CONSISTENCY FOR DISTRIBUTED DATA STORES

By

Mohammad Roohitavaf

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science – Doctor of Philosophy

2019

ABSTRACT

CONSISTENCY FOR DISTRIBUTED DATA STORES

By

Mohammad Roohitavaf

Geo-replicated data stores are one of the integral parts of today's Internet services. Service providers usually replicate their data on different data centers worldwide to achieve higher performance and data durability. However, when we use this approach, the consistency between replicas becomes a concern. At the highest level of consistency, we want strong consistency that provides the illusion of having only a single copy of the data. However, strong consistency comes with high performance and availability costs. In this work, we focus on weaker consistency models that allow us to provide high performance and availability while preventing certain inconsistencies. Session guarantees (aka. client-centric consistency models) are one of such weaker consistency models that prevent some of the inconsistencies from occurring in a client session. We provide modified versions of session guarantees that, unlike traditional session guarantees, do not cause the problem of slowdown cascade for partitioned systems. We present a protocol to provide session guarantees for eBay NuKV that is a key-value store designed for eBay's internal services with high performance and availability requirements. We utilize Hybrid Logical Clocks (HLCs) to provide wait-free write operations while providing session guarantees. Our experiments, done on eBay cloud platform, show our protocol does not cause significant overhead compared with eventual consistency. In addition to session guarantees, a large portion of this dissertation is dedicated to causal consistency. Causal consistency is especially interesting as it is has been proved to be the strongest consistency model that allows the system to be available even during network partitions. We provide CausalSpartanX protocol that, using HLCs, improves current time-based protocols by eliminating the effect of clock anomalies such as clock skew between servers. CausalSpartanX also supports non-blocking causally consistent read-only transactions that allow applications to read a set of values that are causally consistent with each other. Read-only transactions provide a powerful

abstraction that is impossible to be replaced by a set of basic read operations. CausalSpartanX, like other causal consistency protocols, assumes sticky clients (i.e. clients that never change the replica that they access). We prove if one wants immediate visibility for local updates in a data center, clients have to be sticky. Based on the structure of CausalSpartanX, we provide our Adaptive Causal Consistency Framework (ACCF) that is a configurable framework that generalizes current consistency protocols. ACCF provides a basis for designing adaptive protocols that can constantly monitor the system and clients' usage pattern and change themselves to provide better performance and availability. Finally, we present our Distributed Key-Value Framework (DKVF), a framework for rapid prototyping and benchmarking consistency protocols. DKVF lets protocol designers only focus on their high-level protocols, delegating all lower level communication and storage tasks to the framework.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

With millions of users across the world accessing the data in a real-time manner, no major internet service provider can rely on a single copy of its data stored at only one location. To achieve higher performance and availability as well as to increase the durability of the data, most service providers create replicas of their data on different geographical locations–the technique that is known as geo-replication. Geo-replication lets clients query their own local replica thereby reducing the network latency. It also improves the availability and durability of the data, as if one of the replicas fails, the data is still available in other replicas.

Once we create multiple copies of a data item, we need to deal with the consistency among replicas. At the highest level of consistency, we want all replicas to always provide the same data. This level of consistency is called *strong consistency*. Strong consistency lets application developers write the code as if they were writing the code for a system with only one copy of the data. This illusion significantly makes the coding and debugging easier. However, strong consistency comes with its high availability and performance costs. Based on Eric Brewer's conjecture [24], Seth Gilbert and Nancy Lynch [38] proved that it is impossible to have strong consistency and availability in presence of network partitions that prevent some nodes from communicating with each other. This theorem is known as the CAP theorem. To realize why achieving strong consistency and availability is impossible in presence of network partitions consider the following scenario:

Suppose we have data item X replicated in two replicas R_0 and R_1. Suppose there is a network partition preventing R_0 and R_1 from communication, i.e., delivery of any message between R_0 and R_1 is indefinitely delayed. Now, suppose a client writes value V for data item X at replica R_0. After this write, another client reads X from replica R_1. The availability requires to never reject a user request. In other words, all user requests must be satisfied in a finite time. Thus, replica R_1 must answer client's request in a finite time. Since, the communication between R_0 and R_1 is broken, R_1 does not know the value V. Thus, any value returned by R_1 violates strong consistency.

The CAP theorem is about the conflict between availability and strong consistency in presence of network partitions. However, even when there is no partition, we have an inherent conflict between performance and consistency as well. Specifically, to achieve strong consistency, replicas need to communicate with all other replicas to perform a read/write operation. Waiting for all replicas to respond increases the response time for clients which makes strong consistency inefficient for many modern applications such as real-time web applications. To achieve higher availability and performance, most of the current systems sacrifice strong consistency providing much weaker consistency guarantees. Many of new databases such as [10, 17, 27, 31, 44] provide *eventual consistency* which only guarantees that in the absence of new writes if replicas are connected to each other, all replicas eventually become consistent (i.e., provide the same data). Although eventual consistency provides high availability and performance, it causes some data anomalies that make the programming for application developers complex. When coding on top of an eventually consistent system, application developers need to take these data anomalies into consideration which certainly makes coding and debugging challenging. For an example of such data anomalies, consider this example for a social networking application: Suppose Alice uploads a photo and then adds it to an album. The state of the album is stored in one data object, and the value of the photo is stored in another data object. Under eventual consistency, it is possible for a remote replica to receive and make the value of the album visible before receiving the photo. In this situation, the album refers to a photo that the replica has not received. It is not desired, as clients see an album that is referring to a photo, but once they try to see the photo, the system provides no photo.

From discussion above, we can see that at one end, we have strong consistency which provides the easiest interface for application developers, but is expensive in terms of availability and performance. On the other end, we have eventual consistency which has the least availability and performance overhead but does not provide any consistency guarantee except the eventual convergence of the data. In this work, we focus on somewhere in between. Specifically, we focus on consistency models weaker than strong consistency and stronger than eventual consistency that provide high performance and availability as well as preventing some inconsistencies from occurring. Session

guarantees (aka. client-centric consistency models) are one of such weaker consistency models. Session guarantees do not provide perfect consistency as strong consistency, but they prevent some inconsistency in a client session. For instance, monotonic-read prevents a client's view from going backward in time by seeing a version that is older than a version that the client has already seen. As another example, read-your-write guarantees that all versions written in a client session are always visible in that session. These levels of consistency are not guaranteed by eventual consistency.

An obstacle in providing consistency guarantees stronger than eventual consistency is the problem of *slowdown cascades* [19]. The slowdown cascade refers to a situation where the slowdown of one component in a system affects the performance of other components. In case of consistency protocols, slowdown cascade can delay the visibility of updates in all machines because of the slowdown of only one of the servers inside a data center. To prevent slowdown cascades, we provide *per-key* versions of session guarantees that define session guarantee for individual keys which allows us to remove the need for communication between partitions insides a data center thereby eliminating the possibility of slowdown cascades in the system.

We present a protocol to provide session guarantees for eBay NuKV that is a key-value store designed for eBay's internal services with high performance and availability requirements. We utilize Hybrid Logical Clocks (HLCs) in our protocol. HLCs are a combination of conventional logical clocks [45] and physical clocks. The timestamps assigned by HLCs are close to the physical clocks, but at the same time, allows us to provide wait-free write operations while satisfying per-key write session guarantees. Our experiments, done on eBay cloud platform, show our protocol does not cause significant overhead compared with eventual consistency while providing session guarantees.

In recent years, many researchers have focused on another intermediate consistency model called *causal consistency*. What makes causal consistency especially interesting is that causal consistency has been proved to be the strongest consistency model that allows the system to be available even during network partitions [53]. Causal consistency is based on the notion of happens-before relation defined by Lamport in his seminal work [45]. The happens-before relation defines

a partial order for events occurred in a distributed system. We will define happens-before relation in details in Section 2.3, but basically the goal of happens-before relation is to capture the *potential* causality between events. For example, if event e and f are both executed by the same client, and e has executed earlier than f, we say e has happened before f. Now, if e and f are both write operations, we call the value written by event e a *causal dependency* of the value written by event f. For instance, in Alice's photo album explained above, the photo is a causal dependency of the value of the album referring to the photo, because Alice first uploads the photo, and then adds it to the album. Causal consistency guarantees that if a value is visible, its causal dependencies are also visible. In other words, we do not make a value visible to the clients, unless we made sure that we have received all of its causal dependencies. In Alice's example, the remote replica does not make the album referring to the photo visible until it receives the photo.

The broad approach for providing causal consistency is to track the causal dependencies of a version and check them before making the version visible in another replica. Some causal consistency protocols do the dependency tracking explicitly; for each new update, we have a list of all versions that it causally depends on. Once a replica receives the new update, it checks all versions in its dependency list. Explicit dependency tracking becomes challenging when clients read many values before writing new ones. This leads to large dependency metadata that causes performance issues. Another approach is implicit dependency tracking via timestamping versions. Specifically, we assign each version a timestamp that captures its causal dependencies. This information later will be used by servers to decide whether to make a version visible.

We provide a causal consistency protocol with implicit dependency tracking named CausalSpartanX. Since achieving perfectly synchronized clocks in distributed systems is not practical, existing time-based implicit protocols such as GentleRain [36] suffer from clock skew between different machines. CausalSpartanX solves this issue by using HLCs instead of physical clocks. CausalSpartanX, like other causal consistency protocols, assumes sticky clients (i.e. clients never change the replica that they access). We prove if one wants immediate visibility for local updates in a data center, clients have to be sticky. CausalSpartanX also supports causally consistent read-only trans-

actions that allow applications to read a set of values that are causally consistent with each other. Read-only transactions provide a powerful abstraction that is impossible to be replaced by a set of basic read operations. Our read-only transaction algorithm is non-blocking (i.e. the servers involved in the transaction can read the requested data as soon as they received the request), and requires only one round of client-server communication.

Existing protocols for causal consistency, including our own CausalSpartanX, utilize a static approach for tracking and checking dependencies. They are agnostic about the actual usage patterns of clients and treat all clients the same. This static approach penalizes some applications while assisting others. We define the notion of tracking and checking groups as a way to study different causal consistency protocols. We provide our Adaptive Causal Consistency Framework (ACCF) that generalizes existing causal consistency protocols and can be configured to work with different tracking and checking groups. The flexibility of ACCF provides us with a basis to create adaptive causal consistency protocols that monitor the system and adapt themselves. Using ACCF we are also able to treat different clients in different ways.

As mentioned above, consistency protocols for replicated data stores has received much attention in recent years. When the developers intuitively identify a new approach to design such a protocol, the natural question that arises is how to evaluate the new protocol by comparing it with different existing protocols. Distributed data stores are complex systems which makes an accurate analytical performance evaluation infeasible for them. A more practical option is experimental performance evaluation via benchmarking a prototype running the protocol. However, implementing a prototype could be a time-consuming task. To facilitate this process, we have implemented a general framework called Distributed Key-Value Framework (DKVF) that lets a protocol designer conveniently implement a prototype system running a given protocol by writing a piece of code that only focus on the high-level protocol. The protocol designer can delegate all lower-level tasks such as storage and network communication to DKVF. DKVF also comes with a toolset that lets protocol designers easily experiment their prototypes. They can visually design systems running their protocol, run them on the cloud, and perform their experiments. Figure 1.1 shows highlights

of the dissertation in a glance.

Organization: In Chapter 2, we provide the preliminaries including the system assumptions, definitions, and a review on HLCs. Chapter 3 reviews related work. Chapter 4 is dedicated to session guarantees. In Chapter 5, we provide our CausalSpartanX protocol. Chapter 6 provides the proof for the necessity of the sticky clients. The need for adaptive causal consistency together with our ACCF are presented in Chapter 7. We introduce our DKVF in Chapter 8. Chapter 9 provides the future work. Finally, Chapter 10 concludes the dissertation.

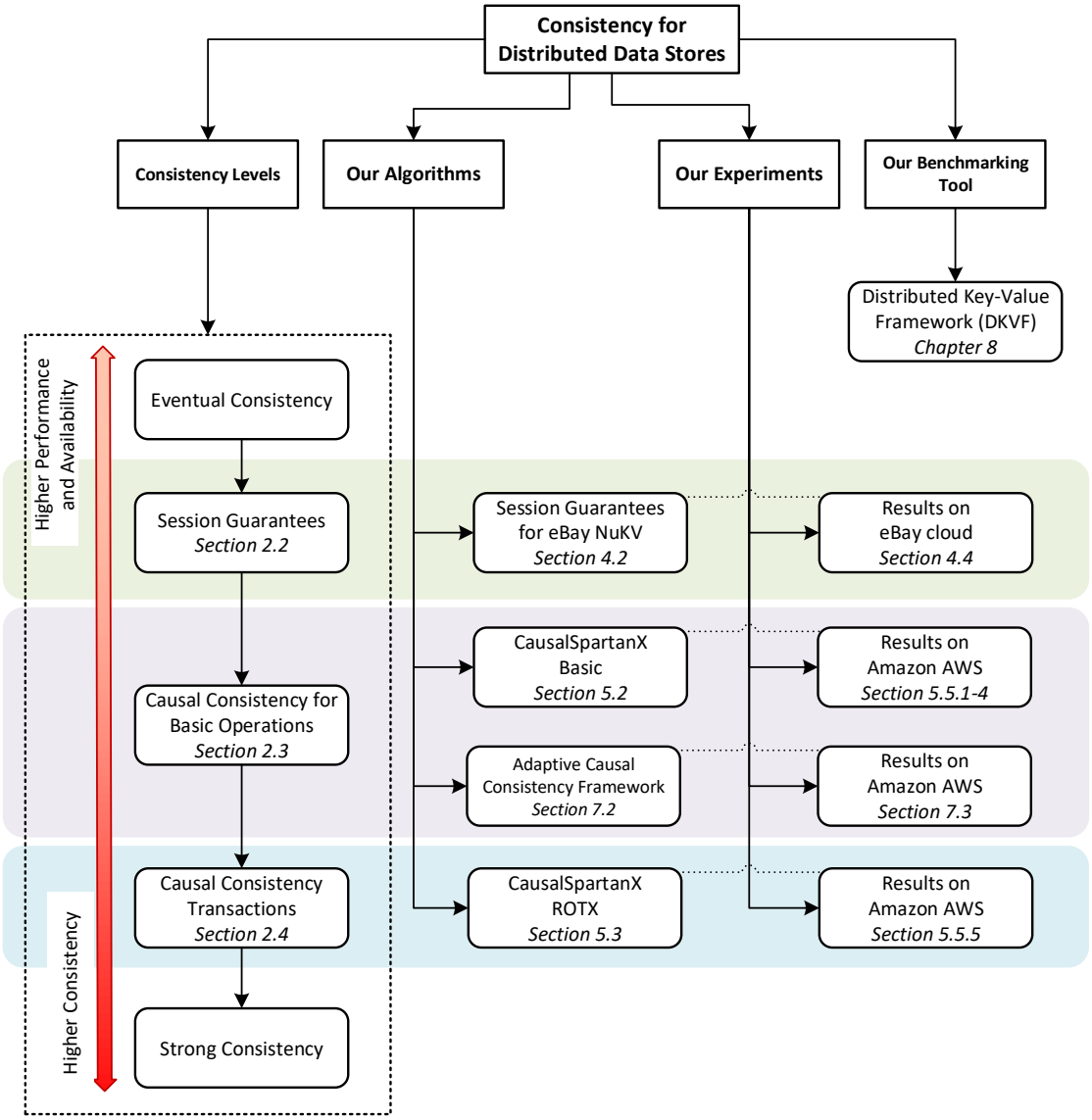


Figure 1.1: Highlights in a glance

CHAPTER 2

PRELIMINARIES

In this chapter, we provide the background that is needed to understand the rest of this dissertation. In Section 2.1, we provide system architecture and assumptions that are the same as the majority of consistency protocols for key-value stores such as [35, 36, 49, 50]. The consistency protocols presented in Chapters 4 and 5 are based on these assumptions. However, in Chapter 7, we explain the need for a flexible architecture that can adapt itself based on actual usage pattern of the clients. Sections 2.2 and 2.3 provide formal requirements for session guarantees and causal consistency, respectively. In Section 2.4, we focus on the requirements for causally consistent read-only transactions. All of the proposed protocols in this dissertation utilize Hybrid Logical Clocks (HLCs). Thus, in Section 2.5, we review HLCs and explain why we utilize them.

2.1 Architecture

We consider a data store whose data is fully replicated into M data centers (i.e., replicas) where each data center is partitioned into N partitions (see Figure 2.1). Partitioning the data means that the data is distributed across multiple machines. Partitioning increases the scalability of a data store, as the size of the data store does not depend upon the capacity of a machine. Replication, on the other hand, increases performance and durability. Specifically, by replicating data in geographically different locations, we let the clients query their local data center for reading or writing data thereby reducing the response time for client operations. In addition, by replicating data, we increase the durability of our data in case of failures at one data center. Like [35,36,49,50], our CausalSpartanX protocol assumes that a client does not access more than one data center. We prove the necessity of this assumption for causal consistency in Chapter 6. NuKV, discussed in Chapter 4, lets clients change their data centers. There might be network failure *between* data centers that causes network partitions. We assume network failure do not happen *inside* data centers. Thus, partitions inside a data center can always communicate with each other.

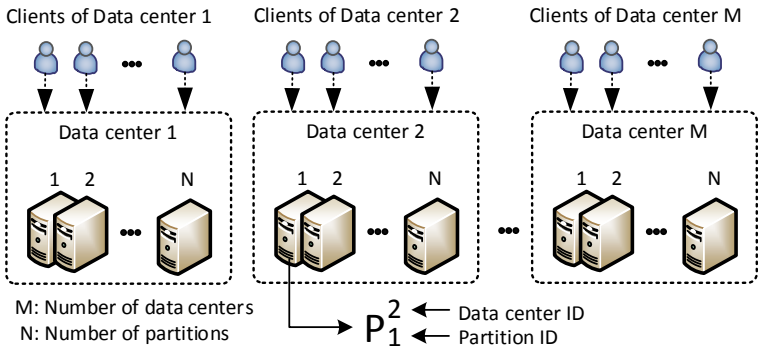


Figure 2.1: A system consisting of M data centers (replicas) each of which consists of N partitions. p_n^m denotes nth partition in mth data center.

We assume multi-version key-value stores that store several versions for each key. A key-value store has two basic operations: $PUT(k, val)$ and $GET(k)$, where $PUT(k, val)$ writes new version with value val for item with key k, and $GET(k)$ reads the value of an item with key k. In addition to basic PUT and GET operations, a key-value store can provide read-only transactions denoted as ROTX. $ROTX(kset)$ returns the value of the set of keys given in $kset$.

2.2 Session Guarantees

In this section, we define various session guarantees for key-value stores. We provide modified versions of session guarantees considered in [65]. This modified versions define session guarantees per keys, which allows us to avoid the problem of slowdown cascade [19] by avoiding cross-partition communications. Specifically, to satisfy definitions provided here, different partitions can check session guarantees independently. Thus, failure/slowdown of a partition does not affect the visibility of updates in other partitions.

First, regarding writes and reads of a specific key we consider following definitions:

Definition 1 *$CommittedWrites(s, k, t)$ is the set of all writes committed for key k at server s at time t.*

Definition 2 *$ClientWrites(c, k, t)$ is the set of all writes done by client c for key k at time t.*

Definition 3 *$ClientsReads(c,k,t)$ is the set of all writes that have written a value for key k read by client c at time t.*

Now, using above definitions, we define various per-key session guarantees as follows:

Definition 4 (Per-key Monotonic-Read Consistency) *Let O be an operation issued by client c on server s at time t that reads the value of key k. Operation O satisfies monotonic-read consistency if any $w \in ClientReads(c,k,t)$ is included in $CommittedWrites(s,k,t')$ where t' is the time when server s actually reads the value for key k.*

Monotonic-read consistency is important for the clients, as the lack of monotonic-read causes user confusion by going backward in time. For example, consider a webmail application. Monotonic-read consistency guarantees that the user is always able to see all emails that has seen before. New emails may be added to the mailbox in the next time that the user checks their mailbox, but no email can disappear once the user saw it, unless the user wants to delete it. This is not guaranteed with only eventual consistency.

Monotonic-read consistency guarantees that the client never misses what it has seen so far. On the other hand, read-you-write consistency guarantees that the client never misses what it has written so far. Specifically,

Definition 5 (Per-key Read-your-write Consistency) *Let O be an operation issued by client c on server s at time t that reads the value of key k. Operation O satisfies read-your-write consistency if any $w \in ClientWrites(c,k,t)$ is included in $CommittedWrites(s,k,t')$ where t' is the time when server s actually reads the value for key k.*

For instance, users must be able to see the posts that they posted on their social network page. This seems to be obvious for a centralized system, but unfortunately, with eventual consistency, it may be violated if the client is routed to another replica when it wants to read the data.

Definition 6 (Per-key Monotonic-write Consistency) *Let O be an operation issued by client c on server s at time t that writes a value for key k. Operation O satisfies monotonic-write consistency if*

for any server s at any time t' if $CommittedWrites(s,k,t')$ includes O, no client accessing s reads a value written by write $w \neq O$ included in $ClientWrites(c,k,t)$.

Suppose a user updates his/her password two times. Per-key monotonic-write consistency together with eventual consistency guarantees that eventually, the user is able to login to the system with the latest passwords in all replicas. Note that just eventual consistency does not guarantee the second version to be the winner version.

Definition 7 (Per-key Write-follows-reads Consistency) *Let O be an operation issued by client c on server s at time t that writes a value for key k. Operation O satisfies write-follows-read consistency if for any server s at any time t' if $CommittedWrites(s,k,t')$ includes O, no client accessing s reads a value written by write $w \neq O$ included in $ClientReads(c,k,t)$.*

As an example for the necessity of per-key write-follows-reads consistency, consider a shared document in a cloud-based document processing service. Suppose one of the contributors reads the current version of a document and then appends a line to the document. To append the line, the application reads the current value of the data object associated with the content of the document, appends the line, and writes the new content of the document back to the system. The per-key write-follows-reads consistency guarantees that in all replicas, the version of the document with the appended line is the winner version.

With Definitions 6 and 7, we can trivially satisfy monotonic-write and write-follows-reads consistency by never returning any committed version. To avoid such a trivial implementation, we assume the following requirements are implicitly required for all definitions:

R1 Any write for a key committed on a server will be eventually committed on all servers hosting that key.

R2 Clients reads only committed versions.

2.3 Causal Consistency

Causal consistency is defined based on the happens-before relation between events [45]. In the context of key-value stores, we define happens-before relation as follows:

Definition 8 (Happens-before) *Let e and f be two events. We say e happens before f, and denote it as $e \rightarrow f$ iff:*

- *e and f are two events by the same client (i.e., in a single thread of execution), and e happens earlier than f, or*
- *e is a $PUT(k, val)$ operation, and f is a $GET(k)$ that returns the value written by e, or*
- *there is another event c such that $e \rightarrow c$ and $c \rightarrow f$.*

Now, we define causal dependency as follows:

Definition 9 (Causal Dependency) *Let v_1 be a version of key k_1, and v_2 be a version of key k_2. We say v_1 causally depends on v_2, and denote it as v_1 dep v_2 iff $PUT(k_2, v_2) \rightarrow PUT(k_1, v_1)$.*

For example, suppose that a user writes a comment for a post on a social network. Since the user, first reads the post, and then writes the comment, the comment causally depends on the post.

A data store is causally consistent if it satisfies these conditions: 1) when a client reads a version, it always remains visible to the client, 2) writes by a client must be immediately visible to the client, and 3) when a version is visible, all of its causal dependencies are also visible.

The definition of causal consistency depends upon the nature of operations allowed on the key-value store. It is possible that the addition of an operation violates the causal consistency provided by the key-value store. For example, if the key value store only supported GET operation and causal consistency is satisfied, adding a new operation such as ROTX can potentially violate causal consistency as the addition of ROTX creates new constraints that have to be satisfied as far as causal consistency is concerned. For this reason, the definitions in this paper are parameterized with a set of operations permitted on the key-value store. This approach allows us to make the definition

generic so that if new operations (e.g., read/write transactions) are added, then the definition can be extended to them. It also allows us to make the proofs in a modular fashion where we can prove correctness with respect to GET and ROTX separately and conclude the correctness of the system that supports both.

Next, we define the notion of visibility for an operation that captures whether a given version (or some concurrent/more recent version) is returned by the operation.

Definition 10 (Visibility) *We say version v of key k is visible for set of operations O to client c iff any operation in O reading k performed by client c returns v' such that $v' = v$ or $\neg(v \ \mathsf{dep} \ v')$.*

Note that in the definition above, O is a set of operation types. For instance, a possible O can be $\{GET\}$. In this case, visible for O means, any $GET(k)$ operation performed by a client returns v' such that $v' = v$ or $\neg(v \ \mathsf{dep} \ v')$. Instead of $\{GET\}$, O could be $\{GET, ROTX\}$. Now, visible for O means, any $GET(k)$ or $ROTX(K)$ that reads k returns v' such that $v' = v$ or $\neg(v \ \mathsf{dep} \ v')$. This way, we can flexibly define visibility for different sets of operations.

Using visibility, we define causal consistency as follows:

Definition 11 (Causal Consistency) *Let k_1 and k_2 be any two arbitrary keys in the store. Let v_1 be a version of key k_1, and v_2 be a version of key k_2 such that $v_1 \ \mathsf{dep} \ v_2$. Let O and R be two sets of operations. We say the store is causally consistent for set of operations O with set of reader operations R, if for any client c conditions below hold:*

- *Any version written by client c is visible for O to c.*
- *Once c reads a version by one of operations of R, the version remains visible for O to c.*
- *If c that has read v_1 by one of operations of R, v_2 is visible for O to c.*

In the above definitions, we ignore the possibility of conflicts in writes. Conflicts occur when we have two writes on the same key such that there is no causal dependency relation between them. For example, when two clients independently write to a key without reading the update made by the other client.

Definition 12 (Conflict) *Let v_1 and v_2 be two versions for key k. We call v_1 and v_2 are conflicting iff $\neg(v_1 \ \mathsf{dep} \ v_2)$ and $\neg(v_2 \ \mathsf{dep} \ v_1)$. (i.e., none of them depends on the other.)*

In case of conflict, we want a function that resolves the conflict. Thus, we define conflict resolution function as $f(v_1, v_2)$ that returns one of v_1 and v_2 as the winner version. If v_1 and v_2 are not conflicting, any f returns the latest version with respect to the causal dependency, i.e., if v_1 dep v_2 then $f(v_1, v_2) = v_1$. Now, we define the notion of visibility that also captures conflicts:

Definition 13 (Visibility+) *We say version v of key k is visible+ for set of operations O for conflict resolution function f to client c iff any operation of O performed by client c reading k returns v' such that $v' = v$ or $v' = f(v, v')$.*

Definition 14 (Causal+ Consistency) *Let k_1 and k_2 be any two arbitrary keys in the store. Let v_1 be a version of key k_1, and v_2 be a version of key k_2 such that v_1 dep v_2. Let O and R be two sets of operations. We say the store is causal+ consistent for set of operations O with set of reader operations R for conflict resolution function f if for any client c conditions below hold:*

- *Any version written by client c is visible+ for O for f to c.*
- *Once c reads a version by one of operations of R, the version remains visible+ for O for f to c.*
- *If c that has read v_1 by one of operations of R, v_2 is also visible+ for O for f to c.*

To achieve the least update visibility latency for local updates, we define causal++ consistency that requires the data store to make all local updates visible to clients immediately.

Definition 15 (Causal++ Consistency) *Let O, R, and I be three sets of operations. A store is causal++ consistent for set of operations O with set of reader operations R and set of immediate-visible operations I for conflict resolution function f if conditions below hold:*

- *The store is causal+ consistent for O with set of reader operations R for f.*

- *Any version v written in data center r is immediately visible+ for operations of I for f to any client accessing r.*

In practice, in addition to the consistency, we want all data centers to eventually converge to the same data. In other words, we want an update occurred in a data center to be reflected in other connected data centers as well. We define two data centers connected, if there is no network partition that prevents them from communication. Now, we define convergence as follows:

Definition 16 (Convergence) *Let v_1 be a version for key k written in data center r.*

- *Let data center r' be continuously connected to data center r, and*
- *for any version v_2 such that v_1 dep v_2, let data center r' be continuously connected to data center r'' where version v_2 is written.*

The data store is convergent+ for set of operations O for conflict resolution function f if v_1 is eventually visible+ for O for f to any client accessing r'.

With the definitions of causal++ consistency and convergence defined in this section, it is straightforward to note the following observations:

Observation 1 *If store S is*

- *causal++ consistent for O, with set of reader operations R_1, and set of instantly visible operations I for conflict resolution f, and*
- *causal++ consistent for O, with set of reader operations R_2, and set of instantly visible operations I for conflict resolution f,*

then, S is causal++ consistent for O, with set of reader operations $R_1 \cup R_2$, and set of instantly visible operations I for conflict resolution f.

Observation 2 *If store S is*

- *causal++ consistent for O_1, with set of reader operations R, and set of instantly visible operations I for conflict resolution f, and*
- *causal++ consistent for O_2, with set of reader operations R, and set of instantly visible operations I for conflict resolution f,*

then, S is causal++ consistent for $O_1 \cup O_2$, with set of reader operations R, and set of instantly visible operations I for conflict resolution f.

Observation 3 *If store S is*

- *causal++ consistent for O, with set of reader operations R, and set of instantly visible operations I_1 for conflict resolution f, and*
- *causal++ consistent for O, with set of reader operations R, and set of instantly visible operations I_2 for conflict resolution f,*

then, S is causal++ consistent for O, with set of reader operations R, and set of instantly visible operations $I_1 \cup I_2$ for conflict resolution f.

Observation 4 *If store S is*

- *convergent for set of operations O_1 for conflict resolution function f, and*
- *convergent for set of operations O_2 for conflict resolution function f,*

then, it is convergent for set of operations $O_1 \cup O_2$ for conflict resolution function f.

2.4 Causally Consistent Read-only Transactions

In this section, we focus on causally consistent read-only transactions. We, first, discuss the motivation for this operation, and then provide requirements for causally consistent read-only transactions.

A causally consistent read-only transaction is a powerful abstraction that can significantly help application developers when working with replicated data stores. A read-only transaction allows application developers to read a set of keys such that the returned versions of the key values are causally consistent with each other as well as with previous reads of the application. To understand the benefit of such abstraction, consider the following example.

Consider a social network such as Facebook where profile pictures are always public. Alice wants to update her profile picture, but she does not want Bob to see her new picture. Since profile pictures are public, the only way for Alice to hide her picture from Bob is to completely block Bob. Thus, she first blocks Bob and then updates her picture (we call this change scenario 1). A data store with only causally consistent PUT and GET operations (without read-only transactions) guarantees that no matter which replicas Alice and Bobs are connected to, the new Alice's picture is visible only when Bob is blocked by Alice. However, it is *not* enough to protect Alice's privacy. Suppose the application first reads Bob's status and finds it unblocked, then it reads Alice's new profile picture. Since it found Bob unblocked, it shows Alice's new picture to Bob that is not acceptable. With a causally consistent read-only transaction, the application could read both values such that they are causally consistent with each other. Thus, the application either 1) reads Alice's old picture, or 2) it reads Alice's new picture, but finds Bob blocked, and both of these cases are acceptable.

Now, suppose after some time, Alice changes her mind. She changes her photo to the old one and then unblocks Bob (we call this change scenario 2). A causally consistent read-only transaction still protects Alice's privacy after this change. With two independent causally consistent GET operations it is impossible to protect Alice's privacy in both cases. Specifically, based on the order that the application issues the two GET operations, it violates Alice's privacy either in scenario 1 or scenario 2.

We formally define this requirement as follows. First, we define visiblity+ for a set of versions:

Definition 17 (Visible+ to a Set of Versions) *Let $vset$ be a set of version. We say version v is visible+ for conflict resolution function f to $vset$, if $vset$ does not include version v' such that v*

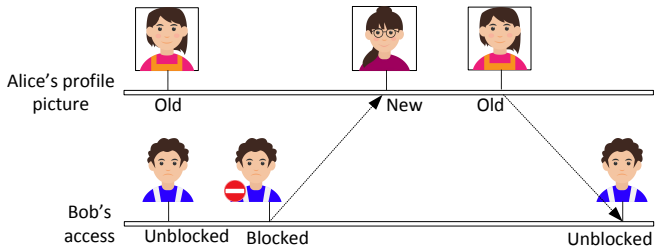


Figure 2.2: Updating profile picture in a social network while blocking another user.

and v' *are two version for the same key,* $v' \neq v$*, and* $v = f(v, v')$*.*

Now, we define causal consistency for a set of versions:

Definition 18 (Causal Consistency for a Set of Versions) *Let* $vset$ *be a set of versions.* $vest$ *is causally consistent for conflict resolution function* f*, if for any version* $v_1 \in vset$*, any version* v_2 *such that* v_1 *dep* v_2 *is visible+ for* f *to* $vest$*.*

2.5 Hybrid Logical Clocks

In this section, we recall HLCs from [42]. HLCs allow us to capture *happens-before* relation [45] while assigning timestamps that are very close to the physical clocks. The HLC timestamp of an event e, is a tuple $\langle l.e, c.e \rangle$. $l.e$ is our best approximation of the physical time when e occurs. $c.e$ is a bounded counter that is used to capture causality whenever $l.e$ is not enough to capture causality. Specifically, if we have two events e and f such that e happens-before f and $l.e = l.f$, to capture causality between e and f, we set $c.e$ to a value higher than $c.f$.

Why using HLCs? We want to timestamp versions such that two following requirements are satisfied:

1. timestamps are close (within clock drift error) to the physical time when the event of writing occurs, and

2. timestamps capture happens-before relation

Logical clocks [45] fail to satisfy the first requirements. Specifically, logical timestamps have no relation to the wall clock time. This causes several problems. First, we cannot use timestamps to provide clients with the version of a data object at a given physical time. More importantly, we cannot resolve conflicting writes based on timestamp. For instance, assume a client writes version v for a data object on replica A. One hour later, another client writes version v' for the same data object on replica B. With logical timestamps, it is possible that the timestamp assigned to v' is smaller than the version assigned to version v. In this situation, if we use timestamps to select the winner version (i.e. the version with higher timestamp is the winner), v will be selected as the winner which is not desired, as we want the version written one hour later to be the winner.

On the other hand, physical clocks fail to satisfy the second requirement. This is due to the fact that perfect clock synchronization is impossible. Thus, the clock skew between different replicas prevents physical clocks from accurately capturing happens-before relation. For instance, imagine a client writes version v for a data object on replica A. Next, the client write version v' on replica B. While using logical clocks we can guarantee that the timestamp assign to v' is higher than that of assigned to v, using physical clocks it is not guaranteed. Specifically, if the clock of replica B is behind A, the timestamp assigned to v' may be smaller than that of v.

HLCs solves both issues explained above, i.e., it allows us to capture the happens-before relation, and at the same time, allows us to provide conflict resolution with respect to the physical time. For completeness, we recall algorithm of HLC from [42] below.

Algorithm 2.1 HLC algorithm

1: **Upon sending a message or local event by process** a
2: $l'.a = l.a$
3: $l.a = max(l'.a, pt.a)$ //tracking maximum time event, $pt.a$ is physical time at a
4: **if** $(l.a = l'.a)$ $c.a = c.a + 1$ //tracking causality
5: **else** $c.a = 0$
6: Timestamp event with $l.a, c.a$

7: **Upon receiving message** m **by process** a
8: $l'.a = l.a$
9: $l.a = max(l'.a, l.m, pt.a)$ //$l.m$ is l value in the timestamp of the message received
10: **if** $(l.a = l'.a = l.m)$ then $c.a = max(c.a, c.m) + 1$
11: **else if** $(l.a = l'.a)$ then $c.a = c.a + 1$
12: **else if** $(l.a = l.m)$ then $c.a := c.m + 1$
13: **else** $c.a = 0$
14: Timestamp event with $l.a, c.a$

CHAPTER 3

RELATED WORK

In this chapter, we review some of the consistency protocols for distributed data stores. We review related work for session guarantees in Section 3.1. In Sections 3.2 and 3.3 we review some of causal consistency protocols for basic operations and read-only transaction, respectively. In addition to session guarantees and causal consistency, we also review two of popular consistency protocols namely eventual consistency and strong consistency in Section 3.4.

3.1 Existing Work on Session Guarantees

Terry et al. provided a protocol for providing session guarantees in [65]. An important improvement over [65], in our work, is utilizing HLCs to avoiding blocking write operations. Also, we consider the per-key versions of session guarantees considered in [65]. This allows us to avoid the overhead of inter-partition communication which in turns eliminates the possibility of slowdown cascade [19] for systems with a large number of partitions. The protocol proposed in [65] for providing session guarantees is used in Bayou architecture [32, 58]. Similar approach is provided in [26, 41]. Bermbach et al. [23] have provided a middleware to provide monotonic-read and read-your-write guarantees on top of eventually consistency systems. There are no experimental results provided in existing works such as [23, 26, 32, 41, 58, 65] to evaluate the cost of providing session guarantees. In our work, we investigate the cost of providing session guarantees in eBay NukV that uses Raft for replication on eBay's cloud platform.

3.2 Existing Work on Causal Consistency for Basic Operations

There are several proposals for causally consistent replicated data stores such as [18, 22, 43, 59] where each replica consists of only one machine. Such an assumption is a serious limitation for the scalability of the system, as the whole data must fit in a single machine. To solve this scalability issue, we can partition the data inside each replica. In this section, we review some of the causal

consistency protocols for partitioned key-value stores. We consider these protocols in two different categories based on how they track causal dependencies. For each protocol, we provide a brief sketch of the algorithm, and then discuss its pros and cons.

3.2.1 Explicit Dependency Tracking

The broad approach for providing causal consistency is to track the causal dependencies of a version, and check them before making the version visible in another replica. The most straightforward way to track dependencies of a version is to track them explicitly, i.e., to keep track of all versions that a given version causally depends on. Specifically, we can accompany each version with a list of all versions that it depends on. COPS [49] and Eiger [50] are two examples of causal consistency protocols that use this approach. Here, we review COPS. Eiger is very similar to COPS for basic PUT and GET operations.

COPS [49] is designed for partitioned key-value stores with the architecture provided in Section 2.1. In COPS, each client maintains a list of all versions that it has read so far. Whenever it wants to write a value, it includes this list of versions with the PUT request. Once the server writes the value, it sends a replicate message to other replicas to inform them about the update to the data. In addition to the key and the value that has been written for it by the client, the server also includes the list of versions read by the client in the replicate message as the dependency list. Once a server receives a replicate message, it first checks the dependency list. If all dependencies of the version are visible in the replica, it makes the version visible. Otherwise, it waits. To check the dependency, a server may need to communicate with other partitions. Specifically, if the dependency list attached with a replicate message received by partition A shows that the replicated version depends on a version that hosted on another partition B, partition A needs to communicate with B to make sure that the dependency is satisfied. To do that, A sends a dependency check message to B. If B has made the dependency visible, it immediately responds to A. Otherwise, it waits. This dependency checking has a high message complexity. Imagine that a client reads a lot of versions before writing a version v. In this case, the dependency list of v is large that results in sending many dependency check

messages in the remote replicas. In addition, the dependency list accompanied with the replicate message is itself an overhead that slows down the communication of replicate messages.

3.2.2 Implicit Dependency Tracking

To reduce the size of dependency metadata, the majority of causal consistency protocol track the dependencies implicitly via timestamps. We review some of these protocols in this section.

Orbe [35] is very similar to COPS, unless instead of the explicit list of actual versions, it uses matrices of timestamps to capture dependencies. In Orbe, each client maintains a $M \times N$ matrix of timestamps where M is the number of replicas, and N is the number of partitions (see Figure 2.1). If entry $\langle m, n \rangle$ of this matrix is t, then the highest timestamp of versions written in partitions n of replica m read by the client is t. Remember causal consistency is transitive (See Section 2.3). Thus, the client may not have read that version directly; it may have read another version that causally depends on a version with timestamp t written in partition n in replica m. Each version also maintains its dependency matrix with it. Whenever a client read the version, it updates its dependency matrix by taking an entry-wise maximum of its dependency matrix and the dependency matrix of the version.

When a client writes a version, the dependency matrix of the client is considered as the dependency matrix of the version. Like COPS, upon creating a new version, the servers sends replicate message to its corresponding partitions in other replicas to inform them about the update. Once a partition receives a replicate message, it first checks its dependencies before making the version visible. Like COPS, the partition sends dependency check messages to other partitions if necessary. Each server maintains the highest timestamps received from its corresponding partitions in other replicas. If it has a timestamp higher than a timestamp required by a dependency check message it responds immediately. Otherwise, it waits.

Although Orbe reduces the size of metadata for tracking dependencies, it still suffers from high message complexity, because like COPS, it needs to contact other partitions for each replicate message. It also needs to store more metadata with each version comparing to the COPS, because

in COPS, we only need to store a single timestamp, but in Orbe, we need to store a matrix of timestamps of $O(M \times N)$.

GentleRain [36] replaces the dependency matrices of Orbe with single scalars. It also changes the mechanism of dependency checking of COPS and Orbe, and reduces the message complexity dramatically. In GentleRain, each client maintains only a single integer called dependency time which is the highest timestamp of the versions that it has read so far. Whenever the client writes a new version, it includes its dependency time. The server creates a new version, and assigns a timestamp to it. The server uses the value of physical clock for the timestamp. it assigns timestamps such that the following condition is satisfied:

$C1$: If version X of object x depends on version Y of object y, then $Y.t < X.t$.

Since the dependency time is the timestamp of versions read by the client before sending the PUT request, we expect it to be less than the current physical time of the server receiving the PUT request. However, in practice, the value of the physical time may not be larger than the client's dependency time due to the inaccuracy of physical clocks. Specifically, physical clocks of different machines may skew. Thus, the clock of one machine may be behind another machine. This leads to situations where a server receives a PUT request with dependency time larger than its current physical time. In this situation, to satisfy $C1$, the server waits until its physical clock shows a value higher than the dependency time.

Like Orbe, each partition keeps the highest timestamps received from its corresponding partitions in other replicas in a version vector. Partitions inside a replica periodically share their version vectors and compute Global Stable Time (GST) as the minimum of all entries of all version vectors. It is straightforward to see:

$C2$: When GST in a node has a certain value T, then all versions with timestamps smaller than or equal to T are visible in the replica.

Now, when a client requests to read a certain key, the server returns the version of the key that has timestamp smaller than GST. $C1$ and $C2$ guarantees that any dependency of the returned version is visible in the replica.

GentleRain dramatically reduces the size of necessary metadata to track dependencies. We only need a single integer to track dependencies. It also reduces the message complexity significantly, because partitions do not need to communicate with each other for each replicate message they receive. Instead, they communicate with each other only periodically. Alongside these benefits, GentleRain suffers from two important problems. The first problem is the sensitivity on the accuracy of the physical clocks. Both the correctness and the performance of GentleRain depends on the accuracy of the physical clocks. Specifically, the correctness of GentleRain is compromised if the physical clock goes backward– an anomaly that happens in practice. The performance is also affected by the wait time before write operations because of clock skew between partitions. The negative impact of this problem is intensified in the context of query amplification where a single end user request translates to many internal requests. We focus on this problem in more detail in Section 5.1. COPS and Orbe do not delay any write operation.

Another issue of GentleRain is the delay before making versions visible in remote replicas. Note that the GST is the minimum entry of all version vectors of all partitions inside a replica. Thus, even only a single entry can hold GST small, increasing the delay before making new version visible to the clients. This leads to high visibility latency when even a single partition is slow (e.g. due to garbage collection, high load, or even failure), even if a version does not have any dependency hosting by that partition. Waiting for a partition that does not host any dependency of a version does not occur in COPS or Orbe. However, even in Orbe and COPS we need to delay the visibility of a version v if the partition hosting one of its dependencies is slow. Such delay is unnecessary for a client never wants to read dependencies of v. Based on this intuition, Mehdi et. al have introduced Occult [54] protocol.

To check dependencies, Occult takes an approach different from the approach of COPS, Orbe, and GentleRain; it moves dependency checking from servers to the clients. Occult assumes a

different architecture than protocol considered so far in this section. Although like other protocols, it assumes the same data placement schema (Figure 2.1), it assumes a single-leader replication policy where for each part of the data, we have a master server. All writes must be done on the master. The master replicates the updates to the followers. Once a server receives a replicate message, it immediately makes the version visible. However, it provides the client with metadata information that let it check if the returned version is consistent with its previous reads. The metadata is similar to the Orbe dependency matrices. However, since Occult uses single-leader replication, the dependency metadata is a vector that has one entry per master partition. To reduce the size of metadata, Occult provides an optimization using structural compression. The details of this optimization can be found in [54]. If the returned version is not causally consistent with client's previous reads, the client has two choices: it can either wait or query the master partition. Transferring dependency checking to the client eliminates the negative impact of slow partitions on the visibility latency, and reduces it to the minimum level, as versions are visible to the client as soon as they arrive. However, it comes with an important cost: Occult does not provide causal consistency and availability in presence of network partitions. As explained in Chapter 1, the most important advantage of causal consistency comparing with strong consistency is that it can be achieved together with availability even in presence of network partitions. By making versions visible as soon as they arrive without any dependency checking, Occult exposes clients to inconsistent versions. In this situation, as explained above, a client has to either wait (i.e., unavailability) or deliver the inconsistent version to the application.

Another problem of Occult is its single-leader replication policy that makes it unsuitable for write-heavy workloads. All other protocols mentioned in this section uses multi-leader replication that lets us distribute the write workload over several machines.

3.3 Existing Work on Causal Consistency for Read-only Transactions

COPS-GT [49] is a read-only transaction algorithm that is based on the COPS [49] algorithm that uses explicit dependency checking. COPS-GT may require two rounds of communication

between the client requesting the transaction and the partitions involved in the transaction. These two rounds of communication increase the response time for the client, especially when network latency between the client and the partitions is high. Compared with COPS-GT, our algorithm requires only one round of communication between the client and one of the partitions involved in the transaction. The rest of communications are done between partitions inside the data center which have negligible network latency. COPS-SNOW [51] is an improvement of COPS-GT algorithm. Unlike COPS-GT, COPS-SNOW requires only one round of communication between the client and the servers. COPS-SNOW achieves this by shifting complexity from read operations to write operations. COPS-SNOW has high metadata overhead. Specifically, in COPS-SNOW, for each version X, we need to store a list of all transactions that have read a version older than X. For each key, we also need to keep track of all transactions that have read the current version. Another problem of COPS-SNOW is doing explicit dependency check for every write operation. To do this explicit dependency check, nodes need to constantly communicate with each that increase the message complexity of the algorithm.

GentleRain [36] provides a blocking ROTX algorithm that may require more than one round of communication with servers in Section 5.5.5. Orbe [35] and ChainReaction [20] are two other blocking algorithms. Contrarian [33] is a time-based causal consistency protocol that provides non-blocking ROTX operations. However, it requires two rounds of client-server communication. Compared with [33], CausalSpartanX protocol provided in Chapter 5 trades one round of client-server communication with one round of server-server communication which results in lower latency for RTOX operations when client-server communication delay is higher than server-server communication delay. For instance, if the client is located in Oregon, and servers are located in California, our experiments show, our CausalSpartanX provides 97% improvement compared with [33] regarding the latency of ROTX operations. Even when the client is co-located with the servers, the latency of client-server communication is typically higher due to firewalls and other security checking across different security zones, because usually, the database servers are in the same zone while application servers are in a different zone.

Wren [64] relies on client cache to provide causally consistent transactions. Wren keeps track of the dependencies of a version with two scalars; one for local updates and one for remote updates. This enables Wren to eliminate the GentleRain's blocking problem. However, since all remote dependencies are tracked by a single scalar, like GentleRain, Wren still suffers from increased update visibility in cases of slow replicas (cf. Section 5.5.3). Another issue with Wren is its requirement for the clients to cache the data that they have written in the system which means more work need to be done on the client-side.

3.4 Other Consistency Protocols

While researchers (especially at academia) are trying to design better causal consistency protocols, many existing industrial distributed data stores rely on eventual consistency and strong consistency. In this section, we review these consistency models.

3.4.1 Eventual Consistency

Due to its performance and availability benefits, eventual consistency is very popular in practical distributed systems. The performance and availability benefits of eventual consistency are especially obvious when we compare it with strongly consistent systems. Many existing NoSQL databases such as Dynamo [31], Cassandra [44], Voldemort [17], and MongoDB [10] provide eventual consistency. Since eventual consistency is basically just a convergence guarantee (i.e. all replica finally provide the same data), implementing it is straightforward: we only need to send any new update to other replicas, and a replica makes a version visible as soon as it receives it. Also, all servers must follow the same function in picking the most recent version between several existing versions for an object. Here we review Dynamo as one of such systems. Other systems are also very similar to Dynamo.

Dynamo [31] is a distributed key-value store built for Amazon platform. Amazon uses Dynamo to manage the state of some of its services with high availability requirements[1]. Dynamo replicates

[1]Dynamo now is also publicly available under the name DynamoDB.

each data object on N nodes referred to as *preference list* of the given key. The first node in the preference list is the key coordinator. Each GET or PUT operation involves the N *healthy and reachable* nodes in the preference list. If some of top N nodes are down or unreachable, we use other less preferred nodes.

When a client sends a $PUT(k, val)$ request, the coordinator creates a new version for k. Dynamo uses vector clock [46] timestamps as version numbers. These vectors have one entry per server. Dynamo does not perform any dependency checking before making a version visible, and version numbers are used only to detect conflicting versions. Dynamo uses a quorum-based write and read policy. Specifically, to write a new version, the coordinator sends the new version to N highest-ranked healthy nodes in the preference list of k. If at least $W - 1$ nodes respond, then the write is considered successful. The value for W can be configured by the application developer.

Similarly, when a client sends a $GET(k)$ request, the coordinator asks N highest-ranked healthy nodes in the preference list of k. The coordinator then waits for R responses. Like W, the value of R is configurable by the application developer. The coordinator may receive different versions of k from different servers. In that situation, two cases are possible: 1) All versions are causally related. In that case, the coordinator reports the most recent version regarding the vector clock timestamps. 2) Some of the versions are not causally related. In that case, the coordinator reports all causally unrelated versions to the client.

One of the advantages of Dynamo is that application developers who use Dynamo can configure it to achieve their desired levels of consistency and availability. Basically, achieving higher availability is possible by assigning lower values for R and W. Some services such as shopping cart should be "always-writable", because the rejection of writes to shopping cart adversely affects customer satisfaction. Setting W to 1 ensures that a PUT operation is never rejected as long as at least one node in the system has written the new version value. However, it may increase the risk of inconsistency. In contrast to shopping cart, some services such as services that maintain product catalogs and promotional items, have high numbers of read requests and small numbers of updates. For such services, typically R is set to 1 and W to N. The typical $\langle N, R, W \rangle$ configuration used in

Dynamo instances is $\langle 3, 2, 2 \rangle$ [31].

3.4.2 Strong Consistency

As explained in Chapter 1, strong consistency gives the application developers the illusion of having a single copy of the data. This is a very powerful abstraction that can significantly simplify coding and debugging applications. It is shown that strong consistency is equivalent to the total order broadcast, and both are equivalent to the consensus. We do not go to the details of consensus, total order broadcast, and strong consistency algorithms in this report. However, briefly discuss the relationship between these problems.

Consensus: Consensus is the problem of reaching an agreement in the distributed systems, i.e. all nodes agree on a decision. Although it may seem easy, it turns out to be a hard problem in presence of node or network failures. In fact, it has been shown it is impossible to design a deterministic algorithm for achieving consensus in an asynchronous network model (when packets may arbitrary delayed) [37]. However, this impossibility results is for a system with a very restrictive assumption (i.e. nodes do not access any timer to use timeouts). In practice, the consensus is used in many practical systems. Paxos [47] is one of the important consensus algorithms.

Total order broadcast: Total order broadcast lets nodes of a distributed systems deliver a sequence of messages all with the same order. The total order broadcast is exactly what a distributed data store needs [40]. Imagine that we have replicated our data in several replicas. All of them apply writes to the data in the same order. Assuming that they have stared with the same data, they will end up in the same state after applying writes in the same order. This idea is a called state machine replication [40].

The problem of total order broadcast is equivalent to the problem of consensus. We do not provide the formal proof of this equivalence here, but to realize it, consider that we can view the total order broadcast as the consensus on the next message that has to be delivered by all nodes.

Strong consistency via total order broadcast: Using total order broadcast, we can provide strong consistency as follows: All nodes share a log with each other.

To write a value: the node appends the write message to the log via total order broadcast. Thus, all nodes see the message on the log in the same order, and apply all writes in the same order.

To read a value: the node appends a message to the log. Then, it starts reading the log. Once it finds the message that it just append to the log, it reads the value that it wants. This way, it is guaranteed that it has applied all writes before it reads the value. Thus, it reads the most recent version.

Viewstamp [48,56], Zab [39], and Raft [57] are some of popular total order broadcast protocols. Zab and Raft are implemented in ZooKeeper [39] and etcd [13], respectively. As mentioned above, Paxos [47] is a consensus protocol. By running Paxos for the next message to deliver, we can provide total order broadcast. However, running total order broadcast directly is more efficient. There is an optimization for Paxos called multi-Paxos that directly provides total order broadcast. Spanner [30] is a strong consistency protocol developed by Google that relies on Paxos.

CHAPTER 4

SESSION GUARANTEES USING HYBRID LOGICAL CLOCKS

In this chapter, we focus on session guarantees. We provide a protocol to provide session guarantees for eBay NuKV that a key-value store designed for services with high performance and availability requirements for internal eBay services. The architecture of NuKV is very similar to the architecture provided in Section 2.1, except we have two levels of replication in NuKV. We start this chapter, by explaining the architecture of NuKV.

4.1 The Architecture of eBay NuKV

The data of a typical NuKV deployment is fully replicated on D data centers. Inside each data center, the data is hosted by P partitions. For each partition, we create several replicas inside a data center. Thus, we have two levels of replication; at one level, we create replicas of the entire data in several data centers, at another level, we create replicas of partitions inside a data center. The replicas of a partition inside a data center form a Raft group [57]. Each group has a leader and other replicas are followers. For each partition, L_d denotes the leader at the data center d. F_d^i denotes the ith follower of L_d (see Figure 4.1).

NuKV follows a multi-leader replication, i.e. clients can write to the leader of any data center. Followers learn new updates via the Raft algorithm [57]. An update on a data center will be asynchronously replicated to the leaders of other data centers by a special follower of the Raft group denoted by XC. The XC servers are stateless. Thus, in case of failure, they can easily recover and resume sending replicate messages to other data centers. Data nodes, on the other hand, detect repeated writes received from XC servers, and avoid applying them. We assume FIFO channels between data centers, i.e., the XC sends updates to the leaders of other data centers with the same order it reads them from its Raft log. Clients usually perform their operations on their local data center. However, it is possible that they access other data centers too. For read operations, clients may use any replica inside a data center, but writes are always on the leaders.

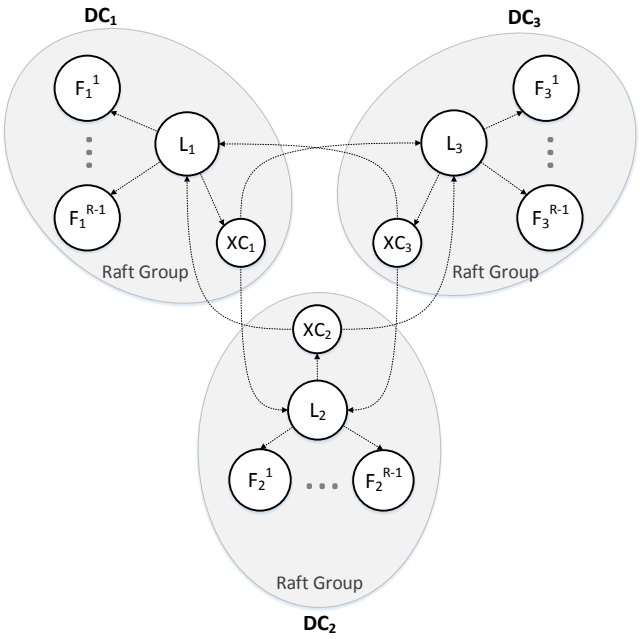


Figure 4.1: An example of a NuKV deployment with three data centers each with R. Note that for *each partition*, we have the architecture shown here.

4.2 Protocol

In this section, we provide a protocol for providing different levels of consistency defined in Section 2.2. The read part of the protocol is basically an adoption of the protocol provided in [65] for our architecture with Raft replication and partitioning explained in Section 4.1. Two basic operations for a key-value store are $PUT(k, v)$ which writes value v for key k and $GET(k)$ which reads the value of key k. Out algorithm may block a GET operation, when the server cannot meet the session guarantee request by the client. For the PUT operations, instead of blocking, we use timestamping with HLC [42] to provide session guarantees.

4.2.1 Client-side

Each client c maintains two $D \times P$ matrices: 1) highest read matrix (hrm), and 2) highest write matrix (hwm). $hrm[d, p]$ is the highest Raft log index of the versions written in partition p of data center d read by client c. $hwm[d, p]$ is the highest Raft log index of the versions written by client c

at partition p of data center d. Note that hrm and hwm are only maintained by the client. They are not sent over the network or stored with keys. Thus, their overhead is negligible. The client also maintains two scalars dt_r and dt_w that maintains the highest HLC timestamps of versions read and written by the client, respectively.

Algorithm 4.1 shows the client-side of our protocol. The client updates its hrm, hwm, dt_r, and dt_w as it reads and writes the data according to the definitions provided above. When the client wants to read a key on partition p, it includes two vectors hrv and hwv with its GET request. The server-side uses these vectors to provide various session guarantees for read operations. Let O be the vector of size D with all entries equal to zero. When the client sends a GET request to partition p, it can send O or the row corresponding to partition p in its hrm, i.e. $hrm[:, p]$ as hrv. Similarly, it can send O or the row corresponding to partition p in its hwm i.e. $hwm[:, p]$ as hwv. By choosing vectors for hrv and hwv, the client can require different session guarantees for its read operations in a per-operation basis as it is shown in Algorithm 4.1. In this algorithm, monotonic-read-your-write means a session guarantee that requires both monotonic-read and read-your-write guarantees.

When the client wants to write a key on partition p, it includes an integer called dependency time denoted by dt in its PUT request. The server-side uses this value to provide various session guarantees for write operations. The client can control its desired session guarantee for a write operation as it is shown in Algorithm 4.1 by choosing values for dt. Choosing 0 results in just eventual consistency, dt_w results in monotonic-write, dt_r results in write-follows-reads, and $max(dt_r, dt_w)$ results in both monotonic-write and write-follows-reads that we refer to by monotonic-write-follows-reads.

4.2.2 Server-side

Each server maintains a stable vector (sv) with size D. $sv[d]$ in server s is the highest log index of the versions written in data center d committed in server s. For each server, PC_d shows the value of the physical clock, and HLC_d shows the value of the hybrid logical clock in L_d.

Algorithm 4.2 shows the server-side of our protocol. In the GET operation handler, the server blocks if for some i, $sv[i]$ is smaller than $hrv[i]$ or $hwv[i]$ sent by the client. This guarantees

Algorithm 4.1 Client-side

1: **GET** (key k, ReadConsistencyLevel l)
2: p = partition id of k
3: **if** (l = EVENTUAL)
4: $hrv = hwv = O$
5: **else if** (l = MONOTONIC-READ)
6: $hwv = O$
7: $hrv = hrm[:, p]$
8: **else if** (l = READ-YOUR-WRITE)
9: $hwv = hwm[:, p]$
10: $hrv = O$
11: **else if** (l = MONOTONIC-READ-YOUR-WRITE)
12: $hwv = hwm[:, p]$
13: $hrv = hrm[:, p]$
14: send $\langle \text{GetReq } k, hwv, hrv \rangle$ to server i
15: receive $\langle \text{GetReply } v, dc\_id, log\_idx, t \rangle$
16: $hrm[dc\_id, p] = max(hrm[dc\_id, p], log\_idx)$
17: $dt_r = max(dt_r, t)$
18: **return** v

19: **PUT** (key k, value v, WriteConsistencyLevel l)
20: p = partition id of k
21: **if** (l = EVENTUAL)
22: $dt = 0$
23: **else if** (l = WRITE-FOLLOWS-READS)
24: $dt = dt_r$
25: **else if** (l = MONOTONIC-WRITE)
26: $dt = dr_w$
27: **else if** (l = MONOTONIC-WRITE-FOLLOWS-READS)
28: $dt = max(dt_r, dt_w)$
29: send $\langle \text{PutReq } k, v, dt \rangle$ to server i
30: receive $\langle \text{PutReply } dc\_id, log\_idx, t \rangle$
31: $hwm[dc\_id, p] = max(hwm[dc\_id, p], log\_idx)$
32: $dt_w = max(dt_w, t)$
33: **return**

that the server has committed all necessary version before reading the value of the requested key. The server, next, returns the version with the highest timestamp among versions written for the requested key.

In the PUT operation handler, the server first updates its HLC using dt sent by the client by calling method *updateHLC* in Line 6. This will call HLC algorithm [42] and guarantees that HLC_d will be higher than any timestamp read/written by the client based on the session guarantee requested by the client. Next, the server creates a new version, timestamps it with the updated HLC_d, and gives it to the Raft algorithm for replication. Once Raft committed, the servers add the version to the version chain of the key and update sv entry for the local data center by the Raft log index. After committing the value on the leader, and updating the local entry of sv, the server returns reply to the client and includes the id of the local data center together with the sv entry for the local data center, and the assigned timestamp. Similar to the leader, other non-XC servers also update their sv upon committing a new version. On the other hand, when an XC server commits version v, if $v.dc\_id$ is the same as the data center id of the server, i.e. the committed write is a local write, it sends a replicate message to the leader of the other groups to propagate a local write to other data centers. It includes the log index of the operation writing the version. Upon receiving a replicate message, the leaders in other data centers append the received message to their Raft log. Followers commit versions received from XC servers like normal local versions and update their sv upon committing in Line 23. If the XC commits a version with a dc_{id} that is different from the id of the local data center, it simply drops the message and does nothing, because the XC of the original data center is responsible for propagating the writes to the other data centers.

4.3 Correctness

In this section, we provide the correctness of the protocol provided in Section 4.2.

Lemma 1 *When a server s has not committed write w with log index log_idx written in data center d, $sv[d] < log\_idx$.*

Proof 1 *By the FIFO assumption of the cross-data center channels, and the total order provided by*

Algorithm 4.2 Server-side

Upon receive $\langle$GetReq $k, hwv, hrv\rangle$
  block while $\exists i$ s.t. $(sv[i] < hrv[i] \vee sv[i] < hwv[i])$
  v = the version for k with the highest timestamp
  send $\langle$GetReply $v.value, v.dc\_id, sv[v.dc\_id], v.t\rangle$ to client

Upon receive $\langle$PutReq $k, value, dt\rangle$ at L_d
  updateHLC (dt)
  $t = HLC_d$
  create new version v
  $v.value \leftarrow value$
  $v.t \leftarrow t$
  $v.dc\_id \leftarrow d$
  append the $\langle k, v\rangle$ to the Raft log
  send $\langle$PutReply $d, sv[d], t\rangle$ to client

updateHLC (t) at L_d
  $l' \leftarrow HLC_d.l$
  $HLC_d.l \leftarrow max(l', PC_d, t.l)$
  if $(HLC_d.l = l' = t.l)$
    $HLC_d.c \leftarrow max(HLC_d.c, t.c) + 1$
  else if $(HLC_d.l = l')$ $HLC_d.c \leftarrow HLC_d.c + 1$
  else if $(HLC_d.l = l)$ $HLC_d.c \leftarrow t.c + 1$
  else $HLC_d.c \leftarrow 0$

Upon commit $\langle k, v\rangle$ with log_idx at a non-XC server
  $sv[v.dc\_id] = log\_idx$
  add v to the version chain of k

Upon commit $\langle k, v\rangle$ at XC_d with log_idx
  if $v.dc\_id = d$
    send $\langle$Replicate $k, v, log\_idx\rangle$ to $L_{d'}$ for all $d' \neq d$

Upon receive $\langle$Replicate $k, v, log\_idx\rangle$ at L_d
  append the $\langle k, v\rangle$ with log_idx to the Raft log

the Raft algorithm [57], we know that versions are committed with the same order that are written in their original data centers. Thus, when write w has not committed on a server, all previously committed writes written in data center d have been written before w. Thus, their log index in smaller than log_idx which leads to $sv[d] < log\_idx$.

Now, using the above lemma, we prove the monotonic-read consistency of the protocol. Specif-

ically,

Theorem 1 (Monotonic-read) *Any operation reading a key on partition p by client c with* $hrv_c = hrm[:, p]$ *satisfies monotonic-read consistency.*

Proof 2 *Let O be an operation by client c that reads the value of key k at time t at server s. If O does not satisfy monotonic-read consistency, there exists a* $w \in ClientReads(c, k, t)$ *written originally at data center d with log index* log_idx *that is not included in* $CommittedWrites(s, k, t')$*, i.e., w has not committed at s at time* t' *where* t' *is the time of reading value for k. Thus, according to Lemma 1,* $sv[d] < log\_idx$ *at time* t'*. Since client c has read w,* $hrv[d] \geq log\_idx$*, according to Line 16 of Algorithm 4.1. Since O returns at time* t'*,* $sv[d] > hrv[d]$ *at time* t'*, according to Line 2. Thus,* $sv[d] \geq log\_idx$ *at time* t' *(contradiction).*

In a similar way, we can prove following theorem:

Theorem 2 (Read-your-write) *Any operation reading a key on partition p by client c with* $hwv_c = hwm[:, p]$ *satisfies read-your-write consistency for* f*.*

Now, we prove the correctness of our protocol for session guarantees of write operations.

Theorem 3 (Monotonic-write) *Any operation writing a key on partition p by client c with* $dt = dt_w$ *satisfies monotonic-write consistency.*

Proof 3 *Let O be an operation by client c that writes version v for key k at time t. If O does not satisfy monotonic-write consistency, there exists server s and time* t' *such that* $CommittedWrites(s, k, t')$ *includes O, but a client accessing s reads a version* v' *written by write* $w \neq O$ *included in* $ClientWrites(c, k, t)$*. By Line 32 of Algorithm 4.1,* dt_w *is higher than* $v'.t$ *at time of writing v. By Line 6 of Algorithm 4.2, and HLC algorithm provided in updateHLC function, we know* $HLC_d > dt_w$*. Thus,* $v.t > v'.t$*. Since O is included in* $CommittedWrites(s, k, t')$*, v is in the version chain for key k according to Line 24. Thus, according to Line 3, server s never returns* v' *with a smaller timestamp than* $v.t$ *(contradiction).*

In a similar way, we can prove following theorem:

Theorem 4 (Write-follows-reads) *Any operation writing a key on partition p by client c with* $dt = dt_r$ *satisfies write-follows-reads consistency.*

Finally, it is straightforward to see that the protocol provided in Section 4.2 satisfies implicit requirements R1 and R2 provided in Section 2.2, as GET always return committed versions, and all servers finally receive any committed version.

4.4 Experimental Results

In this section, we provide the results of benchmarking the protocol defined in Section 4.2 for the architecture explained in Section 2.1. In Section 4.4.1, we provide the experimental setup. In Section 4.4.2, first, we evaluate the overhead of different levels of session guarantees when clients never change their data center. The results of this case reflect the impact of the delay of the Raft on the performance overhead of session guarantees. Then, we consider the case where clients switch to a remote data center for some of their operations. In addition to the Raft delay, the results of this case also show the impact of cross-data center propagation delay on the performance overhead of providing session guarantees. In Section 4.4.3, we investigate the effect of workload characteristics. Section 4.4.4 provides the overall implication of the results.

4.4.1 Experimental Setup

NuKV is implemented in C++. For Raft, it uses an enhanced version of [4]. The client and servers are connected by gRPC [9] and we use Google Protocol Buffers [8] for marshaling/unmarshaling the data.

We did our experiments on a deployment including two data centers, each with three replicas on different machines. Inside a data center, we also have an XC server that forwards the log commits to the leader of the other data center. One data center is located at Phoenix, Arizona, and the other data center is located at Salt Lake City, Utah. The average RTT time between these two data centers

w/o HLC	w/ HLC	Write Guarantee	Read Guarantee
E		Eventual	Eventual
M/E	M/E_{HLC}	Monotonic-write-follows-reads	Eventual
E/M	E/M_{HLC}	Eventual	Monotonic-read-your-write
M/M	M/M_{HLC}	Monotonic-write-follows-reads	Monotonic-read-your-write+

Table 4.1: Legend for Figures 4.2- 4.6

is $\approx$ 15 (ms). We run all replicas and XC servers on machines with the following specification: 4 vCPUs, Intel Core Processor (Haswell) 2.0 GHz, 4 GB memory, 40 GB Storage Capacity running Ubuntu 16.04.3.

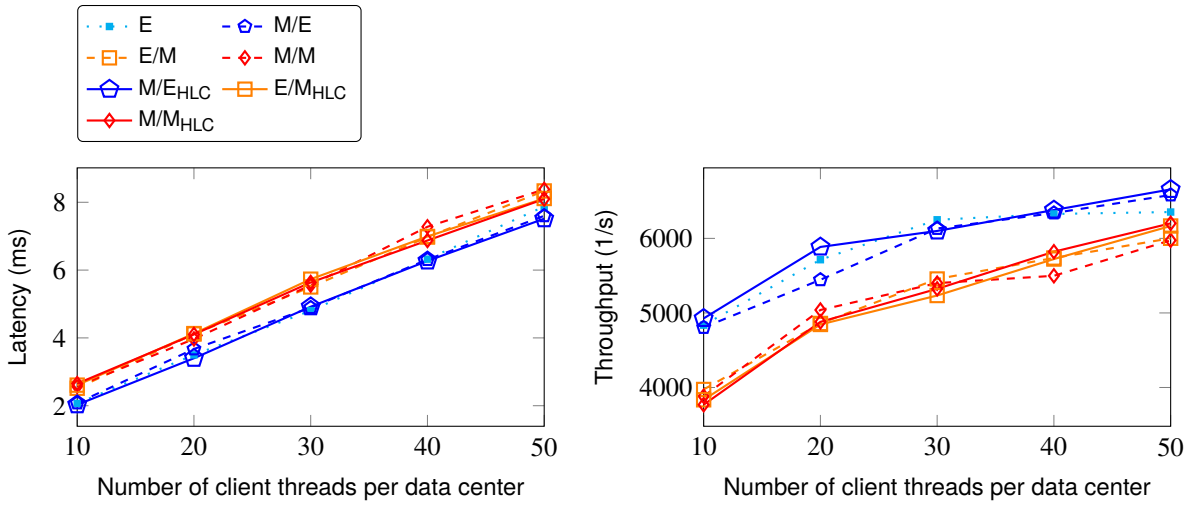
We run client threads from both data centers. We use one machine with following specification for each data center to run the client threads: 8 vCPUs, Intel Core Processor (Haswell) 2.0 GHz, 4 GB memory, 40 GB Storage Capacity running Ubuntu 16.04.3.

Each client thread randomly decides to write or read a key. When it decides to write a key, it randomly picks one of the leaders and writes on it. On the other hand, when it decides to read a key, it randomly picks one of the six replicas and reads from it. These random selections are done with the given probabilities. With different probabilities, we can study the effect of different client usage patterns and the workload characteristics on the performance of the system. We consider 16B keys and 64B values.

4.4.2 Effect of Locality of Traffic

In the first set of experiments, we investigate the effect of locality of traffic on the performance of our key-value store. First, we consider the case where clients only access their local data center.

We consider seven cases as shown in Table 4.1. The cases with HLC subscript show the results for our protocol that uses HLC, and cases without HLC subscript show results when we do not use HLC. If we do not use HLC then the PUT operation must wait until the current server receives all the relevant updates that must occur before the current PUT operation. However, with HLC, we can simply assign the new PUT operation a higher timestamp to ensure that it will be ordered later than previous writes.



(a) Latency

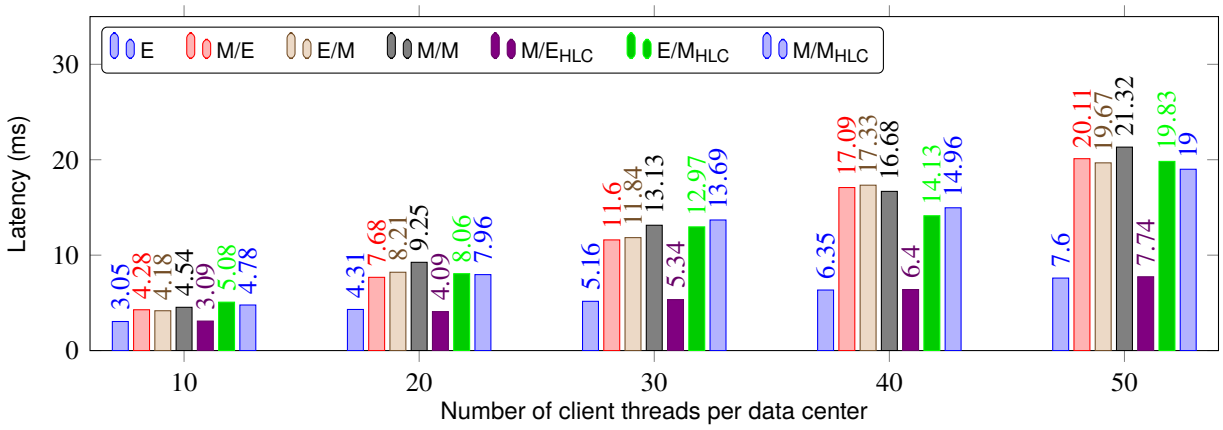
(b) Throughput

Figure 4.2: The effect of load on the latency and throughput of mixed PUT and GET operations with 100% local traffic

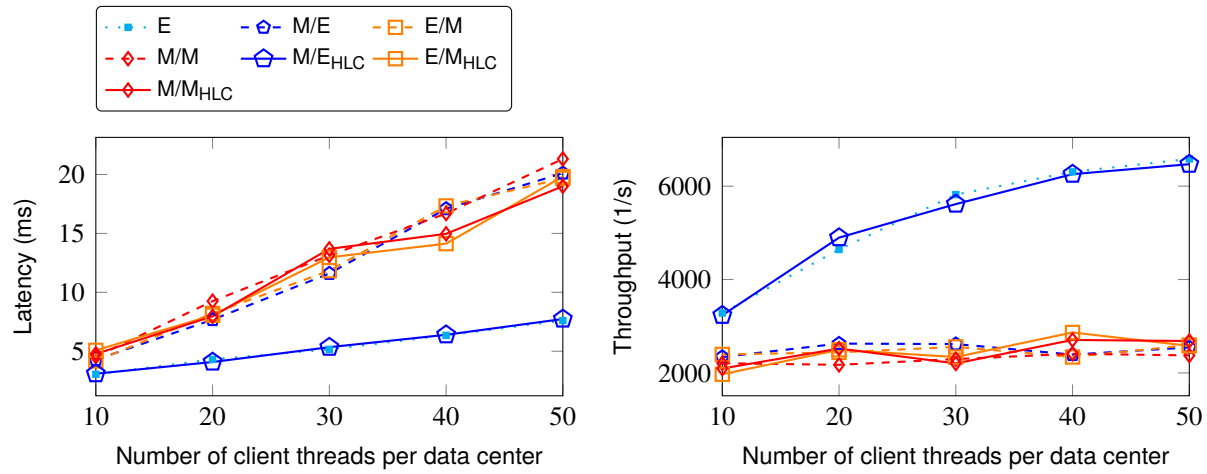
Figure 4.2a and Figure 4.2b show the effect of load on the operation latency and throughput, respectively. In all diagrams with throughout, we report the total operations done by the clients accessing one of the data centers. The results are for 0.5:0.5 put:get ratio, i.e., 50% of operations are PUT, and 50% GET. Note that when clients do not change their data centers, our system actually provides sequential consistency for keys of each partition thanks to the Raft protocol [57], i.e. all updates are applied with the same other on all replicas in one data center. However, regarding the recency of the updates, it is possible that a client reads a version in one replica, but does not find it on another replica.

As expected, the latency and throughput increase in all cases as we run more client threads. Since propagation delay is very small inside one data center, providing session guarantees causes only a negligible overhead such that for some cases we even observed better latency and throughput for stronger guarantees compared with eventual consistency due to the experimental error. However, generally, eventual consistency and M/E, and M/E_{HLC} show better results. In all cases, the additional latency compared with eventual consistency always remains less than 1 (ms).

Next, we consider the case where clients may use the remote data center with 10% probability. Figures 4.3a and 4.3b show the effect of load on the operation latency with 0.5:0.5 put:get ratio.



(a) Latency



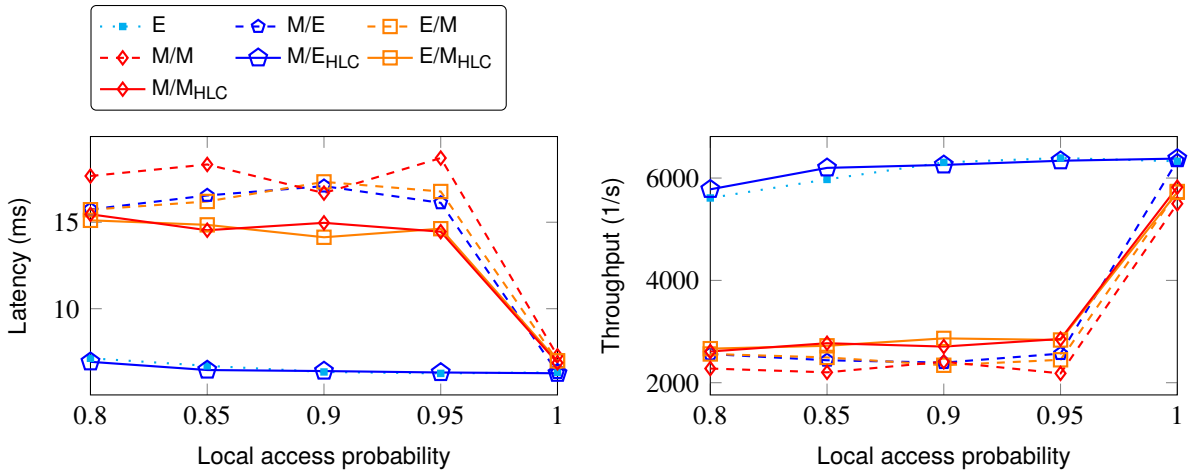
(b) Latency

(c) Throughput

Figure 4.3: The effect of load on the latency and throughput of mixed PUT and GET operations with 10% remote traffic

Unlike the case with 100% local traffic, the difference between eventual consistency and other cases is clear here. In all cases, eventual consistency has the lowest latency. M/E_{HLC} provides the same latency as that of eventual consistency due to wait-free write operations by using HLCs. For 40 client threads, M/E, E/M, and M/M requires an additional ~10 (ms) compared with eventual consistency. When we use HLCs, M/E_{HLC} requires no additional latency, and E/M_{HLC} and M/M_{HLC} requires ~7.7 (ms) and ~8.6 (ms) additional latency. This improvement allows us to process 12-160% additional operations (160% occurs for the case M/E where the use of HLC eliminates the delay in write operations thereby allowing clients to issue more operations)

To further analyze the effect of locality of traffic on the performance, we consider the system



(a) Latency

(b) Throughput

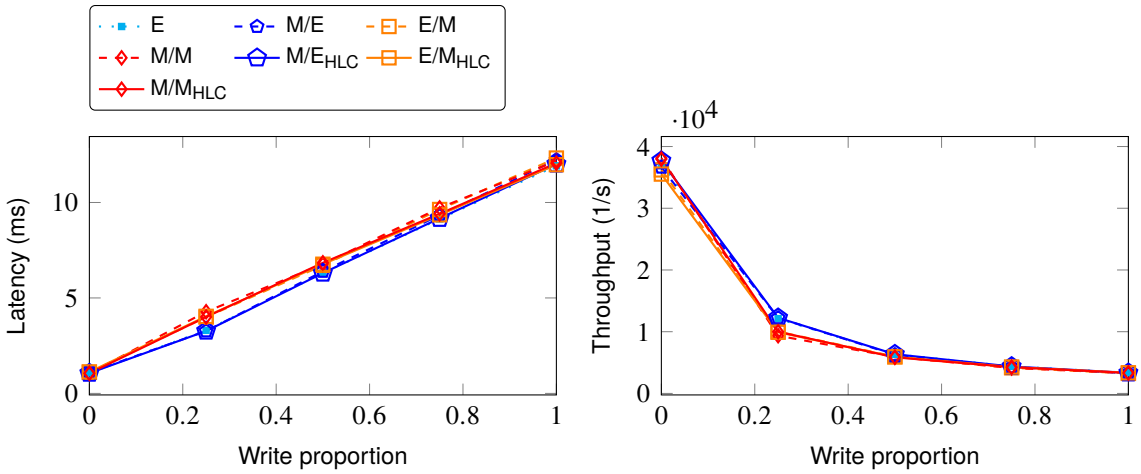
Figure 4.4: The effect of local access probability on the latency and throughput of mixed PUT and GET operations.

with various local access probabilities. Figures 4.4a and 4.4b show how the average latency and throughput change as we increase the local access probability for 40 client threads in each data center. The change in the latency and throughput is not clear, except for case with local access probability 1 where the latency collapses, and throughput increases significantly.

4.4.3 Effect of Workload

In this section, we want to see how the performance changes for workloads with various natures (e.g, read-heavy, write-heavy). The results are for 40 client threads per data center. First, we consider the sticky clients, i.e., 100% local traffic. Figure 4.5a shows the effect of write proportion on the average latency. As write proportion increases the latency increases which is expected, because write operations are expected to be heavier than read operations. Figure 4.5b shows how throughput changes as we change the workload for different levels of consistency. Like Section 4.4.2, the overhead of session guarantees for sticky clients is negligible. For pure-read workload, there is no meaningful overhead, as there is no update and all replicas provide the same data. Thus, all session guarantees are always satisfied.

Figures 4.6a and 4.6b show the effect of write proportion on the average latency for the case



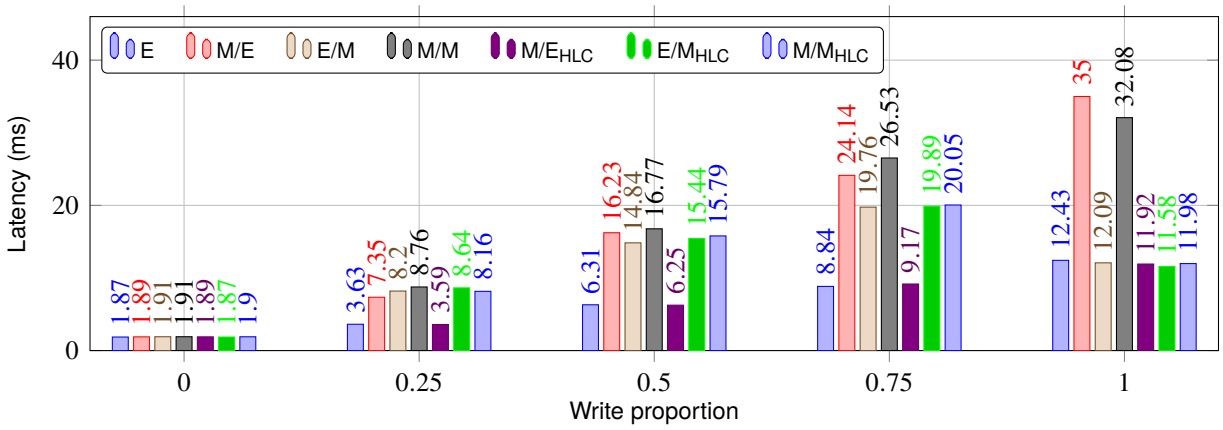
(a) Latency (b) Throughput

Figure 4.5: The effect of workload on latency and throughput of mixed PUT and GET operations with 100% local traffic

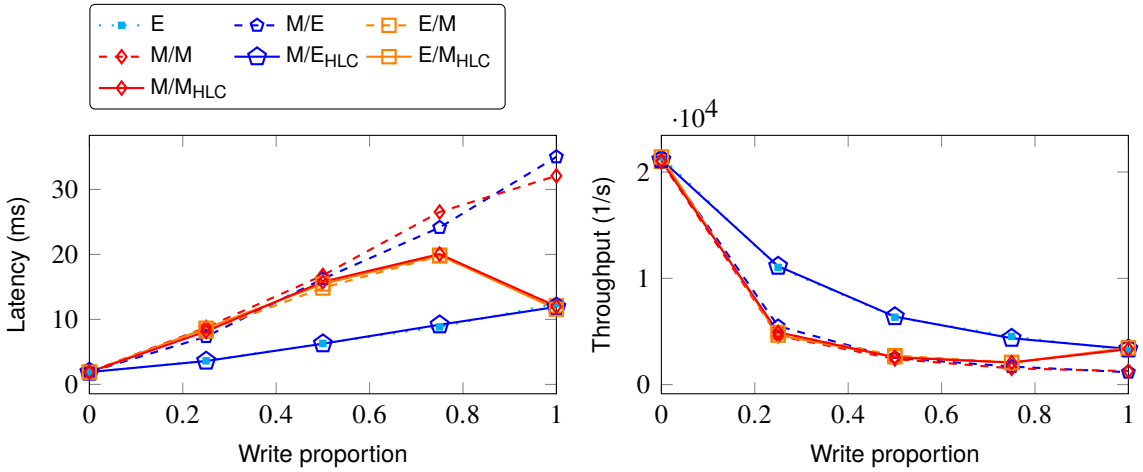
with 10% remote traffic. Again, we see that the difference between eventual consistency and stronger consistency models is more obvious due to inter-data center replication latency. Also, like all other results M/E_{HLC} provide the same performance as eventual consistency. For M/M_{HLC} consistency level, the latency drops as we move from 0.75 write proportion to 1. The reason is with write proportion 1, there is no read operation, and for write operations, M/M_{HLC} is the same as M/E_{HLC}. Thus, it does not require any blocking. Figure 4.6c shows how throughput changes as we change the workload for different levels of consistency with 10% remote access.

4.4.4 Implication of Experimental Results

From these results, we find that if we introduce session guarantees, and the client remains within the same data center, the overhead is within experimental error. Furthermore, even if the client changes its data center, the cost of increased latency is very small, $\approx 10(ms)$. From these results, we find that the session guarantees considered in this paper can be achieved with a very low cost. Without session guarantees, the application can suffer from issues such as the user writes a new value for a key, but obtains the old value when he reads the key, or the user changes the password twice, but finds that the stored password is not the same as the latest one. Such common but highly undesirable problems with eventual consistency can be eliminated with a very low overhead with



(a) Latency



(b) Latency

(c) Throughput

Figure 4.6: The effect of workload on latency and throughput of mixed PUT and GET operations with 10% remote traffic

session guarantees.

Finally, this low overhead also suggests that it is not worthwhile to attempt session guarantees by having clients cache the data they have read or written, as it complicates the design of clients substantially. This is especially important to support computing-challenged clients.

CHAPTER 5

CAUSAL CONSISTENCY USING HYBRID LOGICAL CLOCKS

As explained in Section 3.2.2, GentleRain reduces the overhead of tracking dependencies by taking an implicit approach via physical timestamps. However, it relies on synchronized and monotonic physical clocks for both its correctness and performance. Specifically, it requires that clocks are strictly increasing. This may be hard to guarantee if the underlying service such as NTP causes non-monotonic updates to POSIX time [16] or suffers from leap seconds [2, 7]. In addition, as we will see, the clock skew between physical clocks of partitions may lead to cases where GentleRain must intentionally delay write operations.

The issue of clock anomalies is intensified in the context of query amplification, where a single query (e.g., an update on a Facebook page) results in many (possibly 100s to 1000s) GET/PUT operations [19]. In this case, the delays involved in each of these operations contribute to the total delay of the operation, and can substantially increase the response time for the clients.

Our goal in this chapter is to analyze the effect of clock anomalies to develop a causally consistent data store that is resistant to clock skew among servers. This will allow us to ensure that high performance is provided even if there is a clock skew among servers. It would obviate the need for all servers in a data center to be co-located for the sake of reducing clock anomalies.

To achieve this goal, we develop CausalSpartanX that is based on the structure of GentleRain but utilizes HLCs [42]. Similar to [35, 36, 49, 50], we assume that a client only accesses one replica during its execution. In Chapter 6 we show that this assumption is essential if we want to provide causal consistency while ensuring that all local updates are visible immediately. In other words, we show that if the client could access multiple replicas and a replica makes local updates visible immediately then it is impossible to provide causal consistency.

In this chapter:

- We show that in the presence of clock anomalies, CausalSpartanX reduces the latency of PUT

operations compared with that of GentleRain. Moreover, the performance or correctness of CausalSpartanX is unaffected by clock anomalies.

- We demonstrate that CausalSpartanX is especially effective to deal with delays associated with query amplification.
- We demonstrate that CausalSpartanX reduces the update visibility latency. This is especially important in collaborative applications associated with a data store. For example, in an application where two clients update a common variable (for example, the bid price for an auction) based on the update of the other client, CausalSpartanX reduces the execution time substantially.
- We show that using HLC instead of physical clocks does not have any overhead.
- We demonstrate the efficiency provided by our approach by performing experiments on cloud services provided by Amazon Web Services [1].

5.1 Effect of Clock Anomalies

Thanks to the use of clocks for dependency checking, GentleRain [36] has a throughput higher than that of other causally consistent data stores proposed in the literature such as COPS [49] or Orbe [35]. In particular, GentleRain avoids sending dependency check messages, unlike other systems which leads to a light message complexity which in turn leads to higher throughput. In this section, we discuss the sensitivity of GentleRain on the accuracy of physical clocks and clock synchronization.

5.1.1 Sensitivity on Physical Clock and Clock Synchronization

As explained in Section 3.2.2, to satisfy condition $C1$, in some cases, it may be necessary to wait before creating a new version. Specifically, if a client has read/written a key with timestamp t, then any future PUT operation the client invokes must have a timestamp higher than t. Hence, the client sends the timestamp t of the last version that it has read/written together with a PUT operation. The

partition receiving this request first waits until its physical clock is higher than t before creating the new version. This wait time, as we observed in our experiments, is proportional to the clock skew between servers. In other words, as the physical clocks of servers drift from each other, the incidence and the amount of this wait period increases.

In addition, with GentleRain algorithm sketched in Section 3.2.2, the physical clocks cannot go backward. To illustrate this, consider a system consisting of two data centers A and B. Suppose GSTs in both data centers are 6. That means, both data centers assume all versions with timestamps smaller than 6 are visible (condition $C2$). Now, suppose the physical clock of one of the servers in data center A goes backward to 5. In this situation, if a client writes a new version at that server, condition $C2$ is violated, as the version with timestamp 5 has not arrived in data center B, but its GST is 6 which is higher than 5.

As we explained above, both performance and correctness of GentleRain rely on the accuracy of the physical clocks and the clock synchronization between servers.

5.1.2 Query Amplification

The sensitivity issue identified in Section 5.1.1 is made worse in practice, because a single end user request usually translates to so many possibly causally dependent internal queries. This phenomenon is known as *query amplification* [19]. In a system like Facebook, query amplification may result in up to thousands of internal queries for a single end user request [19]. In such a system, an end user submits the request to a web server. The web server performs necessary internal queries, and then responds to the end user. This implies that the web server needs to wait for all internal queries before responding to the end user request. Thus, any delay in any of internal queries will affect the end user experience [19].

CausalSpartanX solves the issues identified in this section by ensuring that no delays are added to PUT operations. Therefore, CausalSpartanX is unaffected by clock skew even in the presence of query amplification. We achieve this goal by using HLCs instead of physical clocks.

5.2 CausalSpartanX Basic Protocol

In this section, we focus on CausalSpartanX protocol for basic PUT and GET operations [1]. One way to get around the issue of PUT latency identified in Section 5.1 is as follows: Suppose that a client has read a value written at time t and it wants to perform a new PUT operation on a server whose time is less than t. To satisfy $C1$, in [36], PUT operation is delayed until the clock of the server was increased beyond t. Another option is to change the clock of the server to be $t + 1$. However, changing the physical clock is undesirable; it would lead to a violation of clock synchronization achieved by protocols such as NTP. It would also have unintended consequences for applications using that clock.

Using HLC in this problem solves several problems associated with changing the physical clock. Specifically, HLC is a logical clock and can be changed if needed. In the scenario described in the previous paragraph, this would be achieved by increasing the c value which is still guaranteed to stay bounded [42]. At the same time, HLC is guaranteed to be close to the physical clock. Hence, it can continue to be used in place of the physical clock. Also, HLC uses the physical clock as a read-only variable thereby ensuring that it does not affect protocols such as NTP. For these reasons, CausalSpartanX uses HLC.

Another important improvement in CausalSpartanX is the use of Data center Stable Vectors (DSVs) instead of GSTs. DSVs are vectors that have an entry for each data center. If $DSV[j]$ equals t in data center i, then it implies that all writes performed at data center j before time t have been received by data center i. DSVs reduce update visibility latency and allow collaborative clients to work quickly in the presence of some *slow* replicas. Next, we focus on different parts of the CausalSpartanX protocol.

[1] The CausalSpartanX protocol for basic operation is published under name CausalSpartan in [60].

5.2.1 Client-side

A client c maintains a set of pairs of data center IDs and HLC timestamps called dependency set, denoted as DS_c[2]. For each data center i, there is at most one entry $\langle i, h\rangle$ in DS_c where h specifies the maximum timestamp of versions read by client c originally written in data center i. For a given PUT request, this information is provided by the client so that the server can guarantee causal consistency. A client c also maintains DSV_c that is the most recent DSV that the client is aware of.

Algorithm 5.1 shows the algorithm for the client operations. For a GET operation, the client sends the key that it wants to read together with its DSV_c by sending $\langle \textsc{GetReq}\ k, DSV_c\rangle$ message to the server where key k resides. In the response, the server sends the value of the requested key together with a list of dependencies of the returned value, ds, and the DSV in the server, dsv. The client, then, first updates its DSV, and next updates its DS_c by calling $maxDS$ as follows: for each $\langle i, h\rangle \in ds$ if currently there is an entry $\langle i, h'\rangle$ in DS_c, it replaces h' with the maximum of h and h'. Otherwise, it adds $\langle i, h\rangle$ to the DS_c.

For a PUT operation, the client sends the key that it wants to write together with the desired value and its DS_c. In response, the server sends the timestamp assigned to this update together with the ID of the data center. The client then updates its DS_c by calling $maxDS$.

5.2.2 Server-side

In this section, we focus on the server-side of the protocol. We have M data centers (i.e., replicas) each of which with N partitions (i.e., servers). We denote the nth partition in mth replica by p_n^m (see Figure 2.1). We denote the physical clock at partition p_n^m by PC_n^m. Each partition p_n^m stores a vector of size M (one entry for each data center) of (HLC) timestamps denoted by VV_n^m. For $k \neq m$, $VV_n^m[k]$ is the latest timestamp received from server p_n^k by server p_n^m. $VV_n^m[m]$ is the highest timestamp assigned to a version written in partition p_n^m. Partitions inside a data center, periodically

[2]This could be maintained as part of client library as in [49] so that the effective interface of the client does not have to explicitly maintain this information. Alternatively, this could also be maintained by the server for each client. For sake of simplicity, in our discussion, we assume that this information is provided by the client.

Algorithm 5.1 Client operations at client c

```
1:  GET (key k)
2:     send ⟨GetReq k, DSV_c⟩ to server
3:     receive ⟨GetReply v, ds, dsv⟩
4:     DSV_c ← max(DSV_c, dsv)
5:     DS_c ← maxDS(DS_c, ds)
6:  return v

7:  PUT (key k, value v)
8:     send ⟨PutReq k, v, DS_c⟩ to server
9:     receive ⟨PutReply ut, sr⟩
10:    DS_c ← maxDS(DS_c, {⟨sr, ut⟩})

11: maxDS (dependency set ds_1, dependency set ds_2)
12:    for each ⟨i, h⟩ ∈ ds_2
13:       if ∃⟨i, h'⟩ ∈ ds_1
14:          ds_1 ← ds_1 − ⟨i, h'⟩
15:          ds_1 ← ⟨i, max(h, h')⟩
16:       else
17:          ds_1 ← ds_1 ∪ {⟨i, h⟩}
18: return ds_1
```

share their VVs with each other, and compute DSV as the entry-wise minimum of VVs. DSV_n^m is the DSV computed in server p_n^m.

For each version, in addition to the key and value, we store some additional metadata including the (HLC) time of creation of the version, ut, and the source replica, sr, where the version has been written, and a set of dependencies, ds, similar to dependency sets of clients. Note that ds has at most one entry for each data center.

Algorithm 5.2 shows the algorithm for PUT and GET operations at the server-side. Upon receiving a GET request (GetReq), the server first updates its DSV if necessary using DSV value received from the client (see Line 2 of Algorithm 5.2). After updating DSV, the server finds the latest version of the requested key that is either written in the local data center, or all of its dependencies are visible in the data center. To check this, the server compares the DS of the key with its DSV. Note that to find the *latest* version, the server uses the last-writer-wins conflict resolution function that breaks ties by data center IDs as explained in Section 2.3. After finding

the proper value, the server returns the value together with the list of dependencies of the value, and its DSV in a GetReply message. The server also includes the version being returned in the dependency list in Line 4 of Algorithm 5.2 by calling the same $maxDS$ function as defined in Algorithm 5.1.

A major improvement in CausalSpartanX over GentleRain is providing wait-free PUT operations. Once server p_n^m receives a PUT request, the server first updates its DSV with the DS value received from the client. The server, next, updates $VV_n^m[m]$ to calling $updateHLC$ function. It uses maximum number between ds values and $DSV_n^m[m]$ as $updateHLC$ argument. This will guarantee that the new $VV_n^m[m]$ to be higher than this maximum and capture causality. Next, the server creates a new version for the key specified by the client and uses the current $VV_n^m[m]$ value for its timestamp. The server sends back the assigned timestamp $d.ut$ and data center ID m to the client in a PutReply message.

Upon creating a new version for an item in one data center, we send the new version to other data centers via replicate messages. Upon receiving a $\langle Replicate\ d \rangle$ message from server p_n^k, the receiving server p_n^m adds the new version to the version chain of the item with key $d.k$. The server also updates the entry for server p_n^k in its version vector. Thus, it sets $VV_n^m[k]$ to $d.ut$.

Algorithm 5.3 shows the algorithm for updating DSVs. As mentioned before, partitions inside a data center periodically update their DSV values. Specifically, every θ time, partitions share their VVs with each other and compute DSV as the entry-wise minimum of all VVs (see Line 2 of Algorithm 5.3). Broadcasting VVs has a high overhead. Instead, we efficiently compute DSV over a tree like the way GST is computed in [36]. Specifically, each node upon receiving VVs of its children computes entry-wise minimum of the VVs and forwards the result to its parent. The root server computes the final DSV and pushes it back through the tree. Each node, then, updates its DSV upon receiving DSV from its parent. Algorithm 5.3 also shows the algorithm for the heartbeat mechanism. Heartbeat messages are sent by a server if the server has not sent any replicate message for a certain time Δ. The goal of heartbeat messages is updating the knowledge of the peers of a partition in other data centers (i.e., updating VVs).

Algorithm 5.2 PUT and GET operations at server p_n^m

1: **Upon** receive $\langle \text{GetReq } k, dsv \rangle$
2: $DSV_n^m \leftarrow max(DSV_n^m, dsv)$
3: obtain latest version d (the version with lexicographically highest value $\langle ut, sr \rangle$) from version chain of key k s.t.

- $d.sr = m$, or
- for any $\langle i, h \rangle \in d.ds$, $h \leq DSV_n^m[i]$

4: $ds \leftarrow maxDS(d.ds, \{\langle d.sr, d.ut \rangle\})$
5: send $\langle \text{GetReply } d.v, ds, DSV_n^m \rangle$ to client

6: **Upon** receive $\langle \text{PutReq } k, v, ds \rangle$
7: $DSV_n^m \leftarrow maxDS(DSV_n^m, ds)$
8: $dt \leftarrow$ max value in $\{ds.values \cup \{DSV_n^m[m]\}\}$
9: $updateHCL(dt)$
10: Create new item d
11: $d.k \leftarrow k$
12: $d.v \leftarrow v$
13: $d.ut \leftarrow VV_n^m[m]$
14: $d.sr \leftarrow m$
15: $d.ds \leftarrow ds$
16: insert d to version chain of k
17: send $\langle \text{PutReply } d.ut, m \rangle$ to client
18: **for** each server $p_n^k, k \in \{0 \ldots M-1\}, k \neq m$ **do**
19: send $\langle \text{Replicate } d \rangle$ to p_n^k

20: **Upon** receive $\langle \text{Replicate } d \rangle$ from p_n^k
21: insert d to version chain of key $d.k$
22: $VV_n^m[k] \leftarrow d.ut$

23: **updateHLCforPut** (dt)
24: $l' \leftarrow VV_n^m[m].l$
25: $VV_n^m[m].l \leftarrow max(l', PC_n^m, dt.l)$
26: **if** $(VV_n^m[m].l = l' = dt.l)$ $VV_n^m[m].c \leftarrow max(VV_n^m[m].c, dt.c) + 1$
27: **else if** $(VV_n^m[m].l = l')$ $VV_n^m[m].c \leftarrow VV_n^m[m].c + 1$
28: **else if** $(VV_n^m[m].l = dt.l)$ $VV_n^m[m].c \leftarrow dt.c + 1$
29: **else** $VV_n^m[m].c \leftarrow 0$

Algorithm 5.3 HEARTBEAT and DSV computation operations at server p_n^m

1: **Upon** every θ time
2: $DSV_n^m \leftarrow max(DSV_n^m, \text{entry-wise } min_{j=1}^{N}(VV_j^m))$

3: **Upon** every Δ time
4: **if** there has not been any replicate message in the past Δ time
5: $updateHCL()$
6: **for** each server $p_n^k, k \in \{0 \ldots M-1\}, k \neq m$ **do**
7: send $\langle \text{HEARTBEAT } HLC_n^m \rangle$ to p_n^k

8: **Upon** receive $\langle \text{HEARTBEAT } hlc \rangle$ from p_n^k
9: $VV_n^m[k] \leftarrow hlc$

10: **updateHLC** ()
11: $l' \leftarrow VV_n^m[m].l$
12: $VV_n^m[m].l \leftarrow max(VV_n^m[m].l, PC_n^m)$
13: **if** $(VV_n^m[m].l = l')$ $VV_n^m[m].c \leftarrow VV_n^m[m].c + 1$
14: **else** $VV_n^m[m].c \leftarrow 0$

5.3 CausalSpartanX Read-only Transactions

In this section, we provide an algorithm for ROTX, that never blocks, and requires only one round of communication between the client and the servers. Algorithm 5.4 shows both sides of our algorithm for ROTX. Upon request of a ROTX operation from the application, the client sends a ROTX request with the set of requested keys by the application together with its DSV_c and DS_c to one of the servers hosting one of the requested keys to read. In the response, the server sends the values of the requested keys, updated DSV, and the set of dependencies of the returned values. Upon receiving the response, the client updates its DSV_c and DS_c with the new values returned by the server, and return the received values from the server to the application.

On the server-side, upon receiving a ROTX request from a client, first updates it DSV by dsv and ds values received from the client. The server, next, uses the current value of its DSV as the snapshot vector, sv. Then, for each requested key k, the server sends a SLICEREQ request with sv to the partition hosting k. Upon receiving response to its SLICEREQ message, the server reads the returned value, and updates ds. Once the server learned the value of all requested keys, it sends the

response to the client together with its new DSV and the update ds.

Upon receiving a SliceReq request, a partition first updates the local entry of its DSV by the corresponding entry in sv. Next, it retrieves the most recent version s of the request key such that all dependencies of the version are smaller than their corresponding entries in sv, and the update time of the version, $d.ut$, is less than or equal to the $sv[d.sr]$. Then, the partition returns the response to the requesting server with the value of the key and the set of dependencies.

Algorithm 5.4 Algorithm for ROTX

1: //at client c
2: **ROTX** (keys $kset$)
3: send $\langle$ROTX $kset, DSV_c, DS_c\rangle$ to server
4: receive $\langle$ROTXReply $vset, dsv, ds\rangle$
5: $DSV_c \leftarrow max(DSV_c, dsv)$
6: $DS_c \leftarrow maxDS(DS_c, ds)$
7: **return** $vset$

8: //at partition p_n^m
9: **Upon** receive $\langle$ROTX $kset, dsv, ds\rangle$
10: $DSV_n^m \leftarrow max(DSV_n^m, dsv, ds)$
11: $vset \leftarrow \emptyset$
12: $sv \leftarrow DSV_n^m$
13: **for** each $k \in kset$ do {*In parallel*}
14: send $\langle$SliceReq $k, sv\rangle$ to server
15: receive $\langle$SliceReply $v, ds'\rangle$
16: $vset \leftarrow vset \cup \{v\}$
17: $ds \leftarrow maxDS(ds, ds')$
18: send $\langle$ROTXReply $vset, DSV_n^m, ds\rangle$ to client

19: //at partition p_n^m
20: **Upon** receive $\langle$SliceReq $k, sv\rangle$
21: $DSV_n^m[m] \leftarrow max(DSV_n^m[m], sv[m])$
22: obtain latest version d from version chain of key k s.t. for any $\langle i, h\rangle \in maxDS(d.ds, \{\langle d.sr, d.ut\rangle\})$, $h \leq sv[i]$
23: $ds \leftarrow maxDS(d.ds, \{\langle d.sr, d.ut\rangle\})$
24: send $\langle$SliceReply $d.v, ds\rangle$ back to server

5.4 Correctness

In this section, we focus on the correctness of our proposed protocol. We want to show that the data store running our protocol is a causal++ consistent for $R = O = \{GET, ROTX\}$, $I = \{GET\}$, and conflict resolution function last-write-wins. For ROTX operations, we also need to show that the set of values returned by ROTX operations are causally consistent with each other.

5.4.1 Causal Consistency for *GET* Operations

As explained in Section 5.2, we store a dependency set for each version. This set contains at most one entry per data center. Whenever a client reads a version, the client updates its dependency set by the dependency set of the version (see Line 5 of Algorithm 5.1, and Line 6 of Algorithm 5.4). This dependency set later will be used as the dependency set of any version written by this client. Thus, dependencies are transitive. In addition, whenever a client writes a version v, we update the DS_c of the client in Line 10 of Algorithm 5.1 to capture dependency on v. Thus, ds of a version written by client c captures dependency on any other version previously written by c, and any version v previously read by client c along with all dependencies of v. Specifically, we have the following observation,

Observation 5 *Let v_1 and v_2 be two versions for two keys. If v_1 dep v_2, then for any member $\langle i, h\rangle \in v_2.ds$, there exists $\langle i, h'\rangle \in v_1.ds$ such that $h' \geq h$.*

Whenever a server returns a version, it includes the update time of the version in the dependency set returned with the version/versions (see Line 4 of Algorithm 5.2, and Line 23 of Algorithm 5.4). Thus, based on Observation 5, we have the following observation,

Observation 6 *Let v_1 and v_2 be two versions for two keys such that v_2 is written in data center j. If v_1 dep v_2, then $v_1.ds$ has member $\langle j, h\rangle$ such that $h \geq v_2.ut$.*

According to Line 8 of Algorithm 5.2, using HLC algorithm, our protocol assigns a timestamp higher than the timestamps of all of the dependencies of a version. Thus, we have

Observation 7 *Let v_1 and v_2 be two versions for two keys. If v_1 dep v_2, then $v_1.ut > v_2.ut$.*

We consider the last-write-wins policy for conflict resolution. The last-write-wins means, if two versions are conflicting, then the winner version is the one with higher timestamp. If timestamps are equal, the winner version is the one with higher replication id. Specifically, we have

Observation 8 *For conflict resolution function f = last-writer-wins, $f(v, v') = v$ iff*

- *v dep v', or*
- *$\neg(v \text{ dep } v') \wedge \neg(v' \text{ dep } v) \wedge \langle v.ut, v.sr\rangle$ is lexicographically greater than $\langle v'.ut, v'.sr\rangle$.*

Lemma 2 *All versions written in data center i are immediately visible+ for $\{GET\}$ to any client at data center i.*

Proof 4 *Suppose a local version v is not immediately visible+ for $\{GET\}$ to a client. That requires that in response to a GET operation, another version v' is returned such that $f(v, v') = v$. According to Observations 8 and 7, $\langle v.ut, v.sr\rangle$ in any case is lexicographically higher than $\langle v'.ut, v'.sr\rangle$. Thus, since the version is local, the hosting partition has this version in the version chain of the key. Thus, according to Line 3 of Algorithm 5.2, it is impossible to return v' (contradiction).*

A non-local version is not visible for $\{GET\}$ in a data center, either if it has not arrived its hosting partition, or DSV of the hosting partition is behind of one of the dependencies. Specifically, we have the following Lemma,

Lemma 3 *Let v be a version that is written in partition p_n^m. If v is not visible+ for $\{GET\}$ for conflict resolution function f = last-write-wins to a client in data center i, then*

- *v has not arrived partition p_n^i, or*
- *there is a member $\langle k, h\rangle \in v.ds$, such that $DSV_n^i[k] < h$.*

Proof 5 *When v is not visible+ for $\{GET\}$ in data center i, it means in the response of a GET operation another version v' is returned such that $f(v, v') = v$ (see Definition 13). According to Observation 8, two cases are possible. In the first case, according to Observation 7, $v.ut > v'.ut$, thus in both cases $\langle v.ut, v.sr\rangle > \langle v'.ut, v'.sr\rangle$. Now suppose both conditions of the this lemma are false. Then, according to Line 3 of Algorithm 5.2, it is impossible that GET operation returns v' (contradiction).*

We define DSV^i_{real} as entry-wise minimum of all VV of partitions in data center i. In other, words, for all $1 < m < M$, $DSV^i_{real}[i] = \min_{1<n<N} VV^i_n[m]$.

Now, we have the following lemma about the relation of the DSV_{real} and other DSV and DS values inside a data center:

Lemma 4 *In data center i,*

- *for all $1 < n < N$, and $1 < m < M \wedge i \neq m$, $DSV^i_n[m] \leq DSV^i_{real}[m]$, and*
- *for any client c accessing data center i, for all $1 < m < M \wedge i \neq m$, $DSV_c[m] \leq DSV^m_{real}[m]$, and*
- *for any client c accessing data center i, for any $\langle k, h\rangle \in DS_c \wedge k \neq i$, $h \leq DSV^i_{real}[m]$.*

Proof 6 *We prove this lemma, by induction over time.*

At the beginning all VV, DSV, and DS values at servers and client are zero. Thus, $DSV^m_{real} = DSV^m_n$ for any $1 < n < N$. For any client c, $DSV_c = DSV^m_{real}$. Also, for any $\langle k, h\rangle \in DS_c$, $h = DSV^m_{real}[k]$. Now we consider different operations and note how DSV^m_n, DSV_c, or DS_c values changes. For sake of brevity, we assume $1 < n < N$, $1 < m < M \wedge i \neq m$, and $i \neq k$ implicit:

- *DSV **calculation**: At Line 2 of Algorithm 5.3, each server computes the the DSV as the entry-wise minimum of all VVs. Thus, by definition of DSV^i_{real}, $DSV^i_{real} \geq DSV^m_n$.*
- *GET **operation**: By induction hypothesis, we have $dsv \leq DSV^i_{real}$. Thus, when the server updates its DSV^i_n at Line 2 of Algorithm 5.2, still we have $DSV^i_n \leq DSV^i_{real}$. The server*

returns the value to client, and client updates its DSV_c at Line 4 of Algorithm 5.1 using DSV_n^i received from server. Thus, still we have $DSV_c \leq DSV_{real}^i$. The client also updates its DS_c with ds received from server at Line 5 of Algorithm 5.1. If the version is local, all of its dependencies are the version is smaller than DSV_{real}^i by induction hypothesis. Otherwise, since for any $\langle k, h\rangle \in ds$, $h \leq DSV_n^i[k]$, and $DSV_n^i \leq DSV_{real}^i$, for any $\langle k, h\rangle$ in the new DS_c, we still have $h \leq DSV_{real}^i$.

- *PUT* ***operation****: By induction hypothesis, we know for any $\langle k, h\rangle \in ds$, $h \leq DSV_{real}^i[k]$. Thus, when the server updates is DSV at Line 7 of Algorithm 5.2, we still have $DSV_n^i \leq DSV_{real}^i$.*

- *ROTX* ***operation****: By induction hypothesis, we know the dsv and ds received from client is less than or equal to DSV_{real}^i. Thus, when the server updates its DSV_n^i at Line 10 of Algorithm 5.4, still we have $DSV_n^i \leq DSV_{real}^i$. The server returns the values to client, and client updates its DSV_c at Line 5 of Algorithm 5.4 using DSV_n^i received from server. Thus, still we have $DSV_c \leq DSV_{real}^i$. The client also updates its DS_c with ds received from server at Line 5 of Algorithm 5.1. Since for any $\langle k, h\rangle \in ds$, $h \leq sv[k]$, and $sv \leq DSV_n^i \leq DSV_{real}^i$, for any $\langle k, h\rangle$ in the new DS_c, we still have $h \leq DSV_{real}^i$.*

We assume FIFO channels between replicas. Thus, we have the following observation

Observation 9 *If a version v written in data center m, has not arrived data center i, $DSV_{real}^i[m] < v.ut$.*

According to Observation 9, and Lemma 4, we have the following observation:

Observation 10 *If a version v written in data center m, has not arrived data center i, for all $1 \leq n \leq N$, $DSV_n^i[m] < v.ut$.*

According to Line 12 of Algorithm 5.4, we set the value of *sv* for each *ROTX* operation by a DSV value. Thus, according to Lemma 4, we have the following observation

Observation 11 *For any ROTX operation at data center i, for all* $1 \leq m \leq M \wedge i \neq m$, $sv[m] \leq DSV^i_{real}[m]$.

Note that in all lines where we change the DSV values, we always increase them. Thus, we note the following observation:

Observation 12 *For all* $1 \leq i \leq M$, $1 \leq n \leq N$, *and* $1 \leq m \leq M$, $DSV^i_n[m]$ *never decreases.*

Lemma 5 *Once a version becomes visible+ for* $\{GET\}$ *for a client in data center i it remains visible+ for* $\{GET\}$ *for any client in data center i.*

Proof 7 *Suppose a version v was visible+ to a client, but later it is not visible+. That requires that in response to a GET operation, another version* v' *is returned such that* $f(v, v') = v$. *According to Observations 8 and 7,* $\langle v.ut, v.sr\rangle$ *in any case is lexicographically higher than* $\langle v'.ut, v'.sr\rangle$. *Since v was visible, it is either a local version, or for all* $\langle i, h\rangle \in v.ds$, *DSV had a greater value than h. According to Observation 12, DSV never decreases, the condition is still valid. Thus, according to Line 3 of Algorithm 5.2, it is impossible to return* v' *(contradiction).*

When a *GET* operation returns a version, either the version is a local version, or the entries of DSV in the hosting partition is greater than the specified dependency for the version (see Line 3 of Algorithm 5.2). Specifically, we have the following observation,

Observation 13 *Let v be a version for key k that is written in partition* p^m_n. *If* $GET(k)$ *returns v in data center* $i \neq m$, *then for any member* $\langle j, h\rangle \in v.ds$, $DSV^i_n[j] \geq h$.

Lemma 6 *Let* v_1 *be a version written in data center i. Then, for any non-local version* v_2 *such that* v_1 dep v_2, *there must be a client c at data center i that has read either* v_2, *or a non-local version* v_3 *such that* v_3 dep v_2.

Proof 8 *According to Definition 9, (event of writing* v_2*)* $\rightarrow$ *(event of writing* v_1*). Since* v_1 *is a local version, and* v_2 *is not a local version, a non-local version* v_3 *must be read at data center i in event e*

such that (event of writing v_2) $\rightarrow$ e and e $\rightarrow$ (event of writing v_1). By definition 8, (event of writing v_3) $\rightarrow$ e. Thus, $v_3 = v_2$, or (event of writing v_2) $\rightarrow$ (event of writing v_3) that leads to v_3 dep v_2.

Lemma 7 *Once client c reads local version v_1 by a GET operation, any version v_2 such that v_1 dep v_2 is visible+ for GET to c.*

Proof 9 *Wlog assume v_1 and v_2 are versions for keys k_1, and k_2 written in partitions n_1 and n_2, respectively. Also, assume v_1 is written in data center i, and v_2 is written in data center m. According to Lemma 3, we have two cases:*

Case 1 *v_2 has not arrived partition $p^i_{n_2}$*

Since v_1 is a local version, and v_2 is not a local version, according to Lemma 6, we have two cases:

- *there is a client at data center i that has read v_2*

  Contradiction to assumption that v_2 has not arrived partition $p^i_{n_2}$.

- *there is a client at data center i that has read non-local version v_3 such that v_3 dep v_2*

  Wlog, assume v_3 is written in partition n_3.

  Since v_2 has not arrived data center i, according to Observation 9, $DSV^i_{real}[m] < v_2.ut$.

  Since v_3 dep v_2, according to Observation 6, there is $\langle m, h\rangle \in v_3.ds$ such that $h \geq v_2.ut$.

  We have two cases for reading v_3:

  - *v_3 has been read by a GET operation: Since v_3 is not a local version, according to Observation 13 for any $\langle m, h\rangle \in v_3.ds$, $DSV^m_{n_1}[m] \geq h$. According to Lemma 4, $DSV^i_{n_1}[m] \leq DSV^i_{real}[m]$. Thus, $DSV^i_{real}[m] \geq v_2.ut$ (contradiction).*
  - *v_3 has been read by an ROTX operation: for any $\langle m, h\rangle \in v_3.ds$, $sv[m] \geq h$. According to Observation 11, $sv[m] \leq DSV^i_{real}[m]$. Thus, $DSV^i_{real}[m] \geq v_2.ut$ (contradiction).*

Case 2 *there is a member $\langle k, h\rangle \in v_2.ds$, such that $DSV^i_{n_2}[k] < h$.*

Since v_1 dep v_2, according to Observation 5, there is member $\langle k, h' \rangle \in v_1.ds$ such that $h' \geq h$. According to Line 7 of Algorithm 5.2, we update the DSV value with the set of dependecies when we write a new version in a PUT operation. Thus, $DSV^i_{n_1}[k] \geq h'$ that leads to $DSV^i_{n_1}[k] \geq h$. Since client read v_1 before reading k_2, the $DSV^i_{n_2}[k]$ at time of reading k_2 is greater or equal to $DSV^i_{n_1}$, according to Lines 4 of Algorithm 5.1, and Line 2 of Algorithm 5.2. Thus, $DSV^i_{n_2}[k] \geq h$ (contradiction).

Theorem 5 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal+ consistent for $\{GET\}$ with reader operations $\{GET\}$ for f = last-writer-wins.*

Proof 10 *We prove this theorem by showing that CausalSpartanX protocol satisfies Definition 14.*

The first condition is satisfied based on the fact that clients are sticky and Lemma 2. The second condition is satisfied based on Lemma 5. We prove the third condition via contradiction:

Let k_1 and k_2 be any two arbitrary keys in the store residing in partitions n_1 and n_2. Let v_1 be a version of key k_1, and v_2 be a version of key k_2 such that v_1 dep v_2. Now suppose client c reads v_1 in data center i via a GET operation, but v_2 is not visible+ for a GET operation to client c. Let v_2 be a version written in a data center m. According to Lemma 3, two cases are possible:

Case 1 *v_2 has not arrived partition $p^i_{n_2}$.*

Since v_2 has not arrived at data center i, according to Observation 10, $DSV^i_{n_1}[m] < v_2.ut$. Since, v_1 dep v_2, according to Observation 6, $v_1.ds$ has member $\langle m, h \rangle$ such that $h \geq v_2.ut$. According to Lemma 7, v_1 is not a local version. Since $GET(k_1) = v_1$, and v_1 is not a local version, according to Observation 13, for any member $\langle j, h \rangle \in v.ds$, $DSV^i_{n_1}[j] \geq h$. Thus, $DSV^i_{n_1}[m] \geq v_2.ut$ (contradiction).

Case 2 *v_2 has arrived, but there is a member $\langle k, h \rangle \in v_2.ds$, such that $DSV^i_{n_2}[k] < h$.*

Since client c asks for k_2 after reading v_1, according to Line 2 of Algorithm 5.2, $DSV^i_{n_2}[k]$ at the time of reading k_2 is higher than $DSV^i_{n_1}[k]$ at the time of reading k_1 ($DSV^i_{n_1}[k] \leq DSV^i_{n_2}[k]$). Since v_1 dep v_2, according to Observation 5, there is member $\langle k, h' \rangle \in v_1.ds$

such that $h' \geq h$. According to Lemma 7, v_1 is not a local version. Since $GET(k_1) = v_1$, and v_1 is not a local version, according to Observation 13, $DSV^i_{n_1}[k] \geq h'$ that leads to $DSV^i_{n_2}[k] \geq h$ (contradiction).

With Lemma 2 and Theorem 5, we have the following corollary:

Corollary 1 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal++ consistent for $\{GET\}$ for reader operations $\{GET\}$ and instantly visible operations $\{GET\}$ for conflict resolution function $f =$ last-writer-wins.*

Lemma 8 *Once a client c reads version v of key k at partition n by an $\{ROTX\}$ operation at data center i, it remains visible+ for $\{GET\}$ for c.*

Proof 11 *Since the client has read the version, it is obviously written. According to Lemma 2, v_2 is not a local version. Now, according to Lemma 3, two cases are possible:*

Case 1 *v has not arrived partition data center i.*

Contradiction to the assumption that the client has read the version.

Case 2 *v has arrived, but there is a member $\langle j, h\rangle \in v.ds$, such that $DSV^i_n[j] < h$.*

Since client c asks for k after reading v by an ROTX operations, according to Line 2 of Algorithm 5.2, $DSV^i_n[j]$ at the time of reading k using GET is higher than or equal to $DSV_c[j]$ after reading k by ROTX operation. Since ROTX has read v_1, according to Line 22 of Algorithm 5.4, $sv[j] \geq h$. Since $sv[j] \leq DSV_c[j]$, $DSV_c[j] \geq h$ that leads to $DSV^i_n[j] \geq h$ at the time GET operation (contradiction).

Theorem 6 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal+ consistent for $\{GET\}$ with reader operations $\{ROTX\}$ for $f =$ last-write-wins.*

Proof 12 *We prove this theorem by showing that CausalSpartanX protocol satisfies Definition 14.*

The first condition is satisfied based on the fact that clients are sticky and Lemma 2. The second condition is satisfied based on Lemma 8. We prove the third condition via contradiction:

Let k_1 and k_2 be any two arbitrary keys in the store residing in partitions n_1 and n_2. Let v_1 be a version of key k_1, and v_2 be a version of key k_2 such that v_1 dep v_2. Now suppose client c reads v_1 in data center i via an ROTX operation, but v_2 is not visible+ for a GET operation to client c.

According to Lemma 2, v_2 is not a local version. Let v_2 be a version written in a data center m. According Lemma 3, two cases are possible:

Case 1 *v_2 has not arrived partition $p^i_{n_2}$.*

Since v_2 has not arrived at data center i, according to Observation 9, $DSV^i_{real}[m] < v_2.ut$. Since, v_1 dep v_2, according to Observation 6, $v_1.ds$ has member $\langle m, h\rangle$ such that $h \geq v_2.ut$. Since an ROTX operation has read v_1, for any member $\langle j, h\rangle \in v.ds$, $sv[j] \geq h$. According to Observation 11, $sv[m] \leq DSV^i_{real}[m]$. Thus, $DSV^i_{real}[m] \geq v_2.ut$ (contradiction).

Case 2 *v_2 has arrived, but there is a member $\langle k, h\rangle \in v_2.ds$, such that $DSV^i_{n_2}[k] < h$.*

Since client c asks for k_2 after reading v_1, according to Line 2 of Algorithm 5.2, $DSV^i_{n_2}[k]$ at the time of reading k_2 is higher than $DSV_c[k]$ after reading k_1. Since v_1 dep v_2, according to Observation 5, there is member $\langle k, h'\rangle \in v_1.ds$ such that $h' \geq h$. Since ROTX has read v_1, according to Line 22 of Algorithm 5.4, $sv[k] \geq h'$. Since $sv \leq DSV_c$, $DSV_c[k] \geq h'$ that leads to $DSV^i_{n_2}[k] \geq h$ (contradiction).

With Lemma 2 and Theorem 6, we have the following corollary:

Corollary 2 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal++ consistent for $\{GET\}$ for reader operations $\{ROTX\}$ and instantly visible operations $\{GET\}$ for conflict resolution function f = last-writer-wins.*

According to Corollaries 1 and 2, and Observation 1, we have the following corollary:

Corollary 3 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal++ consistent for $\{GET\}$ for reader operations $\{GET, ROTX\}$ and instantly visible operations $\{GET\}$ for conflict resolution function f = last-write-wins.*

5.4.2 Causal Consistency for ROTX Operations

Lemma 9 *Let v be a version that is written in partition p_n^m. If v is not visible+ for $\{ROTX\}$ for conflict resolution function f = last-write-wins to a client in data center i, then*

- *v has not arrived partition p_n^i, or*
- *there is a member $\langle j, h\rangle \in v.ds$, such that $sv[j] < h$, or*
- *$sv[v.sr] < v.ut$.*

Proof 13 *The poof of this lemma is the same as that of Lemma 3, except we must note Line 22 of Algorithm 5.4.*

Lemma 10 *All versions written by client c are visible+ for $\{ROTX\}$ to client c.*

Proof 14 *Suppose version v written by client c is not visible to ROTX. Suppose client c accesses data center i. Now, according to Lemma 9, we have following cases:*

Case 1 *v has not arrived data center i Impossible, since the version is written by client c*

- *there is a member $\langle j, h\rangle \in v.ds$, such that $sv[j] < h$ Since the client has written version v, for any $\langle j, h\rangle \in v.ds$, $DS_c[j] > h$. According to Line 10, $sv[j] > h$ (contradiction).*
- *$sv[v.sr] < v.ut$. Since the client has written version v, $DS_c[v.sr] > v.ut$. According to Line 10, $sv[j] > v.ut$ (contradiction).*

Lemma 11 *Once client c reads local version v_1 by a GET operation, any version v_2 such that v_1 dep v_2 is visible+ for $\{ROTX\}$ to c.*

Proof 15 *Wlog assume v_1 and v_2 are versions for keys k_1, and k_2 written in partitions n_1 and n_2, respectively. Also, assume v_2 is written in data center m. If version v_2 is not visible+ for $\{ROTX\}$, Lemma 9, we have three cases:*

- *v_2 has not arrived partition p_n^i*

  In this case, with the same argument provided in Case 1 of proof of Lemma 7, we face a contradiction.

- *there is a member $\langle k, h\rangle \in v_2.ds$, such that $sv[k] < h$.*

  Since v_1 dep v_2, according to Observation 5, there is member $\langle k, h'\rangle \in v_1.ds$ such that $h' \geq h$. According to Line 7 of Algorithm 5.2, we update the DSV value with the set of dependencies when we write a new version in a PUT operation. Thus, $DSV_{n_1}^i[k] \geq h'$ that leads to $DSV_{n_1}^i[k] \geq h$. Since client read v_1 before reading k_2, the $DSV_c[k]$ at time of reading k_2 is greater or equal to $DSV_{n_1}^i$. According to Line 10, and 12 of Algorithm 5.4, the sv of the ROTX operation is entry-wise greater than DSV_c. Thus, $sv[k] \geq h$ (contradiction).

- *$sv[v_2.sr] < v_2.ut$.*

  Since v_1 dep v_2, according to Observation 6, $\langle v_2.sr, h\rangle \in v_1.ds$ such that $h \geq v_2.ut$. Since client read v_1 before reading k_2, $ds[v.sr] \geq h$ for ROTX operation reading k_2. According to Line 10 and 12, $sv[v_2.sr] \geq v_2.ut$ (contradiction).

Lemma 12 *When client c reads a version v using GET operation, it remains visible+ for $\{ROTX\}$ for c.*

Proof 16 *Suppose v is not visible+ for $\{ROTX\}$ to client c. Since client has read the version before, obviously it is written. According to Lemma 9, there are three cases:*

Case 1 *v has not arrived the hosting partition.*

Contradiction to the assumption that the client has read the version.

Case 2 *there is a member $\langle j, h\rangle \in v.ds$, such that $sv[j] < h$*

Since the client has read v, the $DSV_c[j] \geq h$. According to Line 10, and 12 of Algorithm 5.4, the sv of the ROTX operation is entry-wise greater than DSV_c. Thus, $sv[j] \geq h$ (contradiction).

Case 3 $sv[v.sr] < v.ut$.

Since the client has read v, the $DS_c[v.sr] \geq v.ut$ after reading v. According to Line 10, and 12 of Algorithm 5.4, the sv of the ROTX operation is entry-wise greater than DS_c. Thus, $sv[v.sr] \geq v.ut$ (contradiction).

Theorem 7 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal+ consistent for $\{ROTX\}$ with reader operations $\{GET\}$ for f = last-write-wins.*

Proof 17 *We prove this theorem by showing that CausalSpartanX protocol satisfies Definition 14.*

The first condition is satisfied according to Lemma 10. The second condition is satisfied based on Lemma 12. We prove the third condition via contradiction:

Let k_1 and k_2 be any two arbitrary keys in the store residing in partitions n_1 and n_2. Let v_1 be a version of key k_1, and v_2 be a version of key k_2 such that v_1 dep v_2. Now suppose client c reads v_1 in data center i via a GET operation, but v_2 is not visible+ for $\{ROTX\}$ operation to client c.

Let v_2 be a version written in a data center m. According to Lemma 9, three cases are possible:

Case 1 *v_2 has not arrived partition $p^i_{n_2}$.*

Since v_2 has not arrived at data center i, according to Observation 9, $DSV^i_{real}[m] < v_2.ut$, and according to Observation 10, $DSV^i_{n_1}[m] < v_2.ut$. Since, v_1 dep v_2, according to Observation 6, $v_1.ds$ has member $\langle m, h\rangle$ such that $h \geq v_2.ut$. According to Lemma 11, v_1 is not a local version. Since $GET(k_1) = v_1$, and v_1 is not a local version, according to Observation 13, for any member $\langle j, h\rangle \in v.ds$, $DSV^i_{n_1}[j] \geq h$. Thus, $DSV^i_{n_1}[m] \geq v_2.ut$ (contradiction).

Case 2 *there is a member $\langle j, h\rangle \in v.ds$, such that $sv[j] < h$.*

Since v_1 dep v_2, according to Observation 5, for any member $\langle j, h\rangle \in v_2.ds$, there exists $\langle j, h'\rangle \in v_1.ds$ such that $h' \geq h$. Since the client has read v_1 by a GET operations, the $DSV_c[j] \geq h'$. According to Line 10, and 12 of Algorithm 5.4, the sv of the ROTX operation is entry-wise greater than DSV_c. Thus, $sv[j] \geq h'$. This leads to $sv[j] \geq h$ (contradiction).

Case 3 $sv[v_2.sr] < v_2.ut$.

Since v_1 dep v_2, according to Observation 6, $\langle v_2.sr, h\rangle \in v_1.ds$ such that $h \geq v_2.ut$. Since client read v_1 before reading k_2, $ds[v.sr] \geq h$ for ROTX operation reading k_2. According to Line 10 and 12 of Algorithm 5.4, $sv[v_2.sr] \geq v_2.ut$ (contradiction).

With Lemma 2 and Theorem 7, we have the following corollary:

Corollary 4 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal++ consistent for $\{ROTX\}$ for reader operations $\{GET\}$ and instantly visible operations $\{GET\}$ for conflict resolution function f = last-writer-wins.*

Lemma 13 *When client c read a version v using ROTX operation, it remains visible+ for $\{ROTX\}$ for c.*

Proof 18 *Suppose v is not visible+ for $\{ROTX\}$ to client c. Since client has read the version before, and according to Lemma 9, there are two cases:*

Case 1 *v has not arrived the hosting partition.*

Contradiction to the assumption that the client has read the version.

Case 2 *there is a member $\langle j, h\rangle \in v.ds$, such that $sv[j] < h$*

Since the client has read v, the $DSV_c[j] \geq h$. According to Line 10, and 12 of Algorithm 5.4, the sv of the ROTX operation is entry-wise greater than DSV_c. Thus, $sv[j] \geq h$ (contradiction).

Case 3 $sv[v.sr] < v.ut$.

Since the client has read v, the $DS_c[v.sr] \geq v.ut$ after reading v. According to Line 10, and 12 of Algorithm 5.4, the sv of the ROTX operation is entry-wise greater than DS_c. Thus, $sv[v.sr] \geq v.ut$ (contradiction).

Theorem 8 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal+ consistent for $\{ROTX\}$ with reader operations $\{ROTX\}$ for f = last-writer-wins.*

Proof 19 *We prove this theorem by showing that CausalSpartanX protocol satisfies Definition 14.*

The first condition is satisfied according to Lemma 10. The second condition is satisfied based on Lemma 13. We prove the third condition via contradiction:

Let k_1 and k_2 be any two arbitrary keys in the store residing in partitions n_1 and n_2. Let v_1 be a version of key k_1, and v_2 be a version of key k_2 such that v_1 dep v_2. Now suppose client c reads v_1 in data center i via an ROTX operation, but v_2 is not visible+ for $\{ROTX\}$ operation to client c.

Let v_2 be a version written in a data center m. According to Lemma 9, three cases are possible:

Case 1 *v_2 has not arrived partition $p^i_{n_2}$.*

Since v_2 has not arrived at data center i, according to Observation 9, $DSV^i_{real}[m] < v_2.ut$. Since, v_1 dep v_2, according to Observation 6, $v_1.ds$ has member $\langle m, h\rangle$ such that $h \geq v_2.ut$. Since an ROTX operation has read v_1, for any member $\langle j, h\rangle \in v.ds$, $sv[j] \geq h$. According to Observation 11, $sv < DSV^i_{real}$. Thus, $DSV_{real}[m] \geq v_2.ut$ (contradiction).

Case 2 *there is a member $\langle j, h\rangle \in v.ds$, such that $sv[j] < h$.*

Since v_1 dep v_2, according to Observation 5, for any member $\langle j, h\rangle \in v_2.ds$, there exists $\langle j, h'\rangle \in v_1.ds$ such that $h' \geq h$. Since the client has read v_1 by a ROTX operations, the $sv[j] \geq h'$ where sv_1 is that sv of the ROTX reading v_1. Since $sv_1 < DSV_c$, $DSV_c[j] > h'$. According to Line 10, and 12 of Algorithm 5.4, the sv of the ROTX operation is entry-wise greater than DSV_c. Thus, $sv[j] \geq h'$. This leads to $sv[j] \geq h$ (contradiction).

Case 3 *$sv[v_2.sr] < v_2.ut$.*

Since v_1 dep v_2, according to Observation 6, $\langle v_2.sr, h\rangle \in v_1.ds$ such that $h \geq v_2.ut$. Since client read v_1 before reading k_2, $ds[v.sr] \geq h$ for ROTX operation reading k_2. According to Line 10 and 12 of Algorithm 5.4, $sv[v_2.sr] \geq v_2.ut$ (contradiction).

With Lemma 2 and Theorem 8, we have the following corollary:

Corollary 5 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal++ consistent for* $\{ROTX\}$ *for reader operations* $\{ROTX\}$ *and instantly visible operations* $\{GET\}$ *for conflict resolution function* $f =$ last-writer-wins.

According to Corollaries 4 and 5 and Observation 1 in Section 2.3, we have the following corollary:

Corollary 6 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal++ consistent for* $\{ROTX\}$ *for reader operations* $\{GET, ROTX\}$ *and instantly visible operations* $\{GET\}$ *for conflict resolution function* $f =$ last-writer-wins.

Now, based on Corollaries 3 and 6 and Observation 2, we conclude our desired consistency requirement. Specifically,

Corollary 7 *The data store running CausalSpartanX protocol defined in Section 5.2 is causal++ consistent for* $\{GET, ROTX\}$ *for reader operations* $\{GET, ROTX\}$ *and instantly visible operations* $\{GET\}$ *for conflict resolution function* $f =$ last-writer-wins.

5.4.3 Causal Consistency of Values Returned by $ROTX$ Operations

We formally define this requirement as follows. First, we define visiblity+ for a set of versions:

Definition 19 (Visible+ to a Set of Versions) *Let* $vset$ *be a set of version. We say version* v *is visible+ for conflict resolution function* f *to* $vset$*, if* $vset$ *does not include version* v' *such that* $v' \neq v$ *and* $v = f(v, v')$.

Now, we define causal consistency for a set of versions:

Definition 20 (Causal Consistency for a Set of Versions) *Let* $vset$ *be a set of versions.* $vest$ *is causally consistent for conflict resolution function* f*, if for any version* $v_1 \in vset$*, any version* v_2 *such that* v_1 dep v_2 *is visible+ for* f *to* $vest$.

A group of values returned by any *ROTX* is causally consistent. Specifically,

Theorem 9 *Let $vset$ be a set of versions returned by an ROTX operation. $vset$ is causally consistent for f = last-writer-wins.*

Proof 20 *We prove this theorem by contradiction:*

Let $kset$ and $vset$ respectively be the set of keys read and the set of of versions returned by ROTX operation a performed by client c at data center i. Let v_1 and v_2 be versions for key $k_1 \in kset$ and $k_2 \in kset$, respectively, such that v_1 dep v_2. Suppose $v_1 \in vset$, but v_2 is not visible+. When v_2 is not visible+ to $vset$, it means it is no visible+ for $\{ROTX\}$ to client c.

We have following cases for v_2:

Case 1 *v_2 is not written when ROTX reads k_2:*

*If v_2 is a remote version, this case is the same as **Case 2.1** where v_2 has not been arrived. Thus, we focus on the situation where v_2 is a local version. Now, we consider two cases for v_1:*

Case 1.1 *v_1 is a remote version:*

Suppose v_1 is written in data center j. Since v_1 dep v_2 and v_2 is not written at the time of ROTX begins, v_1 also is not written at that time. Thus, data center could not have received v_1 when it starts the ROTX operation. According to Observation 10, $DSV^i_{real}[j] < v_1.ut$. According to Observation 11, $sv[j] < v_1.ut$. However, since ROTX read v_1, $sv[j] \geq v_1.ut$ (contradiction).

Case 1.2 *v_1 is a local version:*

According to Line 21 of Algorithm 5.4, $DSV^i_{n_2}[i] \geq sv[i]$, after ROTX read k_2. Since v_2 is written after ROTX read k_2, according to Line 8 of Algorithm 5.2 and the algorithm of updateHLC, $v_2.ut > DSV^i_{n_2}[i]$ that leads to $v_2.ut > sv[i]$. Since v_1 dep v_2, according to Observation 6, $\langle i, h \rangle \in v_1.ds$ such that $h \geq v_2.ut$. Thus, $h > sv[i]$. However, since ROTX reads v_1, $sv[i] \geq h$ (contradiction).

Case 2 *v_2 is written when ROTX reads k_2:*

In this case, according to Lemma 9, two cases are possible for v_2:

Case 2.1 *v_2 has not arrived partition $p^i_{n_2}$.*

Since v_2 has not arrived at data center i, according to Observation 9, $DSV_{real}[m] < v_2.ut$. Since, v_1 dep v_2, according to Observation 6, $v_1.ds$ has member $\langle m, h\rangle$ such that $h \geq v_2.ut$. Since the ROTX operation has read v_1, for any member $\langle j, h\rangle \in v.ds$, $sv[j] \geq h$. According to Observation 11, $sv \leq DSV^i_{real}$. Thus, $DSV^i_{real}[m] \geq v_2.ut$ (contradiction).

Case 2.2 *there is a member $\langle j, h\rangle \in v_2.ds$, such that $sv[j] < h$.*

Since v_1 dep v_2, according to Observation 5, for any member $\langle j, h\rangle \in v_2.ds$, there exists $\langle j, h'\rangle \in v_1.ds$ such that $h' \geq h$. Since the client has read v_1 by the ROTX operation, $sv[j] \geq h'$. This leads to $sv[j] \geq h$ (contradiction).

Case 2.3 *$sv[v_2.sr] < v_2.ut$.*

Since v_1 dep v_2, according to Observation 6, $\langle v_2.sr, h\rangle \in v_1.ds$ such that $h \geq v_2.ut$. Since ROTX reads v_1, $sv[v_2.sr] \geq h$ which leads to $sv[v_2.sr] \geq v_2.ut$ (contradiction).

5.4.4 Convergence for *GET* Operations

Now, we focus on the convergence aspect of the protocol. First, we observe that DSV values for connected servers never stop increasing via replicate or heartbeat messages. Thus, we have

Observation 14 *If data center i is connected to data center j, for all $1 \leq k \leq N$, $DSV^j_k[i]$ and $DSV^i_k[j]$ will never stop increasing.*

We have the following theorem about the convergence of our protocol,

Theorem 10 *The data store running CausalSpartanX protocol defined in Section 5.2 is convergent for $\{GET\}$ for conflict resolution function* last-writer-wins.

Proof 21 *We prove this theorem by showing that CausalSpartanX protocol satisfies Definition 16 via proof by contradiction:*

Let v_1 be a version for key k written in partition p_n^m. Let

- *data center i be connected to data center m, and*
- *for any version v_2 such that v_1 dep v_2, data center i be connected to data center j where version v_2 is written.*

Now, suppose v_1 is never visible+ in i.

When v_1 is never visible+ in i, according to Lemma 3, two cases are possible for v_1:

Case 1 *v_1 will never reach partition p_n^i.*

Since data center i is connected to data center m, p_n^i will receive replicate message sent in Line 19 of Algorithm 5.2 (contradiction).

Case 2 *there is a member $\langle j, h\rangle \in v_1.ds$, such that $DSV_n^i[j]$ will always remain smaller than h.*

This is a contradiction to Observation 14 and the second assumption of the theorem.

5.4.5 Convergence for ROTX Operations

Theorem 11 *The data store running CausalSpartanX protocol defined in Section 5.2 is convergent for $\{ROTX\}$ for conflict resolution function* last-writer-wins.

Proof 22 *We prove this theorem by showing that CausalSpartanX-X protocol satisfies Definition 16 via proof by contradiction:*

Let v_1 be a version for key k written in partition p_n^m. Let

- *data center i be connected to data center m, and*
- *for any version v_2 that v_1 dep v_2, data center i be connected to data center j where version v_2 is written.*

Now, suppose v_1 is never visible+ in i.

When v_1 is never visible+ in i, according to Lemma 9, following cases are possible for v_1:

Case 1 *v_1 will never reach at partition p_n^i.*

Since data center i is connected to data center m, p_n^i will receive replicate message sent in Line 19 of Algorithm 5.2 (contradiction).

Case 2 *there is a member $\langle j, h\rangle \in v_1.ds$, such that $sv[j]$ will always remain smaller than h, or $sv[v_1.sr]$ always remains smaller than $v_1.ut$*

This is a contradiction to Observation 14 and the second assumption of the theorem, and the fact that sv is entry-wise greater than DSV.

According to Theorem 10 provided in Section 5.4.4, Theorem 11 provided in this section, and Observation 4 provided in Section 2.3, we conclude our desired convergence requirement. Specifically,

Corollary 8 *The data store running CausalSpartanX protocol defined in Section 5.2 is convergent for $\{GET, ROTX\}$ for conflict resolution function* last-writer-wins.

Corollaries 7 and 8 and Theorem 9 provide the correctness of CausalSpartanX.

5.5 Experimental Results

We have implemented CausalSpartanX protocol in a distributed key-value store called MSU-DB. MSU-DB is written in Java, and it can be downloaded from [11]. MSU-DB uses Berkeley DB [3] in each server for data storage and retrieval. For comparison purposes, we have implemented GentleRian in the same code base.

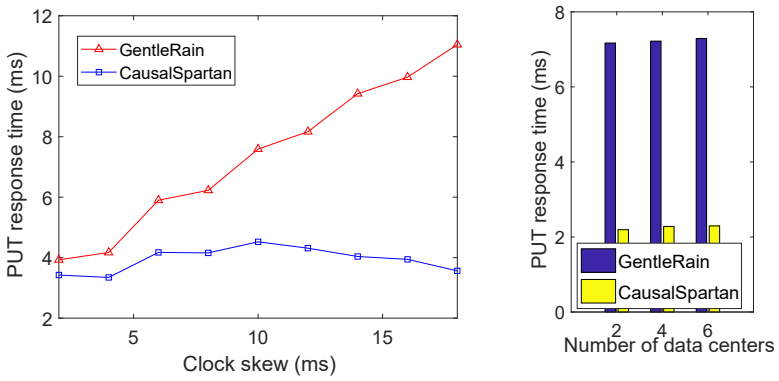
We run all of our experiments on Amazon AWS [1] on `c3.large` instances running Ubuntu 14.04. The specification of servers is as follows: 7 ECUs, 2 vCPUs, 2.8 GHz, Intel Xeon E5-2680v2, 3.75 GiB memory, 2 x 16 GiB Storage Capacity.

First, in Section 5.5.1, we investigate the effect of clock skew on PUT latency. Next, in Section 5.5.2, we evaluate the effect of this increased PUT latency along with query amplification. We analyze the effectiveness of CausalSpartanX in reducing update visibility latency by analysis of a typical collaborative application in Section 5.5.3. Finally, we evaluate the overhead of CausalSpartanX by comparing the throughput of CausalSpartanX and GentleRain in Section 5.5.4 in cases where clocks are perfectly synchronized.

5.5.1 Response Time of PUT Operations

To study the effect of clock skew on the response time accurately, we need to have a precise clock skew between servers. However, the clock skew between two different machines depends on many factors out of our control. To have a more accurate experiment, we consolidate two virtual servers on a single machine and impose an artificial clock skew between them. Then, we change the value of the clock skew and observe its effect on the response time for PUT operations. A client sends PUT requests to the servers in a round robin fashion. Since the physical clock of one server is behind the physical clock of the other server, half of the PUT operations will be delayed by the GentleRain. On the other hand, CausalSpartanX does not delay any PUT operation, and processes them immediately. We compute the average response time for PUT operations with value size 1K. Figure 5.1-(a) shows that average response time for PUT operation in GentleRain grows as the clock skew grows, while the average response time in CausalSpartanX is independent of clock skew.

Next, we do the same experiment when the servers are running on two different machines without introducing any artificial clock skew. We run NTP [14] on servers to synchronize physical clocks. In other words, this simulates the exact condition that is expected to happen in an ideal scenario where we have two partitions within the same physical location. In this setting, the client sends PUT requests to these servers in a round robin manner. Figure 5.1-(b) shows average delay of PUT operations in GentleRain and CausalSpartanX. We observe that in this case, the effect of PUT latency is visible even though the servers are physically collocated and have clocks that are synchronized with NTP.



(a) (b)

Figure 5.1: The effect of clock skew on PUT response time: a) with accurate artificial clock skew when servers are running on the same physical machine b) without any artificial clock skew when servers are running on different physical machines synchronized with NTP.

5.5.2 Query Amplification

In this section, we want to evaluate the effectiveness of CausalSpartanX with query amplification. As explained in Section 5.1.2, a single user request can generate many internal queries. We define *query amplification factor* as the number of internal queries that are generated for a single request. In this section, unlike the previous section where we computed average response time for the queries, we compute the average response time for *requests* each of which contains several queries specified by the query amplification factor.

Now, we want to study how the average response time changes as the query amplification factor changes. We simulate the scenario where the user sends requests to a web server, and each request generates multiple internal PUT operations. The web server sends PUT operations to partitions in a round robin fashion. The user request is satisfied once all PUT operations are done. We compute the average response time for different query amplification factors.

Figure 5.2 shows average response time versus query amplification factor, when we have two partitions for different clock skews. As query amplification factor increases, the response time in both GentleRain and CausalSpartanX increases. This is expected since each request now contains more work to be done. However, the rate of growth in CausalSpartanX is significantly slower.

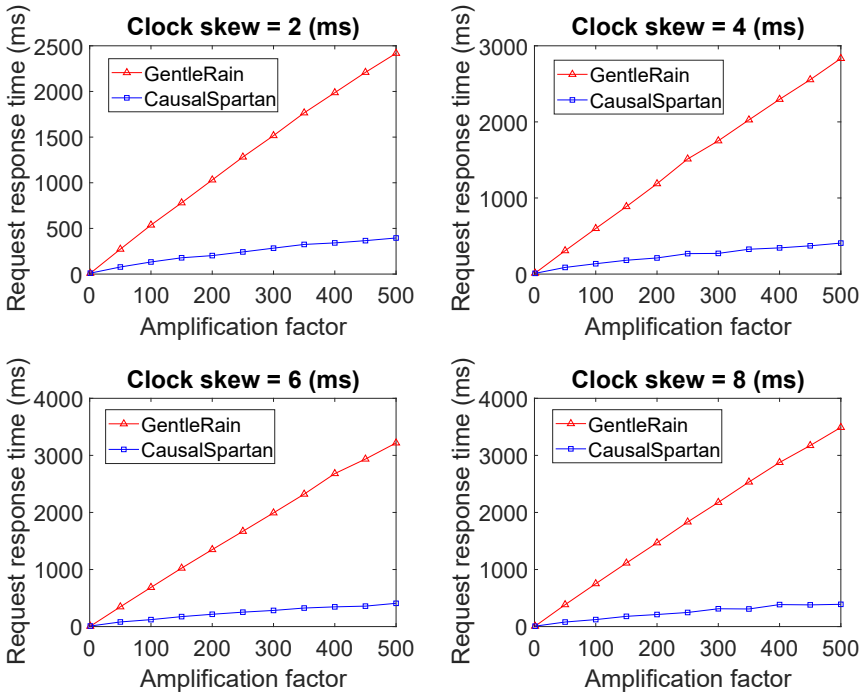
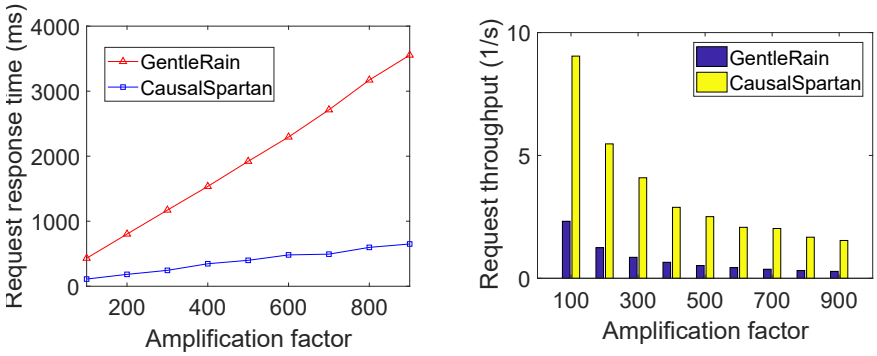


Figure 5.2: The effect of different values of clock skew on request response time for different query amplification factor in GentleRain and CausalSpartanX.

For example, for only 2 (ms) clock skew, the response time of a request with amplification factor 100 in GentleRain is 4 times higher than that in CausalSpartanX. Note that in practical systems higher clock skews are possible [52, 55]. For example, clock skew up to 100 (ms) is possible when the underlying network suffers from asymmetric links [55]. In this case, for a query amplification factor of 100, the response time of GentleRain is 35 times higher than CausalSpartanX.

For results shown in Figure 5.2, we used a controlled artificial clock skew to study the effect of clock skew accurately. Figure 5.3-(a) shows effect of amplification factor on request response time when there is no artificial clock skew, and servers are synchronized with NTP. It shows how real clock skew between synchronized servers that use NTP affects request response time. For instance, for query amplification factor 100, our experiments show that the response time of GentleRain is 3.89 times higher than that of CausalSpartanX. Figure 5.3-(b) also shows the client request throughput in GentleRain and CausalSpartanX for different query amplification factor.



(a) Request response time

(b) Request throughput

Figure 5.3: The effect of amplification factor on client request response time and throughput when we have 8 partitions and 6 data centers, and all partitions are synchronized by NTP without any artificial clock skew.

5.5.3 Update Visibility Latency

In this section, we want to focus on update visibility latency which is another important aspect of a distributed data store. Update visibility latency is the delay before an update becomes visible in a remote replica. Update visibility latency is ultimately important for today's cloud services, as even a few milliseconds matters for many businesses [19]. In GentleRain, only one far replica adversely affects the whole communication in the system by increasing the update visibility latency. In CausalSpartanX, we use a vector (DSV) with one entry for each data center instead of a single scalar (GST) as used in GentleRain. As a result, a long network latency of a data center only affects the communication with that specific data center, and does not affect independent communication between other data centers.

To investigate how CausalSpartanX performs better than GentleRain regarding update visibility latency, we do the following experiment: We run a data store consisting of three data centers A, B, and C. Client c_1 at data center A communicates with client c_2 at data center B via key k as follows: client c_1 keeps reading the value of key k and increments it whenever finds it to be an odd number. Similarly, client c_2 keeps reading the value of key k, and increments it whenever finds it to be an even number. The locations of data centers A and B are fixed, and they are both in California. We change the location of data center C to see how its location affects the communication between c_1

and c_2. Table 5.1 shows the round trip times for different locations of data center C.

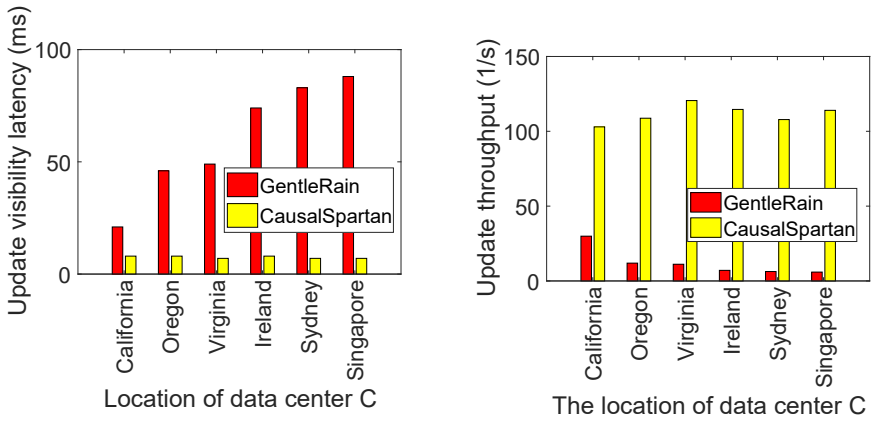
Table 5.1: Round trip times.

Location of data center C	RTT to data center A (ms)	RTT to data center B (ms)
California	1.1709114	0.3201521
Oregon	21.8699663	20.6107391
Virginia	67.0469505	61.2305881
Ireland	138.2809544	139.3212938
Sydney	159.0899451	158.4004238
Singapore	175.6392972	175.6030464

We measure the update visibility latency as the time elapsed between writing a new update by a client and reading it by another client. We use the timestamp of updates to compute the update visibility latency. Thus, because of clock skew, the values we compute are an estimation of actual update visibility latency. Figure 5.4-(a) shows the update visibility latency is lower in CausalSpartanX than that in GentleRain. Also, the update visibility latency in GentleRain increases as the network delay between data center C and A/B increases. For example, when data center C is in Oregon the update visibility latency in CausalSpartanX is 83% lower than that in GentleRain. This value increases to 92% when data center C is in Singapore. Figure 5.4-(b) shows the throughput of communication between clients as the number of updates by a client per second. The location of data center C affects the throughput of GentleRain, while the throughput of CausalSpartanX is unaffected.

5.5.4 Throughput Analysis and Overhead of CausalSpartanX

CausalSpartanX utilizes HLC to eliminates the PUT latency, and utilizes DSV to improve update visibility latency. In this section, we analyze the overhead of these features in the absence of clock skew, query amplification or the collaborative nature of the application. In particular, we analyze the throughput when GET/PUT operations by the client are unrelated to each other.



(a) Update visibility latency

(b) Update throughput

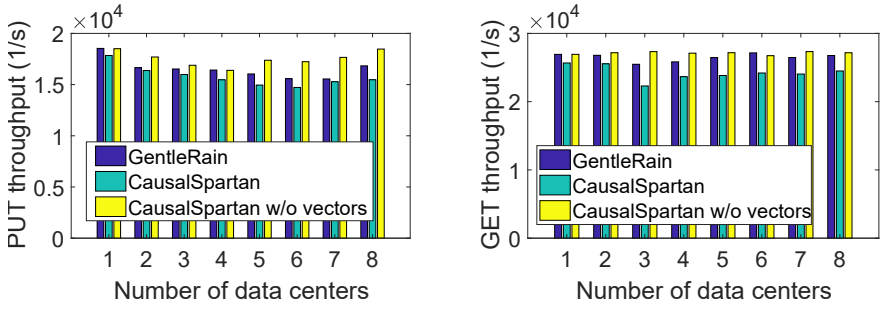
Figure 5.4: How the location of an irrelevant data center adversely affects a collaborative communication in GentleRain, while CausalSpartanX is unaffected.

Since the two features of CausalSpartanX, the use of HLC and the use of DSV are independent, we analyze the throughput with just the use of HLC and with both features. Figure 5.5 demonstrates the throughput of GET and PUT operations. We observe that when GET/PUT operations are independent, then throughout of CausalSpartanX is 5% lower than GentleRain. However, the throughput of CausalSpartanX with just HLC (and not DSV) is virtually identical to that of GentleRain. We note that additional experiments comparing CausalSpartanX with just using HLC are available in [63]. They show that just using HLC does not add to the overhead of CausalSpartanX.

Even though there is a small overhead of CausalSpartanX when GET/PUT operations are unrelated, we observe that with query amplification (that causes some PUT operations to be delayed in GentleRain), the request throughput of CausalSpartanX is higher (cf. Figure 5.3). Thus, while the throughput of basic PUT/GET operations is slightly higher in GentleRain, the throughput of actual requests issued by the end users is expected to be higher in CausalSpartanX.

5.5.5 Performance of ROTX operations

In this section, we compare the performance of ROTX algorithm provided in Section 5.3 with ROTX (named GET-ROTX in [36]) of GentleRain [36]. The sketch of GentleRain's ROTX is as



(a) PUT (b) GET

Figure 5.5: The basic PUT/GET operations throughput in GentleRain and CausalSpartanX.

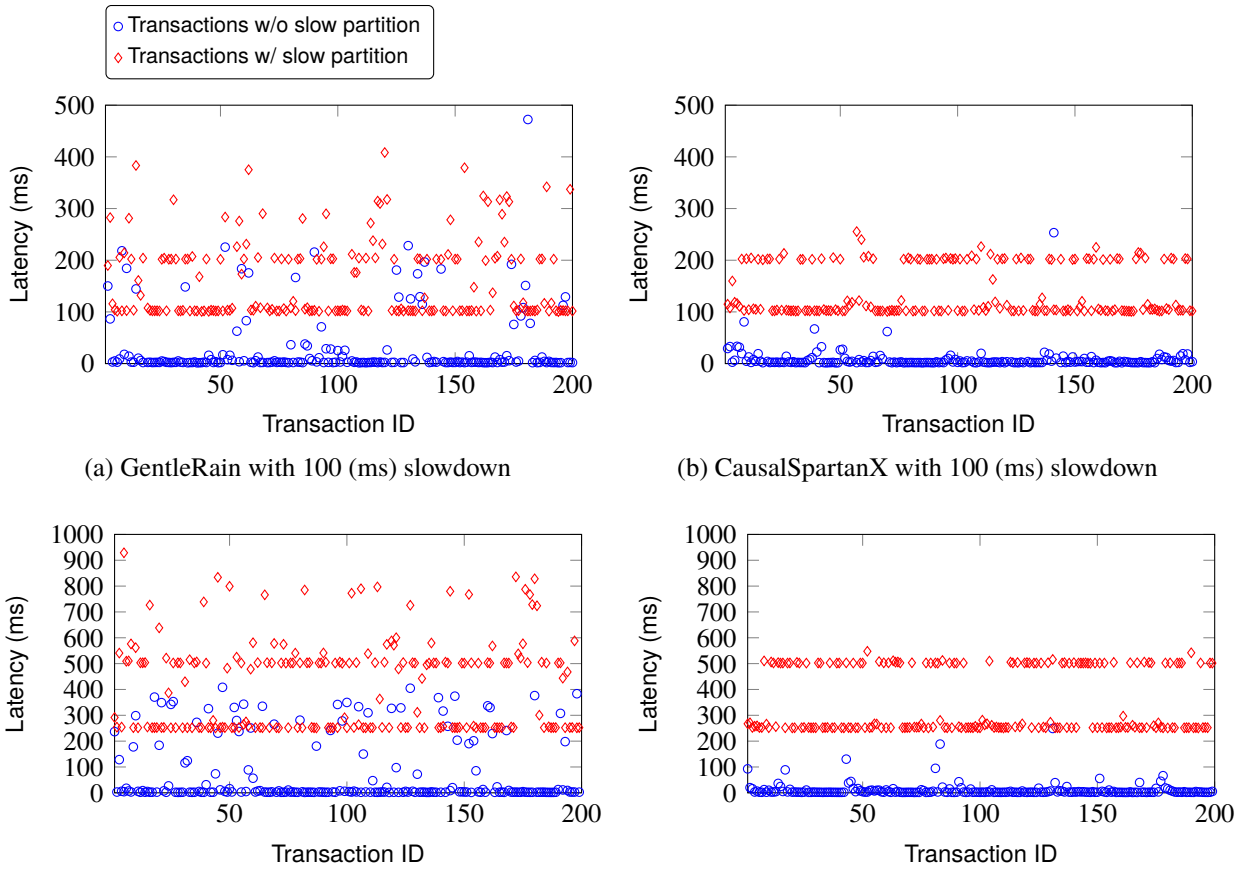
follows: like PUT operations, the client includes its dt with its GET-ROTX operations and sends it to one of the servers as the coordinator. Upon receiving a ROTX operation, if $|dt - GST|$ is smaller than a threshold, sever waits for GST to goes higher than dt. The server sets the snapshot time as its GST and sends requests to all partitions hosting some of the requested keys. In its request, the server includes the snapshot time. The receiving partitions return versions with timestamps smaller than the snapshot time. If $|dt - GST|$ is higher than the threshold, the server runs the Eiger [50]. Since (1) [36] reports that this backup option was never triggered (2) there is a significant increase in metadata and (3) there is up to two more rounds of communication between the client and servers if one intends to use [50] for backup, in our experiments, we use threshold values that guarantee the that the backup option was not necessary.

In Section 5.5.3, we saw how waiting for GST can increase the update visibility latency. In the case of read-only transactions, waiting for GST in GentleRain also increases the response time. Specifically, since GentleRain blocks read-only transactions until GST is high enough, any delay in GST leads to higher response times for the clients and reduced throughput in the system. Latency in GST can be caused by slow replicas (as we saw in Section 5.5.3), slow partitions inside the local replica, or any other delay in GST calculation.

To evaluate the response time of transactions in the presence of a slow partition, we set up the experiment as follows: (1) We consider a set of hot keys that are constantly written by several clients, (2) We have some clients that perform GET with 50% probability and ROTX with 50%

probability on hot keys. (3) We simulate one partition to be slow by intentionally delaying sending messages from it. Specifically, whenever the slow partition wants to send a message to other servers or the clients, we schedule it to be sent after a certain amount of time.

Figure 5.6a shows the latency of 200 ROTX operations with size 3 (i.e. each ROTX reads 3 keys) for GentleRain in a data center with 6 partitions where one partition is slowed by 100 (ms). The circles show the response time of ROTX operations that do not involve any key hosted by the slow partition. Diamonds, on the other hand, show the response time of the ROTX operations involving a key hosted by the slow partition. Figure 5.6b shows the same experiment results for the CausalSpartanX. As it is clear from the plots, in CausalSpartanX, ROTX operations that do not involve the slow partition are not affected by the 100 (ms) slowdown. On the other hand, in GentleRain, some of the ROTX operations that do not involve the slow partition are also affected by the slowdown. The response time of some of ROTX operations of GentelRain involving the slow partition is also significantly higher than that of CausalSpartanX. For 100 (ms) slowdown, the average latency of ROTX operations not involving the slow partition is 29 (ms) for GentelRain while it is 8 (ms) for CausalSpartanX. For ROTX operations involving the slow partition, the response time is 169 and 142 for GentleRain and CausalSpartanX, respectively. The improvement of CausalSpartanX is more significant regarding the *tail-latency* [40]. Figure 5.7a shows the empirical CDFs for both cases of ROTX operations for GentleRain and CausalSpartanX for 100 (ms) slowdown. As it is marked in Figure 5.7a, for 100 (ms) slowdown, the 90th percentile of ROTX operations not involving the slow partition is 129 (ms) for GentelRain while it is only 18 (ms) for CausalSpartanX (i.e., 86.04% improvement). For ROTX operations involving the slow partition, the 90th percentile response time is 289 (ms) and 203 for GentleRain and CausalSpartanX, respectively (i.e. 29.7% improvement). Figure 5.6 also shows results for the case with 250 (ms) slowdown. The improvement of CausalSpartanX is more significant for higher slowdowns. For 250 (ms) slowdown, 90th percentile improvement goes to 93.88%. We note that for higher percentiles also, CausalSpartanX shows clear improvement. Figure 5.7b shows the empirical CDFs for both cases of ROTX operations for GentleRain and CausalSpartanX for 250 (ms) slowdown. For



(a) GentleRain with 100 (ms) slowdown

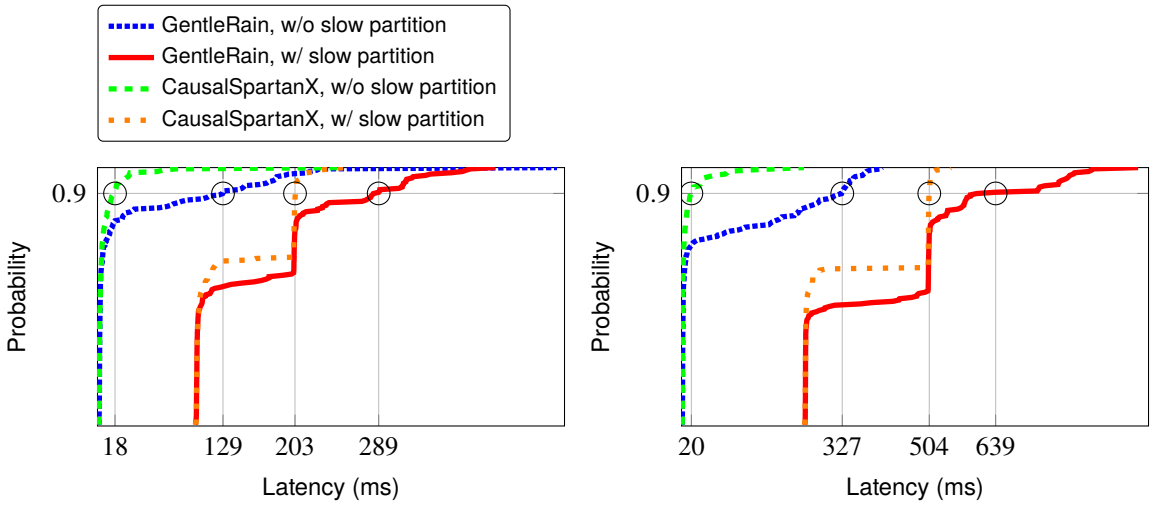
(b) CausalSpartanX with 100 (ms) slowdown

(c) GentleRain with 250 (ms) slowdown

(d) CausalSpartanX with 250 (ms) slowdown

Figure 5.6: The effect of slowdown of one partition on the latency of ROTX operations. Each ROTX operations reads 3 keys.

instance, for the 99th percentile, CausalSpartanX leads to 95.99% and 37.70% for not involving and involving cases, respectively, compared with GentleRain [36]. Note that tail latency is very important, as it directly affects the users' experience. Although it affects a small group of clients (e.g. 1 percent of clients in case of 99th percentile), this small group are usually the most valuable users that perform most of the requests in the system [40]. For this reason, many companies such as Amazon describe response time requirements for their services with 99.9th percentile [40].



(a) Emprical CDFs for 100 (ms) slowdown

(b) Emprical CDFs for 250 (ms) slowdown

Figure 5.7: Empirical CDFs of results of Figure 5.6

CHAPTER 6

NECESSITY OF STICKY CLIENTS FOR CAUSAL CONSISTENCY

In this chapter, we show that in presence of network partitions, if we want availability and causal consistency while providing immediate visibility for local updates, the clients must be sticky.

Theorem 12 *In an asynchronous network (i.e., in presence of network partitions) with non-sticky clients, it is impossible to implement a replicated data store that guarantees following properties:*

- *Availability*
- *Causal++ consistency (for $O = R = I = \{GET\}$ and any conflict resolution function f)*

Proof 23 *We prove this by contradiction. Assume an algorithm A exists that guarantees availability and causal++ consistency for conflict resolution function f in asynchronous network with non-sticky clients. We create an execution of A that contradicts causal++ consistency: Assume a data store where each key is stored in at least two replicas r and r'. Initially, the data center contains two keys k_1 and k_2 with versions v_1^0 and v_2^0. These versions are replicated on r and r'. Next, the system executes as follows:*

- *There is a partition between r and r', i.e., all future messages between them will be indefinitely delayed.*
- *c performs $GET(k_1)$ on replica r, and reads v_1^0.*
- *c performs $PUT(k_1, v_1^1)$ on replica r.*
- *c performs $PUT(k_2, v_2^1)$ on replica r.*
- *c' performs $GET(k_2)$ on replica r. Since v_2^1 is a local update, and in causal++ consistency, local updates have to be immediately visible, the value returned is v_2^1.*

- *c' performed $GET(k_1)$ on replica r'. Because of the network partition, there is no way for replica r' to learn v_1^1. Thus, the value returned is v_1^0.*

Since v_1^1 dep v_1^0, for any conflict resolution function f, $f(v_1^0, v_1^1) = v_1^1$. Thus, according to Definition 13, v_1^1 is not visible+ for f to client c'. It is a contradiction to causal++ consistency, as c' has read v_2^1, but its causal consistency v_1^1 is not visible+ to c'.

The necessity of sticky clients for the causal consistency has been investigated in the literature [21]. The existing proof, however, is based on the impossibility of read-your-write consistency which is part of causal consistency [21, 25]. In other words, since read-your-write consistency is impossible for non-sticky clients for an always-available system in presence of network partition, and because read-your-write consistency is a part of the causal consistency [21,25], the impossibility also applies to the causal consistency. However, the read-your-write consistency can be achieved even for non-sticky clients, if clients cache what they have read or written, and use this cache in the read time if a server has an older value. It is straightforward to see that our proof still holds even when clients can cache their read and writes.

CHAPTER 7

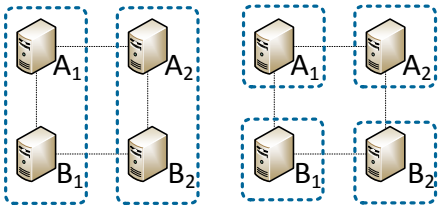
ADAPTIVE CAUSAL CONSISTENCY

Existing causal consistency protocols utilize a static approach in the trade-off between different conflicting requirements (e.g. consistency, visibility, and throughput). They also treat all clients the same, and assume that their usage patterns are always unchanged. For example, they assume clients only access their local data center, and any client may access any part of the data. However, different applications may have different usage patterns.

To illustrate, consider a simple system that consists of two partitions A and B with geographically distributed copies A_1, A_2, B_1 and B_2. Suppose, we are using a causal consistency protocol like [35, 36, 49, 50, 60] that does not make a version visible, unless it made sure all partitions inside a replica are updated enough. We consider two possible ways to organize the replicas: (1) two full replicas each with two partitions, referred to as 2×2 (2) or four partial replicas each with one partition referred to as 4×1. These two organizations are shown in Figure 7.1. Now, consider two applications. The first application, App_1 consists of two clients $C1$ and $C2$ that access A_1 and A_2 respectively for a collaborative work. In App_1, each client updates the data after it reads the new version written by another client. Since each client waits for the other client's update, any increase in update visibility will reduce the throughput of App_1. In the scenario in Figure 7.1a, since A_1 and B_1 are considered in the same replica, A_1 does not make versions visible, unless it made sure B_1 is updated enough. Thus, if the communication between A_1 and B_1 is slow, it takes more time for A_1 to make a version visible. Since the data on B_1 and B_2 is irrelevant for App_1, this delay by A_1 is unnecessary which leads to increased visibility latency which, in turn, leads to a reduced throughput of App_1. Furthermore, if there were a large number of such partitions, this delay would be even more pronounced.

By contrast, there is no such penalty in scenario in Figure 7.1b, as in Figure 7.1b, partitions A_1 and B_1 are considered in different replicas. Thus, they do not check each other.

On the other hand, consider App_2 that consists of one client, say $C3$, and it accesses data from



(a) 2×2 (b) 4×1

Figure 7.1: Two ways to organize replicas

A_1 and B_1. In scenario in Figure 7.1a, C_3 is guaranteed to always read the consistent data. However, in scenario in Figure 7.1b, since A_1 and B_1 do not check the freshness of each other, C_3 may suffer from finding inconsistent versions (or delays or repeated requests to find a consistent version) while accessing A_1 and B_1.

From the above discussion, it follows that no matter how we configure the given key-value store, a system with a static configuration that treats all clients the same will penalize some clients. The goal of this chapter is to develop a broad framework that instead of relying on a fixed set of assumptions, allows the system to be dynamically reconfigured after learning the actual client activities and requirements.

7.1 An Approach for Adaptive Causal Consistency

The broad approach for providing causal consistency is to track the causal dependencies of a version, and check them before making the version visible in another replica. Tracking and checking are usually done using timestamping versions as follows:

- *Dependency Tracking*: Upon creating a new version for a key, we assign a timestamp to the version that *somehow* captures causal dependencies of the version.
- *Dependency Checking*: Upon receiving a version, the receiving replica does not make the version visible to the clients until it makes sure that all of the dependencies of the version are also visible to the clients.

The goal of timestamping is to provide a way to capture causal relation between two versions.

Table 7.1: Tracking and Checking in Some of Causal Systems

Protocol	**Tracking**	**Checking**
COPS [49]	Per key	Per Replica
Eiger [50]	Per key	Per Replica
Orbe [35]	Per server	Per Replica
GentleRain [36]	Per system	Per Replica
Occult [54]	Per Master Server	No checking
Okapi [34]	Per Replica	Per system
CausalSpartan [60]	Per Replica	Per Replica

To satisfy $v_1 \ \mathsf{dep} \ v_2 \Leftrightarrow v_1.t > v_2.t$ (where $v_1 \ \mathsf{dep} \ v_2$ means the event of writing v_2 has happened-before [45] the event of writing v_1, and $v.t$ is the timestamps assigned to v), we need timestamps of size $O(N)$ [28] where N is number of nodes that clients can write on. To solve the issue of large timestamps, causal consistency protocols consider servers in groups and track causality with vectors that have one entry per group. We refer to such groups as *tracking groups*. Tracking dependencies in groups, provides timestamps that satisfies a weaker condition $v_1 \ \mathsf{dep} \ v_2 \Rightarrow v_1.t > v_2.t$. This condition lets us guarantee causal visibility of the versions. However, since it does not provide accurate causality information, we may need to unnecessarily delay the visibility of a version by waiting for versions that are not its real dependencies. Thus, by grouping servers in tracking group, we trade off the visibility of versions for a lower metadata size.

We face a similar trade-off in the dependency checking. Dependency checking determines how conservative we are in making versions visible to the clients. Since checking the whole system is expensive, causal consistency protocols consider systems in groups, and each server only checks servers in its own group. We refer to such groups as *checking groups*. Most of current protocols [35, 36, 49, 50, 60, 61] group servers by their replicas. Thus, a server only checks the dependencies inside the replica that it belongs to. Table 7.1 shows tracking and checking groups for some of the recent causal systems.

When we are designing a causally consistent key-value store, two natural questions arise based on the trade-offs explained above: 1) how much tracking accuracy is enough for a system? 2) how much should we be conservative in making versions visible? We believe the answer to these questions depends on the factors that should be learned at the run-time. A practical distributed

data store performs in a constantly changing environment; the usage pattern of clients can change due to many reasons including time of the day in different time zones or changes in load balancing policies; data distribution can change, because we may need to add or remove some replicas; components may fail or slow down, and so on. These changes can easily invalidate assumptions made by existing causal consistency protocols such as [35, 36, 49, 50, 54, 60] which leads to their reduced performance in practical settings [19]. To solve this issue, we believe that a key-value store must monitor the factors mentioned above and *dynamically* trade-off between different conflicting objectives. We believe dynamically changing tracking and checking grouping based on what we learn from the system is an effective approach to perform such dynamic trade-offs. Using a flexible tracking and checking grouping we are also able to treat different applications in different ways.

To use the above approach, however, we need a protocol that can be easily configured for different groupings. As shown in Table 7.1, existing protocols assume fixed groupings that cannot be changed. To solve this issue, in the next section, we provide a protocol that can be configured to use any desired grouping. This flexible algorithm provides a basis for creating adaptive causal systems. This algorithm also lets us treat clients in different ways, and unlike most of the existing protocols that require a certain data distribution schema, our algorithm allows us to replicate and partition our data any way we like including creating partial replicas.

7.2 Adaptive Causal Consistency Framework

In this section, we provide Adaptive Causal Consistency Framework (ACCF) which is a configurable framework that lets us deal with trade-offs explained in Section 7.1. Specifically, as the input, ACCF receives 1) function T that assigns each server to exactly one tracking group, and 2) function C that assigns each server to a *non-empty set* of checking groups.

7.2.1 Client-side

Algorithm 7.1 shows the client-side of the ACCF. A client c maintains a set of pairs of tracking group ids and timestamps called dependency set, denoted as DS_c. For each tracking group i, there

is at most one entry $\langle i, h\rangle$ in DS_c where h specifies the maximum timestamp of versions read by client c originally written in servers of tracking group i. Each data object has a key and a version chain containing different versions for the object. Each version is a tuple $\langle v, ds\rangle$, where v is the value of the version, and ds is a list that has at most one entry per tracking group that capture dependency of the version on writes on different tracking groups.

Algorithm 7.1 Client operations at client c

Input: Load balancer L

1: **GET** (key k, checking group id cg)
2: $i = L(k)$
3: send $\langle$GetReq $k, cg, DS_c\rangle$ to server i
4: receive $\langle$GetReply $d\rangle$
5: $DS_c \leftarrow max(DS_c, d.ds)$
6: **return** $d.v$
7: **PUT** (key k, value v)
8: $i = L(k)$
9: send $\langle$PutReq $k, v, DS_c\rangle$ to server i
10: receive $\langle$PutReply $tg, ut\rangle$
11: $DS_c \leftarrow max(DS_c, \langle tg, ut\rangle)$

To read the value of an object, the client calls GET method with the desired key to read. The client also specifies the id of the checking group that the server must use. We will see how the server uses this id in Section 7.2.2. We find the preferred server to read the object using the given load balancer service L. After finding the preferred server to ask for the key, we send a GetReq request to the server. In addition to the key and the checking group id, we include the client dependency set DS_c in the request message. The server tries to find the most recent version that is consistent by the client's past reads. In the Section 7.2.2, we explain how the server looks for a consistent version based on the DS_c.

To write a new value for an object, the client calls PUT method. The server writes the version and records client's DS_c as the dependency of the version. After receiving a response from the server for a GET (or PUT) operation, we update DS_c such that any later version written by the client depends on the version read (or written) by this operation.

7.2.2 Server-side

In this section, we focus on the server-side of the protocol. We denote the physical clock at server i by PC_i. To satisfy v_1 dep $v_2 \Rightarrow v_1.t > v_2.t$ condition, and assign timestamps close to the physical clocks, ACCF relies on Hybrid Logical Clocks (HLCs) [42]. HLC_i is the value of HLC at server i. Each server keeps a version vector that has one entry for each tracking group denoted by VV_i. $VV_i[t]$ is the minimum of latest timestamps that server i has received from servers in tracking group t. To keep each other updated, servers send heartbeat messages to each other in case of not sending any replicate message for a specific amount of time.If there is no key that is hosted by both server i and a server in tracking group t, then $VV_i[t] = +\infty$. Each server is a member of one or more dependency checking groups. Servers inside a checking group, periodically share their VVs with each other and compute Stable Version Vector (SVV) as the entry-wise minimum of VVs. SVV_i^{cg} is the SVV computed in server i for checking group cg.

Algorithm 7.2 shows the algorithm for PUT and GET operations at the server-side. When a client asks to read an object, the server waits if there exists $\langle t, h\rangle$ in ds such that $VV_i[t] < h$ which means the server is not updated enough, and reading from the current version chain can violate causal consistency. When the server made sure for any $\langle t, h\rangle$ in ds, $VV_i[t] \geq h$, it checks the SVV_i^{cg}. If for any $\langle t, h\rangle$ in ds, $SVV_i^{cg}[t] \geq h$, the server returns the most recent version for k such that for any $\langle t, h\rangle$ in $k.ds$, $SVV_i^{cg}[t] \geq h$. This guarantees that the client never has to wait if it only reads from servers in checking group cg. If a client uses different checking groups for different reads, it is possible that the server finds $\langle t, h\rangle$ in ds, such that $SVV_i^{cg}[t] < h$. In this situation, server forgets about SVV_i^{cg}, and gives the client the most recent version that has for k. Note that this version is guaranteed to be causally consistent with client's previous reads.

Once server i receives a PUT request, the server updates HLC_i by calling $updateHLC(dt)$ where dt is the highest timestamp in ds. Next, the server creates a new version for the key specified by the client. The server updates $VV_i[T(i)]$ with the new HLC_i value, and sends back its tracking group, $T(i)$, and the assigned timestamp, $d.ds[T(i)]$, to the client in a PutReply message.

Upon creating a new version for an object in one server, we send the new version to other servers

hosting the object via replicate messages. Upon receiving a $\langle \text{REPLICATE } k, d \rangle$ message from server j, the receiving server i adds the new version to the version chain of the object with key k. The server also updates the entry for server $T(j)$ in its version vector (i.e., $VV_i[T(j)]$).

Algorithm 7.2 PUT and GET operations at server i

Input: Tracking grouping function T, Data placement function H

1: **Upon** receive $\langle \text{GETREQ } k, cg, ds \rangle$
2: **while** there is a member $\langle t, h \rangle$ in ds, $h > VV_i[t]$
3: wait
4: **if** for all $\langle t, h \rangle$ in ds, $h > SVV_i^{cg}[t]$
5: d = latest version d from version chain of key k
s.t. for any member $\langle t, h \rangle$ in $k.ds$, $h \leq SVV_i^{cg}[t]$
6: **else**
7: d = latest version d from version chain of key k
8: send $\langle \text{GETREPLY } d \rangle$ to client
9: **Upon** receive $\langle \text{PUTREQ } k, v, ds \rangle$
10: $dt \leftarrow$ maximum value in ds
11: $updateHCL(dt)$
12: Create new item d
13: $d.v, d.ds \leftarrow v, max(ds, \langle T(i), HLC_i \rangle)$
14: insert d to version chain of k
15: update $VV_i[T(i)]$ with HLC_i
16: send $\langle \text{PUTREPLY } T(i), d.ds[T(i)] \rangle$ to client
17: **for** each server $j \neq i$, such that $j \in H(k)$
18: send $\langle \text{REPLICATE } k, d \rangle$ to server j
19: **Upon** receive $\langle \text{REPLICATE } k, d \rangle$ from server j
20: insert d to version chain of key k
21: update $VV_i[T(j)]$ with $d.ds[T(j)]$
22: **updateHLCforPut** (dt)
23: $l' \leftarrow HLC_i.l$
24: $HLC_i.l \leftarrow max(l', PC_i, dt.l)$
25: **if** $(HLC_i.l = l' = dt.l)$
26: $HLC_i.c \leftarrow max(HLC_i.c, dt.c) + 1$
27: **else if** $(HLC_i.l = l')$ $HLC_i.c \leftarrow HLC_i.c + 1$
28: **else if** $(HLC_i.l = l)$ $HLC_i.c \leftarrow dt.c + 1$
29: **else** $HLC_i.c \leftarrow 0$

7.3 Evaluation

We have implemented ACCF using our DKVF [62] that we will explain in Chapter 8. You can find our implementation of ACCF in DKVF repository [5]. In this section, we provide the results

of 2×2 and 4×1 groupings for applications App_1 and App_2 explained above. We run the system explained above consisting of A_1, A_2, B_1, and B_2 on different data centers of Amazon AWS [1]. Note that since we focus on partial replication, there is no assumption about A_1 and B_1 (respectively A_2 and B_2) to be collocated.

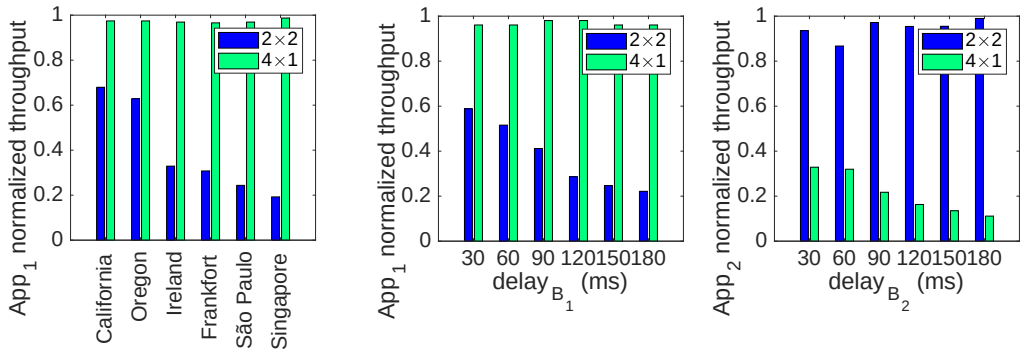
Observations for App_1. App_1 consists of two clients $C1$ and $C2$. $C1$ writes the value 0 using A_1. $C2$ reads 0 (from A_2) and writes 1 (to A_2). Subsequently, $C1$ waits to read 1 and writes 2 and so on. The best scenario for this case is when you have only two partitions A_1 and A_2 in the system. Hence, we normalize the throughput with respect to this.

The results for App_1 are shown in Figure 7.2, where 2×2 (respectively 4×1) corresponds to the organization in Figure 7.1a (respectively, Figure 7.1b). In Figure 7.2a, locations of A_1, A_2 and B_1 are fixed, and we vary the location of B_1 from California to Singapore (ordered based on increasing ping time from A_1 located in California). In Figure 7.2b, we keep the location of B_1 fixed, but artificially add $delay_{B_1}$ to any message sent by B_1. As we can see, by viewing the system as Figure 7.1b, App_1 performance is unaffected whereas viewing the system as Figure 7.1a, performance drops by more than 50%.

Observations for App_2. In App_2, client $C3$ alternates reading from A_1 and B_1. To provide fresh copies, another client writes the same objects on A_2 and B_2 respectively. Here, viewing the system as in Figure 7.1b drops the performance substantially as the message delay of B_2 ($delay_{B_2}$) increases. This is due to blocking the GET operations while waiting for receiving consistent versions. By contrast, by viewing replicas as in Figure 7.1a, performance remains unaffected. Throughputs are normalized with respect to the case where there is no update.

7.4 Comparison with Related Work

Our approach for providing adaptivity in causal consistency is different from other approaches considered in the literature. Occult [54] utilizes structural compression to reduce the size of the timestamps. Other approaches include bloom filters [20]. While these features are intended as a configurable parameter, we believe that it is not possible to dynamically change it at run-time



(a) Chaging the location of B_1 (b) Changing the delay of B_1 (c) Changing the delay of B_2

Figure 7.2: Normalized throughput of App_1 and App_2 for different groupings. App_1 has higher throughput with 4×1, while App_2 has higher throughput with 2×2.

while preserving causal consistency (or detection of its violation). Furthermore, in all these cases, the reconfiguration provided is client-agnostic; it does not take client requests into consideration. By contrast, our framework provides the ability to allow different clients a view of the system in a manner that improves their performance. Finally, it is possible to take client requests into consideration to identify how adaptivity should be provided.

CHAPTER 8

DISTRIBUTED KEY-VALUE FRAMEWORK

Improving consistency protocols for key-value stores has received much attention in recent years. When we design a new protocol, we need to evaluate the protocol with a working prototype to see how our protocol works comparing to existing protocols. One way to have a prototype is building it from scratch. This approach has the advantage of maximum flexibility. However, building our prototype from scratch may take a long time that can slow down the research or development. Furthermore, if the protocol suffers from some undesirable properties (e.g., low performance), a substantial amount of development time is wasted. Another important obstacle is that this approach especially makes the comparison to other protocols hard. Imagine that we want to compare our new protocol to several other existing systems, developed by other groups. Since each of them are implemented in a different code base, we need to implement other protocols with the same code base as ours to have a fair comparison which requires more time.

Another approach is to create our prototype by modifying an already existing system. There are many open-source NoSQL databases that can be modified for prototyping purposes. An important advantage of this approach is that by building a system on top of a tested system, we can benefit from all of its good features, and save time. However, modifying an existing system has its own disadvantages. The most important problem is the lack of flexibility. Although we can always change the code of an open-source data store, to change a system correctly, we need to understand a possibly massive implementation thoroughly, that may take even more time than creating a prototype from scratch. In addition, by changing an existing product, we may lose the advantage of reusing some of its components which was the whole purpose of using an existing product. For instance, suppose an existing system uses a certain replication policy. If the replication policy of our protocol is different, we have to change the whole replication mechanism of the underlying system.

Another problem of changing an existing product is the problem of being locked by that product.

For instance, suppose that we have implemented a prototype to evaluate an algorithm for causal consistency by forking from a current system like Cassandra [44]. If in future we are interested to see how our algorithm would perform if we used another system, say Voldemort [17], we have no choice but building another system based on Voldemort as well. That would be especially necessary if we want to compare our algorithm with another system based on Voldemort.

In addition to the implementation of a protocol, running experiments is also another burden. Different research groups may evaluate their systems in different ways making comparisons unfair. Yahoo! Cloud Serving Benchmark (YCSB) [29] is a good candidate for a unified way of comparing different storage systems. The YCSB drivers required for benchmarking with YCSB are already available for many systems. Although YCSB helps us to benchmark our system, writing the driver, running clusters and clients on several machines, obtaining, and aggregating the results is a task that we have to do everything we want to evaluate a new protocol.

In this chapter, we introduce the Distributed Key-Value Framework (DKVF) that allows protocol designers to quickly create prototypes running their protocols to see how they work in practice. We want to help researchers to only focus on their high-level protocol and let the DKVF do all the lower-level tasks. For instance, consider the GentleRain protocol proposed in [36]. The server side of this protocol is only 31 lines of pseudocode provided in Algorithm 2 of [36]. However, to have a prototype running this protocol, we need to write hundreds of lines of code to handle lower-level tasks that are independent of the protocol. Our goal is to provide a framework that helps researchers to create their prototypes by writing codes that are very close to the pseudocodes that they publish in their research papers. We believe this framework together with a toolset that helps us to run experiments can significantly save time in implementing and benchmarking new protocols. We hope our framework expedites the research on the field.

Followings are the advantages of our framework:

- The framework allows us to easily define our protocol in a high-level abstraction with an event-driven approach. Specifically, we can define our protocol as a set of event handlers which is the same way as researchers typically present their protocols in their papers. It

makes the code much more clear, and reduces the number of lines of code that protocol designers need to write.

- The clear separation of concerns that the framework provides expedites debugging the system, and improves maintainability of our code.

- We can easily compare any two protocols that are implemented on top of the framework, as both of them are implemented with the same code base.

- We provide the implementation of four protocols with this report. Adding other protocols to the repository is part of the future work. Also, other groups can add their protocols to the repository making them publicly available. Having a library of protocol implementations allows researchers to easily compare their protocols to previous ones.

- We can easily change the storage engine of our key-value store without changing the logic of our higher consistency protocol. This makes comparison easy, as we can use the storage engine that another system is built on for comparison purposes.

- The framework and its toolset streamline the use of YCSB for benchmarking protocols. It encourages researchers to use a standardized framework for benchmarking instead of performing experiments in individual ways.

- The framework comes with a command line application called Cluster Manager that lets us conveniently run a cluster over the network. Using Cluster Manager, we can easily run a cluster on cloud systems such as Amazon Web Services (AWSs) on Windows or Linux instances. It allows us to monitor connections, network latencies, current load on nodes, and so on.

- Cluster Manager also lets us specify a set of experiments to benchmark the system. it takes care of running YCSB clients, collecting, and aggregating the results.

- The framework is also accompanied by a graphical tool called Cluster Designer that lets us easily define our cluster and experiments. We can visually create a graph of servers and clients, and define different workloads to run on the cluster.

8.1 Overview of DKVF

DKVF is written in Java. Each key-value store created based on DKVF has two sides: 1) a server side, and 2) a client side. The server-side (respectively, client-side) extends server-side (respectively, client-side) of DKVF by implementing the respective abstract methods and adding new methods required for the protocol at hand.

When we create a new protocol, in addition to actual data consisting of the key-value pairs, we will likely need to store some metadata with each record. For example, we may need to store a timestamp with each version, or we may need to store the ID of the replica where the version has been written. Each protocol requires its own metadata. DKVF relies on Google Protocol Buffers [8] (referred to as protobuf from now on) for marshaling/unmarshalling data for storage and transmission. An important advantage of protobuf is its convenience for the protocol designer to describe the metadata, as the protocol designer only needs to write a simple text file, and protobuf takes cares of creating the necessary code. Another important advantage of protobuf is its effective way of compressing the data using bit variant techniques that saves storage space and network bandwidth. The protobuf description together with the server and client sides of the protocol are components that the protocol designer needs to provide for any key-value store based on DKVF. These components are shown by dark rectangles in Figure 8.1. We will focus on these components in Section 8.2.

Once a key-value store is ready, we can use it for our applications. An application can be any program (website, mobile application, etc.) that needs to access storage resources through the network. We refer to the entity that provides the storage resources as *storage provider*. The storage provider runs the server side of the key-value store. To do that, the storage provides needs to write a configuration file. This configuration file is an XML file according to `Config.xsd` [6],

and describes the cluster and server-side parameters. Once the server side is running, different applications can connect to it through the client side of the key-value and use the storage resources. The application developers also need to write a configuration file that specifies servers to connect and client-side parameters. The server side and client side configurations are shown by white rectangles in Figure 8.1. These configuration files together with three components that the protocol designer needs to write are five components that we need to provide to have a running key-value store based on DKVF.

The `Application` rectangle in Figure 8.1 captures any client program that uses a key-value store server. While the exact application is orthogonal to DKVF, DKVF can be used to provide suitable benchmarks so that application designer can choose the suitable protocol based on these benchmarks. DKVF relies on YCSB [29] for benchmarking. When we benchmark our key-value store using YCSB, the YCSB client becomes our application in Figure 8.1.

DKVF can be configured to use any storage engine provided the storage developer implements the necessary drivers. DKVF comes with a driver for Berkeley-BD. We can configure the default storage to be multi-version or single version. In the case of multi-version, we have to provide a comparator function to order versions with the same key. DKVF also provides a simple approach for the addition of a new storage engine. An important question regarding storage is how we want to replicate data. Data replication can be done either by storage engine itself or by the protocol. The default storage delegates data replication to the protocol. This gives the full control of data replication to the protocol designer. However, we can configure DKVF for the case where the storage engine handles data replication.

8.2 Creating a Prototype using DKVF

The overall usage of DKVF is as shown in Figure 8.1. When a designer intends to develop a new protocol, he/she needs to specify three components (shown by dark rectangles in Figure 8.1). These components are 1) metadata description, 2) the server side of the protocol, and 3) the client side of the protocol. We explain each of these components in this section.

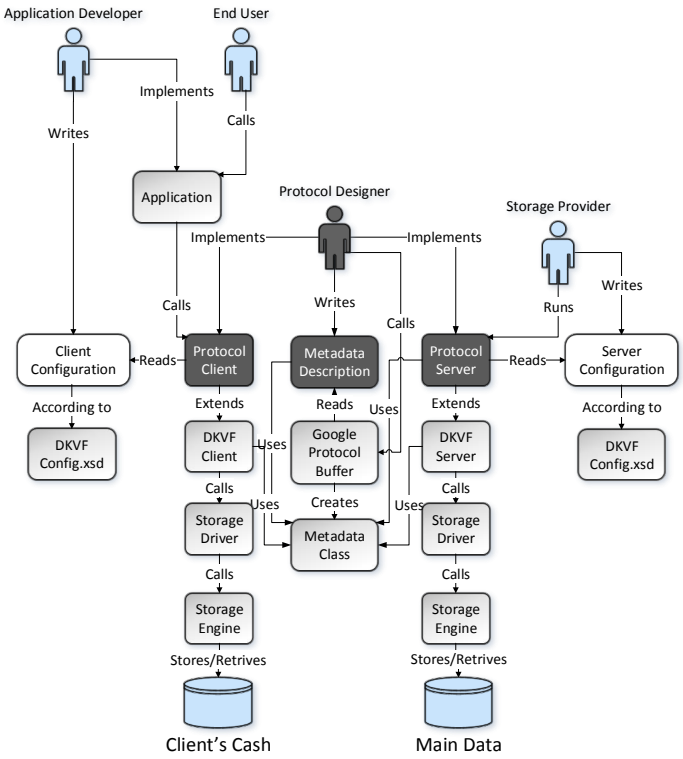


Figure 8.1: Typical usage of DKVF

8.2.1 Metadata Description

Describing metadata is done by writing a text file, `.proto`, that contains a set of *message* blocks. You can think of a message as a class or struct in a programming language. Each message has a set of fields. Each field has a type that is either a primitive type like integer, or another message. Any metadata description written for DKVF must include four messages: 1) `Record`, 2) `ClientMessage`, 3) `ServerMessage`, and 4) `ClientReply`. `Record` describes records that will be stored in the key-value store. For instance, if we want to store a timestamp with each record, we need to add an `int64` field to `Record a` message to store a 64-bit Java `long` variable with each record. `ClientMessage` and `ServerMessage` describe client and server messages, respectively. `ClientReply` describes a response to a client message.

As an illustration, consider the case where we implement GentleRain protocol [36] using DKVF. In GentleRain, each record (i.e., data item) is a tuple as $\langle k, v, ut, sr \rangle$ where k is the key, v is the value,

ut is update time (timestamp of the current version), and *sr* is the ID of the replica where the current version has been written. The following code shows the corresponding protobuf description for GentleRain records. Numbers in front of fields are tag numbers that protobuf uses for optimization. Tag numbers must be unique positive integers, and we should assign smaller values to the fields that are used more frequently [8].

```
message Record {
    string k = 1;
    bytes v = 2;
    int64 ut = 3;
    int32 sr = 4;
}
```

In addition to the metadata for the records, we also use protobuf to describe messages that servers/clients send in our protocol. For instance, consider GetReq message in GentleRain [36]. Each GetReq message has a string to specify the key that we want to read, and an integer for GST value used by the protocol to find a consistent version (see Section 3.2.2). Following code shows the necessary protobuf description for GetReq message. Similarly, we can write description for other messages of the protocol.

```
message GetMessage {
    string k = 1;
    int64 gst = 2;
}
```

After writing the metadata description, we need to compile it using protobuf. The protobuf will create a Java class that contains all necessary data structures to marshaling/unmarshalling our data. This class is shown by `Metadata class` rectangle in Figure 8.1.

8.2.2 Server-side Implementation

To implement the server side of a protocol, we need to write a class that extends the abstract class `DKVFServer`. DKVF follows an event-driven approach to define a protocol. Specifically, we can define a protocol as a set of event handlers. The two main event handlers that will be called by the framework are `handleServerMessage` and `handleClientMessage` of `DKVFServer` class. Inside these two main event handlers, the protocol designer can call detailed event handlers for different events. A protocol can also have other event handlers that do not call by the framework. For instance, GentelRain [36] and CausalSparatan [60] have event handlers that are constantly called at a certain rate. The following code shows the overall structure of GentleRainServer class that implements GentleRain on top of DKVF. The body of event handlers is left blank for sake of presentation.

```
public class GentleRainServer extends DKVFServer {
 @Override
 public void handleClientMessage(ClientMessageAgent cma) {
  if (cma.getClientMessage().hasGetMessage()) {
   handleGetMessage(cma);
  } else if (cma.getClientMessage().hasPutMessage()){
   handlePutMessage(cma);
  }
 }

 @Override
 public void handleServerMessage(ServerMessage sm) {
  if (sm.hasReplicateMessage()) {
   handleReplicateMessage(sm);
  } else if (sm.hasHeartbeatMessage()) {
   handleHearbeatMessage(sm);
  } else if (sm.hasVvMessage()) {
   handleVvMessage(sm);
  } else if (sm.hasGstMessage()) {
   handleGstMessage(sm);
```

```
  }
 }

 private void handleGetMessage(ClientMessageAgent cma){
  //TODO Handle GET messages here
 }
 private void handlePutMessage(ClientMessageAgent cma){
  //TODO Handle PUT messages here
 }
 private void handleReplicateMessage(ServerMessage sm){
  //TODO Handle Replicate messages here
 }
 void handleHearbeatMessage(ServerMessage sm) {
  //TODO Handle Heartbeat messages here
 }
 void handleVvMessage(ServerMessage sm) {
  //TODO Handle VV messages here
 }
 void handleGstMessage(ServerMessage sm) {
  //TODO Handle GST messages here
 }
}
```

`handleServerMessage` receives an object of class `ServerMessage` which is created by protobuf from our metadata description explained in Section 8.2.1. `handleClientMessage` receives an object from class `ClientMessageAgent` that includes an object of class `ClientMessage` created by protobuf.

While we are processing server or client messages in `handleServerMessage` and `handleClientMessage`, we may need to send messages to other servers, or send client responses. To send a message to another server, the framework provides the convenient `sendToServer` method that receives the ID of the destination, and an object of `ServerMessage` class. The mapping between server IDs and their actual addresses must be defined in the configuration file. DKVF takes care of asynchronous reliable FIFO delivery of the message to the destination. Specifically, if the receiver cannot receive the message (e.g., it has crashed, or there is a network partition),

Algorithm 8.1 Pseudocode of the GET request handler of GentleRain protocol

1: **Upon** receive $\langle \text{GETREQ } k, gst \rangle$
2: $GST_n^m \leftarrow max(GST_n^m, gst)$
3: obtain latest version d from version chain of key k s.t. $d.sr = m$, or $d.ut < GST_n^m$
4: send $\langle \text{GETREPLY } d.v, d.ut, GTS_n^m \rangle$ to client

DKVF stores the message and will try to send it later. In the configuration file, we can specify the amount of time to wait before resending the message. Also, we can set the capacity of the queue of undelivered messages. If the limit of waiting messages reaches, DKVF throws an exception. Calling `sendToServer` is thread-safe. Thus, the protocol designer does not need to worry about concurrency or failure issues. To send the response to the client, the `ClientMessageAgent` class provides `sendReply` method that allows us to send the response to the client message.

While we are processing client/server messages, we also need to store or retrieve data from the storage engine. The `DKVFServer` class provides methods that can be used for this purpose. Two main methods are `read (String k, Predicate<Record> p, List<Record> result)` and `insert (String k, Record rec)`. The first method reads all versions of the data item with key k that satisfy predicate p. The second method adds record rec to the version chain of the data item with key k. The default storage can be configured to be multi-version or single-version.

Now, as an example, let us consider an implementation of the GET request handler of GentleRain. Algorithm 8.1 shows the pseudocode of this event handler copied from the original paper [36]. The following code shows the corresponding necessary code for DKVF. First, we update the GST value by calling `updateGst` method. This method basically does what Line 2 of Algorithm 8.1 does in a thread-safe manner. Next, we call `read` of the framework to read the value of the requested data item. We pass a predicate to the read function to find the visible version according to the GentleRain algorithm. Specifically, a version is visible if either it is written locally, or its update time is smaller than GST. Finally, we create `ClientReply` message containing the value of the key, and send it to the client by calling `sendReply` method of the `ClientMessageAgent` object. Note that in the following code, we ignore exception and error handling for sake of presentation.

```
void handleGetMessage(ClientMessageAgent cma) {
 GetMessage gm = cma.getClientMessage().getGetMessage();
 updateGst(gm.getGst()); //Thread-safely update GST
 List Record result = new ArrayList<>();
 StorageStatus ss = read(gm.getKey(), (Record r) -> {
  if (m == r.getSr() || r.getUt() <= gst.get())
   return true;
  return false;}, result);
 Record rec = result.get(0);
 ClientReply cr = ClientReply.newBuilder().setStatus(true).setGetReply(GetReply.newBuilder()
     .setValue(rec.getValue()).setUt(rec.getUt()).setGst(gst.get())).build();
     cma.sendReply(cr);
}
```

8.2.3 Client-side Implementation

To implement the client side of a protocol, we need to extend the client part of the framework. Specifically, we need to write a class that extends class `DKVFClient`. When we extend `DKVFClient`, we have to implement two abstract methods `put` and `get` that are the basic PUT and GET operations of a key-value store. These methods are operations that the protocol designer needs to provide for the application developer. The application developer later can use these methods to use the data store (see Figure 8.1). The protocol designer can also add more complex operations for its implementation, but these two methods are required for any implementation.

To process application requests, the client part needs to send client messages and receive responses from the servers. Finding the correct node to send the request is the problem of *service discovery* [40]. DKVF does not force any service discovery policy, and lets protocol define it. DKVF, on the other hand, provides convenient ways to send/receive messages to/from servers via their IDs specified in the client configuration file. Specifically, `sendToServer(String id, ClientMessage cm)` sends a client message to the server with ID `id`, and `readFromServer (String id)` reads the response from server with ID `id`.

Now, let us consider client side of PUT operation of GentleRain. Algorithm 8.2 shows the PUT operation at client side in the GentleRain. The following code shows the corresponding DKVF code. To find the correct server to send the PUT request, we call `findPartition` function. DKVF `Utils` library provides utilities to distribute the keys according to their hash values. The rest of the handler is clear and identical to the pseudocode.

```
public boolean put(String k, byte[] v) {
 ClientMessage cm = ClientMessage.newBuilder().setPutMessage(PutMessage.newBuilder().\\
 setDt(dt).setKey(key).setValue(ByteString.copyFrom(value))).build();
 String serverId = findPartition(key) //finds server ID
 sendToServer(serverId, cm);
 ClientReply cr = readFromServer(serverId);
 dt = Math.max(dt, cr.getPutReply().getUt());
 return true;
}
```

Algorithm 8.2 Pseudocode of the PUT handler of the client-side of GentleRain protocol

1: **PUT** (key k, value v)
2: send $\langle$PUTREQ $k, v, DT_c\rangle$ to server
3: receive $\langle$PUTREPLY $ut\rangle$
4: $DT_c = max(DT_c, ut)$

8.3 Benchmarking with YCSB

YCSB, originally developed by Yahoo!, is a tool for evaluating the performance of key-value or cloud serving stores [29]. To use YCSB, we need to write a YCSB driver that lets YCSB client class use our key-value store. YCSB has a core workload generator. We can specify different parameters for the core workload generator such as read proportion, insert proportion, value size, number of client threads, number of operations, and so on. Once we specified the workload and driver for YCSB, we can run it to benchmark our system. YCSB gives us different measurements such as throughput and latencies.

DKVF comes with a driver for YCSB. Thus, any key-value store written based on DKVF has its YCSB driver ready. DKVF also includes a workload generator. The DKVF YCSB workload generator extends the YCSB core workload generator by adding new operations such as amplified insert to benchmark the system against query amplification (see Section 3.2.2). This feature allows us to evaluate the performance of macro operations that reveal bottlenecks when a query results in multiple operations on the key-value store.

Figure 8.2 shows the components involving in benchmarking a key-value store created by DKVF. The person who wants to benchmark the system, referred to as benchmark generator, needs to provide two components shown by dark rectangles in Figure 8.2. The first component is the workload properties. The benchmark generator can specify any YCSB core properties for the workload. For benchmarking query amplification, we can specify the amplification factor. The benchmark generator also needs to provide a client configuration file that specifies servers to connect, and other client-side parameters (see Section 8.1). The workload generator is also extensible. Specifically, if we want to benchmark an operation that is not included in DKVF, we need to implement a customized YCSB driver and workload generator. We refer the reader to [29] for details.

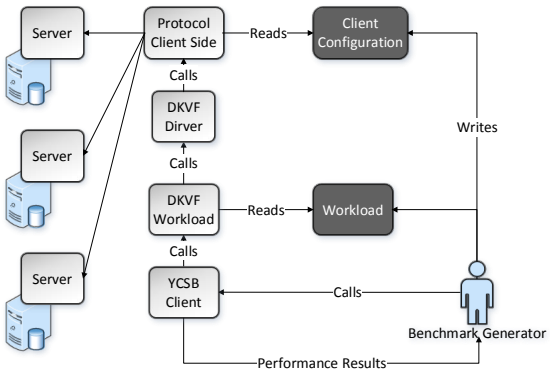


Figure 8.2: Using of YCSB for evaluating a prototype created by DKVF

8.4 Tools

In this section, we introduce two tools that help protocol designers to run and benchmark their distributed key-value stores created with DKVF. These tools can save us a great deal of time and headache in running and benchmarking our systems.

8.4.1 Cluster Manager

Cluster Manager is a command line application to facilitate managing clusters running key-value stores created with DKVF. It also helps us to run distributed YCSB experiments. Using this tool,

we can benchmark our key-value store without directly setting up YCSB; we only need to define our desired workload, and Cluster Manager takes cares of the rest.

To run a cluster, we need to write a cluster descriptor file. This descriptor file is an XML file according to `ClusterDiscriptor.xsd` [6], and specifies various aspects such as the IP address of the servers, port numbers to listen for incoming client/server messages, the topology of the servers, and so on. After loading a cluster descriptor file, we can us Cluster Manager to start all servers. Cluster Manager also enables us to monitor the servers. For instance, we can see if servers have properly started and connected to each other, how much are the network latencies, or how many clients are connected to each server.

Cluster Manager also helps us to test and debug our key-value store. Specifically, after running a cluster we can connect to any server in the cluster and run commands on the servers. For instance, suppose we want to test the convergence of our protocol. We can connect to a replica, and write a value for some key. Next, we can connect to another replica to see if our write has been replicated to the second replica properly. This kind of debugging is very convenient with Cluster Manager. Cluster Manager uses an instance of the client side of our key-value store to interact with the server. Thus, we need to specify our client class for Cluster Manager in the cluster descriptor file.

After running a cluster, and testing it with Cluster Manager, we can conduct an experiment to see how well our protocol performs. We need to write an experiment descriptor file for each experiment. The experiment descriptor file is an XML file according to `ExperimentDescriptor.xsd` [6], and specifies experiment related parameters such as how many clients we want to run, what are the addresses of the client machines, each client is connected to which servers, what are the workloads, and so on. We can define a set of experiments for Cluster Manager in our descriptor file. After loading an experiment descriptor file, we can use Cluster Manager to run the experiment. The Cluster Manager conducts the experiments one by one by running YCSB clients, and gather the results from clients. To aggregate the results, Cluster Manager provides us with a minimal query language that lets us select measurements we want to aggregate and specify how we want to aggregate them (e.g., taking the average).

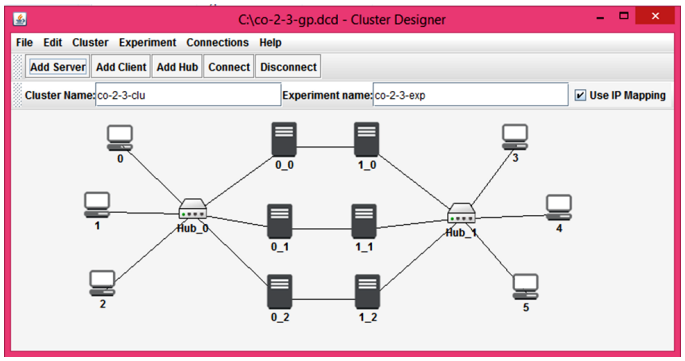


Figure 8.3: The graphical interface of Cluster Designer

8.4.2 Cluster Designer

Although Cluster Manager is a convenient tool that can significantly reduce the time and headache of debugging and benchmarking our protocol, writing cluster and experiment descriptor files can be a tedious and of course error-prone task for larger clusters. To solve this issue, we provide Cluster Designer tool. Cluster Designer is a graphical tool that allows us to define our cluster and experiments visually. The tool provides an area where we can add servers and clients. We can connect servers and clients by lines to specify network connections. When we have several components that need to be all connected to each other, we can use hubs to avoid connecting them one-by-one. Figure 8.3 shows the interface of Cluster Designer. In this network, we have 6 servers and 6 clients. We will talk about this network in more details in 8.5.

We can define default configurations for servers/clients. We later can tailor default configurations for an individual server/client. After designing our cluster and experiments, we can use Cluster Designer to export descriptor files. We can later use Cluster Manager to run our cluster and experiments as explained in Section 8.4.1.

8.5 Experimental Results

In this section, we present some of the results that we obtained from implementing and evaluating three causal consistency protocols namely COPS [49] [1], GentleRain [36], and CausalSpartan [60]

[1] We have implemented a simplified version of COPS without garbage collection.

Table 8.1: The number of lines of code that we wrote to implement different protocols with DKVF.

Protocol	Server Side	Client Side	Metadata
Eventual	95	58	32
COPS	269	84	45
GentleRain	226	61	50
CausalSpartan	292	118	53

(i.e. CausalSpartanX without ROTX operations) using DKVF. We also implemented eventual consistency for comparison. Table 8.1 shows the number of lines of code that we wrote to implement each of these protocols. For each protocol, we have reported the number of lines that we wrote for server side, client side, and describing our metadata in `.proto` file for protobuf. Of course, numbers of lines of code is not an accurate indicator, as different people may write the same program in different ways, but we report them here just to give you an estimate of the coding effort that we needed to put to implement these protocols using DKVF. You can access our implementations in [6].

Without DKVF we needed 843 lines to implement CausalSpartan and 769 lines to implement GentleRain. In implementing CausalSpartan and GentleRain without DKVF, we used Netty [12] for network communications. The number of lines of code that we needed to implement CausalSpartan and GentleRain with DKVF is around 40 percent of what we needed to implement them without DKVF. Note that reducing the number of lines of code is not the only goal of DKVF. Instead, using DKVF has all the benefits that we mentioned in the introduction.

We have not implemented COPS and eventual consistency without DKVF. With DKVF, it took only 2 days to implement COPS based on the description of the protocol in [49]. Furthermore, all the code developed with DKVF essentially required us to convert the pseudocode in the respective papers into Java and add error handling that is generally omitted in the pseudocode. In this sense, writing the code required with DKVF was straightforward.

8.5.1 Experimental Setup

We consider a replicated and partitioned data store shown in Figure 8.3. The data store consists of two replicas. Each replica consists of three partitions. Replica 0 includes partitions 0_0, 0_1, and 0_2. Replica 1, on the other hand, consists of partitions 1_0, 1_1, and 1_2. We assume full replication, i.e., each replica has a copy of the entire key space. The key space inside each replica is partitioned among servers. In Figure 8.3, we have connected servers inside each replica together with a hub. Partitions are also connected to their peers in the other replica. For servers, we use AWS `m3.medium` instances with the following specification: 1 vCPUs, 2.5 GHz, Intel Xeon E5-2670v2, 3.75 GiB memory, 1 x 4 (GB) SSD Storage Capacity.

Connected to each replica, we have a set of clients. We allocate three client machines to run clients. We run 30 threads of YCSB clients on each client machine. All causal consistency protocols that we study here assume locality of traffic, i.e., clients always access one replica. Thus, clients are connected to only one replica as shown Figure 8.3. We run clients on `c3.large` machines with the following specification: 2 vCPUs, 2.8 GHz, Intel Xeon E5-2680v2, 3.75 GiB memory, 2 x 16 (GB) SSD Storage Capacity. We have used more powerful machines for clients to better utilize our servers.

8.5.2 The Effect of Workload on Performance

The workload of different applications has different characteristics. Some workloads are write-heavy, others like those in data analytics are read-heavy. In this section, we want to study how the characteristics of our workload affect the performance of different consistency protocols. In all experiment, we set the size of the values written by clients to 64 bytes.

Figure 8.4 shows how GET:PUT proportion affects the throughput. As we move from the left side of the plot to its right side, the workload nature changes from write-heavy to read-heavy. The throughputs of all protocol increase as the proportion of GET operations increases. This results confirm previous studies [36, 49], and are expected, as GET operations are lighter than PUT. As expected, eventual consistency has the highest throughput. COPS, on the other hand, has the lowest

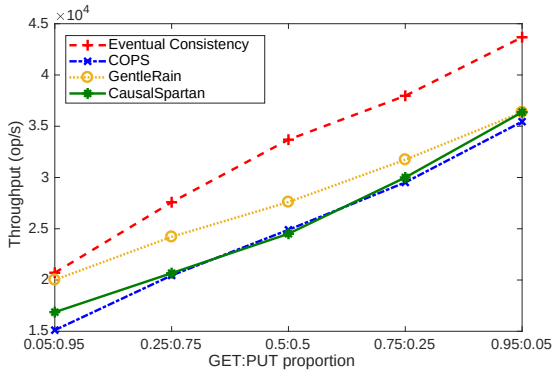
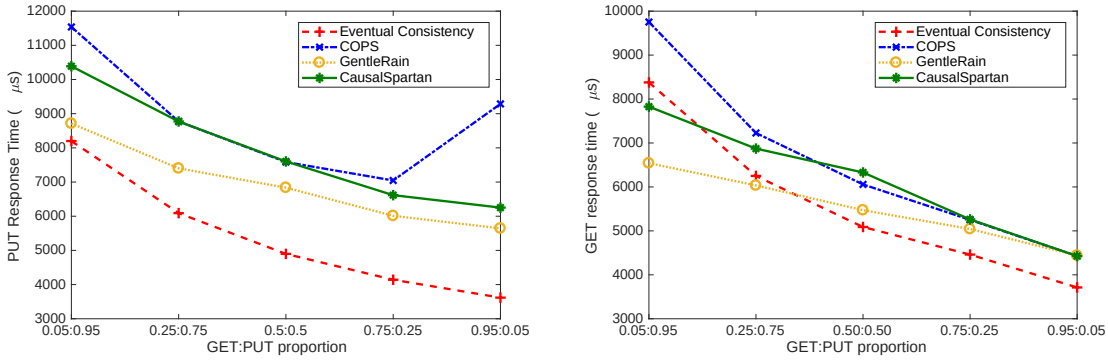


Figure 8.4: Throughput vs. GET:PUT proportion

throughput. This results confirm results published in [36], and is due to the overhead of dependency check messages that partitions send to each other to make sure causal dependencies of an update in other partitions are visible (see Section 3.2.2).

Figure 8.5a shows how GET:PUT proportion affects the response time of PUT operations. In all protocols, the response time of PUT operations decreases as we move to read-heavier workloads. This is due to the less load on servers for read-heavier workloads. The eventual consistency has the shortest response time thanks to its minimal metadata. CausalSpartan has more metadata than GentleRain resulting in higher PUT response time. COPS has the highest response time because of its dependency check messages and its explicit dependency tracking approach. Like other protocols, the trend of PUT response time for COPS is decreasing as we move toward read-heavier workloads that can be explained by less load on the machines. However, for 0.05:0.95, the PUT response time increases. This increase can be understood by considering the dependency tracking mechanism of COPS. At point 0.05:0.95, clients read many keys before writing a key. That results in longer dependency lists which make PUT messages heavier to transmit and process. Note that we have implemented a basic version of COPS protocol without client metadata garbage collection. COPS authors suggest a garbage collection mechanism to cope with this problem [49].

Figure 8.5b shows how GET:PUT proportion affects the response time of GET operations. Like the case of PUT operations, the response time of GET operations also decreases, as we move towards read-heavier workloads. It is interesting that GentleRain and CausalSpartan have



(a) PUT response time

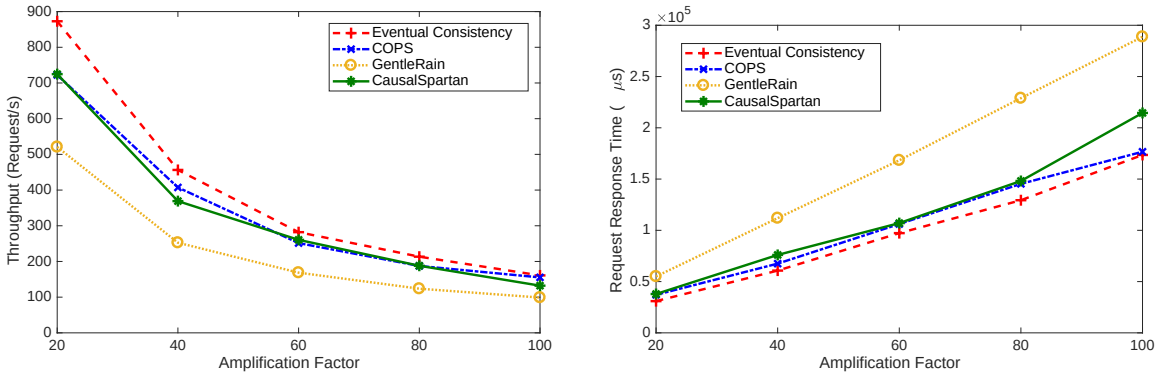
(b) GET response time

Figure 8.5: The effect of GET:PUT ratio on response time

a lower response time for GET operations comparing to the eventual consistency for write-heavy workloads. This can be explained by the synchronization that occurs between threads in GentleRain and CausalSpartan. Specifically, there is a contention between threads while performing PUT operations in GentleRain/CausalSpartan. This contention occurs for obtaining a lock that we used to guarantee updates with smaller timestamps are replicated to other nodes before updates with higher timestamps. This increases the PUT response time that results in lower overall throughput of GentleRain/CausalSpartan for write-heavy workloads. While threads serving PUT operations are waiting for synchronization, the server can handle GET operations. On the other hand, in the eventual consistency, there is no competition between PUT operations. Thus, there are more active threads serving PUT operations leading to higher competition over CPU that finally results in higher GET response time comparing to GentleRain/CausalSpartan. Note that this happens for write-heavy workloads with low GET proportion. Therefore, the eventual consistency still has the highest overall throughout in all cases (See Figure 8.4).

8.5.3 The Effect of Query Amplification

In this section, we study the effect of query amplification on the performance of the system. In this section, we only consider one replica consisting of three partitions. We consider a workload that purely consists of amplified insert operations. Each amplified insert consists of several internal



(a) Request throughput

(b) Request response time

Figure 8.6: The effect of amplification factor on response time and throughput

PUT operations. The number of internal PUT operations is defined by the amplification factor.

Figure 8.6a shows the effect of amplification factor on the client request throughput. Note that this throughput represents the number of client macro operations (not individual PUT operations) that are served in one second. As the amplification factor increases, the throughput of all protocol decreases which is expected, as requests with higher amplification factor include more internal operations which mean more job to do for each request. The eventual consistency has the highest throughput. The pure-write workload is an ideal write scenario for COPS, as dependency lists have at most one entry. Thus, the throughput of COPS is the highest after eventual consistency for this scenario. GentleRain has the lowest throughput. That is due to the delay that GentleRain imposes on PUT operations in case of clock skew between servers. Note that we synchronized the physical clocks of the system with NTP [14], but the effect of clock skew still shows up in the results. These results confirm previous results presented in [60]. CausalSpartan has higher throughput than GentleRain, as CausalSpartan eliminates the need for the delay before PUT operations by utilizing HLCs instead of physical clocks [60]. Figure 8.6b shows the request response time for different protocols. Again, because of delays that GentelRain forces on PUT operations, request response time has the highest value for GentleRain.

CHAPTER 9

FUTURE WORK

In this Chapter, we focus on some of the possible directions that can be pursued as the future work of this dissertation.

9.1 Hybrid Protocol for Causal Consistency

In Chapter 1, we categorized causal consistency protocols to *explicit* and *implicit* protocols. Explicit protocols track and check causal dependencies explicitly by keeping lists of actual causal dependencies of the versions. Implicit protocols such our CausalSpartanX, on the other hand, track causal relations via timestamps and check causal dependencies implicitly via constantly monitoring some sort of the *stable time* in a replica. Implicit protocols reduce metadata overhead and message complexity compared with explicit methods. However, they can leads to higher update visibility especially due to slowdowns of partitions. For example, suppose we have a new update v that has only one causal dependency on partition A. In an explicit protocol, we can easily send a dependency check message to server A to check if the causal dependency is available. On the other hand, an implicit protocol such as GentleRain [36] needs to wait for all servers in the system to send a replicate/heartbeat message with a timestamp higher than that of v to their peers in the replica and partitions inside the replica communicate with each other to find the stable time that is higher than timestamp of v before making v visible. This can delay visibility of v if only one of the partitions in one of the replicas is slowed or failed. We alleviate this problem by keeping track of dependencies for different replicate separately. However, compared with an explicit protocol, our CausalSpartanX still suffers from *unnecessary* delay before making a version visible due to the implicit approach.

We believe an implicit approach is still much more practical than the explicit approach, because it allows us to achieve considerably better performance by introducing a small increase in the update visibility. As mentioned earlier, the overhead of the explicit approach comes from two problems:

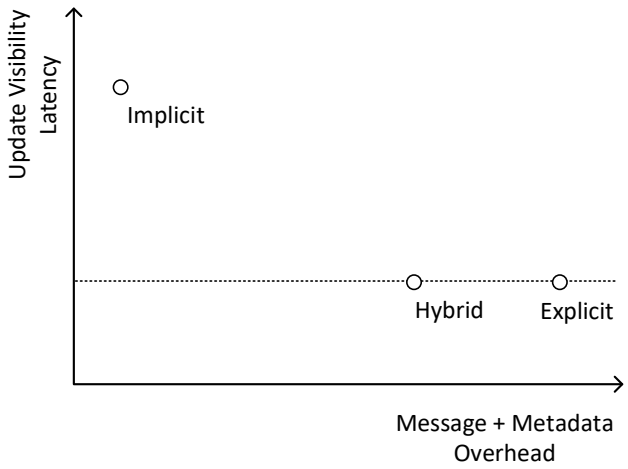


Figure 9.1: How hybrid approach can reduce the overhead of explicit approach while keeping the same update visibility latency

1) metadata overhead of explicit dependency lists, 2) sending dependency check messages. We believe using a hybrid approach, we can prevent the second overhead. Specifically, we can track dependency via dependency lists, and send dependency check messages only when the stable time calculated in a replica is not enough to make sure dependencies are present. We can also postpone checking dependency to when a client wants to read the key– *lazy* dependency checking. This way, we increase the probability of checking dependency by the stable time that allows us to avoid sending unnecessary dependency check messages. The hybrid approach makes visibility latency overhead the same as that of explicit approach while avoiding sending too much dependency check messages. Thus, the hybrid will be superior to the explicit approach. However, since we still need explicit dependency lists, we cannot say the hybrid approach is superior to the implicit approach regarding the metadata overhead. Thus, if we consider two factors 1) update visibility latency and 2) metadata + message complexity overhead, we anticipate that such hybrid approach will position as it is shown in Figure 9.1 compared to the explicit and implicit approach, i.e. it outperforms explicit but not the implicit. Designing a hybrid protocol and experimenting it to see how it works compared with explicit and implicit approach is interesting future work.

9.2 Other Data Models

In this work, we focused on key-value data model. An interesting direction for future work is exploring protocols for other data models such as graph databases or document stores. To do that, we first need to define different consistency levels for such data models, and then design protocols to provide the desired consistency levels. Although we can represent other data models such as graph via key-value, a system that is intended to provide a certain data model can be optimized to provide better performance for operations specific to that data model. Such optimization should be considered in designing protocols for each data model.

9.3 Offline Availability while Providing Consistency

Many of consistency protocols provided in the literature, including those proposed in this dissertation, consider online systems, i.e. clients are always connected to the servers. Such an assumption is reasonable if we consider web servers clients especially when they reside in the data center where data servers reside. However, if consider clients closer to the end user (e.g. laptop machine or cell phones), then the assumption of online systems can be restrictive as the system should be unavailable when clients are not connected to the servers. Interestingly, right now that I am writing these lines, I am on the plane and do not have access to the Internet. Since I do not have access to the Internet, I cannot edit my Latex documents on Overleaf [15] which is a cloud-based Latex editor that allows several people to work on the same document at the same time. It guarantees that all people editing a document see the same version at all time, but it requires clients to be always connected to the servers. To allow clients to continue their work even when there are offline, clients should be able to write and read to and from some sort of client cash. Some protocols such as [66] use client cash for providing offline availability while satisfying certain levels of consistency. However, we believe there is a lot more to investigate in this area.

9.4 General Transactions

In this work, we provided an algorithm for causally consistent read-only transactions. Providing general transactions that allow both read and write is future work. [64] and [54] are examples of protocols that provide causally consistent general transactions.

9.5 Model-based Development of Consistency Protocols

Although there are several theorems such as CAP theorem for replicated data stores, there is a lack of systematic unifying theory to characterize the spectrum of consistency protocols so that the designers can utilize the exact tradeoff that they can obtain. Such a unifying theory could answer questions such as 'is it possible to design a protocol that combines feature A of protocol 1 with feature B of protocol 2?' If so, 'what is the corresponding protocol and will it preserve feature C?'. We can develop the notion of the abstract data store and abstract 'high-level' protocols and demonstrate how existing protocols refine the high-level abstract data store. This way we can find the relation between different consistency protocols and expected constraints invariant properties of the system that implements the given level of consistency. We can see how existing protocols refine the constraints of the abstract protocol. It allows us to see how new protocols can be designed based on alternate approaches to refine the abstract protocol.

We can define our abstract data store using various formalism. One possibility is to define the state of a data store via a set of variables and define a protocol using a set of actions. For example, for the key-value data model, the set of variables are our set of keys. Now, let $v(k)_s$ be the value of key k and $t(k)_s$ be the timestamp assigned to the current version of key k at server s. Also, let $PUT(k,v)_s$ be a predicate that is true when a client requests to write value v for key k, and $GET(k)_s$ be a predicate that is true when a client requests to read value of key k at server s. Moreover, let ret_s be the value returned to the client from server s. Now, we can define the abstract eventual consistency with last-write-wins conflict resolution protocol using following three guarded actions (the left side of $\rightarrow$ is a condition that when is true, the system does what is on the right side of the $\rightarrow$):

- $PUT(k,v)_s \rightarrow v(k)_s = v, t(k)_s = \langle current\_time, server\_id \rangle, ret_s = OK$
- $GET(k)_s \rightarrow ret_s = v(k)_s$
- $\exists s_i, s_j, k : t(k)_{s_i} > t(k)_{s_j} \rightarrow v(k)_{s_j} = v(k)_{s_i}$

Note that the above description is very abstract and can be refined in different ways. For example, to refine the third actions, servers can *asynchronously* send replicate message to their peers to inform them about a new update after returning to the client, or they can do the replication *synchronously* by informing peers before returning to the client. Another alternative is to give the responsibility of replication to the clients (*client-based* replication). We define $REP(k,v,t)_s$ a predicate that represents the event of receiving a replicate message with value v and timestamp t for key k. Also, $PUT(k,v,t)_s$ is the predicate that is true when when a client request to write value v for key k with timestamp t. Now, we can define eventually consistent systems with different replication approaches as follows:

- Asynchronous replication:
  - $PUT(k,v)_s \rightarrow v(k)_s = v; t(k)_s = \langle current\_time, server\_id \rangle; ret_s = OK; \forall s' : REP(k,v,t(k)_s, s') = TRUE$
  - $GET(k)_s \rightarrow ret_s = v(k)_s$
  - $REP(k,v,t)_s \wedge t(k)_s < t \rightarrow v(k)_s = v; t(k)_s = t$
- Synchronous replication:
  - $PUT(k,v)_s \rightarrow v(k)_s = v; t(k)_s = \langle current\_time, server\_id \rangle; \forall s' : REP(k,v,t(k)_s, s') = TRUE; ret_s = OK$
  - $GET(k)_s \rightarrow ret_s = v(k)_s$
  - $REP(k,v,t)_s \wedge t(k)_s < t \rightarrow v(k)_s = v; t(k)_s = t$
- Client-based replication:

- $PUT(k,v)_s \rightarrow v(k)_s = v; t(k)_s = \langle current\_time, server\_id \rangle; ret_s = OK; \forall s' : PUT(k,v,t(k)_s)'_s = TRUE$
- $GET(k)_s \rightarrow ret_s = v(k)_s$
- $PUT(k,v,t)_s \wedge t(k)_s < t \rightarrow v(k)_s = v; t(k)_s = t$

These are just three examples of how we can refine the abstract eventual consistency protocol. Thus, a key-value store that refines eventual consistency does not to be one these three. For instance, NuKV (see Section 4.1) refines synchronous replication for replicas inside the same data center, and uses asynchronous replication for replicas in remote data centers. These protocols can be refined further until we reach concrete protocols. For example, NuKV refines synchronous replication using Raft [57] algorithm, but we could refine it in different ways such as multi-Paxos algorithm [47]. Also, note that all protocols that refine stronger consistency protocols, also refine weaker consistency protocols. For example, our CausalSpartanX that refines causal++ consistency also refines eventual consistency. In fact, CausalSpartanX refines asynchronous eventual consistency protocol. Figure 9.2 shows how these abstract and concrete protocols are related. This figure is just an example of how we can map various consistency protocols and find their relations. Also, note that the formalism provided here is an example of how we can model abstract protocols. We can use different formalism such as timed automata that enable us to model time aspects of consistency protocols.

9.6 Adaptive Causal Consistency

In Chapter 7, we introduced our approach for providing adaptive causal consistency. We provided our framework, ACCF, that facilitates creating such adaptive systems. ACCF is flexible framework that can treat applications individually and can be easily configured by changing its tracking and checking groups. Designing an adaptive causal consistency protocol using ACCF is interesting future. There are some of possible ways for dynamically changing tracking and checking groups to build an adaptive causal consistency protocol:

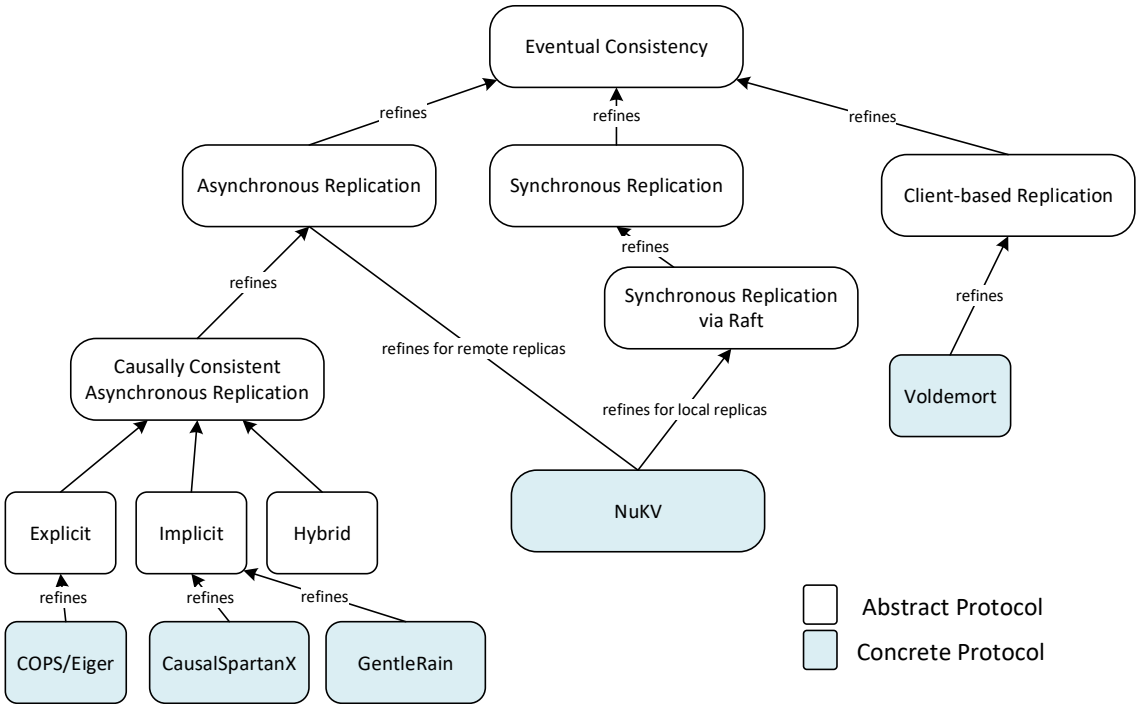


Figure 9.2: Example of how different consistency protocols refines abstract protocols

Dynamically adding or removing checking groups: Adding checking groups is a straightforward process. Each checking group is associated with a data structure (e.g., SVV in the algorithm provided in Section 7.2) that the servers need to maintain (in RAM). Hence, if we want to add a new checking group, the system can run the protocol to initialize these fields and make the new checking group available. Removing a checking group is somewhat challenging especially if some client is using it. In this case, we anticipate that the principle-of-locality would be of help. If a client has not utilized a checking group for a while, in most cases, all the data the client has read has been propagated to all copies in the system. In other words, if a client is using a checking group that has disappeared, we can have the client choose a different checking group. It is unlikely to lead to delays, as all replicas already have the data that the client has read. Two practical questions in removing checking groups are (1) the *time* after which we can remove a checking group and (2) how servers can determine that no client has accessed that checking group in that time. A more difficult question in this work is *when* to add a new checking group and *how many* checking groups to maintain. Clearly, we cannot create a checking group for each possible client, as it would require

exponentially many checking groups.

Utilizing multiple checking groups simultaneously: Yet another question is whether clients could have multiple checking groups or whether clients can change their checking group. The former would be desirable when the system does not offer a checking group that the client needs. However, the client could choose two (or few) checking groups whose union is a superset of the checking group requested by the client. In this case, the server providing the data would have to utilize all of these checking groups –on the fly– to determine which data should be provided to the client.

Learning required checking groups automatically: In this case, the system will learn from client requests to identify when new checking groups should be added and when existing checking groups should be removed. We expect that dynamically changing the checking groups in this manner would be beneficial due to principle-of-locality, where clients are likely to access data that *similar* to the data they accessed before. (Recall that we assume that keys are partitioned with semantic knowledge rather than by approaches such as uniform hashing). We anticipate that learning techniques such as evolutionary or machine learning techniques would be useful to identify the checking groups that one should maintain.

Dynamically changing the tracking groups: Dynamically changing the tracking groups is more challenging but still potentially feasible in some limited circumstances. The reason for this is that while checking groups affect the data maintained by the servers at run-time (in RAM) tracking groups affect storage affected by keys (in long-term permanent storage). In other words, at runtime, we may run into a key that was stored with a different tracking grouping. In this case, it is necessary to convert the data stored with the old tracking grouping into the corresponding data in the new tracking grouping. We expect that principle-of-locality would be of help in this context as well; keys stored long ago are likely to have been updated in all replicas. Conversion of the data stored with keys is protocol specific but still feasible. For example, if we wanted to switch between tracking grouping used by CausalSpartan [60] (where a vector DSV is maintained with one entry per replica) to GentleRain [36] (where only a scalar entry GST is maintained) then we could convert

the DSV entry into a GST entry that corresponds to the minimum of the DSV entries. However, the exact approach to do this for different tracking groupings requires semantic knowledge of those tracking groupings.

9.7 Partial Replication

In this works, like most of the protocols reviewed in Chapter 3, we focused on full replication where each replica has the full copy of the data. Exploring partial replication and designing protocols for it could be future work. Our ACCF introduced in section 7 also provides a basis for causally consistent and partially replicated systems. Partial replication also can be related to the offline availability discussed in Section 9.3, as a client cash can be considered as a partial replica that has only a part of the whole key space.

CHAPTER 10

CONCLUSION

In this dissertation, we considered the problem of consistency for replicated data stores. We focused on intermediate consistency models that are stronger than eventual consistency and weaker than strong consistency. They eliminate many of inconsistencies caused by eventual consistency and at the same, unlike strong consistency, do not cause high performance and availability overhead. Session guarantees are one of such intermediate consistency models. We provided modified definitions of session guarantees that, unlike conventional session guarantees, do not cause slowdown cascades. We presented our analysis of the cost associated with providing session guarantees for NuKV, a key-value store that aims to simultaneously provide high availability and consistency for eBay services. In particular, NuKV maintains the data in multiple data centers and each data center contains multiple replicas of the data. We showed that if we introduce session guarantees, and the client remains within the same data center, the overhead of session guarantees compared with eventual consistency is within experimental error. Furthermore, even if the client changes its data center, the cost of increased latency is very small, $\approx 10(ms)$. We demonstrated this for different types of workloads (100% read, read-heavy, write-heavy and 100% write). Our analysis showed that providing write session guarantees highly benefits from the use of HLCs [42].

In addition to session guarantees, we considered another intermediate consistency model called causal consistency. Causal consistency is especially interesting as it is proved to be the strongest consistency model that remains available in presence of network partitions. We presented CausalSpartanX, a time-based protocol for providing causal consistency for replicated and partitioned key-value stores. Unlike existing time-based protocols such as GentleRain that relies on the physical clocks, our protocol is robust to clock anomalies thanks to utilizing HLCs instead of physical clocks. CausalSpartanX guarantees that the response time for client operations are unaffected by clock anomalies such clock skew. For example, for the clock skew of 10 (ms), the average response time for PUT operations of CausalSpartanX was 4.5 (ms) whereas the average response time of

GentleRain was 7.6 (ms). Also, the correctness of CausalSpartanX is unaffected by NTP kinks such as leap seconds, non-monotonic clock updates and so on. The improvement of CausalSpartanX over GentleRain is more obvious in the context of query amplification where a single end user's request translates to many internal requests. In our experiments, we observed even for 10 (ms) clock skew, CausalSpartanX reduces the average response time from 814 (ms) to 124 (ms) (for a query that consists of 100 operations) and from 3800 (ms) to 407 (ms) (for a query that consists of 500 operations).

Another advantage of CausalSpartanX lies in the fact that it reduces update visibility latency. Specifically, causal consistency protocols need to delay a remote update from being visible to ensure causal consistency. However, such delays can cause substantial increase in latency for collaborative applications where two clients read each other's updates to decide what actions should be executed. As a simple application of this collaborative application, we considered the abstract bidding problem where one client reads the updates (using data center A) from another client (using data center B) and decides to increase its own bid until a limit is reached. We performed this experiment where A and B were in California, but the location of another data center, say C was changed. CausalSpartanX performance remained unaffected by the location of C. By contrast, in GentleRain, the latency increased from 46 (ms) to 88 (ms) when we move data center C from Oregon to Singapore.

In addition to the basic GET and PUT operations, CausalSpartanX provides read-only transaction operation, ROTX, that allows application developers to read a set of keys such that the returned values are causally consistency with each other as well as with the client past reads. CausalSpartanX ROTX is non-blocking, i.e., servers receiving the request can read the requested values immediately, and it only requires one round of communication between the client and the servers. In addition, slow servers that are not involved in the transaction do not affect the response time of it. This feature provides a significant improvement for CausalSpartanX, especially regarding the tail latency, compared with the existing protocol such as GentleRain where the slowdown of a single partition in a data center affects the response time of all transactions in the data center. Our experiment shows that for a 100 (ms) slowdown of a partition, the 90th percentile of ROTX response time of

CausalSpartanX shows 86.04% and 29.7% improvement over that of GentleRain, for transactions not involving the slow partition and transactions involving the slow partition, respectively. For higher slowdowns and higher percentiles, this improvement is more significant. For example, for 500 (ms) slowdown and 99th percentile, these numbers go up to 95.99% and 37.70%, respectively.

CausalSpartanX together with the majority of causal consistency protocols assumes sticky clients i.e., clients only access their local replica. We provided an impossibility result which states in presence of network partitions the stickiness of clients is necessary, to have an always available causally consistent data store that immediately makes local updates visible. This impossibility result is different than the existing impossibility result that requires sticky clients for read-your-writes and still holds even when clients can cache their past read and writes.

Existing causal consistency protocols, including our CausalSpartanX, utilize a static approach in tracking and checking dependencies. In this dissertation, we explained the need for developing a system that provides causal consistency in an adaptive manner. Specifically, we introduced the notion of tracking and checking groups as a way to generalize existing protocols as well as to develop new adaptive protocols. We provided a framework that, unlike existing causal consistency protocols, can be configured to work with different tracking and checking groupings. This flexibility enables us to trade off between conflicting objectives, and provide different views to different applications so that each application gets the best performance. We argue that the approach and the framework introduced in this dissertation provide a basis for adaptive causal consistency for replicated data stores.

To facilitate evaluating future protocols, we developed DKVF which is a framework for rapid prototyping and benchmarking distributed key-value stores. It streamlines the evaluation of the performance of consistency protocols for distributed key-value stores. The great advantage of DKVF is that it allows us to implement our protocols by writing a piece of code that is very close to the pseudocode that protocol designers include in their research papers. Thanks to the convenience of DKVF, we were able to implement each of four consistency protocols namely eventual consistency, COPS, GentleRain, and CausalSpartan in less than 2 days. DKVF relies on

YCSB for benchmarking. The toolset that comes with the framework helps protocol designers to easily evaluate their prototypes.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Amazon aws. https://aws.amazon.com/.

[2] Another round of leapocalypse. http://www.itworld.com/security/288302/another-round-leapocalypse.

[3] Berkeley db. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[4] Cornerstone. https://github.com/datatechnology/cornerstone.

[5] DKVF. https://github.com/roohitavaf/DKVF.

[6] DKVF. https://github.com/roohitavaf/DKVF.

[7] The future of leap seconds. http://www.ucolick.org/~sla/leapsecs/onlinebib.html.

[8] Google protocol buffers. https://developers.google.com/protocol-buffers/.

[9] grpc. https://grpc.io/.

[10] MongoDB. https://www.mongodb.com/.

[11] Msu-db. http://cse.msu.edu/~roohitav/msudb.

[12] Netty. http://netty.io/.

[13] Netty. https://coreos.com/etcd/.

[14] The network time protocol. http://www.ntp.org/.

[15] Overleaf. http://www.overleaf.com.

[16] The trouble with timestamps. http://aphyr.com/posts/299-the-trouble-with-timestamps.

[17] Voldemort. http://www.project-voldemort.com/voldemort/quickstart.html.

[18] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[19] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *HotOS*, 2015.

[20] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 85–98. ACM, 2013.

[21] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, 2013.

[22] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *NSDI*, volume 6, pages 5–5, 2006.

[23] David Bermbach, Jörn Kuhlenkamp, Bugra Derre, Markus Klems, and Stefan Tai. A middleware guaranteeing client-centric consistency on top of eventually consistent datastores. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 114–123. IEEE, 2013.

[24] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[25] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. From session causality to causal consistency. In *PDP*, pages 152–158, 2004.

[26] Jerzy Brzeziński, Cezary Sobaniec, and Dariusz Wawrzyniak. Safety of a server-based version vector protocol implementing session guarantees. In *International Conference on Computational Science*, pages 423–430. Springer, 2005.

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[28] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, 1991.

[29] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[30] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[31] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[32] Alan Demers, Karin Petersen, Mike Spreitzer, Doug Terry, Marvin Theimer, and Brent Welch. The bayou architecture: Support for data sharing among mobile users. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 2–7. IEEE, 1994.

[33] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. Causal consistency and latency optimality: Friend or foe?[extended version]. *Proceedings of the VLDB Endowment*, 11(11), 2018.

[34] Diego Didona, Kristina Spirovska, and Willy Zwaenepoel. Okapi: Causally consistent geo-replication made faster, cheaper and more available. *arXiv preprint arXiv:1702.04263*, 2017.

[35] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013.

[36] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014.

[37] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[38] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[39] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[40] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. " O'Reilly Media, Inc.", 2017.

[41] Anna Kobusinska, Cezary Sobaniec, Marek Libuda, and Dariusz Wawrzyniak. Version vector protocols implementing session guarantees. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 929–936. IEEE, 2005.

[42] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *International Conference on Principles of Distributed Systems*, pages 17–32. Springer, 2014.

[43] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[44] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[45] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[46] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[47] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[48] Barbara Liskov and James Cowling. Viewstamped replication revisited. 2012.

[49] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011.

[50] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, volume 13, pages 313–328, 2013.

[51] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The snow theorem and latency-optimal read-only transactions. In *OSDI*, pages 135–150, 2016.

[52] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 295–310. ACM, 2015.

[53] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11, 2011.

[54] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *NSDI*, pages 453–468, 2017.

[55] David L Mills. Executive summary: computer network time synchronization, 2012.

[56] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.

[57] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[58] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: replicated database services for world-wide applications. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, pages 275–280. ACM, 1996.

[59] Karin Petersen, Mike J Spreitzer, Douglas B Terry, Marvin M Theimer, and Alan J Demers. Flexible update propagation for weakly consistent replication. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 288–301, 1997.

[60] Mohammad Roohitavaf, Murat Demirbas, and Sandeep Kulkarni. Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks. In *Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium on*, pages 184–193. IEEE, 2017.

[61] Mohammad Roohitavaf and Sandeep Kulkarni. Gentlerain+: Making gentlerain robust on clock anomalies. *arXiv preprint arXiv:1612.05205*, 2016.

[62] Mohammad Roohitavaf and Sandeep Kulkarni. Dkvf: A framework for rapid prototyping and evaluating distributed key-value stores. *arXiv preprint arXiv:1801.05064*, 2018.

[63] Mohammad Roohitavaf and Sandeep S. Kulkarni. Gentlerain+: Making gentlerain robust on clock anomalies. *CoRR*, abs/1612.05205, 2016.

[64] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *48th International Conference on Dependable Systems and Networks (DSN'18)*, number CONF, 2018.

[65] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 140–149. IEEE, 1994.

[66] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, pages 75–87. ACM, 2015.