_c$⟩ to $p_n^m$
10:     receive ⟨PUTReply $ut$⟩ from $p_n^m$
11:     $DV_c[m] \leftarrow max\{DV_c[m], ut\}$          ▷ Update client's dependency at local replica
12: end function

13: function RO-TX(key-set $\chi$)
14:     send ⟨RO-TX-Req $\chi, DV_c$⟩ to $p_m^n$
15:     receive ⟨RO-TX-Resp $D$⟩
16:     for ($d \in D$) do
17:         read $d$ as if it was the result of a GET
18:     end for
19: end function
```

In the discussion we indicate a target client as c. We refer to the server which handles c's request as p_n^m. p_n^m can be the partition with which c has established a session, or the partition to which the request has been forwarded (as described in Section 3.3). As said in Section 3.3, concurrent updates to the same key are handled by means of the last-writer-wins rule. The "last" version of a data item is the one with the highest update timestamp. Ties are broken by looking at the source replica id (lowest wins).

GET operation. Client c sends a request ⟨GET k, DV_c⟩, where k is the key of the item to be read. p_n^m checks whether its version vector is entry-wise greater than or equal to DV_c (Algorithm 2 Line 2). This check is not done for the m-th entry because dependencies towards local items are trivially always satisfied. If the check is positive, it means that p_n^m has received all the items of the n-th partition on which c potentially depends. In this case, p_n^m returns the version in k's item chain with the highest update timestamp (Algorithm 2 Line 3). If at least one entry of VV_n^m is smaller than DV_c, it means that c potentially depends on an item d that p_n^m has not replicated yet. Since d can represent an instance associated with key k, p_n^m has to wait for its receipt, or else it might return a version of k that is not causally consistent with c's history. Therefore, p_n^m blocks until $DV_c[i] \leq VV_n^m[i], i \neq m$.

Once p_n^m has determined the correct version d to return, it replies to the client with d's value, update timestamp ut, dependency vector DV and source replica sr. The client then tracks the newly established dependencies, by updating DV_c with d's dependencies DV and $DV_c[sr]$ with d's update time ut.

Algorithm 2 POCC: operations at server p_n^m

```
1:  upon receive ⟨GETReq $k, DV_c$⟩ from $c$ do
        ▷ Ensure $p_n^m$'s state is consistent with c's history
2:      wait until $VV_n^m[i] \geq DV_c[i],\ i = 0 \dots M-1, i \neq m$
3:      $d = argmax_{d.ut}\{d : d.k == k\}$                    ▷ Version of k with highest timestamp
4:      send ⟨GETReply $d.v, d.ut, d.DV, d.sr$⟩ to $c$

5:  upon receive ⟨PUTReq $k, v, DV_c$⟩ from $c$ do
6:      wait until $max\{DV_c\} < Clock_n^m$                   ▷ Deal with clock skew
7:      $d \leftarrow \langle k, v, m, Clock_n^m, DV_c \rangle$
8:      update($d$)                                            ▷ Create new item and store it in the chain
9:      for each server $p_n^j$, $j \in \{0 \dots M-1\}, j \neq m$ do
10:         send ⟨REPLICATE d⟩ to $p_n^j$                      ▷ Replicate d
11:     end for
12:     $VV_n^m[m] \leftarrow$ d.ut                            ▷ Update version vector
13:     send ⟨PUTReply $d.ut$⟩ to $c$

14: upon receive ⟨RO-TX-Req $\chi, DV_c$⟩ from $c$ do
15:     $D = \emptyset$                                        ▷ D represents the set of key-value pairs for the requested keys
16:     $\chi_i = \{k \in \chi : partition(k) == i\}$           ▷ Set of requested keys per partition
17:     $TV = max\{VV_n^m, DV_c\}$                             ▷ Timestamp vector of the transaction snapshot
18:     for $(i : \chi_i \neq \emptyset)$ do
19:         send ⟨SliceREQ $\chi_i, TV$⟩ to $p_i^m$
20:         receive ⟨SliceRESP $D_i$⟩ from $p_i^m$
21:         $D \leftarrow D \cup D_i$
22:     end for
23:     send ⟨RO-TX-Resp $D$⟩ to $c$

24: upon receive ⟨SliceREQ $\chi, TV$⟩ from $p_i^m$ do
25:     wait until $VV_n^m \geq TV$                            ▷ Ensure $p_n^m$ installed all updates in the snapshot
26:     $D = \emptyset$                                        ▷ D represents the set of key-value pairs for the requested keys
27:     for $(k \in \chi)$ do
28:         $D_k = \{d : d.k == k \wedge d.DV \leq TV\}$          ▷ Compute visible versions set
29:         $D \leftarrow D \cup \{argmax_{d.ut}\{d \in D_k\}\}$  ▷ Freshest visible version of k
30:     end for
31:     send ⟨SliceRESP $D$⟩ to $p_i^m$
```

PUT operation. c sends a request ⟨PUT k, v, DV_c⟩, where k is the key of the item to be written, v is the desired value to associate with k and DV_c is the client's dependency vector.

Upon receiving the PUT request, p_n^m, first waits until the local physical clock is higher than the maximum timestamp in DV_c (Algorithm 2 Line 6). In this way, p_n^m ensures that the soon-to-be-created item d has a higher update timestamp than any of its potential dependencies.

Next, p_n^m creates a new version d of k, and inserts d in the version chain corresponding to k. Afterwards, p_n^m replicates the newly created item version d to the remote replicas. Then, p_n^m updates the local entry of its version vector with d's update time. Finally, p_n^m sends a reply with the update time of the newly created item version to c. Upon receiving d as a reply, c sets the $m-$th entry of DV_c to $d.ut$ to track the newly established dependency.

RO-TX. A client c sends a request ⟨RO-TX χ, DV_c⟩ to a server p_n^m, where χ is the set of keys to be read. Upon receiving the request, p_n^m first determines the timestamp vector of the snapshot visible to the transaction TV (Algorithm 2 Line 17). As said in Section 3.4.1, when serving a transaction p_n^m cannot simply return the freshest version of a key. Thus, TV has to be as recent as possible, to avoid returning excessively stale versions of the requested data items, and must include all potential dependencies established by the client. Hence, TV is computed as the entry-wise maximum between VV_n^m and DV_c.

Reads for keys that cannot be served locally are sent in parallel to the corresponding partitions in the form of a request ⟨SliceReq χ, TV⟩ (Algorithm 2 Line 19). Upon receiving a SliceReq, a server first waits until its version vector is entry-wise greater than or equal to the transaction snapshot vector TV (Algorithm 2 Line 25). Then, for each key k in the set of keys to be read χ, the server chooses and returns the freshest visible version, which is the version whose dependency vector DV is less or equal than TV (Algorithm 2 Line 28). When each server has returned to the coordinator, for each key, the version within the snapshot with the highest timestamp, p_n^m returns all the requested keys and values to c.

Updates replication. p_n^m replicates local updates asynchronously by sending them in update timestamp order to its replicas at the other DCs. When a server receives a replicated update d from replica p_n^m, it inserts d in the corresponding version chain and advances its m-th entry of its local version vector to the update time of d (Algorithm 3 Lines 6 – 7).

Heartbeats. If p_n^m does not receive update requests from clients, it does not send replication messages to its replicas either. Therefore, other replicas cannot increase the m-th entry in their version vector. Keeping the version vector up-to-date, as we shall see, is necessary to implement a garbage collection protocol to remove stale versions of data items. Therefore, a partition that does not receive requests for local updates for a time longer than Δ broadcasts its latest clock time to its replicas (Algorithm 3 Line 13). It does so by piggybacking the clock time in the heartbeat messages used by failure detectors. Heartbeat messages and update replication messages are sent in order of increasing update timestamps and clock values.

Garbage collection. Periodically, partitions within a DC exchange the vector corresponding to the aggregate maximum of the TV vectors corresponding to active transactions. If on p_n^m there is no running transaction, p_n^m sends VV_n^m. The servers, then, compute the garbage collection vector GV as the aggregate minimum of the received vectors. Then, each server p_m^m scans the version chain of keys it replicates in descending timestamp order. For each key k, p_n^m retains up to and including the first version d_k such that $d_k.DV \leq GV$. Namely, p_n^m keeps the oldest version of k that can be accessed within the scope of a transaction, and removes oldest versions.

Algorithm 3 POCC: auxiliary functions at server p_n^m

```
1:  function UPDATE($d'$)
2:      create new item $d : \langle d.k, d.v, d.sr, d.ut, d.dv \rangle \leftarrow \langle d'.k, d'.v, d'.sr, d'.ut, d'.dv \rangle$
3:      insert item $d$ in the version chain of key $k$
4:  end function

5:  upon receive ⟨REPLICATE d⟩ from $p_n^j$ do
6:      update($d$)                         ▷ Create new item and store it in the chain
7:      $VV_n^m[j] \leftarrow d.ut$                   ▷ Update version vector

8:  upon every $\Delta$ time do
9:      $ct \leftarrow Clock_n^m$
10:     if ($ct \geq VV_n^m[m] + \Delta$) then
11:         $VV_n^m[m] \leftarrow ct$          ▷ If there are no updates, advance the version vector
12:         for each server $p_n^j$, $j \in \{0 \dots M-1\}, k \neq m$ do
13:             send ⟨HEARTBEAT $ct$⟩ to $p_n^j$         ▷ Send heartbeat to replicas
14:         end for
15:     end if

16: upon receive ⟨HEARTBEAT ct⟩ from $p_n^j$ do
17:     $VV_n^m[j] \leftarrow ct$
```

3.6 Correctness

We now provide an informal proof sketch that POCC provides causal consistency. Namely, we show that an item returned to a GET operation never violates causal consistency, and that the set returned by a RO-TX operation is a causal snapshot according to the definition provided in Section 3.3.

We start by proving two Propositions that we will use to build the correctness proof.

Proposition 1. *Let X, Y be two data items such that $X \rightsquigarrow Y$. Then, $Y.DV[X.sr] \geq X.ut$.*

Proof. This invariant stems from the fact that a client c tracks the dependencies established by any read/written item and stores such dependencies with the data items it writes. Specifically, upon reading an item d, c transitively tracks d's dependencies by computing the entry-wise maximum between DV_c and the $d.DV$, and storing the result as new DV_c (Algorithm 4 Line 4). Further, upon reading/writing an item d, noting i the source replica of d, c tracks the direct dependency on d by setting $DV_c[i] = max\{DV_c[i], d.ut\}$ (Algorithm 4 Line 5 and Line 11, respectively). Finally, when writing a data item d, c stores DV_c with d (Algorithm 2 Line 7). ■

Proposition 2. *Let X, Y be two data items such that $X \rightsquigarrow Y$. Then, $X.ut < Y.ut$.*

Proof. A client c that has established a dependency towards X is such that $DV_c[X.sr] \geq X.ut$. When c tries to write Y on p_n^m, Algorithm 2 Line 6 enforces that $Clock_n^m$ is strictly higher than the maximum value in DV_c. Therefore, because $Clock_n^m$ is used as update timestamp for Y, it follows that $Y.ut > X.ut$. ■

We now show that POCC delivers causal consistency.

Proposition 3. Causally consistent GET operation. *Let client c read Y such that $X \rightsquigarrow Y$. Then, if c later issues a $GET(x)$ operation, c reads X' such that either $X' = X$ or $X' \not\rightsquigarrow X$.*

Proof. The proposition is verified because upon processing a GET operation issued by c for a key x, a server p_n^m waits until it is positive to return a version of x that complies with c's history. We prove this by contradiction, by assuming that there is a $X_0 \rightsquigarrow X \rightsquigarrow Y$ and showing that if p_n^m returns X_0, then it has not respected the protocol of POCC.

Because $X_0 \rightsquigarrow X$, it is, by virtue of Proposition 2, $X_0.ut < X.ut$. In addition, because $X \rightsquigarrow Y$, Proposition 1 enforces that $Y.DV[X.sr] \geq X.ut$. If p_n^m returns X_0 it means that it has not received X yet. If p_n^m had received X, it would have returned it because X has a higher timestamp than X_0 and POCC returns the version of a key with the highest timestamp. To return X, however, p_n^m must pass the waiting condition in Algorithm 2 Line 2. In particular, because $Y.DV[x.sr] = DV_c[X.sr] \geq X.ut \geq X_0.ut$, it must be that $VV_n^m[X.sr] \geq X.ut$. This, however, is impossible because p_n^m has not received X and updates and heartbeats are sent by servers in timestamp order.

■

Proposition 4. Causally consistent RO-TX operation. *The result of a RO-TX operation issued by a client c represents a causally consistent snapshot and is causally consistent with the history of operations issued by c.*

Proof. We first show that the data returned to c are causally consistent with c's history. This stems from the fact that the snapshot vector corresponding to a read-only transaction is computed as the entry-wise maximum between the version vector of the transaction coordinator and the dependency vector of the client (Algorithm 2 Line 17). This means that the snapshot includes every item read or written by c, and any transitive dependency induced by such items. Before serving a transactional read, a server p_n^m waits until its vector clock is at least entry-wise equal to the snapshot vector (Algorithm 2 Line 25). Therefore, for the same reasons presented for the GET operation case, POCC enforces that p_n^m does not return any item that is not causally consistent with c's history.

We now show that a RO-TX operation returns a causal snapshot. Namely, assume that $X \rightsquigarrow X' \rightsquigarrow Y$ and that c reads x and y in the same transaction. Then, if the returned snapshot contains Y, it also contains X'.

By contradiction, assume that the returned snapshot includes Y and X. Because $X \rightsquigarrow X' \rightsquigarrow Y$, it stems from Proposition 1 that $Y.DV[X'.sr] > X'.ut$ and $Y.DV[X.sr] \geq X.ut$. In addition, since Y is in the returned snapshot, it follows that $TV \geq Y.DV$ (Algorithm 2 Line 28). Therefore, it is $TV[X'.sr] \geq Y.DV[X'.sr] > X'.ut$ and $TV[X.sr] \geq Y.DV[X.sr] > X.ut$. Then, by virtue of the blocking condition in Algorithm 2 Line 25, the server replicating x, noted p_x has received both X and X' by the time it serves the RO-TX request from the client. Then, p_x cannot return X instead of X' because $i)$ $X \rightsquigarrow X'$ and Proposition 2 enforces that $X.ut < X'.ut$ and $ii)$ p_x returns the item in the version chain with the highest timestamp (provided that its dependency vector is entry-wise smaller than or equal to TV). ■

3.7 Highly available OCC

3.7.1 Design

In order to circumvent OCC's potential blocking behavior, we propose to augment OCC with a recovery procedure aimed to regain availability during network partitions. This procedure follows the structure outlined by Brewer to breach the CAP boundaries and it is based on three phases [23]. In the first phase, the network partition is detected. In the second one, the system enters a partition-tolerant mode, in which it may give up some functionality or properties. The third phase is the recovery one, during which the system switches back to run its (non network partition tolerant) protocols.

We explain our recovery mechanism starting from the aforementioned blocking condition example. If p blocks while serving a request from c, as soon as p realizes there is a network partition occurring, it closes the session with c. A network partition can be identified by p if it blocks for more than a configurable amount of time. At this point, c re-initializes its session. This new session is managed according to a pessimistic protocol and, therefore, is ensured not to block even during the ongoing network partition. The cost for this session re-initialization is that the client might not be able to see the same version of some data items read or written in the optimistic session. If and when the network partition heals, the session can be promoted again to be managed according to the optimistic approach.

Equipped with this recovery mechanism, OCC can be implemented in a *highly available* fashion, representing a novel and unexplored trade-off between performance and availability/consistency.

We believe that, for many applications, this is a reasonable, if not favorable, trade-off. As already stated, in fact, careful engineering and redundant links make modern geo-distributed data-centers reliable enough to regard network partitions as infrequent events [93, 23, 67, 11]. This is also confirmed by the successful implementation and deployment of strongly consistent geo-replicated storage systems, which, by the CAP theorem, do not tolerate network partitions by design [34].

Further, if a network partition does occur, the cost of re-initializing a client session, both in terms of time and data visibility, is affordable for many applications, e.g., social networks.

In the case of a full DC failure or, equivalently, of a network partition that does not heal, OCC can be subject to what we define the phenomenon of the *lost update*. Assume, for example, there exist X, Y such that $X \rightsquigarrow Y$ and both have been created in DC'. It can happen that Y gets replicated in DC'' and X is prevented to do so because of the failure of DC'. Then, Y can be read in DC'' and new items depending on Y can be written, establishing a dependency towards an item that will never be received by DC''. As a consequence, an OCC would fall-back to run a pessimistic protocol without the possibility to switch back to operating optimistically.

A possible mechanism to recover from this situation is to discard items that depend on a lost update and that have been created after the failure of DC'. This recovery mechanism causes the loss of some updates. As previously discussed, however, data loss is a consequence of a DC failure that is already encompassed also by pessimistic approaches to causal consistency. In pessimistic approaches, however, only updates originated in the failed DC can be lost. In OCC, instead, also updates from healthy DCs might get discarded.

We finally note that the implementation of a mechanism to recovery from full DC failure is a requirement also for other state-of-the-art protocols. In GentleRain [42], for example, this is because a full DC failure prevents servers in other DCs to install remote updates. In Cure [4], instead, a recovery mechanism is needed because healthy DCs might have not received the same set of updates from the failed DC, leading their states to diverge.

3.7.2 Implementation

To achieve high availability in presence of network partitions POCC must be able to fall-back to a pessimistic protocol. Because POCC is based on physical vector clocks, POCC can be easily augmented to fall-back to run the protocol implemented in Cure [4], a recent causally consistent system also based on physical vector clocks [I]. We call Highly Available POCC (HA-POCC) the resulting augmented version of POCC.

In Cure, partitions within a DC periodically run a stabilization protocol. It consists of exchanging their version vectors and computing the aggregate minimum called Globally Stable Snapshot (GSS). $GSS[i] = t$ means that all partitions within a DC have installed all updates originated in the i-th DC. In Cure a remote item d is visible only if $d.DV \leq GSS$, meaning that all d's dependencies have been received. Local items are immediately visible, because they depend only on stable items.

[I]Cure's programming model is transaction-centric. It is, however, straightforward to adapt Cure to support GET and PUT operations.

HA-POCC runs this stabilization protocol much less frequently than Cure, because HA-POCC only needs to use the GSS in the uncommon case of suffering from a network partition. HA-POCC, thus, reduces the overhead associated with the stabilization protocol during normal operational behavior. Because the GSS is infrequently updated, however, this higher resource efficiency comes at the cost of an increased perceived data staleness during network partitions. We argue this is a favorable trade-off, since network partitions are rare events.

During network partitions, HA-POCC runs Cure's protocol except for one detail. In Cure, local data items are always visible, because they depend on stable items. In HA-POCC, however, a local item can depend on remote items that are not stable yet, or that have not even been replicated in the current DC yet. Therefore, HA-POCC needs to distinguish between clients that are running the optimistic protocol and clients that are running the pessimistic protocol. In this way, servers can recognize a local item d created by an optimistic client and make d visible to pessimistic clients only if d is stable according to the pessimistic protocol.

Finally, HA-POCC adopts the garbage collection protocol of Cure, because HA-POCC needs to keep the oldest version of an item that could be accessed by the pessimistic protocol.

3.8 Evaluation

Because OCC departs from the traditional way causal consistency is achieved, in this chapter we primarily want to demonstrate improved data freshness while delivering performance that is competitive with pessimistic designs. Specifically, the focus of this evaluation is on performance during normal operational behavior, i.e., in the absence of network partitions, which is the behavior expected during the vast majority of its running time. We leave the investigation of the behavior of the system during a network partition and its recovery for future work.

To evaluate the effectiveness of OCC, we compare the performance achieved by POCC with the one of Cure*, a re-implementation of Cure [4], a state-of-the-art causally consistent system based on vector clocks. Because Cure only supports transactional operations, we have augmented Cure* with support for simple GET and PUT operations. The comparison between POCC and Cure* is fair because the amount of meta-data exchanged by clients and servers to implement the operations is the same. The two mainly differ in that POCC does not run any stabilization protocol and does not need to search for a stable version of a key when serving a GET.

3.8.1 Experimental environment

Platform. We consider an Amazon AWS deployment consisting of 3 DCs and 32 partitions per DC. The data centers are located in Oregon, North Virginia and Ireland. We use $c4.large$

instances, corresponding to 2 virtual CPU and 3.75 GB of RAM.

Each data partition is composed of one million key-value pairs. We consider small items, with both keys and values of 8 bytes, because they are predominant in many production workloads [80, 9, 53, 97]. Keys are chosen within each partition according to a zipf distribution with parameter 0.99.

Clients are collocated with servers, establish their sessions with the collocated server, and perform operations in closed loop. We introduce a think time between client operations to simulate a more realistic workload, in which a client does not simply injects requests but also processes the result of a performed operation. Naturally, having a think time between operations also lowers the chances that a request blocks when using OCC, because it gives time to servers to receive potentially missing client dependencies. We set the think time to 25 milliseconds. This value is low enough to avoid masking the blocking dynamics in POCC and high enough to allow us to fully load the compared systems without requiring an excessive number of client threads. We note that 25 milliseconds is orders of magnitude lower than the think time used in standard session-based benchmarks like TPC-W [47], TPC-C [68], SpecWeb2005 [50] and Rubis [44], and therefore allows us to benchmark the effectiveness of OCC in a particularly challenging setting.

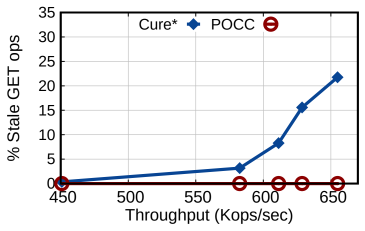
Implementation. We implement POCC and Cure* in the same C++ code-base[II]. We run NTP [82] to keep physical clocks synchronized. As in previous work [4], clocks are synchronized before each experiment. We use the NTP server 0.amazon.pool.ntp.org.

Both POCC and Cure* use the last-writer-wins rule to arbitrate conflicting updates to the same key. Heartbeats are sent by a server if it does not serve any PUT request for 1 millisecond. The stabilization protocol in Cure* is run every 5 milliseconds.

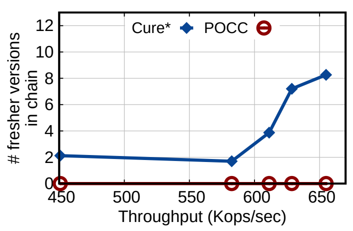
All data is kept in memory and no fault tolerance mechanism is implemented. This allows us to isolate the effects of OCC from the dynamics of the specific fault tolerance mechanism implemented, e.g., primary copy [83], Paxos [65] or chain replication [89].

We start by experimenting with a workload in which clients issue GET and PUT requests. In such a workload, a GET:PUT ratio of $N:M$ means that each client issues N consecutive GET operations followed by one PUT operation. Each GET operation targets a different partition. The PUT operation is issued against a key in a partition chosen uniformly at random.

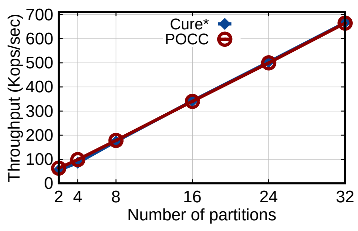
[II]https://github.com/epfl-labos/POCC



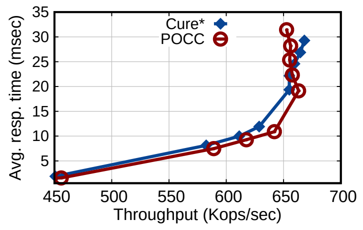
(a) Data staleness.



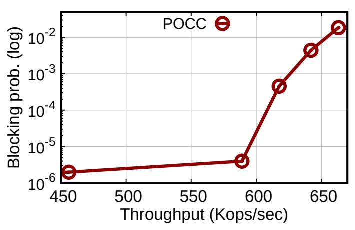
(b) Number of fresher versions in the item chain.



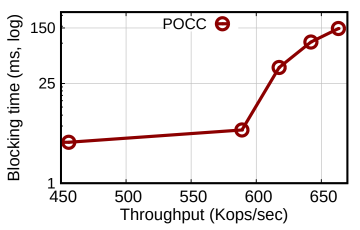
(c) Throughput while varying the number of partitions.



(d) Average response time on 32 partitions with a 32:1 GET:PUT workload.



(e) Blocking probability in POCC.



(f) Blocking time in POCC.

Figure 3.3 – Evaluation of data freshness, performance and blocking incidence of POCC for the single key operation workloads. The results for data staleness (a) and number of fresher versions in the item chain (b) are for the 32:1 GET:PUT workload on 32 partitions. In POCC, single key operations always return the freshest available version, so the percentage of stale reads and the number of fresher versions in the item chain for POCC is 0.

3.8.2 Get-Put workload

We start by experimenting with a workload in which clients issue GET and PUT requests. In such workload, a GET:PUT ratio of $N:1$ means that each client issues N consecutive GETs followed by one PUTs. Each GET operation targets a different partition. The PUT operation is issued against a key in a partition chosen uniformly at random.

Data staleness. As a representative of the protocols that enforce causal consistency in a pessimistic manner, Cure* exposes certain degree of stale data to its clients. Our first goal is to quantify the degree of stale reads in Cure*. For that purpose, we run a workload with 32:1 GET:PUT ratio on 32 partitions in 3 DCs.

To describe the results corresponding to data staleness in Cure* we first provide a definition of *stale* data item. We define a data item “stale" if there is at least one newer version in the item chain when the item is requested, i.e. if the version returned to the client is not the one with the highest timestamp. In order to quantify the data staleness incurred by Cure*, we utilize two metrics: (*i*) the percentage of stale data items returned to the client and (*ii*) the number of fresher item versions available in the version chain of an stale returned data item at the time of the request. We highlight that the percentage of stale data items returned to clients by POCC is 0, as is the number of fresher item versions available. When running workloads composed of single key GET and PUT operations, POCC always returns the freshest available item version, offering maximum data freshness to its clients.

Figure 3.3a shows the percentage of stale data items returned in Cure*. The plot shows that the probability to return stale data increases with the load. This is not only because of a higher remote update rates. It is also because higher contention on physical resources slows down the execution of the stabilization protocol needed to identify stable versions. In fact at high load, but before the saturation point, the percentage of returned stale items reaches 15%. When Cure* is highly loaded, the percentage of returned stale data items grows as high as 22%. In our experiments, the requested keys are chosen according to zipf distribution, which is a common case in practise. This indicates that up to 15%-22% of the requested hot keys will have stale values. A similar dynamic can be observed by Figure 3.3b, which shows the number of fresher versions available in the version chain of an stale returned data item.

Suppose that the ticketing system, discussed in Section 3.2, was deployed on Cure* in our experimental setting. Because we run the stabilization protocol every 5 ms, similarly to the example, it would take up to 7 ms to calculate the new stable time and promote unstable updates to stable and visible. The average one-way trip latency between the closest DCs is ~37 ms (North Virginia to Ireland) and ~65 ms between the furthest (Oregon to Ireland) [1]. Under normal conditions, the lower bound of the visibility latency would be ~44 ms between the closest DCs (37 ms for replication and 7 ms for stabilization) and ~72 ms (65 ms for replication and 7 ms for stabilization) between the furthest DCs. This signifies that the clients would experience at least 15%-10% increase in visibility latency.

We note that in bigger deployments the execution of the stabilization protocol would naturally progress at a lower pace, with a commensurate increase in the perceived staleness. In addition, these results correspond to running the stabilization protocol every 5 milliseconds. Higher values would allow the system to reach a higher throughput (as shown in previous work [42]), but would come at the cost of an increased data staleness. By contrast, POCC is immune to this trade-off.

Scalability and performance. Our next experiments were designed to asses whether the data freshness benefits of POCC came at the cost of scalability and performance penalties.

First we investigate the scalability of both systems. To this end, we deploy POCC and Cure* on a system composed of different partitions (from 2 to 32) and we measure the maximum achievable throughput. The GET:PUT ratio is $p:1$, where p denotes the number of partitions. Figure 3.3c reports the result of this experiment. The plot shows that the two systems achieve basically the same throughput, showing that the optimistic approach can be implemented with no throughput loss with respect to the corresponding pessimistic implementation.

To access the effect on the response time, we fix the number of partitions to 32 and we investigate the average operation response time achieved while increasing the workload intensity. The result of the experiment is depicted by Figure 3.3d. POCC achieves a slightly lower response time than Cure* before reaching the saturation point at about 650 Kops/sec. This is because, as we shall see more in detail later, POCC rarely blocks upon serving an operation and it is more resource efficient than Cure*. POCC, in fact, never traverses an item's version chain, nor needs to run any stabilization protocol (or can run it much more infrequently). Under very high load, POCC performs slightly worse than Cure*, because the better resource efficiency is outweighed by the incidence and extent of operations blocking.

Blocking dynamics. To better understand the performance results presented so far, we investigate the blocking behavior of POCC. To this end, Figure 3.3e reports the blocking probability of POCC and Figure 3.3f reports the blocking time. The results refer to the aforementioned 32:1 GET:PUT workload run over 32 partitions. In both figures, the y-axis where the blocking probability and the blocking time are represented, are expressed in logarithmic scale.

Figure 3.3e, specifically, reports the probability that, in POCC, an operation blocks and the average blocking time for a blocked operation. The plot shows that the blocking probability is negligible under moderate to high load and that becomes noticeable only as the system approaches the maximum achievable throughput (650 Kops/sec). A similar dynamic is exhibited by the blocking time, shown on Figure 3.3f, which is in the order of a few microseconds as long as the system is not heavily loaded. The plot shows that, as long as the workload intensity is not too close to the maximum sustainable (i.e., up to 600 Kops/sec), the possible blocking behavior of POCC has a negligible effect. In particular, up to 600 Kops/sec the blocking probability is lower than 0.001, meaning that the 99.999-th percentile of the operation response times is not affected by the blocking behavior of POCC.

Resource efficiency. Figure 3.4 shows the breakdown of the amount of data exchanged in Cure* for the replication and the stabilization protocols, when varying the GET:PUT ratio from 32:1 to 1:1 on a 3 DCs and 32 partitions deployment. POCC does not run a stabilization protocol, so it only exchanges data for replication. Cure*, on the contrary, has to run a stabilization protocol in order to track the delivery of dependencies. The results are normalized with respect to the amounts of data exchanged in POCC at the same throughput. POCC exchanges up to 37% less metadata than Cure* because it does not need to run any stabilization protocol.

Summary of the results. Figure 3.3a and Figure 3.3b demonstrate the negative effect that the

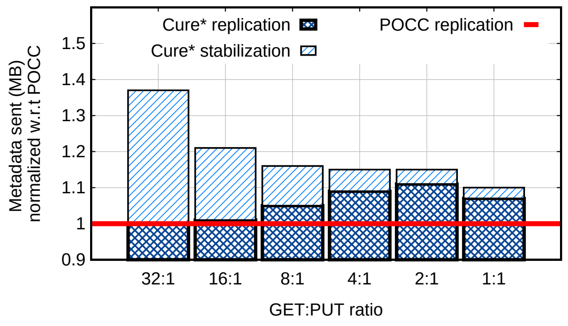


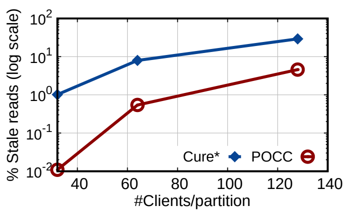
Figure 3.4 – POCC incurs lower overhead than Cure* because it does not need to run a stabilization protocol for tracking the delivery of updates.

pessimistic way of enforcing causal consistency in Cure* has on data freshness. POCC, on the other hand, always returns the freshest values for a requested item for the single key operations. Figure 3.3c reveals that POCC is competitive with Cure* and Figure 3.3d shows that POCC even offers lower latencies until the saturation point is reached. Both figures show that achieving maximal data freshness comes with no significant cost or penalty on performance. Even under high load, when POCC is prone to block more frequently and for a longer amount of time, the performance of POCC remains competitive with the performance of Cure*. POCC achieves this result because blocking is a rare event, thus not hurting the vast majority of operations, and because a blocked operation yields the CPU, thus increasing the amount of computational resources available to quickly serve non-blocked operations. Figure 3.3e attests that the blocking probability is negligible, and Figure 3.3e shows that the blocking time is in order of a few milliseconds. In Cure*, instead, data staleness yields to a higher CPU demand to traverse the version chain upon serving a GET operation and all the requests concurrently compete for the CPU, mutually hurting each other's performance. Figure 3.4 reveals that POCC is more resource efficient than Cure* because it simply does not need to run any stabilization protocol in order to track the delivery of updates.

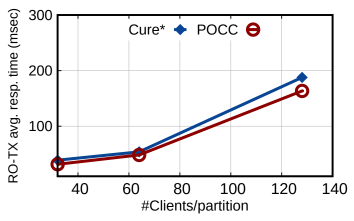
3.8.3 Transactional workload

We now assess the data freshness benefits and the performance of POCC when facing a workload composed of PUT operations and transactional reads. To this end, we run on 32 partitions a workload in which each client first issues a RO-TX to read p items corresponding to p distinct partitions, and then performs a random PUT operation. p is varied from 1 to 32.

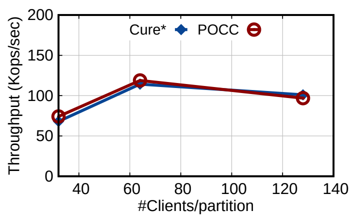
Data staleness. As already explained in Section 3.5.1, with transactions, OCC does not in general return the freshest version of a key. Therefore, both POCC and Cure* are prone to return stale



(a) Data staleness in POCC and Cure* with different number of clients per partition.



(b) Throughput with different number of clients per partition.



(c) Average response time with different number of clients per partition.

Figure 3.5 – Evaluation of POCC and Cure* with a workload composed of PUT and RO-TX operations. The systems are deployed on 32 partitions per DC.

data. In this paragraph we compare the staleness of data returned to client by POCC and Cure*. In particular, Figure 3.5a reports the percentage of stale data items returned by the two systems. To be noted that on Figure 3.5a, the percentage of stale reads, represented on the y-axis, is in logarithmic scale.

The plot shows that the staleness exhibited by POCC is two orders of magnitude lower than Cure*'s. POCC returns much fresher data because it defines the boundaries of items visible within a transaction on the basis of the issuing client's history and the set of items *received* by the transaction coordinator server. Cure*, instead, defines such boundaries in terms of items that are *stable* on the transaction coordinator server, thus being much more prone to read stale values.

Throughput and response time. We now investigate the throughput and response time dynamics while increasing the number of active clients. Figure 3.5b and Figure 3.5c depict the throughput and average RO-TX response time, respectively, achieved by POCC and Cure* when transactions involve half of the partitions in the system. The plots reveal that the two systems exhibit very similar performance dynamics, with POCC achieving up to 12% lower average response time. This demonstrates that POCC achieves two orders of magnitude gain in data freshness, shown on Figure 3.5a, without additional costs on performance. On the contrary, POCC achieves lower response time and slightly higher throughout.

Transactional scalability. Figure 3.6 shows the throughput achieved by POCC and Cure* while increasing the number of partitions involved in a read-only transaction. The plot shows that the performance of POCC and Cure* are comparable –with POCC being slightly better in general– when the number of involved partitions is small. The gain of POCC over Cure* becomes more noticeable (up to 15%) when transactions involve the majority of the partitions in the system. This is because, as the number of involved partition increases, executing a transaction becomes more resource demanding. POCC, then, achieves a higher throughput because it is more resource

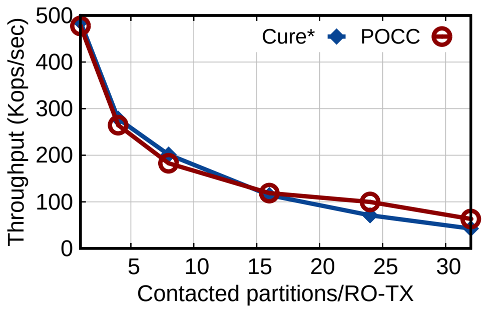


Figure 3.6 – Throughput of POCC and Cure* while varying the number of contacted partitions per transaction. The workload is composed of PUT and RO-TX operations, the systems are deployed on 32 partitions per DC. RO-TX touch 16 partitions.

efficient than Cure*.

Blocking behavior. Figure 3.7 shows two plots: (*i*) the probability that a PUT or a transactional read stalls on a server and (*ii*) the average amount of time a stalled request is delayed. The reported results correspond to the same workload as in the previous paragraph. The plots on Figure 3.7 shows highly non-linear dynamics. The operation blocking probability peaks in correspondence of 64 active threads per partition, which corresponds to the peak throughput. In correspondence of lower throughput, the blocking probability decreases because of the lower update rate in the system. The blocking time of stalled operations instead, decreases from 32 to 64 clients and then quickly grows. We argue that the rationale behind this dynamic is as follows. At low throughput, there is also a low rate of updates. Hence, provided that an operation blocks on a server p, p is expected to wait a relatively long time before receiving the update or the heartbeat that unblocks the pending operation. In correspondence of 64 clients per partition, the throughput is high enough to reduce such waiting time. When the number of clients grows too high, performance and blocking dynamics are mainly determined by the high contention on physical resources. This causes the delayed processing of updates and heartbeats messages, yielding to increased blocking times (and blocking probability).

Summary of the results. We have shown that OCC delivers much fresher data to clients (up to two orders of magnitude) compared to the pessimistic design of Cure*. Besides the improvement in data freshness, OCC is able to achieve higher performance than a pessimistic design in the case of workloads composed of PUT and RO-TX operations. The cause for this performance increase is mainly the higher resource efficiency enabled by OCC. Transactions are, in fact, more demanding both in terms of computational resources and communication. Therefore, with a workload composed of transactions, POCC higher resource efficiency pays off more than the case of a workload consisting of simple GET operations.

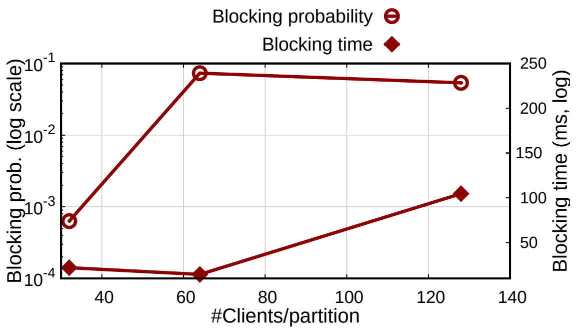


Figure 3.7 – Blocking behavior in POCC with different number of clients per partition. Both y-axes are represented in logarithmic scale.

3.9 Conclusion

We have presented OCC, an optimistic approach to achieve causal consistency in geo-replicated key-value stores. OCC regards network partitions and data center failures as rare events in modern deployments. Therefore, OCC trades some of the availability properties achievable by causal consistency for a higher freshness in the data returned to clients and a higher resource efficiency.

We have implemented OCC in a system named POCC. We have shown that POCC can achieve performance that is comparable or better than its "pessimistic" counterpart in a wide range of workloads, while being able to reduce (or eliminate) the staleness of the data returned to clients.

4 Nonblocking Reads in a Partitioned TCC Data Store

4.1 Introduction

TCC [4] is an attractive consistency level for building geo-replicated data-stores. TCC enforces causal consistency (CC) [2], which is the strongest consistency model compatible with availability [10, 73]. Compared to strong consistency [52], CC does not suffer from high synchronization latencies, limited scalability and unavailability in the presence of network partitions between DCs [20, 63, 69]. Compared to eventual consistency [36], CC avoids a number of anomalies that plague programming with weaker models. In addition, TCC extends CC with interactive read-write transactions, that allow applications to read from a causal snapshot and to perform atomic multi-item writes.

Enforcing CC while offering always-available interactive multi-partition transactions is a challenging problem [72]. The main culprit is that in a distributed environment, unavoidably, partitions do not progress at the same pace. Current TCC designs either avoid this issue altogether, by not supporting sharding [114], or block reads to ensure that the proper snapshot is installed [4]. The former approach sacrifices scalability, while the latter incurs additional latencies.

Wren. This chapter presents Wren, the first TCC system that implements nonblocking reads, thereby achieving low latency, and allows an application to scale out by sharding. Wren implements CANToR (Client-Assisted Nonblocking Transactional Reads), a novel transaction protocol in which the snapshot of the data store visible to a transaction is defined as the union of two components: i) a fresh causal snapshot that has been installed by *every* partition within the DC; and ii) a per-client cache, which stores the updates performed by the client that are not yet reflected in said snapshot. This choice of snapshot departs from earlier approaches where a snapshot is chosen by simply looking at the local clock value of the partition acting as transaction coordinator.

Wren also introduces Binary Dependency Time (BDT), a new dependency tracking protocol, and Binary Stable Time (BiST), a new stabilization protocol. Regardless of the number of partitions and DCs, these two protocols assign only two scalar timestamps to updates and snapshots, corresponding to dependencies on local and remote items. These protocols provide high resource efficiency and scalability, and preserve availability.

Wren exposes to clients a snapshot that is slightly in the past with respect to the one exposed by existing approaches. We argue that this is a small price to pay for the performance improvements that Wren offers.

We compare Wren with Cure [4], the state-of-the-art TCC system, on an AWS deployment with up to 5 DCs with 16 partitions each. Wren achieves up to 1.4x higher throughput and up to 3.6x lower latencies. The choice of an older snapshot increases local update visibility latency by a few milliseconds. The use of only two timestamps to track causality increases remote update visibility latency by less than 15%.

Contributions. In this chapter, we make the following contributions:

1. We present the design and implementation of Wren, the first TCC key-value store that achieves nonblocking reads, efficiently scales horizontally, and tolerates network partitions between DCs.
2. We propose new dependency and stabilization protocols that achieve high resource efficiency and scalability.
3. We experimentally demonstrate the benefits of Wren over state-of-the-art solutions.

The rest of the chapter is organized as follows. Section 4.2 describes TCC and the target system model. Section 4.3 presents the design of Wren. Section 4.4 describes the protocols in Wren. Section 4.5 presents the evaluation of Wren. Section 4.6 concludes the chapter.

4.2 System model

We consider a distributed key-value store whose data-set is split into N partitions. Each key is deterministically assigned to one partition by a hash function. We denote by p_x the partition that contains key x.

The data-set is fully replicated: each partition is replicated at all M DCs. We assume a multi-master system, i.e., each replica can update the keys in its partition. Updates are replicated asynchronously to remote DCs.

The data store is multi-versioned. An update operation creates a new version of a key. Each version stores the value corresponding to the key and some meta-data to track causality. The system periodically garbage-collects old versions of keys.

At the beginning of a session, a client c connects to a DC, referred to as the local DC. All c's operations are performed within said DC to preserve availability [12] [I]. c does not issue another operation until it receives the reply to the current one. Partitions communicate through point-to-point lossless FIFO channels (e.g., a TCP socket).

4.3 The design of Wren

We first illustrate the challenge in providing nonblocking reads, by showing how reads can block in the state-of-the-art Cure system [4]. We then present CANToR (§ 4.3.2), and BDT and BiST (§ 4.3.3). We discuss fault tolerance and availability in Wren (§ 4.3.4).

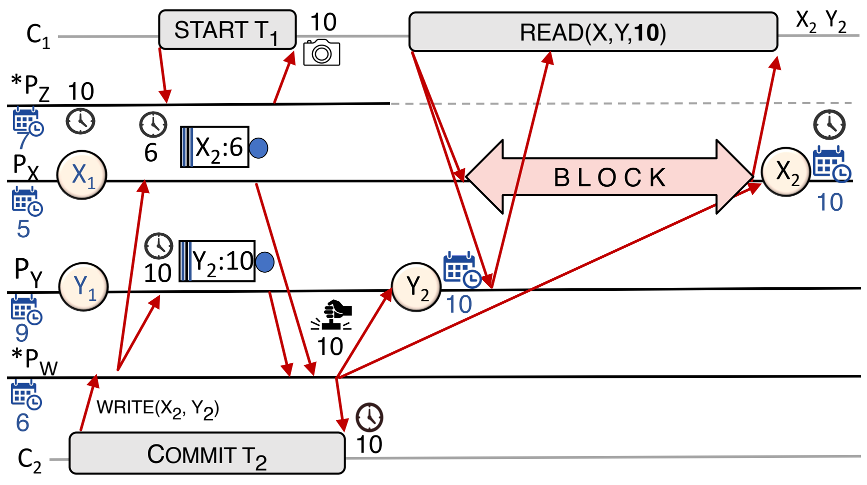
4.3.1 The challenge in providing nonblocking reads

For the sake of simplicity, we assume that a transaction snapshot S is defined by a logical timestamp, denoted st. We say that a server has *installed* a snapshot with timestamp t if the server has applied the modifications of all committed transactions with timestamp up to and including t. Once a server installs a snapshot with timestamp t, the server cannot commit any transaction with a timestamp $\leq t$.

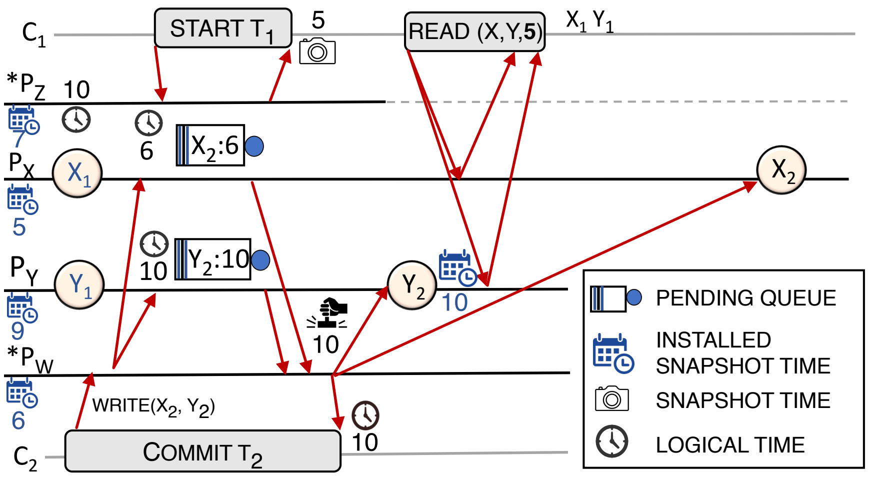
Achieving nonblocking reads in TCC is challenging, because they have to preserve consistency and respect the atomicity of multi-item (and hence multi-partition) write transactions. Assume that a transaction writes X and Y. A transaction T that reads x and y, must either see both X and Y or neither of them. The complexity of the problem is increased by the fact that the reads on individual keys in a transactional READ may proceed in parallel. In other words, a $READ(T_{ID}, x, y)$ sends in parallel a $read(x)$ operation to p_x and a $read(y)$ operation to p_y, and the $read(x)$ taking place on p_x is unaware of the item returned by the $read(y)$ on p_y.

Cure [4] provides the state-of-the-art solution to this problem. When a client c starts a transaction T, T is assigned a causal snapshot S by a randomly chosen *coordinator* partition. S includes all previous snapshots seen by c. To this end, the coordinator sets st as the maximum between the highest snapshot timestamp seen by c and the current clock value at the coordinator. When T commits, it is assigned a commit timestamp by means of a two-phase commit (2PC) protocol. Every partition that stores an item modified by T proposes a timestamp (strictly higher than st),

[I] Wren can be extended to allow a client c to move to a different DC by blocking c until the last snapshot seen by c has been installed in the new DC.



(a) Blocking reads in existing systems.



(b) Nonblocking reads in Wren.

Figure 4.1 – In existing systems (a), a transaction can be assigned a snapshot that has not been installed by every partition in the local DC. c_1's transaction is assigned timestamp 10, but p_x has not installed snapshot 10 by the time c_1 reads. This leads p_x to block c_1's read. In Wren (b), c_1's transaction is assigned a timestamp corresponding to a snapshot installed by every partition in the local DC, thus avoiding blocking. The trade-off is that older versions of x and y are returned.

and the coordinator picks the maximum as the commit timestamp ct of T. All items written by T are assigned ct as timestamp. Because $ct > st$, all such writes carry the information that they depend on the items in S, whose timestamps are less than or equal to st.

Cure achieves causality and enforces atomicity. If a transaction is assigned a snapshot timestamp st, the individual read operations of a READ transaction can in parallel read the version of any requested key with the highest timestamp $\leq st$. This protocol, however, enforces causality and atomicity at the cost of potentially blocking read operations. We show this behavior by means of an example, depicted in Figure 1a.

To initiate T_1, client c_1 contacts a coordinator partition, in this case p_z. T_1 is the first transaction issued by c_1, so c_1 does not piggyback any snapshot timestamp to initiate a transaction. The local time on p_z is 10. To maximize the freshness of the snapshot visible to T_1, p_z assigns to T_1 a timestamp equal to 10. In the meantime, c_2 commits T_2, which writes X_2 and Y_2. During the 2PC, p_x proposes 6 as commit timestamp, i.e., the current clock's value on p_x. Similarly, p_y proposes 10. The coordinator of T_2, p_w, picks the maximum between these two values and assigns to T_2 a commit timestamp 10. p_y receives the commit message, writes Y_2 and installs a snapshot with timestamp 10. p_x, instead, does not immediately receive the commit message, and its snapshot still has the value 5.

At this point, c_1 issues its READ(T_1, x, y) operation by sending a request to p_x and p_y with the snapshot timestamp of T_1, which is 10. p_y has installed a snapshot that is fresh enough, and returns Y_2. Instead, p_x has to block the read of T_1, because p_x cannot determine which version of x to return. p_x cannot safely return X_1, because it could violate CC and atomicity. p_x cannot return X_2 either, because p_x does not yet know the commit timestamp of X_2. If X_2 were eventually to be assigned a commit timestamp > 10, then returning X_2 to T_1 violates CC. p_x can install X_2 and the corresponding snapshot only when receiving the commit message from p_w. Then, p_x can serve c_1's pending read with the consistent value X_2.

Similar dynamics characterize also other CC systems with write transactions, e.g., Eiger [70].

4.3.2 Nonblocking reads in Wren

Wren implements CANToR, a novel transaction protocol that, similarly to Cure, is based on snapshots and 2PC, but avoids blocking reads by changing how snapshots visible to transactions are defined. In particular, a transaction snapshot is expressed as the union of two components:

1. a fresh causal snapshot installed by every partition in the local DC, which we call *local stable snapshot*, and
2. a client-side cache for writes done by the client and that have not yet been included in the local stable snapshot.

1) Causal snapshot. Existing approaches block reads, because the snapshot assigned to a transaction T may be "in the future" with respect to the snapshot installed by a server from which T reads an item. CANToR avoids blocking by providing to a transaction a snapshot that only includes writes of transactions that have been installed at all partitions. When using such a snapshot, then clearly all reads can proceed without blocking.

To ensure freshness, the snapshot timestamp st provided to a client is the largest timestamp such that all transactions with a commit timestamp smaller than or equal to st have been installed at all partitions. We call this timestamp the *local stable time* (LST), and the snapshot that it defines the *local stable snapshot*. The LST is determined by a stabilization protocol, by which partitions within a DC gossip the latest snapshots they have installed (§ 4.3.3). In CANToR, when a transaction starts, it chooses a transaction coordinator, and it uses as its snapshot timestamp the LST value known to the coordinator.

Figure 1b depicts the nonblocking behavior of Wren. p_z proposes 5 as snapshot timestamp (because of p_x). Then c_1 can read without blocking on both p_x and p_y, despite the concurrent commit of T_2. The trade-off is that c_1 reads older versions of x and y, namely X_1 and Y_1, compared to the scenarion in Figure 1a, where it reads X_2 and Y_2.

Only assigning a snapshot slightly in the past, however, does not solve completely the issue of blocking reads. The local stable snapshot includes *all* the items that have been written by *all* clients up until the boundary defined by the snapshot and on which c (potentially) depends. The local stable snapshot, however, might not include the most recent writes performed by c in earlier transactions.

Consider, for example, the case in which c commits a transaction T, that includes a write on item x, and obtains a value ct as its commit timestamp. Subsequently, c starts another transaction T' and obtains a snapshot timestamp st'smaller than ct, because ct has not yet been installed at all partitions. If we were to let c read from this snapshot, and it were to read x, it would not see the value it had written previously in T.

A simple solution would be to block the commit of T until $ct \geq LST$. This would guarantee that c can issue its next transaction T' only after the modifications of T have been applied at *every partition in the DC*. This approach, however, introduces high commit latencies.

2) Client-side cache. Wren takes a different approach that leverages the fact that the only causal dependencies of c that may not be in the local stable snapshot are items that c has written itself in earlier transactions (e.g., x). Wren therefore provides clients with a private cache for such items: all items written by c are stored in its private cache, from which it reads when appropriate, as detailed below.

When starting a transaction, the client removes from the cache all the items that are included in the causal snapshot, in other words all items with commit timestamp lower than its causal snapshot time st. When reading x, a client first looks up x in its cache. If there is a version of x in the cache, it means that the client has written a version of x that is not included in the transaction snapshot. Hence, it *must* be read from the cache. Otherwise, the client reads x from p_x. In either case, the read is performed without blocking [II].

4.3.3 Dependency tracking and stabilization protocols

BDT. Wren implements BDT, a novel protocol to track the causal dependencies of items. The key feature of BDT is that every data item tracks dependencies by means of only two scalar timestamps, regardless of the scale of the system. One entry tracks the dependencies on local items and the other entry summarizes the dependencies on remote items.

The use of only two timestamps enables higher efficiency and scalability than other designs. State-of-the-art solutions employ dependency meta-data whose size grows with the number of DCs [4, 114], partitions [43] or causal dependencies [72, 69, 70, 5]. Meta-data efficiency is paramount for many applications dominated by very small items, e.g., Facebook [80, 9], in which meta-data can easily grow bigger than the item itself. Large meta-data increases processing, communication and storage overhead.

BiST. Wren relies on BDT to implement BiST, an efficient stabilization protocol to determine when updates can be included in the snapshots proposed to clients within a DC (i.e., when they are *visible* within a DC). BiST allows updates originating in a DC to become visible in that DC without waiting for the receipt of remote items. A remote update d, instead, is visible in a DC when it is *stable*, i.e., when all the causal dependencies of d have been received in the DC.

BiST computes two cut-off values that indicate, respectively, which local and remote items can become visible to transactions within a DC. The local component computed by BiST is the LST, which we described earlier. The remote component is the Remote Stable Time (RST), that, similarly to the LST, indicates a lower bound on remote snapshots that have been installed by every node within the local DC.

By decoupling local and remote items, BiST allows transactions to determine the visibility of local items without synchronizing with remote DCs [4], in contrast to systems that use a single scalar timestamp for dependency tracking [42, 43]. This decoupling enables availability and nonblocking reads also in the geo-replicated case, because a snapshot visible to a transaction includes only remote items that have already been received in the local DC.

With BiST, periodically partitions within a DC exchange the commit timestamp of the latest

[II]The client can avoid contacting p_x, because Wren uses the last-writer-wins rule to resolve conflicting updates (see § 2.4.3). With other conflict resolution methods, the client would always have to read the version of x from p_x, and apply the updates(s) in the cache to that version.

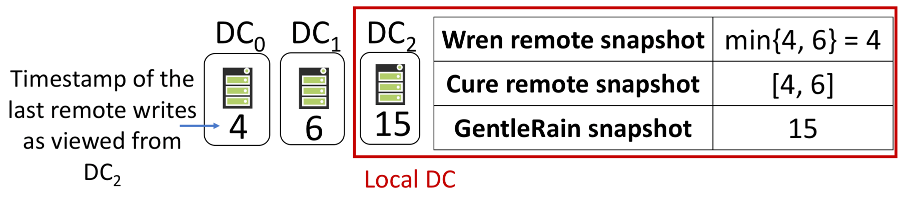


Figure 4.2 – Resource efficiency vs freshness in BiST (one partition per DC). DC_2 is the local DC.

local and remote transactions they have applied. Then, each partition computes the LST, resp., RST, as minimum of the received timestamps corresponding to local, resp., remote transactions. Therefore, LST and RST reflect local and remote snapshots that have been already installed by all partitions in the DC, and from which transactions can read without blocking, as we explain in the following.

Snapshots and nonblocking reads. When a transaction T starts, the local, resp., remote, entry of the corresponding snapshot S is set to be the maximum between the LST, resp., RST on the coordinator and the highest LST, resp. RST, value seen by the client, ensuring that clients see monotonically increasing snapshots.

T uses the timestamps in S to determine the version of an item that it can read, namely the freshest version of an item that falls within the visible snapshot (let alone the items that are read from the client-side cache, as described in § 4.3.2). S includes local, resp., remote, items whose timestamp is lower than the LST, resp., the RST. Because both LST and RST reflect snapshots installed by every partition in the DC, T can read from S in a nonblocking fashion.

Trade-off. BiST enables high scalability and performance at the expense of a slight increase in the time that it takes for an update to become visible in a DC (the so called *visibility latency*). By using BiST, Wren only tracks the lower bound on the commit time of local transactions (LST) and replicated transactions coming from all the remote DCs (RST). We describe this trade-off by sketching in Figure 4.2 how BiST and other existing stabilization protocols work at a high level.

In the example, the local DC (DC2) has committed transactions with timestamp up to 15. It has received commits from DC0 with timestamp up to 4, and from DC1 with timestamp up to 6. Wren exposes to transactions remote items with timestamp up to 4, the minimum of 4 and 6. Cure [4] uses one timestamp per DC, so transactions can see items from DC_0 with timestamp up to 4 and from DC_1 with timestamp up to 6. GentleRain [42] uses a single timestamp (the local one) to encode both local and remote snapshots, so transactions can see all items up to timestamp 15. However, they have to block until the local DC has received *all* remote updates with timestamps lower than or equal to 15.

Timestamps. So far, we have assumed that Wren uses logical, Lamport clocks to generate timestamps. Wren, instead, uses Hybrid Logical Physical Clocks (HLC) [61]. In brief, an HLC is a logical clock whose value on a partition is the maximum between the local physical clock and the highest timestamp seen by the partition plus one. HLCs combine the advantages of logical and physical clocks. Like logical clocks, HLCs can be moved forward to match the timestamp of an incoming event. Like physical clocks, they advance in the absence of events and at approximately the same pace.

Wren's use of HLCs improves the freshness of the snapshot determined by BiST, which, as a by-product, also reduces the amount of data stored in the client-side caches. HLCs have previously been employed by CC systems to avoid waiting for physical clocks to catch up when generating timestamps for updates [49, 90]. HLCs alone, however, do not solve the problem of blocking reads with TCC. A snapshot timestamp can be "in the future" with respect to the installed snapshot of a partition, regardless of whether the clock is logical, physical or hybrid.

4.3.4 Fault tolerance and Availability

Fault tolerance (within a DC). Similarly to previous transactional systems based on 2PC, Wren can integrate fault-tolerance capabilities by means of standard replication techniques such as Paxos [65].

Wren preserves nonblocking reads, even if such fault tolerance mechanisms are enabled. In blocking systems, instead, fault tolerance increases the latency incurred by transactions upon blocking, because it increases the duration of a commit.

The failure of a server blocks the progress of BiST, but only during the short amount of time during which a backup partition has not yet replaced the failed one. The failure of a client does not affect the behavior of the system. The clients only keep local meta-data, and cache data that have already been committed to the data-store.

Availability (between DCs). BiST is always available. Transactions are never blocked or delayed as a result of the disconnection or failure of a DC. The disconnection of DC_i causes the RST to freeze in all DCs that get disconnected from DC_i. However, because BiST decouples local from remote dependencies, any RST assigned to a transaction refers to a snapshot that is already available in the DC, and the transaction can thus proceed. The LST, instead, always advances, ensuring that clients can prune their local caches even if a DC disconnects.

Symbol	Definition
N	Number of partitions
M	Number of replicas per partition
p_n^m	The $m-$th replica of the $n-$th partition
lst_c	Client's local stable time
rst_c	Client's remote stable time
RS_c	Client's read set
WS_c	Client's write set
WC_c	Client-side cache
$Clock_n^m$	Physical clock time at p_n^m
HLC_n^m	Hybrid logical clock time at p_n^m
VV_n^m	Version vector of p_n^m
TX	Transaction context
$Prepared_n^m$	List of pending transactions of p_n^m
$Committed_n^m$	List of transactions of p_n^m, ready to be committed
lst_n^m	Local stable time of p_n^m
rst_n^m	Remote stable time of p_n^m
d	A tuple $\langle k, v, ut, rdt, id_t, sr \rangle$
k	A key
v	A value
ut	Update time
rdt	Remote dependency time
id_t	Transaction ID
sr	Source replica id

Table 4.1 – Symbols used in Wren's protocol description and corresponding meaning.

4.4 Protocols of Wren

We now describe in more detail the meta-data stored and the protocols implemented by clients and servers in Wren.

4.4.1 Meta-data

Items. An item d is a tuple $\langle k, v, ut, rdt, id_T, sr\rangle$. k and v are the key and value of d, respectively. ut is the timestamp of d which is assigned upon commit of d and summarizes the dependencies on local items. rdt is the remote dependency time of d, i.e., it summarizes the dependencies towards remote items. id_T is the id of the transaction that created the item version. sr is the source replica of d.

Client. In a client session, a client c maintains id_c which identifies the current transaction, and lst_c and rst_c, that correspond to the local and remote timestamp of the transaction snapshot, respectively. c also stores the commit time of its last update transaction, represented with hwt_c. Finally, c stores WS_c, RS_c and WC_c corresponding to the client's write set, read set and client-side cache, respectively.

Servers. A server p_n^m is identified by the partition id (n) and the DC id (m). In our description, thus, m is the local DC of the server. Each server has access to a monotonically increasing physical clock, $Clock_n^m$. The local clock value on p_n^m is represented by the hybrid clock HLC_n^m.

p_n^m also maintains VV_n^m, a vector of HLCs with M entries. $VV_n^m[i], i \neq m$ indicates the timestamp of the latest update received by p_n^m that comes from the n-th partition at the i-th DC. $VV_n^m[m]$ is the version clock of the server and represents the local snapshot installed by p_n^m. The server also stores lst_n^m and rst_n^m. $lst_n^m = t$ indicates that p_n^m is aware that every partition in the local DC has installed a local snapshot with timestamp at least t. $rst_n^m = t'$ indicates that p_n^m is aware that every partition in the local DC has installed all the updates generated from all remote DCs with update time up to t'.

Finally, p_n^m keeps a list of prepared and a list of committed transactions. The former stores transactions for which p_n^m has proposed a commit timestamp and for which p_n^m is awaiting the commit message. The latter stores transactions that have been assigned a commit timestamp and whose modifications are going to be applied to p_n^m.

4.4.2 Operations

Start. Client c initiates a transaction T by picking at random a *coordinator* partition (denoted p_n^m) and sending it a start request with lst_c and rst_c. p_n^m uses these values to update its lst_n^m and rst_n^m, so that p_n^m can propose a snapshot that is at least as fresh as the one accessed by c in previous transactions. Then, p_n^m generates the snapshot visible to T. The local snapshot timestamp is lst_n^m. The remote one is set as the minimum between rst_n^m and $lst_n^m - 1$. Wren enforces the remote snapshot time to be lower than the local one, to efficiently deal with concurrent conflicting

Algorithm 4 Wren client c (open session towards p_n^m).

```
1: function START
2:     send ⟨StartTxReq $lst_c, rst_c$⟩ to $p_n^m$
3:     receive ⟨StartTxResp $id, lst, rst$⟩ from $p_n^m$
4:     $rst_c \leftarrow rst$; $lst_c \leftarrow lst$; $id_c \leftarrow id$
5:     $RS_c \leftarrow \emptyset; WS_c \leftarrow \emptyset$
6:     Remove from $WC_c$ all items with commit timestamp up to $lst_c$
7: end function

8: function READ($\chi$)
9:     $D \leftarrow \emptyset$; $\chi' \leftarrow \emptyset$
10:    for each $k \in \chi$ do
11:        $d \leftarrow$ check $WS_c$, $RS_c$, $WC_c$ (in this order)
12:        if ($d \neq NULL$) then $D \leftarrow d$
13:    end for
14:    $\chi' \leftarrow \chi \setminus D.keySet()$
15:    send ⟨TxReadReq $id_c$,⟩ $\chi'$ to $p_n^m$
16:    receive ⟨TxReadResp $D'$⟩ from $p_n^m$
17:    $D \leftarrow D \cup D'$
18:    $RS_c \leftarrow RS_c \cup D$
19:    return $D$
20: end function

21: function WRITE($\chi$)
22:    for each $\langle k, v\rangle \in \chi$ do                 ▷ Update $WS_c$ or write new entry
23:        if ($\exists d \in WS : d == k$)then $d.v \leftarrow v$ else $WS_c \leftarrow WS_c \cup \langle k, v\rangle$
24:    end for
25: end function

26: function COMMIT                                               ▷ Only invoked if $WS \neq \emptyset$
27:    send ⟨CommitReq $id_c, hwt_c, WS_c$⟩ to $p_n^m$
28:    receive ⟨CommitResp $ct$⟩ from $p_n^m$
29:    $hwt_c \leftarrow ct$                                     ▷ Update client's highest write time
30:    Tag $WS_c$ entries with $hwt_c$
31:    Move $WS_c$ entries to $WC_c$                              ▷ Overwrite (older) duplicate entries
32: end function
```

updates. Assume c wants to read x, that c has a version X_l in its private cache with commit timestamp $ct > lst_n^m$, and that there exist a visible remote X_r with commit timestamp $\geq ct$. Then, c must retrieve X_r, its commit timestamp and its source replica to determine whether X_l or X_r should be read according to the last writer wins rule. By forcing the remote stable time to be lower than lst – and hence of ct – the client knows that the freshest visible version of x is X_l, which can be read locally from the private cache [III].

After defining the snapshot visible to T, p_n^m also generates a unique identifier for T, denoted id_T, and inserts T in a private data structure. p_n^m replies to c with id_T and the snapshot timestamps.

Upon receiving the reply, c updates lst_c and rst_c, and evicts from the cache any version with timestamp lower than lst_c. c can prune the cache using lst_c because p_n^m has enforced that the

[III] The likelihood of rst_n^m being higher than lst_n^m is low given that i) geo-replication delays are typically higher than the skew among the physical clocks [71] and ii) rst_n^m is the minimum value across all timestamps of the latest updates received in the local DC.

Algorithm 5 Wren server p_n^m - transaction coordinator.

```
1:  upon receive ⟨StartTxReq $lst_c, rst_c$⟩ from $c$ do
2:      $rst_n^m \leftarrow max\{rst_n^m, rst_c\}$                        ▷ Update remote stable time
3:      $lst_n^m \leftarrow max\{lst_n^m, lst_c\}$                        ▷ Update local stable time
4:      $id_T \leftarrow generateUniqueId()$
5:      $TX[id_T] \leftarrow \langle lst_n^m, \min\{rst_n^m, lst_n^m - 1\}\rangle$      ▷ Save TX context
6:      send ⟨StartTxResp $id_T, TX[id_T]$⟩                          ▷ Assign transaction snapshot

7:  upon receive ⟨TxReadReq $id_T$, $\chi$⟩ from $c$ do
8:      $\langle lt, rt\rangle \leftarrow TX[id_T]$
9:      $D \leftarrow \emptyset$
10:     $\chi_i \leftarrow \{k \in \chi : partition(k) == i\}$            ▷ Partitions with ≥ 1 key to read
11:     for ($i : \chi_i \neq \emptyset$) do
12:         send ⟨SliceReq $\chi_i, lt, rt$⟩ to $p_i^m$
13:         receive ⟨SliceResp $D_i$⟩ from $p_i^m$
14:         $D \leftarrow D \cup D_i$
15:     end for
16:     send ⟨TxReadResp $D$⟩ to $c$

17: upon receive ⟨CommitReq $id_T, hwt, WS$⟩ from $c$ do
18:     $\langle lt, rt\rangle \leftarrow TX[id_T]$
19:     $ht \leftarrow max\{lt, rt, hwt\}$                               ▷ Max timestamp seen by the client
20:     $D_i \leftarrow \{\langle k, v\rangle \in WS : partition(k) == i\}$
21:     for ($i : D_i \neq \emptyset$) do                                 ▷ Done in parallel
22:         send ⟨PrepareReq $id_T, lt, rt, ht, D_i$⟩ to $p_i^m$
23:         receive ⟨PrepareResp $id_T, pt_i$⟩ from $p_i^m$
24:     end for
25:     ct $\leftarrow max_{i:D_i \neq \emptyset}\{pt_i\}$                     ▷ Max proposed timestamp
26:     for ($i : D_i \neq \emptyset$) do send ⟨Commit $id_T, ct$⟩ to $p_i^m$ end for
27:     delete TX[$id_T$]                                                ▷ Clear transactional context of c
28:     send ⟨CommitResp $ct$⟩ to $c$
```

highest remote timestamp visible to T is lower than lst_n^m. This ensures that if, after pruning, there is a version X in the private cache of c, then $X.ct > lst$ and hence the freshest version of x visible to c is X.

Read. The client c provides the set of keys to read. For each key k to read, c searches the write-set, the read-set and the client cache, in this order. If an item corresponding to k is found, it is added to the set of items to return, ensuring read-your-own-writes and repeatable-reads semantics. Reads for keys that cannot be served locally are sent in parallel to the corresponding partitions, together with the snapshot from which to serve them. Upon receiving a read request, a server first updates the server's LST and RST, if they are smaller than the client's (Alg. 5 Lines 2–3). Then, the server returns to the client, for each key, the version within the snapshot with the highest timestamp (Alg. 6 Lines 6–10). c inserts the returned items in the read-set.

Write. Client c locally buffers the writes in its write-set WS_c. If a key being written is already present in WS_c, then it is updated; otherwise, it is inserted.

Algorithm 6 Wren server p_n^m - transaction cohort.

```
1:  upon receive ⟨SliceReq $\chi, lt, rt$⟩ from $p_i^m$ do
2:      $rst_n^m \leftarrow max\{rst_n^m, rst\}$                         ▷ Update remote stable time
3:      $lst_n^m \leftarrow max\{lst_n^m, lst\}$                         ▷ Update local stable time
4:      $D \leftarrow \emptyset$
5:      for ($k \in \chi$) do
6:          $D_k \leftarrow \{d : d.k == k\}$                            ▷ All versions of k
7:          $D_{lv} \leftarrow \{d : d.sr == m \wedge d.ut \leq lt \wedge d.rst \leq rt\}$   ▷ Local visible
8:          $D_{rv} \leftarrow \{d : d.sr \neq m \wedge d.ut \leq rt \wedge d.rst \leq lt\}$   ▷ Remote visible
9:          $D_{kv} \leftarrow \{D_k \cap \{D_{lv} \cup D_{rv}\}\}$              ▷ All visible versions of k
10:         $D \leftarrow D \cup \{argmax_{d.ut}\{d \in D_{kv}\}\}$        ▷ Freshest visible vers. of k
11:     end for
12:     reply ⟨SliceResp D⟩ to $p_i^m$

13: upon receive ⟨PrepareReq $id_T, lt, rt, ht, D_i$⟩ from $p_i^m$ do
14:     $HLC_n^m \leftarrow max(Clock_n^m, ht+1, HLC_n^m+1)$              ▷ Update HLC
15:     pt $\leftarrow HLC_n^m$                                        ▷ Proposed commit time
16:     $lst_n^m \leftarrow max\{lst_n^m, lt\}$                          ▷ Update local stable time
17:     $rst_n^m \leftarrow max\{rst_n^m, rt\}$                          ▷ Update remote stable time
18:     $Prepared_n^m \leftarrow Prepared_n^m \cup \{id_T, rt, D_i\}$       ▷ Append to pending list
19:     send ⟨PrepareResp $id_T, pt$⟩ to $p_i^m$

20: upon receive ⟨CommitReq $id_T, ct$⟩ from $c$ do
21:     $HLC_n^m \leftarrow max(HLC_n^m, ct, Clock_n^m)$                  ▷ Update HLC
22:     $\langle id_T, rst, D \rangle \leftarrow \{\langle i, r, \phi \rangle \in Prepared_n^m : i == id_T\}$
23:     $Prepared_n^m \leftarrow Prepared_n^m \setminus \{\langle id_T, rst, D \rangle\}$      ▷ Remove from pending
24:     $Committed_n^m \leftarrow Committed_n^m \cup \{\langle id_T, ct, rst, D \}$   ▷ Mark to commit
```

Commit. The client sends a commit request to the coordinator with the content of WS_c, the id of the transaction and the commit of its last update transaction hwt_c, if any. The coordinator contacts the partitions that store the keys that need to be updated (the cohorts) and sends them the corresponding updates and hwt_c. The partitions update their HLCs, propose a commit timestamp and append the transaction to the pending list. To reflect causality, the proposed timestamp is higher than the snapshot timestamps and hwt_c. The coordinator then picks the maximum among the proposed timestamps [41], sends it to the cohort partitions, clears the local context of the transaction and sends the commit timestamp to the client. The cohort partitions move the transaction from the pending list to the commit list, with the new commit timestamp.

Applying and replicating transactions. Periodically, the servers apply the effects of committed transactions, in increasing commit timestamp order (Alg. 7 Lines 6-20). p_n^m applies the modifications of transactions that have a commit timestamp lower than the lowest timestamp present in the pending list. This timestamp represents the lower bound on the commit timestamps of future transactions on p_n^m. After applying the transactions, p_n^m updates its local version clock and replicates the transactions to remote DCs. When there are more transactions with the same commit time ct, p_n^m updates its local version clock only after applying the last transaction with the same ct and packs them together to be propagated in one replication message (Alg. 7 Lines 10–17).

Algorithm 7 Wren server p_n^m - auxiliary functions.

```
1: function UPDATE($k, v, ut, rdt, id_T$)
2:     create $d : \langle d.k, d.v, d.ut, d.rdt, d.id_T, d.sr\rangle \leftarrow \langle k, v, ut, rdt, id_T, m\rangle$
3:     insert new item $d$ in the version chain of key $k$
4: end function

5: upon Every $\Delta_R$ do
6:     if ($Prepared_n^m \neq \emptyset$) then $ub \leftarrow min_{\{p.pt\}}\{p \in Prepared_n^m\} - 1$
7:     else $ub \leftarrow max\{Clock_n^m, HLC_n^m\}$ end if
8:     if ($Committed_n^m \neq \emptyset$) then                    ▷ Commit tx in increasing order of ct
9:         $C \leftarrow \{\langle id, ct, rst, D\rangle\} \in Committed_n^m : ct \leq ub$
10:        for ($T \leftarrow \{\langle id, rst, D\rangle\} \in (group\ C\ by\ ct)$) do
11:            for ($\langle id, rst, D\rangle \in T$) do
12:                for ($\langle k, v\rangle \in D$) do update ($k, v, ct, rst, id$) end for
13:            end for
14:            for ($i \neq m$) send ⟨Replicate $T, ct$⟩ to $p_n^i$ end for
15:            $Committed_n^m \leftarrow Committed_n^m \setminus T$
16:        end for
17:        $VV_n^m[m] \leftarrow ub$                                 ▷ Set version clock
18:    else
19:        $VV_n^m[m] \leftarrow ub$                                 ▷ Set version clock
20:        for ($i \neq m$) do send ⟨Heartbeat $VV_n^m[m]$⟩ to $p_n^i$ end for
21:    end if

22: upon receive ⟨Replicate $T, ct$⟩ from $p_n^i$ do
23:     for ($\langle id, rst, D\rangle \in T$) do
24:         for ($\langle k, v\rangle \in D$) do update ($k, v, ct, rst, id$) end for
25:     end for
26:     $VV_n^m[i] \leftarrow ct$                          ▷ Update remote snapshot of i-th replica

27: upon receive ⟨Heartbeat t⟩ from $p_n^i$ do
28:     $VV_n^m[i] \leftarrow t$                           ▷ Update remote snapshot of i-th replica

29: upon Every $\Delta_G$ do                   ▷ Compute remote and local stable snapshots
30:     $rst_n^m \leftarrow min_{\{i=0,\ldots,M-1, i\neq m; j=0,\ldots,N-1\}} VV_j^m[i]$        ▷ Remote
31:     $lst_n^m \leftarrow min_{\{i=0,\ldots,N-1\}} VV_i^m[m]$                  ▷ Local
```

If a server does not commit a transaction for a given amount of time, it sends a heartbeat with its current HLC to its peer replicas, ensuring the progress of the RST.

BiST. Periodically, partitions within a DC exchange their version vectors. The LST is computed as the minimum across the local entries in such vectors; the RST as minimum across the remote ones (Alg. 7 Lines 29–31). Partitions within a DC are organized as a tree to reduce communication costs [42].

Garbage collection. Periodically, the partitions within a DC exchange the oldest snapshot corresponding to an active transaction (p_n^m sends its current visible snapshot if it has is no running transaction). The aggregate minimum determines the oldest snapshot S_{old} that is visible to a running transaction. The partitions scan the version chain of each key backwards and keep the all the versions up to (and including) the oldest one within S_{old}. Earlier versions are removed.

4.4.3 Correctness

In this section we provide a high-level argument to show the correctness of Wren.

Snapshots are causal. To start a transaction, a client c piggybacks the freshest snapshot it has seen, ensuring the monotonicity of the snapshot seen by c (Alg. 8 Lines 1–6). Commit timestamps reflect causality (Alg. 8 Line 19), and BiST tracks a lower bound on the snapshot installed by every partition in a DC. If X is within the snapshot of a transaction, so are its dependencies, because i) dependencies generated in the same DC where X is created have a timestamp lower than X and ii) dependencies generated in a remote DC have a timestamp lower than $X.rdt$. On top of the snapshot provided by the coordinator, the client applies its writes that are not in the snapshot. These writes cannot depend on items created by other clients that are outside the snapshot visible to c.

Writes are atomic. Items written by a transaction have the same commit timestamp and RST. LST and RST are computed as the minimum values across all the partitions within a DC. If a transaction has written X and Y and a snapshot contains X, then it also contains Y (and vice-versa).

4.5 Evaluation

We evaluate the performance of Wren in terms of throughput, latency and update visibility. We compare Wren with Cure [4], the state-of-the-art approach to TCC, and with H-Cure, a variant of Cure that uses HLCs. By comparing with H-Cure, we show that using HLCs alone, as in existing systems [90, 37], is not sufficient to achieve the same performance as Wren, and that nonblocking reads in the presence of multi-item atomic writes are essential.

4.5.1 Experimental environment

Platform. We consider a geo-replicated setting deployed across up to 5 replication sites on Amazon EC2 (Virginia, Oregon, Ireland, Mumbai and Sydney). When using 3 DCs, we use Virginia, Oregon and Ireland. In each DC we use up to 16 servers (m4.large instances with 2 VCPUs and 8 GB of RAM). We spawn one client process per partition in each DC. Clients issue requests in a closed loop, and are collocated with the server partition they use as coordinator. We spawn different number of client threads to generate different load conditions. In particular, we spawn 1, 2, 4, 8, 16 threads per client process. Each “dot" in the curve plots corresponds to a different number of threads per client.

Implementation. We implement Wren, H-Cure and Cure in the same C++ code-base [IV]. All protocols implement the last-writer-wins rule for convergence. We use Google Protobufs for communication, and NTP to synchronize physical clocks. The stabilization protocols run every 5 milliseconds.

Workloads. We use workloads with 95:5, 90:10 and 50:50 r:w ratios. These are standard workloads also used to benchmark other TCC systems [4, 114, 76]. In particular, the 50:50 and 95:5 r:w ratio workloads correspond, respectively, to the update-heavy (A) and read-heavy (B) YCSB workloads [32]. Transactions generate the three workloads by executing 19 reads and 1 write (95:5), 18 reads and 2 writes (90:10), and 10 reads and 10 writes (50:50). A transaction first executes all reads in parallel, and then all writes in parallel.

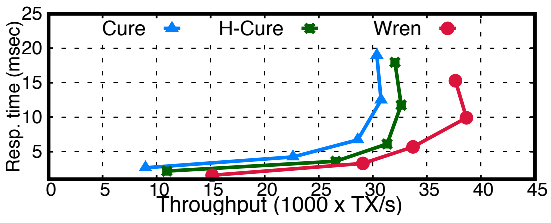
Our default workload uses the 95:5 r:w ratio and runs transactions that involve 4 partitions on a platform deployed over 3 DCs and 8 partitions. We also consider variations of this workload in which we change the value of one parameter and keep the others at their default values. Transactions access keys within a partition according to a zipfian distribution, with parameter 0.99, which is the default in YCSB and resembles the strong skew that characterizes many production systems [9, 78, 17]. We use small items (8 bytes), which are prevalent in many production workloads [9, 78]. With bigger items Wren would retain the benefits of its nonblocking reads. The effectiveness of BDT and BiST would naturally decrease as the size of the items increases, because meta-data overhead would become less critical.

4.5.2 Performance evaluation

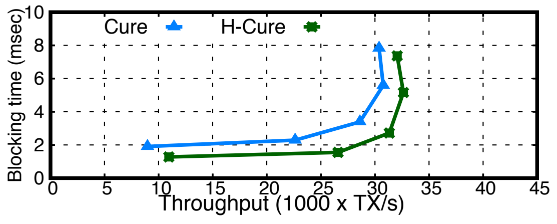
Latency and throughput. Figure 4.3a reports the average transaction latency vs. throughput achieved by Wren, H-Cure and Cure with the default workload. Wren achieves up to 2.33x lower response times than Cure, because it never blocks a read due to clock skew or to wait for a snapshot to be installed. Wren also achieves up to 25% higher throughput than Cure. Cure needs a higher number of concurrent clients to fully utilize the processing power left idle by blocked reads. The presence of more threads creates more contention on the physical resources and implies more synchronization to block and unblock reads, which ultimately leads to lower throughput.

Wren also outperforms H-Cure, achieving up to 40% lower latency and up to 15% higher throughput. HLCs enable H-Cure to avoid blocking the read of a transaction T because of clock skew. This blocking happens on a partition if the local timestamp of T's snapshot is t, there are no pending or committed transactions on the partition with commit timestamp lower than t, but

[IV] https://github.com/epfl-labos/wren



(a) Throughput vs average TX latency.



(b) Mean blocking time in Cure and H-Cure. Wren never blocks.

Figure 4.3 – Performance of Wren, H-Cure and Cure on 3 DCs, 8 partitions/DC, 4 partitions involved per transaction, and 95:5 r:w ratio. Wren achieves better latencies because it never blocks reads (a). H-Cure achieves performance in-between Cure and Wren, showing that only using HLCs does not solve the problem of blocking reads in TCC. Cure and H-Cure incur a mean blocking time that grows with the load (b). Because of blocking, Cure and H-Cure need higher concurrency to fully utilize the resources on the servers. This leads to higher contention on physical resources and to a lower throughput (a).

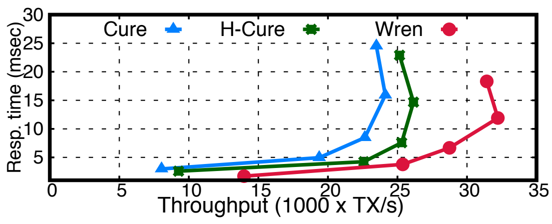
the physical clock on the partition is lower than t. HLCs, however, cannot avoid blocking T if there are pending transactions on the partition, and T is assigned a snapshot that has not been installed on the partition.

Statistics on blocking in Cure and H-Cure. Figure 4.3b provides insights on the blocking occurring in Cure and H-Cure, that leads to the aforementioned performance differences. The plots show the mean blocking time of transactions that block upon reading. A transaction T is considered as blocked if at least one of its individual reads blocks. The blocking time of T is computed as the maximum blocking time of a read belonging to T.

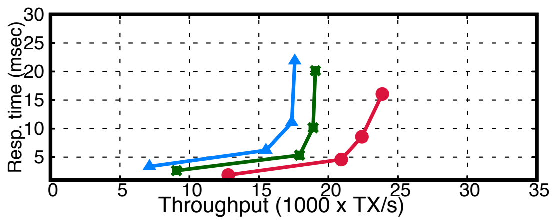
Blocking can take up a vast portion of a transaction execution time. In Cure, blocking reads introduce a delay of 2 milliseconds at low load, and almost 4 milliseconds at high load (without considering overload conditions). These values correspond to 35-48% of the total mean transaction execution time. Similar considerations hold for H-Cure. The blocking time increases with the load, because higher load leads to more transactions being inserted in the pending and commit queues, and to higher latency between the time a transaction is committed and the corresponding snapshot is installed.

4.5.3 Varying the workload

Figure 4.4a and Figure 4.4b report the average transaction latency as a function of the load for the 90:10 and 50:50 r:w ratios, respectively. Figure 4.5a and Figure 4.5b report the same metric with the default r:w ratio of 95:5, but with $p = 2$ and $p = 8$ partitions involved in a transaction,

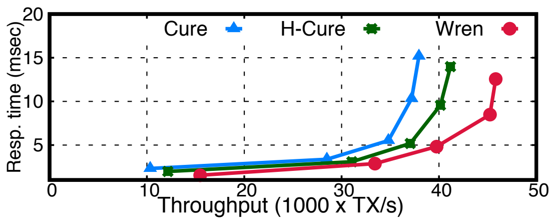


(a) Throughput vs average TX latency (90:10 r:w).

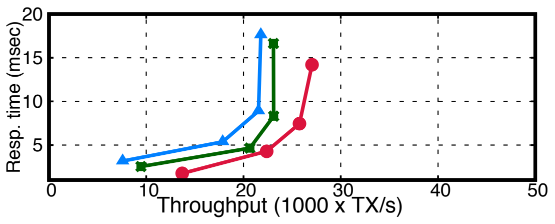


(b) Throughput vs average TX latency (50:50 r:w).

Figure 4.4 – Performance of Wren, Cure and H-Cure with different 90:10 (a) and 50:50 (b) r:w ratios, 4 partitions involved per transaction (3DCs, 8 partitions). Wren outperforms Cure and H-Cure for both read-heavy and write-heavy workloads.



(a) Throughput vs average TX latency (p=2).



(b) Throughput vs average TX latency (p=8).

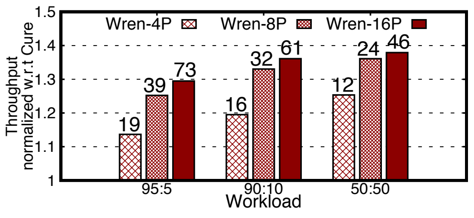
Figure 4.5 – Performance of Wren, Cure and H-Cure with transactions that read from 2 (a) and 8 (b) partitions with 95:5 r:w ratio (3DCs, 8 partitions). Wren outperforms Cure and H-Cure with both small and large transactions.

respectively. These figures show that Wren delivers better performance than Cure and H-Cure for a wide range of workloads. It achieves transaction latencies up to 3.6x lower then Cure, and up to 1.6x lower than H-Cure. It achieves maximum throughput up to 1.33x higher than Cure and 1.23x higher than H-Cure. The peak throughput of all three systems decreases with a lower r:w ratio, because writing more items increases the duration of the commit and the replication overhead. Similarly, a higher value of p decreases throughput, because more partitions are contacted during a transaction.

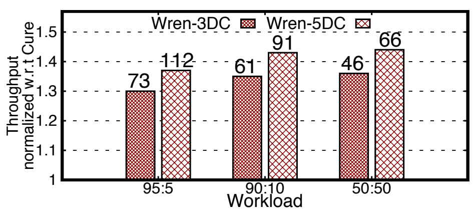
4.5.4 Varying the number of partitions

Figure 4.6a reports the throughput achieved by Wren with 4, 8 and 16 partitions per DC. The bars represent the throughput of Wren normalized with respect to the throughput achieved by Cure in the same setting. The number on top of each bar represents the absolute throughput achieved by Wren.

The plots show three main results. First, Wren consistently achieves higher throughput than Cure, with a maximum improvement of 38%. Second, the performance improvement of Wren is more



(a) Throughput when varying the number of partitions/DC (3 DCs).



(b) Throughput when varying the number of DCs (16 partitions/DCs).

Figure 4.6 – Throughput achieved by Wren when increasing the number of partition per DC (a) and DCs in the system (b). Each bar represents the throughput of Wren normalized w.r.t. to Cure (y axis starts from 1). The number on top of each bar reports the absolute value of the throughput achieved by Wren in 1000 x TX/s. Wren consistently achieves better throughput than Cure and achieves good scalability both when increasing the number of partitions and the number of DCs.

evident with more partitions and lower r:w ratios. More partitions touched by transactions and more writes increase the chances that a read in Cure targets a laggard partition and blocks, leading to higher latencies, lower resource efficiency, and worse throughput. Third, Wren provides efficient support for application scale-out. When increasing the number of partitions from 4 to 16, throughput increases by 3.76x for the write-heavy and 3.88x for read-heavy workload, approximating the ideal improvement of 4x.

4.5.5 Varying the number of DCs

Figure 4.6b shows the throughput achieved by Wren with 3 and 5 DCs (16 partitions per DC). The bars represent the throughput normalized with respect to Cure's throughput in the same scenario. The numbers on top of the bars indicate the absolute throughput achieved by Wren.

Wren obtains higher throughput than Cure for all workloads, achieving an improvement of up to 43%. Wren performance gains are higher with 5 DCs, because the meta-data overhead is constant in BiST, while in Cure it grows linearly with the number of DCs. The throughput achieved by Wren with 5 DCs is 1.53x, 1.49x, and 1.44x higher than the throughput achieved with 3 DCs, for the 95:5, 90:10 and 50:50 workloads, respectively, approximating the ideal improvement of 1.66x. A higher write intensity reduces the performance gain when scaling from 3 to 5 DCs, because it implies more updates being replicated.

4.5.6 Resource efficiency

Figure 4.7a shows the amount of data exchanged in Wren to run the stabilization protocol and to replicate updates, with the default workload. The results are normalized with respect to the

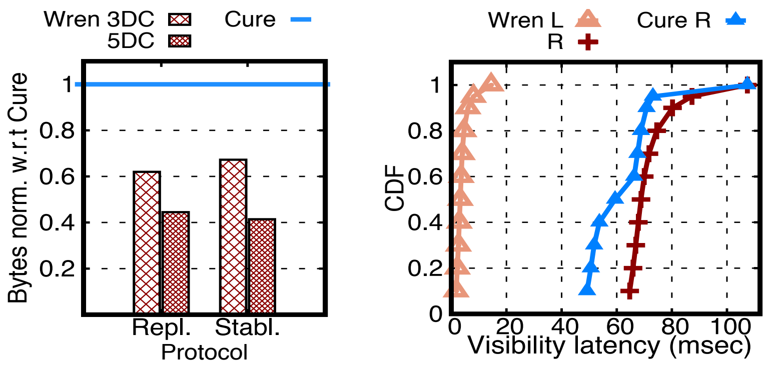


Figure 4.7 – (a) BiST incurs lower overhead than Cure to track the dependencies of replicated updates and to determine transactional snapshots (default workload). (b) Wren achieves a slightly higher **R**emote update visibility latency w.r.t. Cure, and makes **L**ocal updates visible when they are within the local stable snapshot (3 DCs).

amounts of data exchanged in Cure at the same throughput. With 5 DCs, Wren exchanges up to 37% fewer bytes for replication and up to 60% fewer bytes for running the stabilization protocol. With 5 DCs, updates, snapshots and stabilization messages carry 2 timestamps in Wren versus 5 in Cure.

4.5.7 Update visibility

Figure 4.7b shows the CDF of the update visibility latency with 3 DCs. The visibility latency of an update X in DC_i is the difference between the wall-clock time when X becomes visible in DC_i and the wall-clock time when X was committed in its original DC (which is DC_i itself in the case of local visibility latency). The CDFs are computed as follows: we first obtain the CDF on every partition and then we compute the mean for each percentile.

Cure achieves lower update visibility latencies than Wren. The remote update visibility time in Wren is slightly higher than in Cure (68 vs. 59 milliseconds in the worst case, i.e., 15% higher), because Wren tracks dependencies at the granularity of the DC, while Wren only tracks local and remote dependencies (see § 4.3.3 Figure 4.2). Local updates become visible immediately in Cure. In Wren they become visible after a few milliseconds, because Wren chooses a slightly older snapshot. We argue that these slightly higher update visibility latencies are a small price to pay for the performance improvements offered by Wren.

4.6 Conclusion

We have presented Wren, the first TCC system that at the same time implements nonblocking reads thereby achieving low latency and allows applications to scale-out by sharding. Wren implements a novel transactional protocol, CANToR, that defines transaction snapshots as the union of a fresh causal snapshot and the contents of a client-side cache. Wren also introduces BDT, a new dependency tracking protocol, and BiST, a new stabilization protocol. BDT and BiST use only 2 timestamps per update and per snapshot, enabling scalability regardless of the size of the system. We have compared Wren with the state-of-the-art TCC system, and we have shown that Wren achieves lower latencies and higher throughput, while only slightly penalizing the freshness of data exposed to clients.

5 TCC with Non-blocking Reads and Partial Replication

5.1 Introduction

In geo-replicated environments, partial replication is an effective technique to reduce storage requirements and replication costs. In partial replication, each DC stores only a subset of the partitions, as opposed to the full replication case, where each DC stores the whole dataset. Hence, the system can scale to a higher number of partitions with respect to a full replication approach, and updates performed in one DC are propagated to fewer replicas.

This chapter presents PaRiS, the first system that implements TCC on a partially replicated data platform, and that supports non-blocking parallel read operations (and hence, non-blocking read-only transactions). Parallel non-blocking reads are an important requirement to guarantee good performance [72, 105, 34], especially for the large class of important read-intensive applications [107, 80, 81].

Achieving non-blocking parallel transactional reads with partial replication is challenging, because different reads within the same transaction may be served in parallel by servers in different DCs. With full replication, a transaction executes entirely within a single DC, and therefore protocols for full replication are not prepared to deal with the situation of reads of the same transaction being served in different DCs. For instance, different sets of transactions may have been committed in different DCs, and naive application of full replication protocols may lead to reads returning inconsistent results.

PaRiS addresses this issue by means of a new causal dependency tracking protocol, that we call Universal Stable Time (UST). In short, UST identifies a snapshot of the dataset that has been installed in *all* DCs. Hence, a transaction in any DC can read from such a snapshot without blocking. In addition to the snapshot defined by UST, PaRiS equips clients with a private cache, in which clients store their own updates that are not yet reflected in the snapshot identified by the

UST. The combination of these two techniques, UST and private caches, suffices to implement TCC with non-blocking reads in a partial replication setting. PaRiS computes UST efficiently by means of a periodic, lightweight intra- and inter-DC gossiping protocol. In addition, PaRiS uses only one timestamp to track dependencies and to define transactional snapshots, thus enabling scalability both in terms of number of DCs and number of partitions per DC.

Overall, PaRiS achieves low latency, low storage requirements and rich transactional semantics. This combination represents a significant improvement over existing systems that either block read operations to preserve consistency, do not support partial replication, or do not support generic read-write transactions.

The trade-off made by PaRiS –which is provably unavoidable [105]– is to expose to transactions a view of the data that is slightly in the past. We argue that a moderate increase in data staleness is a reasonable price to pay for the performance benefits brought about by PaRiS.

We evaluate PaRiS on an AWS deployment comprising of up to 10 DCs, and with heterogeneous workloads with different degrees of data access locality. We show that PaRiS scales well with the number of DCs and partitions, while being able to handle larger datasets than existing solutions that assume full replication.

Contributions. The key contributions of this chapter are:

1. We present the design and implementation of PaRiS, the first TCC system that supports partial replication and implements non-blocking parallel read operations.
2. We propose a novel dependency tracking protocol call Universal Stable Time (UST) that a snapshot of the dataset that has been installed in all DCs.
3. We experimentally demonstrate the benefits of PaRiS over a blocking alternative

The outline of the rest of this chapter is as follows. Section 5.2 describes TCC and the target system model. Section 5.3 presents the design of PaRiS. Section 5.4 describes the protocols in PaRiS. Section 5.5 presents the evaluation of PaRiS. Section 5.6 concludes the chapter.

5.2 System Model

5.2.1 System model

We consider a distributed key-value store whose dataset is split into N partitions. We assume that each server is assigned a single partition, and we denote by p_x the server responsible for key x.

Each partition p_i is replicated at R different DCs, where R is the replication factor. There are M DCs in total where $M > R$, hence, only a subset of the full dataset is present in each DC. A client reads locally whenever possible, otherwise can contact any remote replica of a key.

We assume a multi-master system, i.e., all replicas can update keys for which they are responsible. Updates are replicated asynchronously to remote DCs.

We assume a data store where each item can have multiple versions, and with every update operation, a new version of a data item is created. To track causality, each version of a data item stores metadata, together with the value of the key. However, old versions of items are periodically garbage-collected. Partitions communicate through point-to-point lossless FIFO channels (e.g., TCP sockets).

At the beginning of a session, a client c connects to a partition p in one DC according to some load balancing scheme. This DC is referred to as the local DC. The partition p serves all c's operations. If p does not store a key k targeted by an operation, p transparently forwards the operation to a partition storing k, possibly in a different DC. c does not issue the next operation until it receives the reply to the current one.

Availability. We use the term availability to indicate that a client operation is never blocked in the presence of a network partition [4, 96].

5.2.2 APIs

PaRiS's programming interface offers the following operations for interactive read-write transactions:

- $< T_{ID}, S > \leftarrow START - TX()$: starts an interactive transaction T and returns T's transaction identifier T$_{ID}$ and the causally consistent snapshot S visible to T.
- $\langle vals \rangle \leftarrow READ(T_{ID}, k_1, ..., k_n)$: reads in parallel the set of items corresponding to the input set of keys for a transaction identified by T$_{ID}$.
- $WRITE(T_{ID}, \langle k_1, v_1 \rangle, ..., \langle k_n, v_n \rangle)$: updates a set of keys, given as input, to the corresponding values for a transaction with T$_{ID}$.
- $COMMIT - TX(T_{ID})$: finalizes the transaction T$_{ID}$ and atomically updates items that have been modified by means of a WRITE operation in the scope of the transaction, if any.

Clients can issue multiple read and write operations that can operate on multiple keys, between the start and the commit of T.

Under the TCC programming model, conflicting updates are resolved rather than forbidden. Therefore, transactions never abort due to conflicts. Although transactions can abort by means of system-related issues, e.g., not enough space on a server to perform an update, for simplicity we do not consider aborts in this dissertation.

5.3 Design of PaRiS

PaRiS implements, in a scalable manner, TCC with parallel transactional reads in a partially replicated and sharded system. We first illustrate the challenges involved in doing so in Section 5.3.1. Next, in Section 5.3.2 we present how PaRiS overcomes these challenges by a novel dependency tracking protocol and the use of a small client-side cache. Finally, in Section 5.3.3 we discuss fault tolerance and availability in PaRiS.

5.3.1 Challenges of partial replication

Since TCC must simultaneously guarantee the preservation of causal consistency and the atomicity of multi-item writes, achieving non-blocking reads while maintaining TCC is challenging. In a fully replicated environment, the task of enforcing this behavior is eased by two invariants: i) all remote updates are received by all DCs, and hence every DC receives the dependencies of each update, and ii) all updates of a transaction are performed within the same DC, and hence all the updates of the same transaction can be found in each DC. As example, assume that $X \rightsquigarrow Y$, and that X and Y are the latest versions of their keys, x and y, respectively. Causal consistency dictates that if a client reads x and y in a transaction, then if Y is returned, X has to be returned as well. Furthermore, assume that Z, the last version of key z, has been written by the same transaction as Y. TCC further implies that either both Y and Z are visible to the transaction, or none of them is. In full replication, some sort of communication among partitions in the *same* DC is enough to ensure that Y is visible in the DC only after X, and that Y and Z are atomically visible.

Partial replication, instead, violates the two invariants described above. This leads to a new set of challenges of enforcing consistency and atomicity. First, tracking consistency is harder. In the previous example with keys x and y, X may be replicated from DC_0 to DC_1, and Y from DC_0 to DC_2. Then, assume that a transaction from DC_3 reads x in DC_1 and y in DC_2. The transaction has to ensure that Y is read in DC_2 only if also X is read, concurrently, in DC_1. Similarly, enforcing atomicity is harder. Assume that Z is replicated from DC_0 to DC_1, and Y from DC_0 to DC_2, and that a transaction from DC_3 reads y and z. Then, the transaction in DC_3 has to ensure that either both Y and Z or none of them are read in an atomic fashion.

Addressing these two challenges is made more difficult by the fact that a read operation can target any replica of the target key. Hence, consistency and atomicity have to be preserved despite the fact that different transactions targeting the same keys can hit different replicas of those keys. The complexity of the problem is further exacerbated by the fact that different replicas of a version X may be in different DCs that store different sub-sets of the dependencies of X.

One possible solution to these challenges could be allowing more than one round of client-server communication to perform a single parallel read operation. Servers can return possibly inconsistent versions in the first round(s), and the client can detect and fix these violations by issuing additional read requests [69, 70, 76].

Another possible solution could be blocking a read on a partition until the partition knows that all other involved partitions are serving the read operations from the same causally consistent snapshot of the data store [5, 42, 4].

Clearly, these solutions increase the latency experienced by the transactions and reduce the achievable throughput, because they introduce waiting times or require extra communication.

5.3.2 Non-blocking reads in partial replication by PaRiS

PaRiS addresses these challenges by a combination of a novel dependency tracking protocol, called UST, and a client-side cache. UST identifies snapshots of the data store that can be read by transactions without blocking. These snapshots are such that they have been already installed by every DC, so they are slightly in the past. The client-side cache stores the versions written by the client that are not yet reflected in the snapshot determined by UST, allowing clients to observe monotonically increasing snapshots even if UST identifies slightly stale snapshots. We now explain how PaRiS leverages UST and the client-side cache in its transactional protocol.

Transactions in PaRiS. PaRiS identifies key versions and snapshots by means of a scalar timestamp. Upon starting, a transaction is assigned a snapshot timestamp st that, together with the content of the client-side cache, determines the snapshot visible to the transaction. Upon completing, each transaction that writes at least one key is assigned a commit timestamp that reflects causality, determined by means of a two-phase commit (2PC) protocol.

Non-blocking reads in PaRiS. The key idea in PaRiS is to identify a snapshot that has been installed by each DC. We define such snapshot as *stable*. A stable snapshot with timestamp ts contains all versions with a timestamp $\leq ts$, and indicates that *every* transaction T with a commit timestamp $\leq ts$ has been applied in *every* data center that stores a replica of the keys written by T.

Hence, a transaction can read from a stable snapshot without blocking or running multiple client-server rounds, regardless of the DC in which the individual reads of the transaction are performed.

A coordinator partition is responsible for assigning a stable snapshot to a transaction that starts. Any node can act as the coordinator of any transaction. The coordinator enforces the monotonicity of the snapshots assigned to transactions issued by the same client. To this end, the client piggybacks its last observed snapshot timestamp on the transaction start message to the coordinator.

UST. UST is the new protocol implemented by PaRiS to identify, in a scalable fashion, stable snapshots. Each partition maintains a version vector that indicates the timestamps of the latest applied transactions, both the ones executed by the partition itself and the ones received from remote replicas. Periodically, partitions within the same DC and across DCs exchange, by means of a gossiping protocol, the minimum of the timestamps in their version vectors. The aggregate minimum of the exchanged values identifies a timestamp such that all transactions with lower timestamps have been applied by all partitions in every DC.

UST identifies stable causally consistent snapshots with a single timestamp, regardless of the scale of the system. This enables high scalability and efficiency, by reducing partition-to-partition and client-to-partition communication overhead.

Cache. UST alone cannot enforce causality. In fact, the commit timestamp assigned to a transaction T issued by client c is higher than the stable snapshot assigned to T, which allows the commit timestamps to reflect causality. The commit timestamp of T may be higher than the snapshot assigned by the next transaction issued by c. In that case, such snapshot would not include the modifications performed by c in T, which may lead to violation of the read-your-own-writes property required by causal consistency.

PaRiS overcomes this issue by storing on the client the versions written by the client. Upon receiving a snapshot timestamp st, a client c removes from the cache all versions with timestamp $\leq st$. These versions are already included in the snapshots visible to any future transaction issued by c. When reading key x, c first checks its cache. If a version exists in the cache, that version has to be read by c to enforce the read-your-own-writes property. Else, c issues a read request to a replica. In both cases, the read completes without blocking.

Generating timestamps. As in recent proposals [96, 76, 90, 49], PaRiS uses Hybrid Logical Physical Clocks (HLC) [61] to generate timestamps. An HLC is a logical clock whose value on a partition is the maximum of the local physical clock and the highest timestamp seen by the partition plus one. Like logical clocks, HLCs can be moved forward to match the timestamp of an incoming event, without blocking to wait for the local physical clock to catch up with the timestamp of the incoming event. Like physical clocks, HLCs advance in the absence of events and at approximately the same pace. Hence, HLCs improve the freshness of the snapshot determined by UST over a solution that uses logical clocks, which can advance at very different rates on different partitions.

5.3.3 Fault tolerance

Failures (within a DC). PaRiS can tolerate failures of a server by integrating existing solutions for 2PC-based systems, e.g., based on Paxos [65]. Reads remain non-blocking with such mechanisms enabled, because they access a snapshot corresponding to transactions that have *already* been committed.

As in previous systems based on dependency aggregation protocols, the failure of a server blocks the progress of UST, but only as long as a backup has not taken over.

Client failures are transparent to the system. The clients only keep local meta-data and cache data that have already been committed to the data-store. The transaction contexts corresponding to failed clients are cleaned up after a timeout.

Availability (among DCs). PaRiS achieves availability in a DC as long as one replica per partition is reachable by the DC, so that remote items can be read, and the 2PC can be executed. In fact, remote reads can be performed without blocking by any replica, because the snapshot visible to a transaction is already installed in all DCs. In addition, local operations never block.

If a DC partitions from the rest of the system, that DC cannot serve any transaction that involves reading or writing a remote item. The remaining DCs remain available and clients can still issue transactions, but the UST freezes at all DCs, because it is computed as a system-wide minimum. As a result, transactions see increasingly stale snapshots of the data, and the client cache cannot be pruned.

5.4 Protocols of PaRiS

We now describe the meta-data stored and the protocols implemented by the clients and servers in PaRiS.

Symbol	Definition
N	Number of partitions
M	Number of replicas per partition
p_n^m	The $m-$th replica of the $n-$th partition
ust_c	Client's universal stable time
hwt_c	Client's highest write time
RS_c	Client's read set
WS_c	Client's write set
WC_c	Client-side cache
$Clock_n^m$	Physical clock time at p_n^m
HLC_n^m	Hybrid logical clock time at p_n^m
VV_n^m	Version vector of p_n^m
GSV_n^m	Global stable times vector of p_n^m
ust_n^m	Universal stable time of p_n^m
TX	Transaction context
$Prepared_n^m$	List of pending transactions of p_n^m
$Committed_n^m$	List of transactions of p_n^m, ready to be committed
d	A tuple $\langle k, v, ut, id_t, sr \rangle$
k	A key
v	A value
ut	Update time
id_t	Transaction ID
sr	Source replica id

Table 5.1 – Symbols used in PaRiS's protocol description and corresponding meaning.

5.4.1 Meta-data

Items. An item d is represented as the tuple $\langle k, v, ut, id_T, sr \rangle$. k and v are the key and value of d, respectively. ut is the timestamp of d which is assigned upon the commit of the transaction that created d and determines the snapshot to which d belongs. id_T is the id of the transaction that created the item version. sr is the id of the DC where the item is created.

Clients. In a client session, a client c maintains the highest stable snapshot timestamp known to c, denoted by ust_c, and the commit time of its last update transaction, noted hwt_c. The client also maintains a private cache WC_c, which stores items written by c for which c does not yet know whether they belong to the stable snapshot. Finally, the client maintains the meta-data and data of the transaction that is currently running: id_c, which is the unique identifier of the transaction, and WS_c and RS_c, which correspond to the transaction's write set and read set, respectively.

Server. A server p_n^m is identified by the partition id (n), and the DC id (m). p_n^m stores the replica id (r), where $r \leq R$, the replication factor of partition p_n^m.

Each server has access to a physical clock $Clock_n^m$, that generates timestamps in a monotonically increasing order. We assume that the clocks are loosely synchronized by using a time synchronization protocol, such as NTP [82]. The correctness of our protocol does not depend on the precision of the time synchronization. The local clock value on p_n^m is represented by the hybrid logical clock HLC_n^m. p_n^m also maintains two vector clocks VV_n^m and GSV_n^m, that represent vectors of HLCs. VV_n^m, has R entries, one for each replica of partition n. $VV_n^m[i], i \neq r$, indicates the timestamp of the latest update received by p_n^m that comes from the i-th replica of partition n. $VV_n^m[r]$ is the version clock of the server and represents the local snapshot installed by p_n^m. GSV_n^m, or Global Stabilization Vector, has M entries. $GSV_n^m[i] = t$ means that p_n^m is aware that all partitions in DC_m have installed all events generated in the DC_i with timestamp up to t. The server also stores the UST of p_n^m, denoted by ust_n^m. $ust_n^m = t$ indicates that p_n^m is aware that every partition in every DC has installed a snapshot with timestamp at least t.

Finally, following standard practice for systems that perform a 2PC protocol, p_n^m keeps two queues, one with prepared and one with committed transactions. The former, denoted by $Prepared_n^m$, stores transactions for which p_n^m has proposed a commit timestamp and for which p_n^m is waiting for the commit message. The latter, denoted by $Committed_n^m$ stores transactions that have been assigned a commit timestamp and whose modifications are going to be applied to p_n^m.

5.4.2 Operations

Algorithm 8 provides pseudo-code for the client protocol, Algorithm 9 and Algorithm 10 for the protocols executed by a server to run a transaction, for the cases in which the server is or is not the transaction coordinator, respectively, and Algorithm 11 for the replication and the UST protocols.

Start. Client c starts a transaction T by picking at random a coordinator partition (denoted p_n^m) in the local DC and sending it a start request with ust_c. p_n^m uses ust_c to update ust_n^m, so that p_n^m can assign to the new transaction a snapshot that is at least as fresh as the one accessed by c

Algorithm 8 PaRiS client c (open session towards p_n^m).

```
1: function START
2:     send ⟨StartTxReq $ust_c$⟩ to $p_n^m$
3:     receive ⟨StartTxResp $id, ust$⟩ from $p_n^m$
4:     $id_c \leftarrow id$; $ust_c \leftarrow ust$;
5:     $RS_c \leftarrow \emptyset; WS_c \leftarrow \emptyset$
6:     Remove from $WC_c$ all items with commit timestamp up to $ust_c$
7: end function

8: function READ($\chi$)
9:     $D \leftarrow \emptyset$    ▷ D represents the set of key-value pairs for the requested keys
10:    for each $k \in \chi$ do
11:        $v \leftarrow$ look up $k$ in $WS_c$, $RS_c$, $WC_c$ (in this order)    ▷ The value might already be available in the client
12:        if ($v \neq NULL$) then $D \leftarrow D \cup \langle k, v\rangle$
13:    end for
14:    $\chi' \leftarrow \chi \setminus keys(D)$
15:    send ⟨ReadReq $id_c$,⟩ $\chi'$ to $p_n^m$
16:    receive ⟨ReadResp $D'$⟩ from $p_n^m$
17:    $D \leftarrow D \cup D'$
18:    $RS_c \leftarrow RS_c \cup D$
19:    return $D$
20: end function

21: function WRITE($\chi$)
22:    for each $\langle k, v\rangle \in \chi$ do    ▷ Update $WS_c$ or write new entry
23:        if ($\exists d \in WS : d == k$)then $d.v \leftarrow v$ else $WS_c \leftarrow WS_c \cup \langle k, v\rangle$
24:    end for
25: end function

26: function COMMIT    ▷ Only invoked if $WS \neq \emptyset$
27:    send ⟨CommitReq $id_c, hwt_c, WS_c$⟩ to $p_n^m$
28:    receive ⟨CommitResp $ct$⟩ from $p_n^m$
29:    $hwt_c \leftarrow ct$    ▷ Update client's highest write time
30:    Tag $WS_c$ entries with $hwt_c$
31:    Move $WS_c$ entries to $WC_c$    ▷ Overwrite (older) duplicate entries
32: end function
```

in previous transactions. p_n^m uses its updated value of ust_n^m as snapshot for T. p_n^m also generates a unique identifier for T, denoted id_T, and inserts T in a private index of transactions TX. p_n^m replies to c with id_T and the snapshot timestamp ust_n^m.

Upon receiving the reply, c updates ust_c and evicts from the cache any version with timestamp lower than or equal to ust_c. c can prune such versions because the UST protocol ensures that they are included in the snapshot installed by any partition in the system. This means that if, after pruning, there is a version X in the private cache of c, then $X.ct > ust$ and hence the freshest version of x visible to c is X.

Read. For each key k to read, c searches the write-set, the read-set and the client cache, in this order. If an item corresponding to k is found, it is added to the set of items to return, ensuring read-your-own-writes and repeatable-reads semantics. Reads for keys that cannot be served locally are sent to the transaction coordinator p_n^m together with the transaction id. p_n^m retrieves

Algorithm 9 PaRiS server p_n^m - transaction coordinator.

```
1:  upon receive ⟨StartTxReq $ust_c$⟩ from $c$ do
2:      $ust_n^m \leftarrow max\{ust_n^m, ust_c\}$                    ▷ Update universal stable time
3:      $id_T \leftarrow generateUniqueId()$
4:      $TX[id_T] \leftarrow ust_n^m$                                   ▷ Save TX context
5:      send ⟨StartTxResp $id_T, TX[id_T]$⟩                        ▷ Assign transaction snapshot

6:  upon receive ⟨ReadReq $id_T$,$\chi$⟩ from $c$ do
7:      $ust \leftarrow TX[id_T]$
8:      $D \leftarrow \emptyset$
9:      $\chi_i \leftarrow \{k \in \chi : partition(k) == i\}$         ▷ Partitions with ≥ 1 key to read
10:     for ($i : \chi_i \neq \emptyset$) do                           ▷ Done in parallel
11:         $j = getTargetDCForPartition(i)$     ▷ Returns an id of a DC that replicates partition i
12:         send ⟨ReadSliceReq $\chi_i, ust$⟩ to $p_i^j$
13:         receive ⟨ReadSliceResp $D_i$⟩ from $p_i^j$
14:         $D \leftarrow D \cup D_i$
15:     end for
16:     send ⟨ReadResp $D$⟩ to $c$

17: upon receive ⟨CommitReq $id_T, hwt, WS$⟩ from $c$ do
18:     $\langle ust \rangle \leftarrow TX[id_T]$
19:     $ht \leftarrow max\{ust, hwt\}$                                 ▷ Max timestamp seen by the client
20:     $D_i \leftarrow \{\langle k, v \rangle \in WS : partition(k) == i\}$
21:     for ($i : D_i \neq \emptyset$) do                              ▷ Done in parallel
22:         $j = getTargetDCForPartition(i)$     ▷ Returns an id of a DC that replicates partition i
23:         send ⟨PrepareReq $id_T, ust, ht, D_i$⟩ to $p_i^j$
24:         receive ⟨PrepareResp $id_T, pt_i$⟩ from $p_i^j$
25:     end for
26:     ct $\leftarrow max_{i:D_i \neq \emptyset}\{pt_i\}$              ▷ Max proposed timestamp
27:     for ($i : D_i \neq \emptyset$) do                              ▷ Done in parallel
28:         $j = getTargetDCForPartition(i)$     ▷ Returns an id of a DC that replicates partition i
29:         send ⟨CommitReq $id_T, ct$⟩ to $p_i^j$
30:     end for
31:     delete TX[$id_T$]                                               ▷ Clear transactional context of c
32:     send ⟨CommitResp $ct$⟩ to $c$
```

the snapshot corresponding to the transaction, and sends to each involved partition the set of keys to be read, in parallel. Because each DC only stores a subset of the full data set, some of the contacted partitions may belong to a remote DC that replicates the partitions where the keys belong. Remote DCs can be chosen depending on geographical proximity or on some load balancing scheme.

Upon receiving a read request, regardless of whether it originates from the local DC or from a remote one, p_n^m first updates its ust_n^m, if it is smaller than the transaction's snapshot (Alg. 10 Line 2). Next, the server returns, for each key, the version with the highest timestamp below the snapshot timestamp (Alg. 10 Lines 4–7). As we shall see shortly, the commit protocol of PaRiS allows concurrent updates on the same key, both within a DC and in different DCs. This is typically the case in TCC systems to avoid costly validation protocols for update transactions [4, 96]. In case multiple versions of a key are assigned the same timestamp, PaRiS totally orders versions by a concatenation of timestamp, transaction id and source data center id, in this order.

Algorithm 10 PaRiS server p_n^m - transaction cohort.

```
 1: upon receive ⟨ReadSliceReq $\chi, ust$⟩ from $p_i^j$ do
 2:     $ust_n^m \leftarrow max\{ust_n^m, ust\}$                ▷ Update universal stable time
 3:     $D \leftarrow \emptyset$
 4:     for ($k \in \chi$) do
 5:         $D_{sv} \leftarrow \{d : d.k == k \wedge d.ut \leq ust\}$        ▷ Universally visible
 6:         $D \leftarrow D \cup \{argmax_{d.ut}\{d \in D_{sv}\}\}$      ▷ Freshest visible vers. of k
 7:     end for
 8:     send ⟨SliceResp $D$⟩ to $p_i^j$

 9: upon receive ⟨PrepareReq $id_T, ust, ht, D_i$⟩ do from $p_i^j$
10:     $HLC_n^m \leftarrow max(Clock_n^m, ht+1, HLC_n^m+1)$         ▷ Update HLC
11:     $ust_n^m \leftarrow max\{ust_n^m, ust\}$                ▷ Update universal stable time
12:     pt $\leftarrow max\{HLC_n^m, ust_n^m\}$                  ▷ Proposed commit time
13:     $Prepared_n^m \leftarrow Prepared_n^m \cup \{id_T, pt, D_i\}$     ▷ Append to pending list
14:     send ⟨PrepareResp $id_T, pt$⟩ to $p_i^j$

15: upon receive ⟨CommitReq $id_T, ct$⟩ do from $p_i^j$
16:     $HLC_n^m \leftarrow max(HLC_n^m, ct, Clock_n^m)$           ▷ Update HLC
17:     $\langle id_T, pt, D\rangle \leftarrow \{\langle i, r, \phi\rangle \in Prepared_n^m : i == id_T\}$
18:     $Prepared_n^m \leftarrow Prepared_n^m \setminus \{\langle id_T, pt, D\rangle\}$      ▷ Remove from pending
19:     $Committed_n^m \leftarrow Committed_n^m \cup \{\langle id_T, ct, D\rangle\}$        ▷ Mark to commit
```

Once p_n^m has received the reply from all the partitions contacted, p_n^m sends the items to the client, which inserts them in its read-set.

Write. Client c locally buffers the writes in its write-set WS_c. If a key being written is already present in WS_c, then it is updated; otherwise, it is inserted in WS_c.

Commit. To finalize the transaction, the client sends a commit request to the coordinator with the content of WS_c, the id of the transaction and the commit timestamp of its last update transaction hwt_c, if any. The commit protocol of PaRiS is based on the two-phase commit (2PC) protocol. The coordinator contacts the partitions that store the keys that need to be updated and sends them the corresponding updates and hwt_c. Some of the contacted partitions may belong to a remote DC. Each partition involved first updates its clock to be at least as high as the maximum of the transaction's snapshot timestamp and hwt_c. Then, each partition increases its clock and sends the updated clock value to the coordinator as a commit timestamp. The proposed timestamp reflects causality because it is higher than both the snapshot timestamp and hwt_c. Each partition also inserts the transaction id, the set of keys to be modified on the partition and the proposed timestamp in the queue of pending transactions.

The coordinator picks the maximum among the proposed timestamps, sends it to the partitions involved in the transaction, removes the transaction from the private index of transactions TX and sends the commit timestamp to the client. Upon receiving the commit message, a partition increases its clock to match the commit time, if needed, and moves the transaction from the

Algorithm 11 PaRiS server p_n^m - auxiliary functions.

```
1:  function UPDATE($k, v, ut, id_T$)
2:      create $d : \langle d.k, d.v, d.ut, id_T, d.sr\rangle \leftarrow \langle k, v, ut, id_T, m\rangle$
3:      insert new item $d$ in the version chain of key $k$
4:  end function

5:  upon Every $\Delta_R$ do
6:      if ($Prepared_n^m \neq \emptyset$) then $ub \leftarrow min_{\{p.pt\}}\{p \in Prepared_n^m\} - 1$
7:      else $ub \leftarrow max\{Clock_n^m, HLC_n^m\}$ end if
8:      $\rho_n \leftarrow Replicas(n)$
9:      if ($Committed_n^m \neq \emptyset$) then                         ▷ Commit tx by increasing order of ct
10:         $C \leftarrow \{\langle id, ct, D\rangle\} \in Committed_n^m : ct < ub$
11:         for ($T \leftarrow \{\langle id, D\rangle\} \in (group\ C\ by\ ct)$) do
12:             for ($\langle id, D\rangle \in T$) do
13:                 for ($\langle k, v\rangle \in D$) do update ($k, v, ct, id$) end for
14:             end for
15:             for ($j \in \rho_n \wedge j \neq r$) do send ⟨Replicate $T, ct$⟩ to $p_n^j$ end for
16:             $Committed_n^m \leftarrow Committed_n^m \setminus T$
17:         end for
18:         $VV_n^m[m] \leftarrow ub$                                                  ▷ Set version clock
19:     else
20:         $VV_n^m[m] \leftarrow ub$                                                  ▷ Set version clock
21:         for ($j \in \rho_n \wedge j \neq r$) do send ⟨Heartbeat $VV_n^m[m]$⟩ to $p_n^j$ end for   ▷ Send heartbeat to replicas
22:     end if

23: upon receive ⟨Replicate $T, ct$⟩ from $p_n^j$ do
24:     for ($\langle id, D\rangle \in T$) do
25:         for ($\langle k, v\rangle \in D$) do
26:             update ($k, v, ct, id$)
27:         end for
28:     end for
29:     $i \leftarrow GetReplicaIdForDC(j)$
30:     $VV_n^m[i] \leftarrow ct$                    ▷ Update version clock of i-th replica of n-th partition

31: upon receive ⟨Heartbeat t⟩ from $p_n^j$ do
32:     $i \leftarrow GetReplicaIdForDC(j)$
33:     $VV_n^m[i] \leftarrow t$                     ▷ Update version clock of i-th replica in n-th partition

34: upon every $\Delta_G$ time do                                   ▷ Gather global stable times from other DCs
35:     $GSV_n^m[j] \leftarrow min\{VV_i^m[j]\}, \forall j = 0 \ldots M-1, \forall i = 0 \ldots N-1$

36: upon every $\Delta_U$ time do                                          ▷ Compute universal stable time
37:     $minGST \leftarrow min\{GSV_i^m[j]\}, \forall j = 0 \ldots M-1, \forall i = 0 \ldots N-1$
38:     $ust_n^m \leftarrow max\{minGST, ust_n^m\}$                              ▷ Enforce monotonicity of $ust_n^m$
```

pending queue to the commit queue, with the new commit timestamp.

Applying and replicating transactions. Periodically, the effects of transactions committed by p_n^m are applied on p_n^m, in increasing commit timestamp order (Alg. 11 Lines 6-21). p_n^m applies the modifications of transactions that have a commit timestamp strictly lower than the lowest timestamp present in the pending list. This timestamp represents the lower bound on the commit timestamps of future transactions on p_n^m. After applying the transactions, p_n^m updates its local version clock and replicates the update operations in the applied transactions to its remote replicas.

If p_n^m does not commit a transaction for a given amount of time, p_n^m updates its local clock, and sends it to its peer replicas by means of a heartbeat message. This ensures the progress of the UST also in the absence of updates.

Stabilization protocol. Every Δ_G time units, partitions within a data center exchange the minimum of their version vectors to compute the global stable time (GST) of the local data center. Similarly to previous work [42, 4], PaRiS organizes nodes within a DC as a tree to reduce message exchanges. The GST is progressively aggregated from the leaves to the root, and then propagated from the root to all the nodes in the DC. Next, all the roots from each DC exchange their GST values.

Every Δ_U time units, the roots compute the ust_n^m as the aggregate minimum of the received GSTs and propagate it to all the other nodes in the DC.

Garbage collection. Periodically the partitions exchange the oldest snapshot corresponding to an active transaction (p_n^m sends its current stable snapshot timestamp if it has no running transactions). The aggregate minimum determines the oldest snapshot S_{old} that is visible to a running transaction. The partitions scan the version chain of each key backwards and remove all versions older than S_{old}. The same protocol that computes the UST also computes S_{old}.

5.4.3 Correctness

We now provide an informal proof sketch that PaRiS provides causal consistency by showing that i) reads observe a causally consistent snapshot and ii) writes are atomic.

Lemma 1. *The snapshot time sn_T of a transaction T is always lower than the commit time of T, $sn_T < T.ct$.*

Proof. Let T be a transaction with snapshot time sn_T and commit time ct. The snapshot time is determined during the start of the transaction (Alg. 9 Line 4). The commit time is calculated in the commit phase of the 2PC protocol, as maximum value of the proposed prepare times of all partitions participants in T (Alg. 9 Line 26). In order to reflect causality when proposing a prepare timestamp, each partition proposes higher timestamp than the snapshot timestamp (Alg. 10 Line 12). Thus, the commit time of a transaction, ct, is always greater than the snapshot time, sn_T. ■

Proposition 5. *If an update u_2 causally depends on an update u_1, $u_1 \rightsquigarrow u_2$, then $u_1.ut < u_2.ut$.*

Proof. Let c be the client that wrote u_2. There are three cases upon which u_2 can depend on u_1, described in Section 2.4.2: *1*) c committed u_1 in a previous transaction; *2*) c has read u_1, written in a previous transaction and *3*) c has read u_3, and there exists a chain of direct dependencies that lead from u_1 to u_3, i.e. $u_1 \rightsquigarrow ... \rightsquigarrow u_3$ and $u_3 \rightsquigarrow u_2$.

Case 1. When a client commits a transaction, it piggybacks the last update transaction commit time hwt_c, if any, to its commit request for the transaction coordinator (Alg. 8 Line 27) which is, furthermore, piggybacked as ht in its prepare requests to the involved partitions (Alg. 9 Line 23). To reflect causality when proposing a commit timestamp, each partition proposes higher timestamp than both ht and the snapshot timestamp (Alg. 10 Lines 10–14). The coordinator of the transaction chooses the maximum value from all proposed times from the participating partitions (Alg. 9 Line 26) to serve as commit time, ct, for all the updated items in the transaction. The new version of the data item is written in the key-value store with ct as its update time, ut (Alg. 11 Lines 2 and 13). When c commits the transaction that updates u_2, it piggybacks the commit time of the transaction that updated u_1. Hence, from the discussion above it follows that $u_1.ct < u_2.ct$. Because the commit time of a transaction is the update time of all the data item versions updated in the transaction (Alg. 11 Lines 2 and 13), we have $u_1.ut < u_2.ut$.

Case 2: c could have read u_1 either from c's client cache or from the transaction's causally consistent snapshot sn_T.

If c has read u_1 from c's client cache, then c has written u_1 either in a previous transaction in the same thread of execution or in the current one. If c wrote u_1 in the same transaction where u_2 is also written, then it is not possible to have $u_1 \rightsquigarrow u_2$ because all the updates from that transaction will be given the same commit, i.e. update timestamp, indicating that $u_1.ut = u_2.ut$. Thus, u_1 must be written in a previous transaction and from *Case 1* it follows that $u_1.ut < u_2.ut$.

Next, we will consider the case when c read u_1 from the causally consistent snapshot sn_T that contains u_1. When a transaction T is started, the snapshot sn_T is determined by Alg. 9 Line 2, $sn_T = max\{ust_c, ust_n^m\}$. From Alg. 10 Line 5 we have that $u_1.ut \leq ust = sn_T$. From Lemma 1 it follows that $u_1.ut \leq sn_t < u_2.ct = u_2.ut$. Therefore, $u_1.ut < u_2.ut$.

Case 3: If u_2 depends on u_1 because of a transitive dependency out of $c's$ thread-of-execution, it means that there exists a chain of direct dependencies that lead from u_1 to u_2, i.e., $u_1 \rightsquigarrow ... \rightsquigarrow u_3$ and $u_3 \rightsquigarrow u_2$. Each pair in the transitive-chain, belongs to either *Case 1* or *Case 2*. Hence, the proof of *Case 3* comes down to chained application of the correctness arguments from *Case 1* and *Case 2*, proving that each element has an update time lower than its successor's. ■

Proposition 6. *A partition vector clock $VV_n^m[i] = t$ implies that p_n^m has received all updates from $i - th$ replica with commit time, $ct \leq t$.*

Proof. We need to to prove that Proposition 6 holds for both local and remote updates. To show the former, we demonstrate that there are no pending local updates with commit timestamp $ct \leq t$. When p_n^m updates the local replica vector clock entry $VV_n^m[r]$, it finds the minimum prepare timestamp of all transactions that are currently in the prepare phase (Alg 11. Line 6). Because the commit time is calculated as the maximum of all prepare times (Alg. 9 Line 26) and the HLC clock is monotonic (Alg. 10 Lines 10 and 16), all future transactions are guaranteed to have a commit time that is not lower than the minimum prepare timestamp. Thus, when $VV_n^m[r]$ is assigned the minimum of the prepare times of all transactions in the prepare queue minus 1 (Alg. 11 Line 6), p_n^m has already received all updates for the snapshot $VV_n^m[i] = t$.

To show the latter, we use proof by contradiction. Let's assume there is a remote update u from $i - th$ replica such that $u.ct < t$, and p_n^m has not received u. By Alg. 11 Line 30, the partition would have received an update u' such that $u'.ct = t$. The updates are sent in the order of their commit timestamps (Alg. 11 Lines 9–16). Hence, if $u'.ct > u.ct$ the p_n^m could not have received another update u' before u. Therefore, $u.ct > t$, implying that $u.ct \not< t$, leading to the contradiction. ■

Proposition 7. *Snapshots in PaRiS are causal.*

Proof. To start a transaction, a client c piggybacks the freshest snapshot it has seen, ensuring the monotonicity of the snapshot seen by c (Alg. 9 Line 2). Commit timestamps reflect causality (Alg. 9 Line 26), and UST tracks a lower bound on the snapshot installed by every partition in all DCs (Alg. 11 Lines 36-38). If X is within the snapshot of a transaction, so are its dependencies (Proposition 5). On top of the snapshot provided by the coordinator, c also can read the writes, that are not yet included in the snapshot, from the cache. These writes cannot depend on items created by other clients that are outside the snapshot visible to c. ■

Proposition 8. *Writes are atomic.*

Proof. Although updates are made visible independently on each partition p_n^m involved in the commit phase, either all updates are made visible or none of them are. All updates from a transaction belong to the same snapshot, because they all receive the same commit timestamps (Alg. 9 Line 29). The updates are being installed in the order of their commit timestamps (Alg. 11 Lines 9–16). The visibility of the item versions is determined by the transaction snapshot (Alg. 10 Line 5), which is based on the value of p_n^m's universal stable time ust_n^m. ust_n^m is computed by the UST protocol as the aggregate minimum of the version vectors entries of all partitions of all data centers (Alg. 11 Lines 34–38). ■

PaRiS implements TCC, as every transaction reads from a causally consistent snapshot (Proposition 7) that includes all effects (Proposition 8) of its causally dependent transactions.

5.5 Evaluation

Competitor systems. To assess the advantages of non-blocking reads, we compare PaRiS against a blocking protocol, which we call Blocking Partial Replication, or BPR, inspired by [43]. In BPR, the snapshot of a transaction T of client c is determined as the maximum of the highest causally consistent snapshot seen by c and the clock value of the transaction coordinator. BPR uses one timestamp to encode transactional snapshots, so we can compare fairly the resource efficiency of PaRiS versus the one of BPR. BPR also favors the freshness of the snapshots that are visible to transactions. BPR, however, implies having blocking transactional reads, because the server must ensure that the returned version belongs to a causally consistent snapshot. To this end, a partition blocks a read operation with snapshot timestamp t until it has applied all local and remote transactions with timestamp up to t. By being blocking and choosing the freshest possible snapshot, BPR does not need a stabilization protocol.

We do not compare to the second alternative possible design, discussed in Section 5.3, because to our best knowledge, a two-round protocol designed for a multi-master TCC partially replicated system currently does not exist. The two existing two-round protocols, [70] [76], are both designed for a fully replicated system, and it is very difficult to adapt them to partial replication. The replication scheme of [70] is based on dependency checking, and [76] has a master-slave system model which, additionally, can abort even read-only transactions. None of the systems supports partial replication and general-purpose read-write transactions.

To measure the overhead of implementing causal consistency in PaRiS, we additionally compare PaRiS to an implementation that does not guarantee causal consistency, which we refer to as NoCC. NoCC is implemented in the same codebase as PaRiS, with several key differences: *i)* it does not read from a causally consistent snapshot; *ii)* it does not have the start phase of transactions (one round trip less than PaRiS); *iii)* it does not traverse an item version chain because it always returns the latest received item version; *iv)* it does not run a stabilization protocol and *v)* for a fair comparison with PaRiS, NoCC provides transaction atomicity by a simplified variation of the 2PC in the commit phase where there are no adjustments made to the HLC clock (each participant in the prepare phase just proposes its clock value + 1).

Comparison with fully replicated systems is beyond the scope of this dissertation, but the trade-offs are rather intuitive. To achieve non-blocking reads, PaRiS uses a snapshot that is installed in every DC, complemented with the client-side cache. A fully replicated system, besides the client-side cache, only needs a snapshot installed in all partitions in the local DC to avoid blocking. A fully replicated system has lower latency on reads and writes, because they are local, and lower visibility latency, because it needs to track only the lower bound on item versions installed in the local DC. However, a partial replication system reduces the replication costs and storage requirements.

	OR	NV	IR	MU	SY
Oregon		85.72	144.52	238.11	157.13
N. Virginia	88.28		80.4	190.13	207.98
Ireland	139.32	76.47		137.53	279.75
Mumbai	253.07	190.42	132.74		268.66
Sydney	148.69	224	291.34	307.53	

Table 5.2 – Round-trip latencies in ms between AWS DCs used for the default experimental configuration.

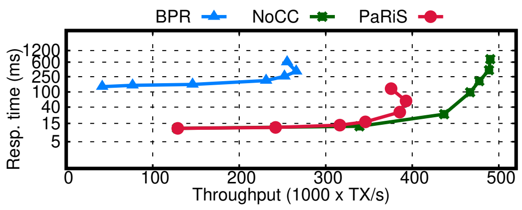
5.5.1 Experimental environment

Platform. We consider a geo-replicated setting deployed across up to 10 replication sites on Amazon EC2 (North Virginia, Oregon, Ireland, Mumbai, Sydney, Canada, Seul, Frankfurt, Singapore and Ohio). When using 3 DCs, we use North Virginia, Oregon and Ireland. When using 5 DCs, we use the previously mentioned 3 DCs plus Mumbai and Sydney and the round-trip latencies among them are shown on Figure 5.2. In each DC we use up to 18 servers (c5.xlarge instances with 4 VCPUs and 8 GB of RAM). The replication factor is 2. We choose this value because it allows us to use 3 as minimum number of DCs in our experiment and use partial replication. We use hash-based partitioning scheme, where each key is deterministically assigned to one partition by a hash function.

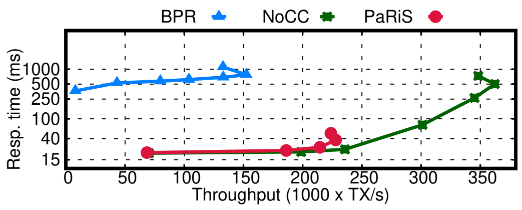
We spawn one client process per partition in each DC. Clients are co-located with the server partition they use as a transaction coordinator. The clients issue requests in a closed loop. To generate different load conditions, we spawn different number of threads per client process. Depending on the type of the workload or the protocol, a different number of threads is needed to saturate the target system. Increasing the number of threads past that point leads the systems to deliver lower throughput despite serving a higher number of client threads. Each “dot" in the curve plots we report corresponds to a different number of active threads per client process.

Implementation. We implement PaRiS, BPR and NoCC in the same C++ codebase. All the protocols implement the last-writer-wins rule for convergence. We use Google Protobufs for communication, and NTP to synchronize physical clocks. For PaRiS, the stabilization protocols run every 5 milliseconds.

Workloads. We use workloads with 50:50 and 95:5 r:w ratios that correspond to the write-heavy (A) and read-heavy (B) profiles of YCSB workloads [32]. These are standard workloads also used to benchmark other TCC systems [4, 76, 114, 96]. To accommodate the transactional nature of PaRiS, we extend YCSB by wrapping the reads and writes into a transaction, with two additional operations for the start and for the commit of a transaction. Transactions generate both workloads



(a) Throughput vs average TX latency (95:5 r:w ratio).



(b) Throughput vs average TX latency (50:50 r:w ratio).

Figure 5.1 – Performance of PaRiS, BPR and NoCC (y-axis in logarithmic scale) with 95:5 (a) and 50:50 (b) r:w ratios, 4 partitions involved per transaction (5 DCs, 45 partitions, replication factor is 2, 18 machines per DC). PaRiS outperforms BPR for both read-heavy and write-heavy workloads. As long as the systems are not overloaded, the throughput of PaRiS is almost the same as NoCC, whereas NoCC achieves higher max throughput under high load. The last points in each line represent overload conditions, that lead throughput to decrease and latency to increase.

by executing 19 reads and 1 write (95:5), and 10 reads and 10 writes (50:50). Hence, in each workload a transaction executes 20 operations per transaction, excluding the start and commit operation. A transaction first executes all the reads in parallel, and then all the writes in parallel.

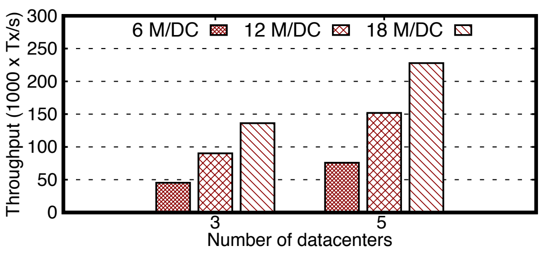
A transaction can target only partitions in the local DC, or can touch random partitions in remote DCs. In the first case, we say that a transaction is “local-DC"; else, we say it is “multi-DC". When accessing a remote partition, a client can choose among two replicas. We assign to every client in a DC the same preferred remote replica for each partition. We vary the preferred replica in the DCs using a round-robin assignment, to balance the load. To evaluate the effect of the partial replication, we use workloads with 100:0, 95:5, 90:10 and 50:50 local-DC:multi-DC ratios.

The default workload we consider uses a 95:5 r:w ratio, a 95:5 local-DC:multi-DC ratio and runs transactions that involve 4 partitions on a platform deployed over 90 machines spread over 5 DCs. The default deployment has 45 partitions that are replicated with replication factor 2. Hence, each DC has a total of 18 machines.

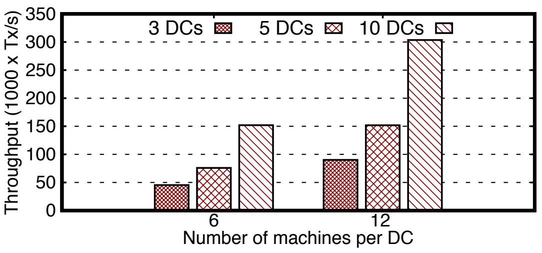
We also consider variations of this workload in which we change the value of one parameter and keep the others at their default values. Transactions access keys within a partition according to a zipfian distribution, with parameter 0.99, which is the default in YCSB and resembles the strong skew that characterizes many production systems [9, 78, 17]. We use small items (8 bytes), which are prevalent in many production workloads [9, 78].

5.5.2 Latency and throughput

Blocking vs. non-blocking. Figures 5.1a and 5.1b report the average transaction latency vs.



(a) Throughput when varying the number of machines per DC.



(b) Throughput when varying the number of DCs.

Figure 5.2 – Throughput achieved by PaRiS when increasing the number of machines per DC (a) and DCs (b). PaRiS achieves good scalability both when increasing the number of machines per DCs and DCs.

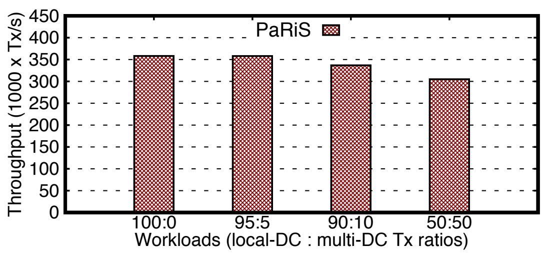
throughput achieved by PaRiS and BPR with the 95:5 (the default) and with the 50:50 r:w ratios. In the read-heavy case, PaRiS achieves up to 5.91x lower response times and up to 1.47x higher throughput than BPR. PaRiS also achieves up to 20.56x lower response times and up to 1.46x higher throughput than BPR for the write-heavy workload. PaRiS achieves lower latencies because it never has to wait for a snapshot to be installed. PaRiS achieves higher throughput because it does not incur any overhead to block/unblock read requests. Because BPR is a blocking protocol, it needs a higher number of concurrent client threads to fully utilize the processing power left idle by blocked reads, creating more contention on the physical resources and more synchronization overhead to block and unblock reads, which ultimately leads to lower throughput.

Blocking time. The average blocking time of the read phase of a transaction in BPR is 29 ms for the top throughput in the read-dominated workload (Figure 5.1a) and 41 ms for the top throughput in the write-heavy workload (Figure 5.1b).

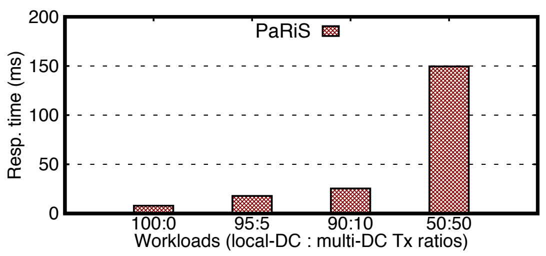
Causal vs. non-causal. Figures 5.1a and 5.1b show that PaRiS achieves the same latency as NoCC for both read-heavy and write-heavy workloads, as long the systems are not overloaded. NoCC achieves up to 1.24x higher maximum throughput for the read-heavy workload and up to 1.59x higher maximum throughput for the write-heavy workload. This performance difference is due to the overhead incurred by PaRiS to implement causal consistency.

5.5.3 Scalability

Varying the number of machines per DCs. Figure 5.2a reports the throughput achieved by PaRiS when using 6, 12 and 18 machines/DC. We consider two geo-replicated deployments that use 3 and 5 DCs. In both cases, PaRiS achieves the ideal improvement of 3x when scaling from 6



(a) Throughput when varying the locality of the transactions.



(b) Latency when varying the locality of the transactions.

Figure 5.3 – Throughput (a) and latency (b) achieved by PaRiS when varying the locality of the transactions with 100:0, 95:5, 90:10 and 50:50 local-DC:multi-DC ratio for the default workload.

to 18 machines/DC. This result showcases the ability of PaRiS to scale horizontally regardless of the number of DCs on which it is deployed.

Varying the number DCs. Figure 5.2b reports the throughput achieved by PaRiS when deployed on 3, 5 and 10 DCs. We consider two cases corresponding to 6 and 12 machines/DC. In both cases PaRiS achieves the ideal improvement of 3.33x, when scaling from 3 to 10 DCs. This result shows that PaRiS scales well to higher numbers of DCs for different sizes of the platform within each DC.

5.5.4 Varying data access locality

Figure 5.3a reports the maximum throughput achieved by PaRiS when varying the locality ratio (local-DC:multi-DC) of transactions from 100:0 to 50:50, for the default workload. Figure 5.3b shows the average transaction latency corresponding to the throughput values reported in Figure 5.3a. Performance deteriorates as the percentage of local accesses decreases. The maximum achievable throughput drops slightly, from 350 to 300 KTx/sec. As expected, latency is more heavily penalized, increasing from 8 to 150 ms. The number of threads needed to saturate the system increases as the locality decreases (from 32 to 512 in this case), because requests spend more time traveling between DCs, explaining the small drop in maximum throughput of only 16% compared to the order-of-magnitude increase in latency.

As any partially replicated system, PaRiS targets workloads with high locality in the data access pattern. In case of limited locality, the performance penalty incurred by PaRiS, and partial replication in general, is the inevitable price to pay to enable higher storage capacity.

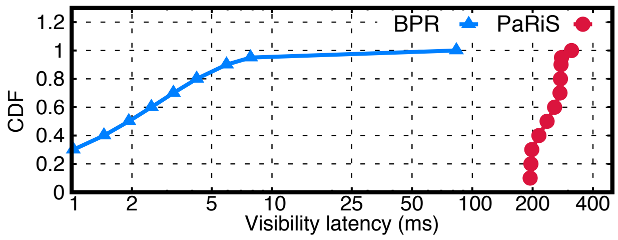


Figure 5.4 – PaRiS has higher update visibility latency than BPR (logarithmic scale), for the default workload.

5.5.5 Data staleness

We measure the staleness of the data returned by PaRiS by measuring the *visibility latency* of updates. The *visibility latency* of an update X in DC_i is the difference between the wall-clock time when X becomes visible in DC_i and the wall-clock time when X was received in DC_i. We factor out the latency of the update propagation from the DC where the item was created to DC_i, because this communication delay is always present in any replication protocol.

Figure 5.4 shows the CDF of the update visibility latency achieved by PaRiS and BPR with 5 DCs and the default workload. Each point in the CDF is computed as the average across the corresponding values on each node.

The update visibility time in PaRiS is higher than in BPR. That is to be expected because UST identifies a lower bound of the update time of transactions applied in the whole cluster. The worst-case difference between the visibility latency in PaRiS and BPR is around 200 ms. This aligns with the communication latency between the two farthest DCs, Sydney and Mumbai, as shown on Figure 5.2. This latency is incurred by these two DCs to exchange their knowledge about versions that are known to have been installed in other DCs. BPR trades data freshness for performance, because it exposes more recent snapshots of the data at the cost of blocking reads, therefore achieving much lower performance than PaRiS.

5.5.6 Resource efficiency

Figure 5.5 shows the percentage of the total data exchanged in PaRiS to run UST and to replicate updates, when varying the locality ratio (local-DC:multi-DC) of transactions from 100:0 to 50:50 for the default workload. For the stabilization protocol, we measured the total amount of data exchanged within each DC (shown as INTRA-DC) and between DCs (shown as INTER-DC). As shown on Figure 5.5, the major part of the total amount of exchanged data (64% - 72%) is sent for the purpose of replication, 26% -34% for the intra-DC part of the UST protocol and, finally, only 2% for the inter-DC part of the UST protocol. UST's stabilization messages carry

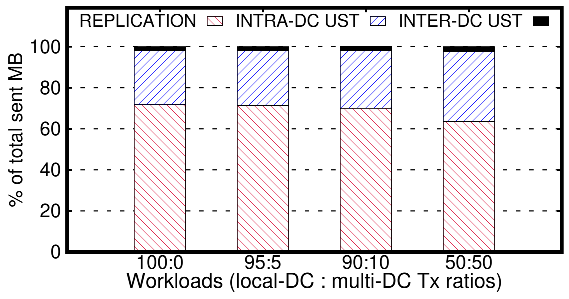


Figure 5.5 – The overhead of UST compared to replication.

only a single timestamp, regardless of the number of partitions and DCs in the system, which contributes to the high resource efficiency in PaRiS.

5.6 Conclusion

We present PaRiS, the first system that implements TCC in a partially replicated system and achieves non-blocking read operations. PaRiS implements a novel dependency tracking protocol, called UST, which requires only one timestamp to track dependencies. UST identifies a snapshot of the data that is available at *every* DC, thereby enabling non-blocking reads regardless of the DC in which the read takes place.

We evaluate PaRiS on a data platform replicated on up to 10 DCs. PaRiS scales well and achieves lower latency than the blocking alternative, while being able to handle larger datasets than existing solutions that assume full replication.

6 Related Work

This chapter presents selected papers that are closely related to the work presented in this dissertation. We provide a short analysis of the main technique of how they implement causal consistency, the system model that they are designed for, how they track dependencies, whether they implement nonblocking reads, etc.

Our research is also related to strongly consistent and eventually consistent systems. Those systems, however, have different design goals and are envisioned to satisfy the needs of different types of applications. Therefore, we consider strongly and eventually consistent systems out of the scope of this chapter, as our focus is solely oriented towards causally consistent systems.

We group related causally consistent systems into five categories according to whether they *i*) are designed for non-partitioned system model, *ii*) are designed for partitioned system model, *iii*) implement transactional causal consistency, *iv*) implement optimistic causal consistency, *v*) implement partial replication. Table 6.1 shows a taxonomy of the selected CC systems, covered in this chapter. Figure 6.1 shows a graphical distribution of the selected CC systems, compared form a point of view of two aspects: the transactional semantics that they achieve, and the metadata size. Finally, we briefly discuss some other aspects related to our research.

System	Part.	TXs	Nonbl. reads	Partial repl.	Meta-data	Technique	OCC
ISIS [20]	✗	ROT/WOT	✗	✗	O(M)	sequencer	✗
Lazy Rep [63]	✗	✗	✗	✗	O(M)	sequencer	✗
Bayou [99]	✗	✗	✓	✗	O(M)	sequencer	✗
PRACTI [18]	✗	✗	✗	✓	O(1)	sequencer	✗
TACT [112, 113]	✗	ROT/WOT	✗	✗	O(1)	sequencer	✗
SwiftCloud [114]	✗	Generic	✓	✗	O(M)	sequencer	✗
COPS [69]	✓	ROT	✓	✗	O(|deps|)	explicit check	✗
Eiger [70]	✓	ROT/WOT	✓	✗	O(|deps|)	explicit check	✗
ChainReaction [5]	✓	ROT	✗	✗	O(M)	sequencer	✗
Bolt-on CC [12]	✓	✗	✓	✗	O(|deps|)	explicit check	✗
Orbe [43]	✓	ROT	✗	✗	O(M x N)	explicit check	✗
GentleRain [42]	✓	ROT	✗	✗	O(1)	stabilization	✗
CausalSpartan [90]	✓	✗	✓	✗	O(M)	stabilization	✗
COPS-SNOW [72]	✓	ROT	✓	✗	O(|deps|)	explicit check	✗
Contrarian [38]	✓	ROT	✓	✗	O(M)	stabilization	✗
Cure [4]	✓	Generic	✗	✗	O(M)	stabilization	✗
AV [105]	✓	Generic	✓	✗	O(M)	stabilization	✗
OCCULT [76]	✓	Generic	✗	✗	O(N)	lazy resolution	✓
Saturn [22]	✓	✗	✓	✓	O(1)	sequencer	✗
C3 [46]	✓	✗	✓	✓	O(M)	stabilization	✗
Karma [74]	✓	ROT	✓	✓	O(|deps|)	explicit check	✗
Xiang, Vaidya [111]	✓	✗	✗	✓	O(1)	stabilization	✗
Opt-Track-CPR [55]	✗	✗	✓	✗	O(M)	sequencer	✗
Full/Opt-Track [55]	✗	✗	✓	✓	O(M)	sequencer	✗
Mongo [106]	✓	✗	✗	✗	O(1)	stabilization	✗
Kronos [45]	✓	-	-	✓	O(graph)	sequencer	✗
EunomiaKV [49]	✓	✗	✓	✗	O(M)	sequencer	✗
POCC [94]	✓	ROT	✗	✗	O(M)	lazy resolution	✓
WREN [96]	✓	Generic	✓	✗	O(2)	stabilization	✗
PaRiS [95]	✓	Generic	✓	✓	O(1)	stabilization	✗

Table 6.1 – Taxonomy of the main CC systems covered in this chapter. *M* is the number of DCs, and *N* is the number of partitions. *Part.* designates whether the system supports partitioning or not. For systems that do not support transactions, the *non-blocking read* property refers to single-item reads. *ROT* stands for read-only transaction and *WOT* stands for write-only transaction. *Generic* stands for interactive read-write transactions.

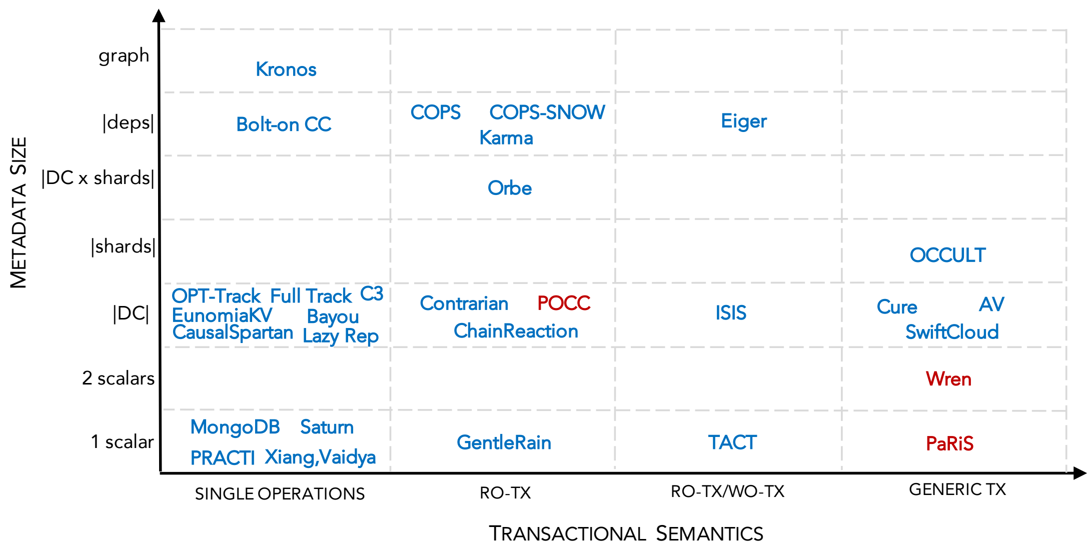


Figure 6.1 – Graphical distribution of the selected CC systems based on two aspects: the transactional semantics that they provide and the metadata size. The systems presented in this disertation are marked in red, and other systems are in blue.

6.1 Non-partitioned causally consistent systems

The concept of potential causality, or the "happens-before" relation, was fist defined by Lamport in his seminal paper [66] in 1978. In 1995, Ahamad et all. introduced the concept of causal memory [2] based on Lamport's potential causality. Several earlier systems in previously published papers, such as the ISIS toolkit [20], Lazy Replication [63] and Bayou [99], provided causal consistency guarantees as well. They assume single-machine replicas and do not consider partitioned datasets. These systems are considered as the first breed of causally consistent systems and they deeply influenced and inspired further causal consistency research.

Occult is a recently developed system that is affiliated with this category. However, it differs from the previously mentioned systems, since it belongs into the next generation of CC systems, both from a timeline perspective, as well as from its design goals.

The presented systems are not scalable as they all are designed for single-machine replicas. POCC, Wren and PaRiS, instead, are designed for applications that scale out by sharding and enforce causal consistency in a scalable way by utilizing lightweight protocols.

The ISIS toolkit [20] is a distributed programming environment that introduces protocols based on virtually synchronous [19] process groups and group communications. Its protocols are based on multicast primitives. The causal broadcast primitive (*cbcast*) can be used to implement causally consistent geo-replicated data stores. Its system model assumes single-machine replicas under full replication setting. In order to track causal dependencies, the (*cbcast*) primitive uses vector clocks, having one entry per DC. The authors extend the (*cbcast*) primitive into a totally ordered multicast primitive called atomic broadcast (*abcast*).

Lazy Replication [63] represents another replication scheme that enables applications to enforce causal consistency. This method enables clients to define a partial order of data operations by indicating the causal order of operation calls through a front-end. It supports two types of operations, update and query operations. Each operation is labeled with its causal dependencies and send asynchronously to the rest of the replicas. In order to track causal dependencies, their implementation used vectors having one entry per replica. When a replica receives an update, it has to wait for the update's causal dependencies to be satisfied before it installs the update. Lazy replication is designed for the single-machine replica model. Each replica first creates a sequential log that contains all operations issued to that replica. The log is then send to other replicas through a gossip protocol. Next, each replica merges its log with the gossiped logs from the other replicas. Finally, each replica applies the operations in causal order.

Bayou [99] is a causally consistent replicated database system which is primarily designed for mobile applications that operate under diverse and unreliable network conditions. Bayou relies on an anti-entropy protocol for propagating updates between replicas in a lazy and incremental fashion. It is designed for single-machine replicas under full replication. The main goal is to allow clients to read and make updates even when the replica they contact is disconnected. Bayou stores updates in a log, by causally ordering them. To ensure causal order, each replica timestamps updates with a monotonically increasing scalar timestamps. To ensure that each replica eventually reaches the same final state, Bayou's replicas synchronize by pair-wise exchange of their logs. Each replica keeps two version vectors, one for already committed and one for tentative updates. The anti-entropy information that is exchanged between two replicas that are synchronizing, is a pair of committed version vectors plus the updates that are not present to one of the replicas. Bayou does not support partial replication.

The **PRACTI [18]** paradigm states that an ideal replication framework should provide the three PRACTI properties: partial replication, arbitrary consistency and topology independence. Arbitrary consistency means that a system can provide both weak and strong consistency guarantees. Topology independence means that any replica can exchange updates with any other replica. The authors propose their prototype of a system that integrates the PRACTI properties. Their implementation supports causal consistency as the weakest consistency level. The PRACTI prototype is still limited to the single-machine replication model. It uses peer-to-peer log-based replica synchronization protocol. Dependencies are tracked using a single scalar timestamp and it maintains vector clocks on each replica to reduce the amount of data exchanged between the replicas. The PRACTI prototype scales through non-genuine partial replication: some replicas still need to receive updates on data items that are not replicated locally.

TACT [112, 113] represents a middleware layer that enforces arbitrary consistency bound among replicas using three metrics defined to capture the consistency spectrum: *numerical error, order error* and *staleness*. Numerical error limits the total weight of updates that can be applied across all replicas before being propagated to a given replica. Order error limits the number of tentative updates, or updates whose global order is still undetermined and can be subject to reordering, that can be outstanding at any one replica. Staleness places a real-time bound on the delay of update propagation among replicas. The weakest form of consistency that the TACT system provides is causal consistency. The design of TACT is similar to the one of Bayou and PRACTI in the sense that they all are designed for a single-machine replicas under full replication. Each replica maintains a logical timestamps version vector, similar to the one in Bayou. Each replica exchanges its updates using anti-entropy sessions. Updates are timestamped by a monotonically increasing logical clock timestamp. Before being applied, the replica checks for conflicts with the underlying data store. Updates might have to be reordered or rolled back.

SwiftCloud [114] addressed the problem of providing transactional causal consistency to the client machine in a way that tolerated full-replica failures. In SwiftCloud [114] clients declare the items in which they are interested, and the system sends them the corresponding updates, if any. SwiftCloud uses a sequencer-based approach, which totally orders updates, both those generated in a DC and those received from remote DCs. The sequencer-based approach ensures that the stream of updates pushed to clients is causally consistent. However, sequencing the updates also makes it cumbersome to achieve horizontal scalability. SwiftCloud is designed to ensure that updates are made visible to clients when they have been applied at a predefined number (k) of replicas. SwiftCloud uses dependency vectors to enforce causal consistency. It is designed for single-machine replicas under full replication model.

6.2 Partitioned causally consistent systems

The exponential data growth made scalability one of the key properties of modern distributed systems. This sparkled the emergence of a new generation of partitioned causally consistent systems such as COPS [69], Eiger [70], Bolt-on CC [12], Orbe [43], GentleRain [42], ChainReaction [5], CausalSpartan [90], SNOW-COPS [72], Contrarian [38], MongoDB [77]. Our work POCC belongs into this category and it distinguishes from them by introduces a novel way of enforcing causal consistency.

COPS [69] The COPS system (Clusters of Order-Preserving Servers) is a geo-replicated key-value store that enforces causal+ consistency guarantees. Lloyd at all. [70] coined the term ALPS for distributed systems that provide four desirable properties: availability, low latency, partition tolerance and scalability. To provide all these properties, ALPS systems must sacrifice strong consistency. In the quest for the strongest form of consistency that is achievable under those constraints, the authors define and propose a causal+ consistency model.

COPS is designed for partitioned and geo-replicated data stores under the full replication model. In order to provide causal consistency, COPS tracks causal dependencies at the granularity of data items. A client stores every accessed item as dependency metadata in its client context. Additionally, this metadata is stored in the data store with each issued update operation. Conflicting updates are handled by last-writer-wins or application-specific reconciliation. When an update is replicated from one replica to another, the dependency metadata is propagated as well. An update is installed at remote replicas only when all of its dependencies are satisfied at that replica.

COPS supports only single-read and single-write operations. To support causally consistent non-blocking read-only transactions, the authors extend the COPS protocol into a variant called COPS-GT. COPS-GT provides read transactions that give clients a causal+ consistent snapshot of multiple keys. Although COPS and COPS-GT garbage-collect the dependency metadata periodically, the metadata size could still potentially grow large and affect the performance.

Lloyd at all. [70] present **Eiger** as a follow-up work on Cops, designed for the rich column-family data model. Eiger is a partitioned and geo-replicated causally consistent storage system, build on top of Cassandra [64]. Eiger shares some similarities with COPS in terms of data item granularity of dependency tracking, tracking every accessed item at the client side and garbage collection of stable versions. However, Eiger offers stronger semantics, due to the support of write-only transactions besides the read-only transactions. Additionally, Eiger improves the read-only protocol used in COPS, by relying on lightweight Lamport's Logical clocks [66] instead of requiring the explicit dependencies of each version of the item being read to compute the snapshot. To achieve this, the read operations sometimes need a third round, contrary to COPS, where only one round is needed in the single-read operations and maximum of two rounds in read-only transactions.

Bolt-on causal consistency [12] represents a "bolt-on" shim layer, mounted on top of existing widely-deployed eventually consistent data stores, that provides causal consistency. The general idea of Bailis et all. is to separate the consistency-related safety properties from availability and durability in distributed data stores. When issuing requests, clients interact directly with the "bolt-on" shim layer that integrates a local data store. A read request goes directly to the shim layer. An update request is first stored in the local data store of the shim layer, and it is eventually propagated to the eventually consistent data store. Bolt-on causal consistency tracks dependencies at a finer granularity, by explicitly storing the item dependencies, in a similar fashion as COPS and Eiger. The metadata size is proportional with the client's causal history. Bolt-on causal consistency does not support partial replication.

Orbe [43] addresses the problem of providing a scalable and efficient implementation of causal consistency for partitioned and replicated data stores. It uses two-dimensional matrices to track causal dependencies in a client session, with one entry per partition per data center. Each update is associated with the dependency matrix from the client session. To reduce the dependency metadata size, Du et all. propose dependency cleaning by leveraging the fact that once an update and all its dependencies are replicated, any subsequent reads will not introduce new dependencies to the client session. Orbe also supports read-only transactions, and it uses loosely synchronized physical clocks. However, Orbe does not support non-blocking transactional reads.

ChainReaction [5] is a geo-replicated key-value data store that offers causal consistency by leveraging a variant of chain replication [89]. In ChainReaction, the client library stores metadata about the client accessed state. ChainReaction supports read-only transactions, and it relies on loosely synchronized physical clocks. It uses one sequencer per data center to order all updates and all reads that are part of a read-only transaction. However, the sequencer represents a potential performance bottleneck, and it introduces an extra round-trip within the data center, which increases the latency of all update operations. ChainReaction does not support non-blocking transactional reads. For dependency tracking, it uses version vector having one entry per data center which is associated to every update. A remote update is made visible when all its causal dependencies have been fully propagated in the local data center. ChainReaction does not support partial replication.

GentleRain [42] is a causally consistent, partitioned, geo-replicated key-value data store that is designed to have throughput comparable to eventually consistent data stores. To achieve its design goal, GentleRain eliminates dependency checking messages for data updates. It tracks dependencies by using a single physical timestamp, and it ensures causal consistency by using a background stabilization protocol. Compressing metadata to a single physical timestamp comes at the price of increased visibility latency of remote updates. Local updates are immediately visible. The lower-bound of the visibility latency of remote updates is the latency between the destination data center and the data center that is furthest from it. GentleRain implements read-only transactions, but it does not provide non-blocking transactional reads. It is designed for full replication.

GentleRain implements causally consistent snapshot reads and read-only transactions. A read from a causally consistent snapshot only includes the item versions that strictly belong to the snapshot. The read-only transactions have the same semantics as the causally consistent snapshot reads, however, in addition they include any values previously read by the client that might not be part of the snapshot. If those values are excluded, causal consistency will be violated. The protocol for the snapshot reads takes only one round and is wait-free. However, the read-only transactions may block. If the expecting blocking time exceeds a predefined threshold, GentleRain employs Eiger's read protocol, which completes in a maximum of three rounds.

CausalSpartan [90] is a partitioned, geo-replicated key-value data store. It is designed for full replication model. Similarly as Wren and PaRiS, CausalSpartan utilizes Hybrid Logical Clocks (HLCs) [61] for event to timestamping. CausalSpartan augments GentleRain with HLCs to make PUT operations robust against clock skew. The stabilization protocol is the same as in GentleRain. CausalSpartan does not support transactions. In CausalSpartan, dependencies are tracked using version vectors having one entry per data center.

Lu et all. [72] show an impossibility result that states that no read-only transaction algorithm can provide both the lowest latency and the highest power. They define the SNOW properties as the following pairs of properties: strict serializability (S) and write transactions that conflict (W), which provide the highest power, and nonblocking operations (N) and one response per read (O) which provide lowest latency. The SNOW Theorem states that no read-only transaction algorithm provides all of the SNOW properties. They showed that it is impossible to perform non-blocking causally consistent read-only transactions using one-round of communication and sending only a single version of the objects involved. The authors implemented two systems: COPS-SNOW and Rococo-SNOW. COPS-SNOW is designed to be a latency-optimal (LO) system by providing non-blocking read-only transactions combined with one response per read. Rococo-SNOW is designed to be SNOW-optimal, as in providing three of the SNOW properties (S + O + W). However, both designs push the overhead towards the writes.

Didona et all. [38] show that LO read-only transactions, as defined by Lu et all. [72] introduce additional cost on writes that is so high that they, in practice, exhibit performance inferior to alternative designs, even in read-heavy workloads. This cost manifests itself not just by negatively influencing throughput, but also by causing higher resource contention, and hence higher latencies. The authors demonstrate this counter-intuitive result theoretically and practically. Theoretically they show that the extra cost imposed on writes to achieve LO read-only transactions is inherent to causal consistency, and it cannot be avoided by any causally consistent system that implements LO read-only transactions. Practically they support their claim by designing Contrarian, a causally consistent partitioned geo-replicated data store that achieves non-blocking read-only transactions and is single-versioned but it requires two rounds of communication. Contrarian uses hybrid logical clocks for timestamping events. It tracks causal dependencies using version vectors having one entry per data center. Contrarian is designed for full replication setting.

Tyulenev et all. [106] discuss how they implemented causal consistency in MongoDB [77]. MongoDB is a distributed database that supports replication and partitioning. Each partition in MongoDB represents a replica set that contains a primary node and one or more secondary nodes. The primary node is selected by consensus election, and it receives the writes for its particular dataset partition. MongoDB timestamps events using hybrid logical clock. It tracks causal dependencies using a single-scalar timestamp, called Stable Cluster Time (SCT). SCT is monotonically increased by performing write operations. Each partition maintains an operation log, where the SCT is persisted. Although they are considering to implement causally consistent transactions as their future work, the currently described version of MongoDB does not support causally consistent transactions. MongoDB does not support partial replication.

6.3 Transactional causally consistent systems

Transactional Causal Consistency (TCC) extends causal consistency with interactive read-write transactions, and is therefore particularly appealing for geo-replicated platforms. Implementing interactive read-write transactions, however, is challenging and there are just a few systems that support TCC. From this category, we present Cure [4], AV [105], Occult [76]. SwiftCloud, which we already introduced in the non-partitioned CC systems, is another system that supports interactive read-write transactions. However, it does not scale. Our work Wren is the first TCC system that at the same time i) implements nonblocking read operations and ii) allows an application to efficiently scale out within a replication site by sharding.

Akkoorath et al. [4] introduced TCC by presenting their system **Cure**. Cure is a distributed key-value data store whose system model assumes partitioning and geo-replication under the full replication data scheme. It implements interactive read-write transactions that read from a causally consistent snapshot and guarantee atomicity of writes. However, the reads in Cure could block because a transaction T can be assigned a snapshot that has not been installed by some partitions. If T reads from any of such laggard partitions, it will block. Cure uses CRDTs to guarantee convergence. It represents causal dependencies by dependency vectors with one entry per DC. Cure's protocol relies on loosely synchronized physical clocks. Cure runs a periodic dependency stabilization protocol to identify stable items.

Tomsic et all. [105] study the trade-offs between the read guarantees, delay, and freshness of transactional reads in distributed systems. They show several impossibility results. The authors implement three systems, **CV**, **OP** and **AV** by modifying the implementation of Cure. Each system has the properties of Cure such as it implements TCC, uses version vector for dependency tracking, uses physical clock timestamps, etc. However, each system has different design goal. AC achieves minimal delay, bu choosing an older snapshot, similar to Wren. CV ensures Read Committed Isolation. OP provides causally-consistent snapshot reads and atomic updates, similar to Wren. Wren is more efficient than AV, because it provides BiST and BDT, an efficient dependency tracking and stabilization protocols that need only two timestamps to track dependencies, and AV relies on version vector, which limits the scalability of the system.

6.3.1 Optimistic causally consistent systems

Occult [76] is another geo-replicated distributed key-value store that implements general-purpose transactions. Moreover, to our best knowledge, it is the only other OCC system that implements causal consistency. Occult was designed in parallel with our system POCC. It provides causal consistency to its clients without exposing the system to the effects of slowdown cascades [3]. Occult implements a weaker variant of PSI [93] called Per-Client Parallel Snapshot Isolation (PC-PSI). Occult supports general-purpose transactions. However, Occult implements master-slave replication model in which only the master replica of a partition accepts writes. Thus, the commit of a transaction may span multiple DCs, which causes performance degradation. Occult offers non-blocking reads, however, a read operation may have to be retried several times in case of missing dependencies, and may even have to contact the remote master replica, which might not be accessible due to a network partition. The consequence of retrying read operations in Occult has a negative impact on performance, which is comparable to blocking the read to receive the right value to return. Additionally, Occult can abort even the read-only transactions. By contrast, POCC implements a multi-master replication model and all of its operations complete wholly within a DC. POCC never aborts read-only transactions or retries read operations.

6.4 Partial replication in causally consistent systems

As it can significantly reduce the storage requirements and replication costs, partial replication gains increased popularity as a replication model. There are only handful of systems that provide causal consistency in a partial replication setting: Saturn [22], C^3 [46], Karma [74], Xiang and Vaidya [111]. Among them, our system PaRiS is the first TCC system that supports partial replication and implements non-blocking parallel read operations.

Saturn [22] is a metadata service for geo-replicated systems that guarantees causal consistency. Saturn keeps the size of the metadata small and constant by using a single scalar to track causality. It is designed to provide genuine partial replication. The main design goal of Saturn is to relieve the data store from managing consistency across data centers. Saturn disseminates metadata using a tree-based technique via a shared tree among the data centers. All updates between data centers are serialized and transmitted through the shared tree. The tree-based dissemination technique has several weaknesses. It represents the bottleneck of the system, because all metadata has to be sent through the tree. In case of a tree-node failure, the new tree needs to be recomputed, which is time consuming process. Saturn does not support transactions.

C^3 [46] represents a causally consistent geo-replicated data store designed for partial replication. Similarly as Saturn, C^3 separates the data store layer from the metadata layer. C^3 tracks causal dependencies using version vector, named executing clock, with one entry per data center. When the client makes an update, the request is first forwarded to the causality layer. The causality layer increments the local operation counter and uses it to assign a timestamp to the operation. It captures the update's dependencies by storing the executing clock value with the update metadata into the update label. Then the label is propagated to the causality layer in every data center that replicates the modified item. C^3 does not support transactions.

Karma [74] is a partitioned geo-replicated causally consistent data store that supports partial replication. Since partial replication, unavoidably, results in remote accesses which inquire high latency, Karma tries minimize those remote accesses by relying on per-DC caches and persistent write-buffers. In Karma's architecture, one full replica dataset is disjointly distributed among several DCs that from a consistent-hashing ring. The number of rings is equal to the number of full replicas. This limits the flexibility and generality of Karma's architecture. In order to ensure causal consistency, Karma employs dynamic ring restrictions (DRR) which force subsequent client reads to go to the same ring. It tracks in-flight objects (or objects that are still not stable and whose exposure could violate causal consistency) in order to put the threads accessing such objects in restrictive mode. Tracking dependencies in Karma is similar to COPS, because they both track dependencies per data item. To track causal dependencies, Karma requires per-client state and to track in-flight versus stable items, it tracks per-item state. Karma can support read-only transactions by minimally modifying Eiger's read-only transactions protocol in order to address the dynamic ring binding and caching.

Xiang and Vaidya [111] propose a framework for supporting general partial replication with causal consistency using global stabilization. Global stabilization a technique for achieving causal consistency, that, for example, was used in GentleRain and Cure. The authors propose an algorithm designed for distributed multi-version key-value stores with general partial replication. The algorithm is inspired by GentleRain and shares several similarities with it, such as using a single physical clock time. This timestamp is associated with every update. In order to determine its value, a global stabilization protocol is run. However, partial replication complicates the stabilization protocol. The algorithm that Xiang and Vaidya propose augments the protocols from GentleRain by allowing arbitrary replication across all the servers, and enabling the client to communicate with an arbitrary subset of serves and migrate without extra delay. However, their protocol does not support transactions.

Full-Track and **Opt-Track** are algorithms proposed by Hsu et all. [55] for implementing causal consistency under partial replication. However, a data center, or site in their terminology, is a single-machine replica that does not support partitioning. Both algorithms track causal dependencies using a matrix clock of size n × n, where n is the number of sites. This clock is associated with every update and is piggybacked when the update is being replicated. Opt-Track represents an optimization of Full-Track algorithm. It minimizes the size of meta-data carried on messages and stored in local logs by utilising the transitivity rule of causal consistency. Furthermore, the authors additionally extend the Opt-Track algorithm into an algorithm that ensures causal consistency under full replication, named Opt-Track-CPR. To track causal dependencies, Opt-Track-CPR uses a version vector with one entry per replica.

6.5 Other related aspects

6.5.1 Client-side caching

Caching at the client side is a technique primarily used to support disconnected clients, especially in mobile and wide area network settings [21, 87, 115]. Wren and PaRiS, instead, uses client-side caching to guarantee consistency.

6.5.2 Speculative systems

Optimistic Causal Consistency (OCC) is also related to the literature on *speculative* systems. Speculation consists of optimistically processing an operation even if previous operations have not been fully completed yet and only their tentative result is available. If the tentative result differs from the later final one, the system rollbacks to a consistent state. Speculation has been investigated in a variety of contexts, including processor architectures [51], operating systems [79], simulators [56], concurrency control schemes for transactional systems [62], software transactional memory [86] and recent systems that can concurrently support different consistency levels [48]. The primary aim of speculation is to reduce operation response times by avoiding to wait for the expensive computation of the final value of an operation. OCC, instead, embraces an optimistic approach to maximize the freshness of data returned to client and to reduce the overhead for dependency checking/stabilization. As we have shown, however, in many cases the better resource efficiency enabled by OCC can also yield performance gains in terms both of response times and achievable throughput.

Speculation has been widely investigated in the context of strongly consistent transactional systems [60, 26, 84] and, recently, also in the context of systems achieving lower consistency guarantees. With reference to a transactional systems, speculation entails executing a transaction even if the outcome of previous, potentially conflicting transaction has not been determined yet. If one transaction T waiting for validation aborts, it may cause a domino effect, aborting all or part of the transactions that have been started after T. Speculation optimistically assumes a low contention rate and aims to reduce the response time of a transaction by overlapping the execution and validation phase of subsequent transactions. Our proposal is, instead, framed in the context of a weaker consistency model and is primarily aimed to maximize the freshness of data returned to client.

6.5.3 Ordering services

Kronos. [45] represents a fault-tolerant event ordering service that tracks dependencies and provides time ordering for distributed applications. The main idea behind Kronos is to separate the task of tracking causal relationships from distributed subsystems and centralize them in a separate event ordering service. On the one hand, because it is designed for geo-replicated applications, clients have to pay the cost of potentially higher latencies due to the large round trips to use the service.

Gunawardhana et all. [49] propose **Eunomia**, a service responsible for metadata serializing in an order consistent with causality and by operating out of the clients' critical operational path. The design goal of Eunomia was to replace the traditional sequencers, by letting client operations to execute without synchronous coordination. They implement their service on a variant of Riak in **EunomiaKV**, a geo-replicated partitioned key-value store, where each DC relies on an instance of Eunomia. By using background site stabilization procedure, Eunomia establishes a serialization of all updates originating in the local data center in causal order consistent. After determining the order, Eunomia notifies other data centers about it. EunomiaKV uses hybrid logical clock timestamps, generated by the local data center. To track causal dependencies, metadata is represented with a version vector with one entry per DC. EunomiaKV does not support transactions and partial replication.

6.5.4 Highly-available transactional systems

Bailis et al. [14, 16] propose several flavors of transactional protocols that are available and support read-write transactions. These protocols rely on fine-grained dependency tracking and enforce a consistency level that is weaker than CC.

GSP [25] is an operational model for replicated data that supports highly available transactions. GSP targets non-partitioned data stores and uses a system-wide broadcast primitive to totally order the updates.

TARDiS [35] is a multi-master transactional key-value store explicitly designed for weakly-consistent systems. TARDiS introduces the branch-on-conflict concurrency control mechanism. It supports merge functions over conflicting *states* of the application, rather than at key granularity. This flexibility requires a significant amount of metadata and a resource-intensive garbage collection scheme to prune old states. TARDiS does not implement sharding.

7 Conclusions and Future Work

Causal consistency is an attractive consistency model for building modern replicated datastores. Compared to strong and eventual consistency, it captures best of both worlds: it does not suffer from the long latencies and partition intolerance of the former and it avoids some of the anomalies that are possible under the latter. In the past decade, the desirability of causal consistency provoked an increased interest from the research community. However, the industry is still reluctant to make a widespread use of it, so when it comes to designing geo-replicated storage systems, system designers are still viewing their options as a binary choice between eventual and strong consistency. Of course, applications that require global ordering, such as banking and financial systems, will always need strongly consistent solutions. Nonetheless, other types of applications, currently implemented with eventual consistency, such as social networks, ticketing systems, collaborative systems etc., could certainly benefit from the advantages that causal consistency offers.

In this dissertation, we contribute to decreasing the performance gap between causal and eventual consistency. We present three novel systems, and each system addresses different challenges and explores novel trade-offs in the design space of causally consistent geo-replicated and partitioned key-value stores.

Maximizing data freshness. First we address the challenge of maximizing data freshness - an underrepresented requirement of applications that can benefit from increased data freshness, such as e-commerce systems, social networks, trading systems. For that purpose, we have introduced a novel approach to implementing causal consistency in geo-replicated key-value stores, which we named Optimistic Causal Consistency (OCC). OCC's main goal is to increase data freshness. In OCC, a server always returns the most recent available version of an item. Its design was inspired by two observations. First, recent research [110, 15, 71] has revealed that data items are typically replicated and accessed after their dependencies have already been propagated following a naturally consistent order. Second, network partitions (especially complete DC failures) are relatively rare events [15, 93, 23, 67]. These observations led us to question whether existing protocols for ensuring causal consistency in modern data centers are too pessimistic. Existing systems enforce causal consistency by ensuring that they will not expose any replicated item versions to clients until all of their causal dependencies have been received as well. This invariant demands the implementation of dependency checking or stabilization protocols, which delay the visibility of new data items and result in computational and communication overhead. We argue that this behaviour is too pessimistic for modern systems and most of the time unnecessary. On the contrary, in OCC, the dependency check needed to enforce CC is done only when it is necessary: upon serving a client operation, rather than on the receipt of a replicated data item as in existing systems.

In order to increase data freshness, OCC exposes unstable data items. To enforce CC while exposing unstable items, we introduced a new technique called client-assisted lazy dependency resolution. By empowering the client with the ability to track dependencies, client-assisted lazy dependency resolution frees the data store from the burden of running distributed protocols to track the delivery of updates.

With OCC, we explore a novel trade-off in the landscape of CC protocols. Its potentially blocking behavior makes it vulnerable to network partitions. Because network partitions are rare in practice, however, OCC trades availability to maximize data freshness and reduce the communication overhead. We further propose a recovery mechanism that allows an OCC system to fall back on a pessimistic protocol to remain available during network partitions. POCC is an implementation of OCC based on physical clocks. We show that OCC improves data freshness, while offering comparable or better performance than its pessimistic counterpart.

Low latency, rich transactional semantics and scalability. Next, we explored how to provide a desirable and practically meaningful combination of properties such as low latency, rich transactional semantics and scalability. Our observation was that state-of-the art transactional read algorithms of TCC systems suffer from latency overheads that have prevented their adoption at scale. To overcome that challenge, we have designed Wren, the first TCC system that at the same time implements nonblocking reads thereby achieving low latency and allows applications to scale-out by sharding.

Enforcing CC while offering always-available interactive multi-partition transactions is a challenging problem. Wren implements a novel transactional protocol, CANToR, that defines transaction snapshots as the union of a fresh causal snapshot and the contents of a client-side cache.

Wren also introduces BDT, a new dependency tracking protocol, and BiST, a new stabilization protocol. BDT and BiST use only two timestamps per update and per snapshot, enabling scalability regardless of the size of the system. By decoupling local and remote items, BiST allows transactions to determine the visibility of local items without synchronizing with remote DCs, enabling availability and nonblocking reads also in the geo-replicated case.

We have compared Wren with a state-of-the-art TCC system, and we have shown that Wren achieves lower latencies and higher throughput, while only slightly decreasing the freshness of data exposed to clients.

Partial replication. The final contribution of this dissertation is PaRiS, the first system that implements TCC in a partially replicated system and achieves non-blocking read operations. With the exponential growth of data, partial replication could soon be the favorable replication model for key-value data stores since it can significantly reduce the storage requirements and replication costs that a full replication model incurs.

Nonetheless, implementing TCC under partial replication is a not an easy challenge, due to the fact that different reads within the same transaction may be served in parallel by servers in different DCs. It also requires devising a clever and efficient way to represent and track causal dependencies. We address these challenges by implementing a novel dependency tracking protocol, called UST, which requires only one timestamp to track dependencies. UST identifies a snapshot of the data that is available at *every* DC, thereby enabling non-blocking reads regardless of the DC in which the read takes place. In addition to the UST-defined snapshot, PaRiS equips clients with a private cache, where clients store their own updates that are not yet reflected in the UST-defined snapshot. The combination of the cache and the UST-defined snapshot suffices to implement TCC with non-blocking reads in the partial replication setting. Hence, transactions can consistently read from such a snapshot on any server in any replication site without having to block. Moreover, PaRiS requires only one timestamp to track dependencies and define transactional snapshots, thereby achieving resource efficiency and scalability.

To achieve these benefits, PaRiS makes a trade-off that is provably unavoidable: it exposes to transactions a view of the data that is slightly in the past. We argue that a moderate increase in data staleness is a reasonable price to pay for the performance benefits accomplished by PaRiS.

We evaluated PaRiS on a data platform replicated on up to 10 DCs. PaRiS scales well and achieves lower latency than the blocking alternative, while being able to handle larger datasets than existing solutions that assume full replication. We furthermore show that PaRiS scales well with the number of DCs and partitions, while being able to handle larger data-sets than existing solutions that assume full replication.

Applications. Depending on the use case, the presented systems can be used for both cloud computing and edge computing model. Because of their design, Wren and PaRiS are better suited for the cloud-computing model, and POCC can be a good fit for the edge computing environment.

In conclusion, this dissertation presents three novel systems that explore and efficiently solve different challenges in the design space of causally consistent geo-replicated data stores: improving data freshness, freeing the data store from the responsibility of tracking causal dependencies, providing low latency reads with richer transactional semantics in partitioned fully and partially replicated system architectures, introducing efficient dependency tracking and stabilization protocols tailored for full and partial replication settings.

7.1 Future work

Extending POCC. In the first part of this dissertation, we have focused on assessing the benefits of OCC during normal operational behavior, i.e., in the absence of network partitions. As future work, we plan to quantitatively assess the performance and behavior of POCC in presence of network partitions and full data center failures. We would like to implement and evaluate the highly available extension of OCC that we propose as a fall-back mechanism during network partitions.

Exploring partition fault tolerance methods. In a similar fashion as for OCC, we would like to explore which fault tolerant mechanisms would be best suited for Wren and for PaRiS in the unfortunate case a partition goes down in a DC. Possible techniques that could be used are chain replication, Paxos-based or Raft-based replication of partitions in the local DC.

Moving towards the edge. Nowadays, with the recent technological advance in wearable gadgets and smart IoT devices, as well as the proliferation of portable and mobile devices, there is a shift from the cloud computing paradigm towards the edge computing paradigm. The key idea in the edge computing model is to push away the computations and data storage from the centralized nodes towards the data sources, thus avoiding the unnecessary data transfers to the cloud, as well as providing faster responses to the clients. However, the current edge-centered solutions are still in the inception phase and at this point are either limited from scalability or performance point of view, or they require development of specialized hardware. Furthermore, the edge devices have limited network bandwidth, computational power and storage capacity, and are less reliable than a classical data center.

Our future work will be focused toward designing novel key-value data stores envisioned to fill the gap between the traditional cloud computing model and the edge computing model.

Bibliography

[1] Matt Adorjan. *AWS Inter-Region Latency Monitoring*. URL: https://www.cloudping.co/grid.

[2] Mustaque Ahamad et al. “Causal Memory: Definitions, Implementation, and Programming”. In: *Distributed Computing* 9.1 (1995), pp. 37–49.

[3] Phillipe Ajoux et al. “Challenges to Adopting Stronger Consistency at Scale”. In: *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*. HOTOS’15. Switzerland: USENIX Association, 2015, p. 13.

[4] Deepthi Devaki Akkoorath et al. “Cure: Strong semantics meets high availability and low latency”. In: *Proc. of ICDCS*. 2016.

[5] Sérgio Almeida, João Leitão, and Luiés Rodrigues. “ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication”. In: *Proc. of EuroSys*. 2013.

[6] *Amazon Found Every 100ms of Latency Cost them 1% in Sales*. URL: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/.

[7] Eric Anderson et al. “What Consistency Does Your Key-value Store Actually Provide?” In: *Proc. of HotDep*. 2010.

[8] Hewlett Packard Enterprise Antonio Neri. *World Economic Forum Interview*. 2019.

[9] Berk Atikoglu et al. “Workload Analysis of a Large-scale Key-value Store”. In: *Proc. of SIGMETRICS*. 2012.

[10] Hagit Attiya, Faith Ellen, and Adam Morrison. “Limitations of Highly-Available Eventually-Consistent Data Stores”. In: *Proc. of PODC*. 2015.

[11] Peter Bailis and Kyle Kingsbury. “The Network is Reliable”. In: *Queue* 12.7 (July 2014), 20:20–20:32. ISSN: 1542-7730. DOI: 10.1145/2639988.2639988. URL: http://doi.acm.org/10.1145/2639988.2639988.

[12] Peter Bailis et al. “Bolt-on Causal Consistency”. In: *Proc. of SIGMOD*. 2013.

[13] Peter Bailis et al. “Highly Available Transactions: Virtues and Limitations”. In: *PVLDB* 7.3 (2013), pp. 181–192. DOI: 10.14778/2732232.2732237. URL: http://www.vldb.org/pvldb/vol7/p181-bailis.pdf.

[14] Peter Bailis et al. “Highly Available Transactions: Virtues and Limitations”. In: *Proc. VLDB Endow.* 7.3 (Nov. 2013), pp. 181–192. ISSN: 2150-8097. DOI: 10.14778/2732232.2732237. URL: http://dx.doi.org/10.14778/2732232.2732237.

[15] Peter Bailis et al. “Probabilistically Bounded Staleness for Practical Partial Quorums”. In: *Proc. VLDB Endow.* 5.8 (Apr. 2012), pp. 776–787.

[16] Peter Bailis et al. “Scalable Atomic Visibility with RAMP Transactions”. In: *Proc. of SIGMOD*. 2014.

[17] Oana Balmau et al. “TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores”. In: *Proc. of ATC*. 2017.

[18] Nalini Belaramani et al. “PRACTI Replication”. In: *Proc. of NSDI*. 2006.

[19] Kenneth P. Birman and Thomas A. Joseph. “Reliable Communication in the Presence of Failures”. In: *ACM Trans. Comput. Syst.* 5.1 (Jan. 1987), pp. 47–76.

[20] Kenneth Birman, André Schiper, and Pat Stephenson. “Lightweight Causal and Atomic Group Multicast”. In: *ACM Trans. Comput. Syst.* 9.3 (Aug. 1991), pp. 272–314. ISSN: 0734-2071. DOI: 10.1145/128738.128742. URL: http://doi.acm.org/10.1145/128738.128742.

[21] Magnus E. Bjornsson and Liuba Shrira. “BuddyCache: High-performance Object Storage for Collaborative Strong-consistency Applications in a WAN”. In: *Proc. of OOPSLA*. 2002.

[22] Manuel Bravo, Luıés Rodrigues, and Peter Van Roy. “Saturn: A Distributed Metadata Service for Causal Consistency”. In: *Proc. of EuroSys*. 2017.

[23] Eric Brewer. “CAP Twelve Years Later: How the "Rules" Have Changed”. In: *Computer* 45.2 (2012), pp. 23–29.

[24] Eric A. Brewer. “Towards Robust Distributed Systems (Abstract)”. In: *Proc. of PODC*. 2000.

[25] Sebastian Burckhardt et al. “Global Sequence Protocol: A Robust Abstraction for Replicated Shared State”. In: *Proceedings of ECOOP*. 2015.

[26] Ugur Cetintemel, Peter J. Keleher, and Michael J. Franklin. “Support for Speculative Update Propagation and Mobility in Deno”. In: *Proc. of ICDCS*. 2001.

[27] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. In: *ACM Trans. Comput. Syst.* 26.2 (June 2008). ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: https://doi.org/10.1145/1365815.1365816.

[28] Jennifer Chu. *Data freshness, not speed, most important for IoT*. URL: http://news.mit.edu/2018/keeping-data-fresh-wireless-networks-0605.

[29] Matthew Clark. *How the BBC builds websites that scale*. URL: https://www.creativebloq.com/features/how-the-bbc-builds-websites-that-scale.

[30] IBM Marketing Cloud. *10 Key Marketing Trends For 2017*. 2017.

[31] *Cook Case Study*. URL: https://www.nccgroup.trust/globalassets/resources/uk/case-studies/web-performance/cook-case-study.pdf.

[32] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proc. of SoCC*. 2010.

[33] Brian F. Cooper et al. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1277–1288. ISSN: 2150-8097. DOI: 10.14778/1454159.1454167. URL: https://doi.org/10.14778/1454159.1454167.

[34] James C. Corbett et al. "Spanner: Google's Globally Distributed Database". In: *ACM Trans. Comput. Syst.* 31.3 (Aug. 2013), 8:1–8:22.

[35] Natacha Crooks et al. "TARDiS: A Branch-and-Merge Approach To Weak Consistency". In: *Proc. of SIGMOD*. 2016.

[36] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proc. of SOSP*. 2007.

[37] D. Didona, K. Spirovska, and W. Zwaenepoel. "Okapi: Causally Consistent Geo-Replication Made Faster, Cheaper and More Available". In: *ArXiv e-prints, https://arxiv.org/abs/1702.04263* (Feb. 2017). arXiv: 1702.04263 [cs.DC].

[38] Diego Didona et al. "Causal Consistency and Latency Optimality: Friend or Foe?" In: *Proc. VLDB Endow.* 11.11 (July 2018).

[39] Phil Dixon. *Shopzilla site redesign: We get what we measure*. Velocity Conference Talk. 2009.

[40] *Driving user growth with performance improvements*. URL: https://medium.com/pinterest-engineering/driving-user-growth-with-performance-improvements-cfc50dafadd7.

[41] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. "Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks". In: *Proc. of SRDS*. 2013.

[42] Jiaqing Du et al. "GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks". In: *Proc. of SoCC*. 2014.

[43] Jiaqing Du et al. "Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks". In: *Proc. of SoCC*. 2013.

[44] Sameh Elnikety et al. "Predicting Replicated Database Scalability from Standalone Database Profiling". In: *Proc. of EuroSys*. 2009.

[45] Robert Escriva et al. “Kronos: the design and implementation of an event ordering service”. In: *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*. Ed. by Dick C. A. Bulterman et al. ACM, 2014, 3:1–3:14. DOI: 10.1145/2592798.2592822. URL: https://doi.org/10.1145/2592798.2592822.

[46] P. Fouto, J. Leitão, and N. Preguiça. “Practical and Fast Causal Consistent Partial Geo-Replication”. In: *Proceedings of NCA*. 2018.

[47] Daniel F Garcıéa and Javier Garcıéa. “TPC-W e-commerce benchmark evaluation”. In: *Computer* 36.2 (2003).

[48] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. “Incremental Consistency Guarantees for Replicated Objects”. In: *Proc. of OSDI*. 2016.

[49] Chathuri Gunawardhana, Manuel Bravo, and Luıés Rodrigues. “Unobtrusive Deferred Update Stabilization for Efficient Geo-Replication”. In: *Proc. of ATC*. 2017.

[50] Rema Hariharan and Ning Sun. “Workload characterization of SPECweb2005”. In: *SPEC Benchmark Workshop*. 2006.

[51] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. Morgan Kaufmann Publishers Inc., 2011.

[52] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects”. In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492.

[53] *How much text versus metadata is in a tweet?* http://goo.gl/EBFIFs.

[54] John A. Hoxmeier, Ph. D, and Chris Dicesare Manager. “System response time and user satisfaction: An experimental study of browser-based applications”. In: *Proceedings of the Association of Information Systems Americas Conference*. 2000, pp. 10–13.

[55] T. Hsu and A. D. Kshemkalyani. “Value the Recent Past: Approximate Causal Consistency for Partially Replicated Systems”. In: *IEEE TPDS* 29.1 (2018).

[56] D. Jefferson et al. “Time Warp Operating System”. In: *Proc. of SOSP*. 1987.

[57] Jepsen. *Consistency Models*. URL: https://jepsen.io/consistency.

[58] Steve Jones. *Data freshness, not speed, most important for IoT*. URL: https://www.networkworld.com/article/3281126/data-freshness-not-speed-most-important-for-iot.html.

[59] Steve Jones. *Why Data Freshness Is More Important Than Speed For IoT*. URL: https://it.toolbox.com/blogs/stevejones/why-data-freshness-is-more-important-than-speed-for-iot-080918.

[60] Bettina Kemme et al. “Processing Transactions over Optimistic Atomic Broadcast Protocols”. In: *Proc. of ICDCS*. 1999.

[61] Sandeep S. Kulkarni et al. “Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases”. In: *Proc. of OPODIS*. 2014.

[62] H. T. Kung and John T. Robinson. “On Optimistic Methods for Concurrency Control”. In: *ACM TODS* 6.2 (June 1981).

[63] Rivka Ladin et al. “Providing High Availability Using Lazy Replication”. In: *ACM Trans. Comput. Syst.* 10.4 (Nov. 1992), pp. 360–391.

[64] Avinash Lakshman and Prashant Malik. “Cassandra: A Decentralized Structured Storage System”. In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: https://doi.org/10.1145/1773912.1773922.

[65] Leslie Lamport. “The Part-time Parliament”. In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169.

[66] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (July 1978), pp. 558–565.

[67] Vincent Liu et al. “F10: A Fault-tolerant Engineered Network”. In: *Proc. of NSDI*. 2013.

[68] Diego R Llanos and Belén Palop. “TPCC-UVa: an open-source TPC-C implementation for parallel and distributed systems”. In: *Proc. of IPDPS*. 2006.

[69] Wyatt Lloyd et al. “Don’t Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS”. In: *Proc. of SOSP*. 2011.

[70] Wyatt Lloyd et al. “Stronger Semantics for Low-latency Geo-replicated Storage”. In: *Proc. of NSDI*. 2013.

[71] Haonan Lu et al. “Existential Consistency: Measuring and Understanding Consistency at Facebook”. In: *Proc. of SOSP*. 2015.

[72] Haonan Lu et al. “The SNOW Theorem and Latency-Optimal Read-Only Transactions”. In: *In Proc. of OSDI*. 2016.

[73] P. Mahajan, L. Alvisi, and M. Dahlin. *Consistency, Availability, Convergence*. Tech. rep. TR-11-22. Computer Science Department, University of Texas at Austin, May 2011.

[74] T. Mahmood et al. “Karma: Cost-effective Geo-replicated Cloud Storage with Dynamic Enforcement of Causal Consistency”. In: *IEEE Transactions on Cloud Computing* (2018).

[75] *Marissa Mayer at Web 2.0*. URL: http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.

[76] Syed Akbar Mehdi et al. “I Can’t Believe It’s Not Causal! Scalable Causal Consistency with No Slowdown Cascades”. In: *Proc. of NSDI*. 2017.

[77] *MongoDB*. URL: https://www.mongodb.com/.

[78] Faisal Nawab et al. “Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores”. In: *Proc. of SIGMOD*. 2015.

[79] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. “Speculative Execution in a Distributed File System”. In: *ACM TOCS* 24.4 (Nov. 2006).

[80] Rajesh Nishtala et al. “Scaling Memcache at Facebook”. In: *Proc. of NSDI*. 2013.

[81] Shadi A. Noghabi et al. “Ambry: LinkedIn’s Scalable Geo-Distributed Object Store”. In: *Proc. of SIGMOD*. 2016.

[82] *NTP: The Network Time Protocol*. http://www.ntp.org.

[83] Brian M. Oki and Barbara H. Liskov. “Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems”. In: *Proc. of PODC*. 1988.

[84] Roberto Palmieri, Francesco Quaglia, and Paolo Romano. “AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing”. In: *Proc. of NCA*. 2010.

[85] D. S. Parker et al. “Detection of Mutual Inconsistency in Distributed Systems”. In: *IEEE Trans. Softw. Eng.* 9.3 (May 1983), pp. 240–247. ISSN: 0098-5589.

[86] Sebastiano Peluso et al. “SPECULA: Speculative Replication of Software Transactional Memory”. In: *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*. SRDS ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 91–100. ISBN: 978-0-7695-4784-8. DOI: 10.1109/SRDS.2012.67. URL: https://doi.org/10.1109/SRDS.2012.67.

[87] Dorian Perkins et al. “Simba: Tunable End-to-end Data Consistency for Mobile Apps”. In: *Proc. of EuroSys*. 2015.

[88] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. “Conflict-Free Replicated Data Types CRDTs”. In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Zomaya. Cham: Springer International Publishing, 2018, pp. 1–10. ISBN: 978-3-319-63962-8. DOI: 10.1007/978-3-319-63962-8_185-1. URL: https://doi.org/10.1007/978-3-319-63962-8_185-1.

[89] Robbert van Renesse and Fred B. Schneider. “Chain Replication for Supporting High Throughput and Availability”. In: *Proc. of OSDI*. 2004.

[90] Mohammad Roohitavaf, Murat Demirbas, and Sandeep Kulkarni. “CausalSpartan: Causal Consistency for Distributed Data Stores using Hybrid Logical Clocks”. In: *SRDS*. 2017.

[91] Eric Schurman and Jake Brutlag. *The user and business impact of server delays, additional bytes, and http chunking in web search*. Velocity Conference Talk. 2009.

[92] Marc Shapiro et al. “Conflict-free Replicated Data Types”. In: *Proc. of SSS*. 2011.

[93] Yair Sovran et al. “Transactional Storage for Geo-replicated Systems”. In: *Proc. of SOSP*. 2011.

[94] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. “Optimistic Causal Consistency for Geo-Replicated Key-Value Stores”. In: *Proc. of ICDCS*. 2017.

[95] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. “PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication”. In: *Proceedings of ICDCS*. 2019.

[96] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. “Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store”. In: *Proceedings of DSN*. 2018.

[97] *Storing hundreds of millions of simple key-value pairs in Redis*. http://goo.gl/ieeU17.

[98] Roshan Sumbaly et al. “Serving Large-Scale Batch Computed Data with Project Voldemort”. In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies*. FAST’12. San Jose, CA: USENIX Association, 2012, p. 18.

[99] D. B. Terry et al. “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System”. In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP ’95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 172–182. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224070. URL: http://doi.acm.org/10.1145/224056.224070.

[100] D. B. Terry et al. “Session guarantees for weakly consistent replicated data”. In: *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. Sept. 1994, pp. 140–149. DOI: 10.1109/PDIS.1994.331722.

[101] *The Cost of Latency*. URL: https://www.digitalrealty.com/blog/the-cost-of-latency.

[102] *The DAO Hack Explained: Unfortunate Take-off of Smart Contracts*. URL: https://medium.com/@ogucluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562.

[103] *The need for mobile speed: How mobile latency impacts publisher revenue*. URL: https://www.thinkwithgoogle.com/intl/en-154/insights-inspiration/research-data/need-mobile-speed-how-mobile-latency-impacts-publisher-revenue/.

[104] Robert H. Thomas. “A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases”. In: *ACM Trans. Database Syst.* 4.2 (June 1979), pp. 180–209.

[105] Alejandro Z. Tomsic, Manuel Bravo, and Marc Shapiro. “Distributed Transactional Reads: The Strong, the Quick, the Fresh & the Impossible”. In: *Proceedings of the 19th International Middleware Conference*. Middleware ’18. 2018.

[106] Misha Tyulenev et al. “Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB”. In: *SIGMOD ’19*. 2019.

[107] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. “Wikipedia Workload Analysis for Decentralized Hosting”. In: *Comput. Netw.* 53.11 (July 2009).

[108] Paolo Viotti and Marko Vukolic. "Consistency in Non-Transactional Distributed Storage Systems". In: *ACM Comput. Surv.* 49.1 (June 2016). ISSN: 0360-0300. DOI: 10.1145/2926965. URL: https://doi.org/10.1145/2926965.

[109] Werner Vogels. "Eventually Consistent". In: *Commun. ACM* 52.1 (Jan. 2009), pp. 40–44.

[110] Hiroshi Wada et al. "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective". In: *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. 2011, pp. 134–143.

[111] Zhuolun Xiang and Nitin H. Vaidya. "Global Stabilization for Causally Consistent Partial Replication". In: *CoRR* abs/1803.05575 (2018). arXiv: 1803.05575. URL: http://arxiv.org/abs/1803.05575.

[112] H. Yu and A. Vahdat. "Building replicated Internet services using TACT: a toolkit for tunable availability and consistency tradeoffs". In: *Proceedings Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems. WECWIS 2000*. 2000, pp. 75–84.

[113] Haifeng Yu and Amin Vahdat. "Design and Evaluation of a Continuous Consistency Model for Replicated Services". In: *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation - Volume 4*. OSDI'00. San Diego, California: USENIX Association, 2000.

[114] Marek Zawirski et al. "Write Fast, Read in the Past: Causal Consistency for Client-Side Applications". In: *Proc. of Middleware*. 2015.

[115] Irene Zhang et al. "Diamond: Automating Data Management and Storage for Wide-Area, Reactive Applications". In: *Proc. of OSDI*. 2016.

KRISTINA SPIROVSKA

tina.spirovska@gmail.com
Lausanne, Switzerland
www.linkedin.com/in/kristina-spirovska

WORK EXPERIENCE

Research and teaching assistant École Polytechnique Fédérale de Lausanne	Lausanne, Switzerland	Sep 2014 – Present
MTS Intern at Nutanix Objects Team	San Jose, CA, USA	Nov 2019 – Jan 2020
Research Intern at Intel Labs Systems and Software Research Lab	Hillsboro, OR, USA	Jun 2016 – Sep 2016
Research and teaching assistant Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University	Skopje, Macedonia	Jan 2011 – Jun 2013
Teaching student assistant Institute of Informatics, Faculty of Natural Sciences and Mathematics, Ss. Cyril and Methodius University	Skopje, Macedonia	Jun 2008 – Dec 2010

EDUCATION

PhD in Computer Science École Polytechnique Fédérale de Lausanne	GPA: 5.25 / 6	2014 – Present
MSc in Engineering of Intelligent Systems Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Skopje	GPA: 10 /10	2011 – 2013
BSc in Computer Science Institute of Informatics, Faculty of Natural Sciences and Mathematics, Ss. Cyril and Methodius University, Skopje	GPA: 9.88 / 10	2005 – 2009

TECHNICAL SKILLS

PROGRAMMING LANGUAGES	WEB TECHNOLOGIES	ML TOOLS	DATABASE
C++, Python, Java	HTML, CSS, JQuery, Bootstrap, Flask	Mathematica, MATLAB, Shogun	SQL, MySQL

PEER-REVIEWED PUBLICATIONS

PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication, *Kristina Spirovska*, *Diego Didona and Willy Zwaenepoel;* The 39th IEEE International Conference on Distributed Computing Systems (**ICDCS'19**), Dallas, Texas, July 2019		2019
Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store, *Kristina Spirovska*, *Diego Didona and Willy Zwaenepoel;* The 48th International Conference on Dependable Systems and Networks (**DSN'18**), Luxembourg, June 2018	Best Paper Award	2018
Optimistic Causal Consistency for Geo-Replicated Key-Value Stores, *Kristina Spirovska, Diego Didona and Willy Zwaenepoel*; The 37th IEEE International Conference on Distributed Computing Systems (**ICDCS'17**), Atlanta, GA, USA, June 2017		2017
Improvement of the Generic Classification Model based on a Multiple Kernel Data Fusion, *Kristina Spirovska and Ana Madevska Bogdanova;* The 10th International Conference for Informatics and Information Technology (CIIT 2013), Bitola, Macedonia, April 2013		2013
Model of a Generic Classification System based on a Multiple Kernel Data Fusion, *Kristina Spirovska and Ana Madevska Bogdanova* ICT Innovations 2012, Ohrid, Macedonia, September 2012		2012
Modeling of the Immune System Using Pray and Predator Model, *Kristina Spirovska and Nevena Ackovska*, The 8th International Conference for Informatics and Information Technology (CIIT 2011)		2011

RESEARCH PROJECTS

Partially Replicated Causally Consistent Transactions
Fall 2018 – Present

Designing, implementing and evaluating novel transactional causally consistent data stores with partial geo-replication.

Technology: C++, Protobuf, C++ Bootstrap, Python, Boto, AWS, Docker

Transactional Causally Consistent Geo-replicated Key-value Stores
2017 – 2018

Designing, implementing and evaluating a novel transactional causally consistent geo-replicated key-value data store.

Technology: C++, Protobuf, C++ Bootstrap, Python, Boto, AWS, Docker

Optimistic Geo-replication Protocols
2015 – 2017

Designing, implementing and evaluating a novel optimistic protocol for geo-replication with causal consistency.

Technology: C++, Protobuf, C++ Bootstrap, Python, Boto, AWS, Docker

Implementing and Evaluating Chain Replication on a Variant of COPS
Spring 2015

Enhancing and evaluating an optimized version of COPS using chain replication.

Technology: C++, Protobuf, C++ Bootstrap, Python

Transforming Memcached with SwissTM Transactional Memory
Fall 2014

Transforming the lock-based implementation of Memcached to a lock-free implementation using SwissTM transactional memory.

Technology: C++, Memcached, SwissTM

Generic Distributed Classification System based on a Multiple Kernel Data Fusion
2012 – 2013

Designing, implementing and evaluating a distributed classification system that does data preprocessing, intelligent choice of SVM-based classification methods, proposes a solution and automatically tunes the parameters.

Technology: Python, Schogun, Flask, HTML, JQuery, Bootstrap

Building a Model of the Immune System Using Prey and Predator Model
2011 – 2012

Designing a novel mathematical model of the immune system, based on a "prey and predator" method.

VOLUNTEERING

GirlsCoding Mentor	Lausanne, Switzerland	2017 – present
Member of NGO "Computer Science Society"	Skopje, Macedonia	2008 – 2014

ACADEMIC HONORS – CERTIFICATES - RECOGNITIONS

EPFL PhD Fellowship	2014 – 2015
Recognition for achieved results as a mentor of students from "Kuzman Josifovski Pitu" Elementary school, Skopje of XXII State competition of Informatics	2007 – 2009
Academic honors for exceptional academic achievements during the four years of bachelor studies at Institute of Informatics at Ss. Cyril and Methodius University, Skopje	2006 – 2009
State scholarship for talented students from the Government of Macedonia	2006 – 2007

LANGUAGES

English	●●●●●	French	●●○○○
German	●○○○○	Spanish	●○○○○
Macedonian	●●●●●	Serbian	●●●○○