J Grid Computing (2023) 21:8
https://doi.org/10.1007/s10723-022-09640-z

Efficient Causal Access in Geo-Replicated Storage Systems

Stanley Lima · Filipe Araujo · Miguel de Oliveira Guerreiro · Jaime Correia · Andre Bento · Raul Barbosa

Received: 10 September 2022 / Accepted: 5 December 2022 / Published online: 28 January 2023

Abstract We consider a setting where applications, such as websites or games, need causal access to objects available in geo-replicated cloud data stores. Common ways of implementing causal consistency involve hiding objects while waiting for their dependencies or waiting for server replicas to synchronize. To minimize delays and retrieve objects faster, applications may try to reach different server replicas at once. This entails a cost because providers charge for each reading request, including reading misses where the causal copy of the object is unavailable. Therefore, latency and cost are conflicting goals, which we control by selecting where to read and when. We formulate this challenge as a multi-criteria optimization problem and propose five non-dominated reading strategies, four of which are Pareto optimal, in a setting constrained to two server replicas. We validate these solutions on the following real cloud storage services: AWS S3, DynamoDB and MongoDB. Savings of as much as 50% on reading costs, with no significant or even a positive impact on latency, demonstrate that both clients and cloud providers could benefit from richer services compatible with these retrieval strategies.

Keywords Cloud storage systems · Causal consistency · Geo-replication · Caching

S. Lima
Anima Holding S.A. (Ânima Educação), São Paulo, Brazil
e-mail: stanleylima@dei.uc.pt

F. Araujo (✉) · M. de Oliveira Guerreiro · J. Correia · A. Bento · R. Barbosa
University of Coimbra, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, 3030-290 Coimbra, Portugal
e-mail: filipius@uc.pt

M. de Oliveira Guerreiro
e-mail: mguerreiro@dei.uc.pt

J. Correia
e-mail: jaimec@dei.uc.pt

A. Bento
e-mail: apbento@dei.uc.pt

R. Barbosa
e-mail: rbarbosa@dei.uc.pt

M. de Oliveira Guerreiro
INESC-ID, Instituto Superior Técnico, University of Lisbon, Rua Alves Redol, 9, Lisboa, 1000-029, Portugal

1 Introduction

For the sake of ensuring High Availability (HA) and low latency, many applications resort to geo-replicated data stores on the cloud. Under replication, as clients concurrently read and write to different replicas, they may observe inconsistent out-of-order message delivery, including causal order violations, where effects seem to appear before their causes. The lack of consistency in data stores is undesirable, as it increases the complexity of applications, which have to deal explicitly with seeing their data in an incorrect order.

Unfortunately, consistency is difficult to achieve. Message propagation times, network partitions, server crashes, or network reconfigurations may prevent applications from reading their writes if they cannot reach the same replica, while other timing issues related to the triangle inequality in message delays or data storage performance may lead to other causality violations. Many applications cannot afford such cases to surface. For example, in an online gaming scenario, where players connect to geo-replicated servers, if Alice adds an object to a map and sends the object reference to Bob, he must be able to see the new object, regardless of the replica he is using.

Usually, to ensure consistency, applications need to wait for some of the objects or their dependencies to arrive or become visible at a replica server. Instead of imposing this delay, we let the application read any object as soon as possible. If an object is not available at the closest replica, the application may retry the read operation, possibly multiple times, or go straight to another replica where the object is available, such as the source replica where it was first written. To curb waiting times, the application could also try to simultaneously read from the local and the source replicas. Cloud providers, however, usually charge for each read attempt, thus turning latency and cost into conflicting goals. In this paper, we formulate the problem of reading specific versions of objects from cloud data storage, in causal order, while aiming to minimize both latency and cost.

The idea of ensuring causal ordering is decades-old and assumes multiple forms. Message passing libraries, like ISIS [12], can ensure causal delivery of messages to participants. Unlike this generic approach, replicated storage systems [43, 46] offer causal access to data. The precise mechanism in which they offer causality can vary. Servers may use some lazy approach to get up-to-date [43] or they may defer access to items to ensure they never expose objects with missing dependencies [46]. Other systems, like GentleRain [22] or CausalSpartan [54] resort to physical or hybrid clocks to ensure the same guarantee. A different proposal consists of using a shim layer atop standard cheap storage to control what the client can see. In Bolt-on [7], objects with missing causal dependencies are not made available to the client, i.e., they remain invisible until all the dependencies are also available. A common pattern we observe here is that causal system implementations tend to require some form of background synchronization before objects become available for applications.

To enable faster reads, we make all objects visible and accessible at once, by pushing the responsibility of controlling causal dependencies to the application. This has one great advantage of enabling clients to interact with more than a single kind of storage at any given time. This fine-grained approach can support strict causal dependencies, instead of the "happened-before" relation, which creates unnecessary dependencies, but the formulation we propose can apply to one case or the other. When updating an object, a client must keep track of whatever dependencies it used and were relevant for that update. When another client reads the same object and follows to read one of its dependencies, it may try the replica that best suits it. The rationale is to make read operations as fast as possible by doing a judicious choice of the replica. Our goal is to read a specific object version from geo-replicated storage as soon as possible, at the smallest possible cost.

As we discuss in Section 2, we can express the latency-cost challenge as a two-criteria optimization problem based on the idea that two locations are reasonable when a client is looking for a specific object: the local region closest to the client and the source region, where the object comes from. We restrict our optimality analysis to a setting having only these two replicas. We assume that the client has a reference to the object, including name, version, source region, and creation time. These elements may come in the form of references from another newer object, they may come from a peer application explicitly providing a reference, or from a library that keeps track of all object accesses, for the sake of ensuring the "happened-before" relationship. To solve the problem, the client also needs data regarding round-trip times to replicas and replication timings. Based on these data, in Section 3, we propose non-dominated and Pareto-optimal solutions that explore different possibilities, like reading from the closest replica first, delaying the access to the closest replica, going straight to the source, or doing simultaneous accesses to both replicas.

The experiments we performed in Section 4 with Simple Storage Service (S3) [2], DynamoDB [4], and MongoDB [8] show that one can easily observe triangle inequality violations in applications resorting to geo-replicated data stores. Consequently, immediately reading from the closest replica may not be the best option for retrieving data and multiple reading strategies, as we propose, are possible. Our results show that issuing read requests at the right time and to the right replica can cut costs as much as 50%, with little or no time penalties, sometimes even faster than immediately reading from the local replica.

When compared to previous work, which we review in Section 5, our approach requires the cooperation of clients but ensures causal access to data across different storage systems. To make this approach practical, storage systems should provide the following operations: i) enable replica selection for object retrieval and ii) blocking object retrievals, i.e., applications should be able to retrieve an object immediately or, should it be unavailable at the replica, wait for the object to arrive. This entails potential gains for could providers and clients, as long as the price (for the client) and cost (for the provider) for such operation lies in between one and two read attempts.

The main contribution of this paper is, therefore, a set of object access policies that aim at ensuring causality with quality of service and economic advantages for the user and the cloud provider.

The rest of the paper is organized as follows: in Section 2 we provide a detailed overview of consistency before formalizing the optimization problem we propose. In Section 3 we propose different strategies to solve this problem. In Section 4 we observe that our formulation applies in real storage systems. In Section 5 we review related work and in Section 6 we conclude the paper.

2 Model and Problem Formulation

In this section, we present some consistency models, the object system model and formulate the problem to solve.

2.1 Consistency Trade-Offs

To persist their data, programs need to interact with storage, including distributed storage, according to a contract: storage offers an Application Programming Interface (API) and, in exchange, offers some guarantees regarding the data, like durability. As part of such contract, *consistency* usually refers to ordering properties that the storage system ensures in response to multiple, possibly concurrent, program operations, like reads and writes. For example, the system might need to ensure that the data is available or that the application has access to the latest written version, while the program might have to define transaction boundaries or use certain features of the API.

Since clients can exchange messages with each other and with different replicas, they may see seemingly contradictory information, like missing files, older versions of files they have already updated, or more subtle violations, as discussed in the literature [23, 28]. Stronger consistency models provide better guarantees, but are slower, less available, and less scalable, whereas weaker consistency models force programmers to cope with the insufficiencies of the contract. A large number of consistency models exist, namely eventual [11], PRAM [14], causal [13], sequential and linearizable [5] or strict [52], among others [56].

2.2 Eventual Consistency

If no new updates are made to replicated data, eventually, all accesses will return the last updated value [11]. That is, in the absence of further updates, all replicas converge to identical copies of the write with the latest timestamp. This model is only concerned with ensuring that, at some point, the updates are consistent across all replicas, but it says nothing about time or sequence. This consistency model is thus weak, but scalable. Cloud storage providers often have an option for eventual consistency in some of their services. This is the case of Azure CosmosDB [51], Google Cloud Datastore [30], MongoDB or DynamoDB, which can also offer strong consistency guarantees.

2.3 Session Guarantees

In the context of the Bayou project, Terry et al. [57] defined four session guarantees for weakly consistent data. Session guarantees refer to the interaction of a single client with the storage, as the client issues successive read and write operations, possibly involving different replicas. Terry et al. [57] proposed the following guarantees: read your writes, monotonic writes (replicas must see the writes from a given client in the order it issued them), writes follow reads (a replica receiving a write operation of some value v must have the updates that the client read and used to produce that specific write of value v), and monotonic reads (this ensures that the client will not read data from a replica that is missing writes whose effects it has already seen before). Brzezinski et al. [15] showed that causal consistency requires the previous four session guarantees.

The read your writes guarantee means that a client must be able to see the effects of his or her previous writes [39, 57]. Consider two operations a and b on the same client, such that $a \rightarrow b$, being a a write operation on some object, and b a read operation on the same object, happening *after* the write. Regardless of the replica where the client reads (event b), the replica must have already received the write operation (issued in a). AWS S3 used to be eventually consistent in its early versions, while also ensuring the additional useful property that the latest version of an object would eventually spread to all replicas (last write wins). It now evolved to offer read your writes consistency [2] (known as "read-after-write").

2.4 Causal Consistency

Informally, causal consistency refers to the guarantee that causes are observed before their effects regardless of the replicas a client accesses. We can resort to the "happened-before" definition of Lamport [44], to determine whether two events have a potential causal relation, $\rightarrow$. Event a "happened-before" event b, $a \rightarrow b$, if i) there is one process, where a occurred before b; ii) a is a send event, while b is the corresponding receive event; or iii) $\exists c | a \rightarrow c \wedge c \rightarrow b$, where c is another event. Lamport logical clocks [44] capture the "happened-before" relation, in the sense that $a \rightarrow b \implies C_a < C_b$, where C_e is the logical timestamp of event e.

Unlike logical clocks, which can only ensure that $C_a \leq C_b \implies b \not\rightarrow a$, the more complex mechanism of vector clocks [9] can actually order any two events in the system (they may also be concurrent), because $a \rightarrow b \iff V_a < V_b$, where V_e is the vector timestamp of event e. The problem with vector clocks is that they occupy considerable space.

2.5 Other Consistency Models

Applications may use vector clocks to order message delivery in multicast settings. Nonetheless, some applications may require stronger consistency models, because causality provides no guarantees concerning concurrent updates. Different replicas may end up having and serving different contents without ever violating causality, as different processes, unaware of each other, may write their views of the data. To avoid this problem, systems like COPS [46], add eventual consistency to causal order, a guarantee known as Causal+. This, however, does not serve other scenarios, such as distributed state machines, where replicas need to receive reads and writes in the same order, thus requiring total order broadcast [21, 44]. Total ordered message delivery may or may not respect causality.

Other stronger consistency models, such as linearizability [33] entail a considerable performance penalty.

2.6 System Model

We aim at ensuring causal access across storage systems, including object stores, relational and non-relational databases, distributed file systems, and others. Some or all of these may have geo-replication. While some distributed storage systems may have strong internal consistency—stronger than causal—, when combined they offer no consistency.

We assume that programs deal with objects, which are an instantiation of a data structure, like an array of bytes, a dictionary, or a register in a database table. When doing a write, programs may add causal relationships to previous objects, by adding metadata references to them. They do this explicitly, rather than adding transaction boundaries or depending on some layer to keep track of all causal dependencies. This is a requirement that also exists in other causal storage systems, such as GentleRain [22]. Causally related objects thus define a Directed Acyclic Graph (DAG). We show a causal DAG for illustrative purposes in Fig. 1.

Assume that some client reads object A, r_A, before writing object B, w_B, i.e., $r_A \rightarrow w_B \implies w_A \rightarrow w_B$. If, in addition to this, the client uses data from A to create B, B causally depends on A, $A \mapsto B$. Applications must keep track of all causal relations between objects and discard other irrelevant "happened-before" ones between events. The condition we must meet is that, if $A \mapsto B$, once a client reads B it must be able to read A. This requirement is straightforward in immutable storage systems [32].

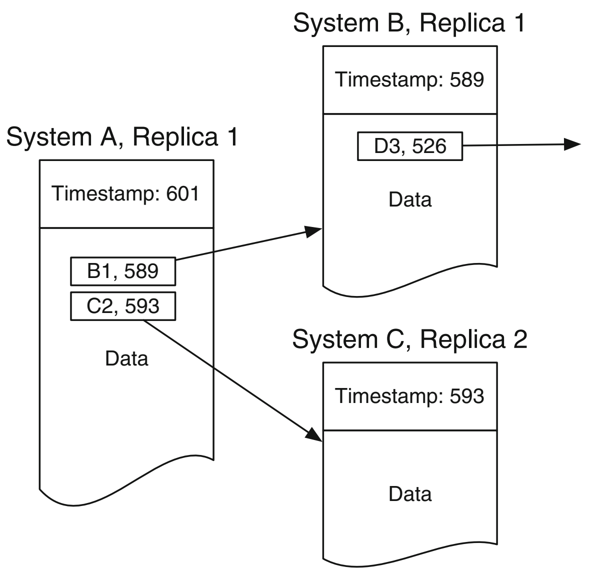

Fig. 1 Causal Directed Acyclic Graph (DAG)

Immutable file systems have been in existence for a long time, e.g., in the old VAX/VMS [48]; they still exist in newer systems, such as Azure Blob [49], Wasabi [59], or S3 versioning [2]. Immutability is often a building block for more complex systems, as in the case of Dropbox MagicPocket [17]. It is also a requirement in some legal, health, and business-critical applications.

Applications must be careful not to delete objects that are still referenced, as clients would later be unable to differentiate between a version yet to arrive and one already deleted. Also, if clients can modify objects, applications may have to enforce versioning on top of the mutable storage system, e.g., by adding versions to the object metadata. Newer versions of an object do not violate causality, as they simply signal the application that the version they were looking for was deleted.

To help programs select the region from where they should fetch objects, object references should contain their source region. We regard a cluster of servers in the same region as a single replica: in real systems, clients are usually not aware of and cannot select among the replicas that exist inside the same region.

2.7 Problem Formulation

Since accessing objects, even if they are absent, imposes a load on the server, cloud providers usually charge a fee for each read operation. The challenge is, therefore, to quickly retrieve DAG objects at a minimum cost. More formally, given a reference to an object at time $t = t_0$ ("now"), program p should minimize the time at which it receives the object, T, and the number of reading requests it issues to the provider, R (a proxy for cost). We consider two replicas: the closest replica to p plus some other replica (farther away from p) where another client first wrote the object, the source replica. The whole process consists of a write operation to the most distant replica, next p knows about the object before trying to read it. The challenge is that reading the object immediately from the closest replica might not be the optimal strategy.

Before the read operation, the client must know i) the specific object and the minimum version it needs to read; ii) the source replica for that object version; iii) a wall clock timestamp associated with object creation at the source region; iv) average round-trip times to the local and source regions; and v) an empirical Cumulative Distribution Function (CDF) of replication times from the source region to its local region.

We assume that replicas are always available. If they are not, clients may have to block for an indeterminate time, waiting for the recovery of a crashed or partitioned storage provider.

3 Optimal Causal Access

In this section, we propose a few strategies to minimize T and R. Some of these are Pareto optimal.

3.1 Distributed Setting

We consider a setting with two clients C_1 and C_2, and two servers S and R. C_1 is closer to server S (source), while C_2 is closer to server R (replica). Figures 2 and 3 show the interactions and respective times, when C_2 goes to R. Figure 2 illustrates the order of the interactions, while Fig. 3 shows their duration. Steps 2 and 2′ take place in parallel. X_S and X_R are random variables that represent, respectively, the latency of C_2's request to S, and R. C_1 writes an object to server S and, upon getting a confirmation, it immediately sends a message to C_2. This prompts C_2 to start the program p to read the object. While computing

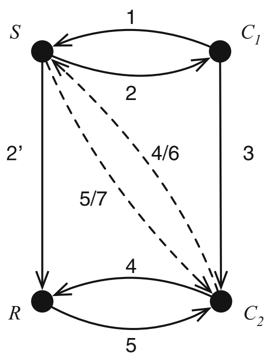

Fig. 2 Node interactions between clients and server replicas

the best strategy for C_2, we assume that the response takes the same time as the request. This assumption is common [18] and seems reasonable for small objects.

Note that according to the triangle inequality $|SR| < |SC_1| + |C_1R| < |SC_1| + |C_1C_2| + |C_2R|$, where $|WZ|$ is the Euclidean distance between W and Z. If this relationship extended to message processing and propagation times, C_2 would always find the right (or a newer) version of the object in R. This is, however, not the case. We thus evaluate some strategies that C_2 can use to fetch the object, namely the following:

1. going to S first (cheapest);
2. trying R first and then going to S if the former fails;
3. or simultaneously going to S and R (most expensive).

3.2 Source-Only Strategy

We call "Source-Only" (SO) to the simplest approach of reaching for the data source S. If C_2 sends a request for the object at time t ($t \geq t_0$), the expected time, $T(t)$, for C_2 to retrieve data is simply $E[T(t)] = t + 2E[X_S]$, while the number of requests is 1. This solution is Pareto optimal because C_2 issues the smallest possible number of reading requests.

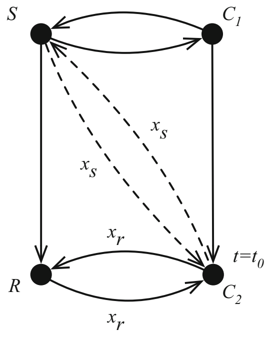

Fig. 3 Interaction times as seen from C_2

3.3 Replica-Then-Source Strategy

We now evaluate a strategy of going to R first and trying S if the read fails. We call this "Replica-then-Source" (RTS). Let random variable $H(t)$ represent the probability that C_2 finds the object at R reading at time t (a hit). Equation 1 gives the expected time, $T(t)$, at which C_2 receives the object:

$$E[T(t)] = t + 2E[X_R] + 2E[X_S](1 - H(t)) \quad (1)$$

The time to get the object is a round-trip to R or, if R still does not have a copy, which happens with probability $1 - H(t)$, the request takes an extra round-trip to S. How does this time compare to direct access to S? By setting (1) equal to $t_e + 2E[X_S]$, where t_e is the time of equilibrium, we get the result of (2):

$$H(t_e) = \frac{E[X_R]}{E[X_S]} \quad (2)$$

If $t > t_e$, p should go to R, otherwise to S. For example, if going to S takes 10 times more than going to R, C_2 should try R first, as soon as R reaches a 10% probability of having the object. For a reading time $t = t_0$, such that $H(t_0) > E[X_R]/E[X_S]$, the Replica-then-Source (RTS) Strategy is non-dominated compared to picking the source first and the replica again, should the first read fail. The former would take more time on average, while the latter would take more messages on average.

In (3) we compute the number of reading requests of this strategy:

$$R = 1 + (1 - H(t_0)) = 2 - H(t_0) \quad (3)$$

3.4 Delayed Replica-Then-Source Strategy

We can improve the RTS strategy, by postponing the request to a more favorable moment, t, when the probability of p finding the data in R is higher. This generalizes the base RTS. We call this strategy "Delayed Replica-then-Source" (DRTS).

In (4) we determine the most favorable moment to read the data, t_m, by solving the equation $dE[T(t)]/dt = 0$, with $E[T(t)]$ as defined in (1):

$$H'(t_m) = \frac{1}{2E[X_S]} \quad (4)$$

Let Z be a random variable that represents the time at which the object copy arrives at R, as seen by C_2, such that $Z = t_a$ means that a request from C_2 at $t = t_a$ arrives at R at the same time as the object copy. $H(t) = P(Z \leq t)$ is the CDF of Z and $H'(t)$ the corresponding Probability Density Function (PDF). If we consider the horizontal line $y = 1/(2E[X_S])$, (4) does not have any solution whenever $H'(t)$ is below this line, i.e., when the PDF has no peak or any peak it has is very low. This will happen whenever the arrival times of the object copy are very scattered. In this case, C_2 should not wait and should get data from either S or R immediately, depending on $H(t)$ and (2). We consider a few real cases in Section 4.

The number of read accesses now depends on t and is $R(t) = 2 - H(t)$. If (4) has a solution that corresponds to a global minimum, the Delayed Replica-then-Source (DRTS) strategy is faster than going to S first and faster than going for R at any other time. Since it uses less than 2 messages on average, this strategy is Pareto-optimal.

3.5 Parallel and Delayed Parallel Strategy

If C_2 simultaneously accesses S and R at time t, (5) gives the expected time to retrieve the object, while $R = 2$:

$$E[T(t)] = t + 2E[X_R]H(t) + 2E[X_S](1 - H(t)) \quad (5)$$

We call this strategy "Parallel" (Par.). Expected time T is strictly lower than in Source-Only (SO) or RTS, but the expected number of requests, R, is always higher. We can also delay the access to R, to improve the chances of Parallel (Par.) and thus get the "Delayed Parallel" (DP) strategy. The most straightforward approach is to go to S as soon as possible and delay the access to R to the time t_m that minimizes (1). None of our previous solutions dominates Delayed Parallel (DP), because this latter is faster.

To illustrate the relation between the different algorithms, in Fig. 4, we show a possible outcome of using the different strategies for some reading start time $t = t_0$. One should notice that this is a mere example and many different results could be possible, such as having DRTS and DP slower than RTS and Par., respectively.

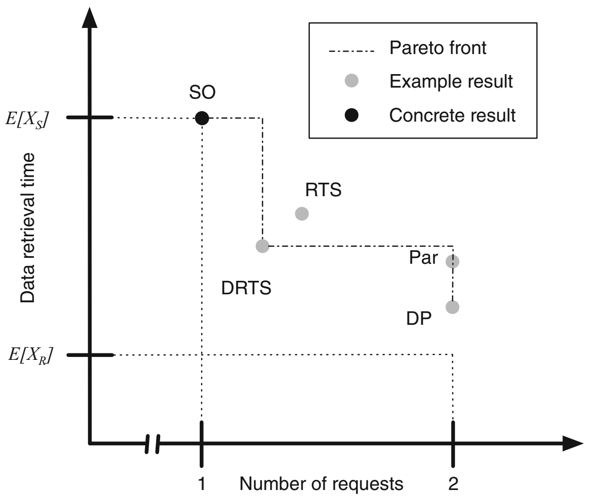

Fig. 4 Possible results for the different reading strategies

3.6 Comparison

Refer to Table 1. We assume that current time is $t = t_0 \geq 0$. Columns have the strategies, rows have the different relative locations for t_0, t_m and t_e. We only care for minima in the present or future, i.e., $t_m \geq t_0$. The "X" tells us which strategies are optimal under which circumstances. The SO strategy is always in the Pareto frontier. Regardless of the time, no other strategy can issue fewer requests. The Parallel option is also optimal for $t_m = t_0$, as no other option could be faster. Still for $t_m = t_0$, the RTS strategy is also in the Pareto frontier after the equilibrium point, i.e., if $t_0 > t_e$. If $t_m > t_0$, for $t_m > t_e$, DRTS, starting at t_m is better

Table 1 Access strategies

Relative times	SO	RTS	DRTS	Par.	DP
$t_0 = t_m < t_e$	P			P	
$t_0 < t_m < t_e$	P				X
$t_0 < t_e < t_m$	P		P		X
$t_e < t_0 = t_m$	P	P		P	
$t_e < t_0 < t_m$	P		P		X
$t_e < t_m < t_0$			–		
$t_m < t_0 < t_e$			–		
$t_m < t_e < t_0$			–		

Evaluation occurs at time $t = t_0$ ("now"). Time t_m is the minimum of (1), such that $t_m \in [t_0, \infty)$. Time t_e is the equilibrium time as in (2). "P" stands for Pareto optimal, "X" for non-dominated among the strategies in this table than RTS. DP is not dominated by the SO or any of the RTS strategies whenever $t_m > t_0$.

3.7 Practical Concerns

In our experiments of Section 4, we set $t = 0$ to the moment when C_1 sends the notification to C_2, according to its clock. This makes the computation of $H(t)$ at C_2 independent of the delay between C_1 and C_2 but forces a subtraction of timestamps from different clocks. The impact of this subtraction is small if clock drifts between C_1 and C_2 are small compared to the time scale that C_2 needs to collect information of $H(t)$. Variations in the interaction times between C_1 and S also impact $H(t)$.

If a single process, say C_1, wrote all the objects, it could set the wall clock time itself; if different objects come from different processes, having S setting the timestamp of the creation operation seems better. With this approach, $H(t)$ and $H'(t)$ become independent of the client that created the object. Moreover, R can help with the computation of $H(t)$ and $H'(t)$, by computing the CDF of replication times it gets from S, $H_{replica}(t)$. Given an object arriving at S at time t_{source} and reaching R at time $t_{replica}$, R computes the difference $t_{replica} - t_{source}$, to determine the replication time. This does not require the clocks of S and R to be synchronized. $H(t)$ can be approximated at C_2 by $H(t) \approxeq H_{replica}(t+E[X_R])$. Using $H_{replica}(t)$ and $H'_{replica}(t)$ (for different file size ranges), plus $E[X_S]$ and $E[X_R]$, C_2 can estimate t_e and t_m. To determine $E[X_S]$ and $E[X_R]$, it can keep track of past interactions with the geographical replicas of the service. This is simple to do, even for a thin client.

4 Experimental Results and Analysis

The strategies from the previous section are non-dominated and some are in the Pareto front. Nonetheless, the question of knowing which cases (rows) of Table 1 exist in practice naturally follows. To determine this, we evaluated some real storage systems.

4.1 Storage Systems Used in the Experiments

In our experiments we considered three different systems, replicated across two different geographical regions:

- AWS Simple Storage Service (S3) [2].
- AWS DynamoDB [4].
- MongoDB [8].

S3 is a low-cost cloud-based storage system. It is a key-value store, providing access to objects in exchange for a key. S3 separates objects into different buckets serving as namespaces. S3 can replicate the contents of buckets using cross-region replication. Once a client puts an object in the source region bucket, S3 will eventually put a copy in the replicated bucket. The reverse, however, is not true.

DynamoDB is a highly available key-value storage oriented toward low latency "always-on" service. One of the goals of DynamoDB is to reply to 99.9% of the requests within 300 milliseconds [20].

We also considered MongoDB due to its popularity and ability to make use of geo-replication features that allow the client to manage and select the globe regions where the data can be written or read via MongoDB Atlas, an elastic Software as a Service (SaaS) that provides easy cloud deployments of MongoDB clusters. We used a replica set of size 3 in S, with a master and two secondaries, while in R we had a read-only secondary replica. To perform the MongoDB experiments, we used the asynchronous writing mode of the Mongo native library for Python, `pymongo` [25], to ensure that C_1 would not wait in any blocking call for an acknowledgment to be received from one of the replicas. This is known as write concern *unacknowledged*.

A common feature of these three storage systems is that clients can control the region from where they read, an option that is absent from most other technologies but that we defend to be worthwhile.

4.2 Experiments

Refer to Fig. 5. In our experiments with the three technologies, we took Ireland as the source (S) and São Paulo as the replicated region (R). We ran our experiments between June and November of 2020. We used a virtual machine in Ireland to run a Python script that writes a small number of bytes (10 plus a couple of identifiers) to the repository, which is a bucket in S3, a table in DynamoDB and a collection in MongoDB. Next, the script invokes a Python AWS Lambda function in São Paulo, which reacts to this request, by trying to read that data (e.g., the same object from

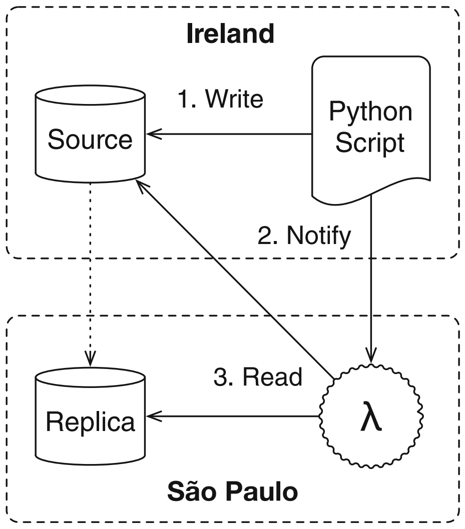

Fig. 5 Replication time measurement

the S3 replicated bucket). The AWS Lambda function may try Ireland or São Paulo or both, depending on the strategy from Section 3.

We ran two different experiments. To compute empirical CDFs ($H(t)$) and the respective PDFs, we repeated the cycle of Fig. 5 200 times in sequence always going for the replica in São Paulo. We consider the time since C_1 in Ireland starts notifying the Lambda function to the moment this function (C_2) successfully starts reading the object from São Paulo. For DynamoDB and MongoDB we found it worthwhile to run a second experiment, to test the DRTS strategy. In this case, the Lambda function sleeps for a predetermined amount of time before trying São Paulo first, followed by Ireland, should the first attempt fail. For each sleep period, we repeated the experiment 30 times.

4.3 Retrieval and Round-Trip Times

We start by showing in Table 2 statistic indicators of retrieval times per technology. We also need the average round-trip times from C_2 to the replica, $E[X_R]$, and the source, $E[X_S]$. We used reading times from the Lambda function in São Paulo to derive the values of Table 3. Differences are considerable, especially concerning S3. These differences may result from the fact that our MongoDB setting had a single replica, while the other technologies may have more replicas; S3 certainly does.

Table 2 Retrieval times per technology for the sample with size 200

Technology	Mean	Standard deviation	Max - Min
S3	25.57	1.48	5.32
DynamoDB	0.96	0.37	1.48
MongoDB	0.27	0.03	0.22

Values in seconds

4.4 S3 Results

Refer to Fig. 6, where we show the empirical CDF, $H(t)$, and respective PDF. Since S3 systematically takes between 21 and 28 seconds to replicate the object, clock skews and round-trip times have little relevance. This very large replication time was a surprise for us because in earlier versions of this experiment the copy would be ready almost always before 5 seconds. We cannot explain this behavior, as it may happen for multiple reasons internal to AWS. This delay happens even with “Replication Time Control” active. Under Replication Time Control 99.99% of the objects are replicated within 15 minutes. We had this feature switched off, as it made most of our replications slower.

Refer to Figs. 7 and 8. With S3, the decision of picking the source or the replica is straightforward. Since, in this case, (4) has no solutions, we have no best and no worst case (local minimum and local maximum) for the delay strategy. Up to the equal point, 24.92 seconds, computed according to (2), one should try SO, while after that point one should try RTS. The

Table 3 Half round-trip times from the Lambda function in São Paulo to the source and replica services (resp. Ireland and São Paulo)

Technology	$E[X_R]$	$E[X_S]$
S3	30.804	106.50
MongoDB	1.36	92.01
DynamoDB	6.48	94.64

Values in milliseconds

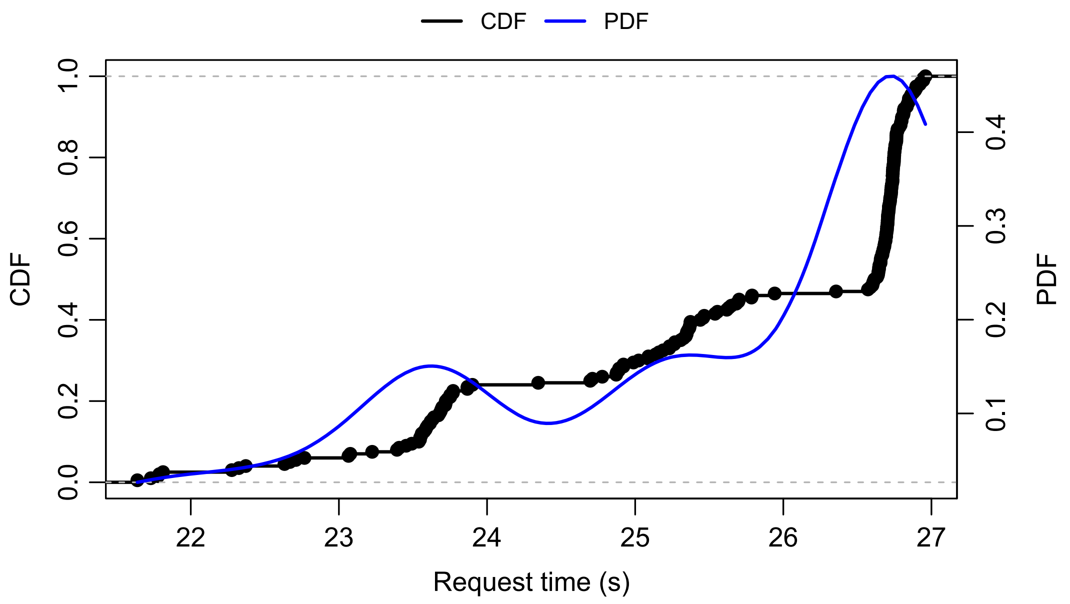

Fig. 6 AWS S3 CDF and PDF

Parallel strategy is also an option. While according to Fig. 7, the choice seems of little importance, the sharp decline in the number of reading requests of Fig. 8 suggests that C_2 should only try the replica after the second 27. We compute the number of requests in this figure using (3) and the empirical CDF.

4.5 DynamoDB Results

Figure 9 shows that replication is much faster in DynamoDB than in S3. This is expected because these technologies have complementary targets. The PDF is bimodal, but most replications occur around $t = 1$

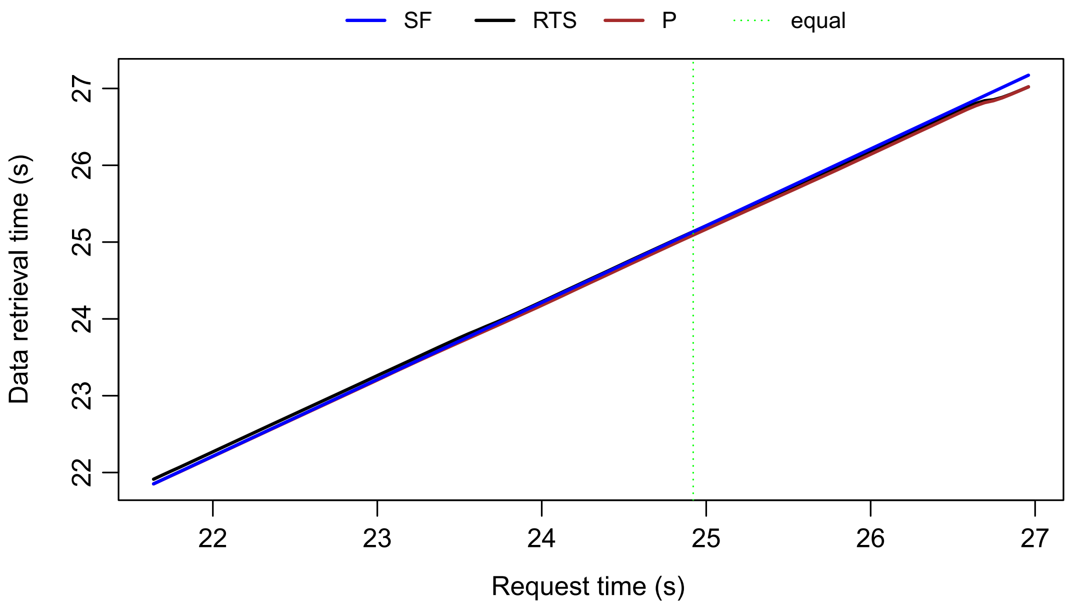

Fig. 7 AWS S3 retrieval times per algorithm. The "equal" line refers to the equilibrium time t_e of (2)

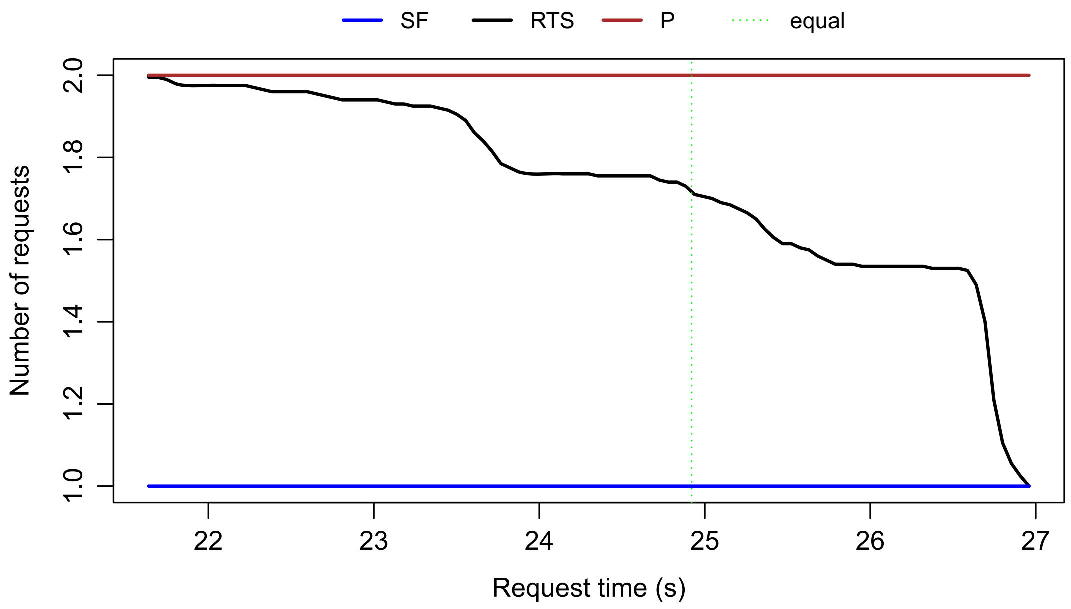

Fig. 8 AWS S3 number of retrieval requests

second. Again, (4) is impossible, which means that the delaying strategy is not worthwhile; as we show in Fig. 10, C_2 should immediately go for S if $t < 0.25$ seconds or for R after that, being Parallel a possibility as well.

In Fig. 11, we compute the foreseen number of requests. This number drops almost linearly with time, but saving in requests entails a growing cost in time. Figure 12 shows actual observations of the RTS (average of 30 measurements) and their predictable continuation. Points in the bottom-right correspond to sooner request times, while points in the top-left correspond to later request times. Once the probability of finding the object in R approaches 1, delaying the request will

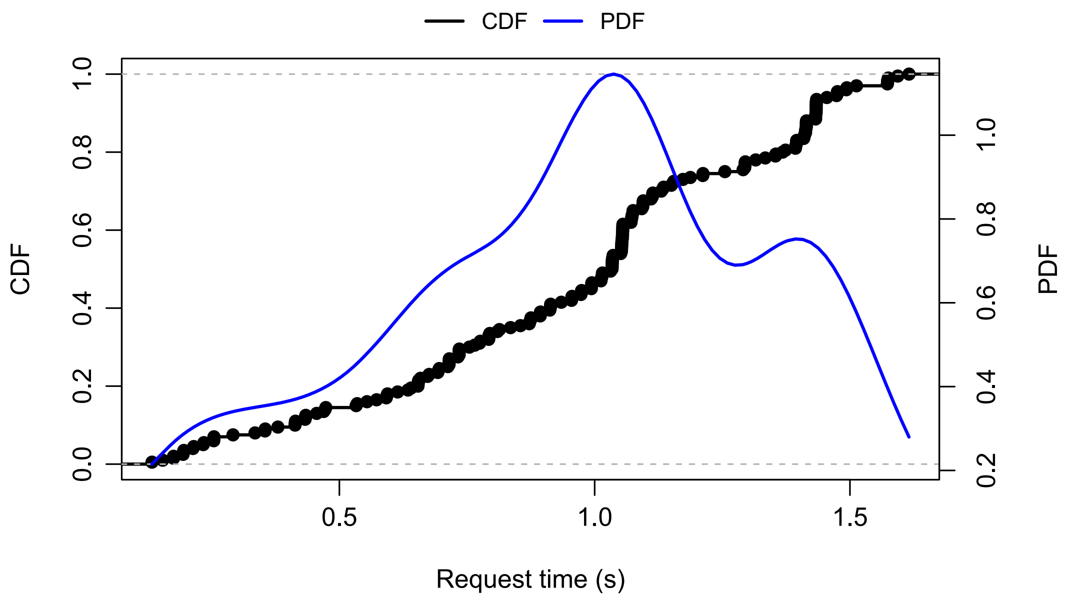

Fig. 9 DynamoDB CDF and PDF

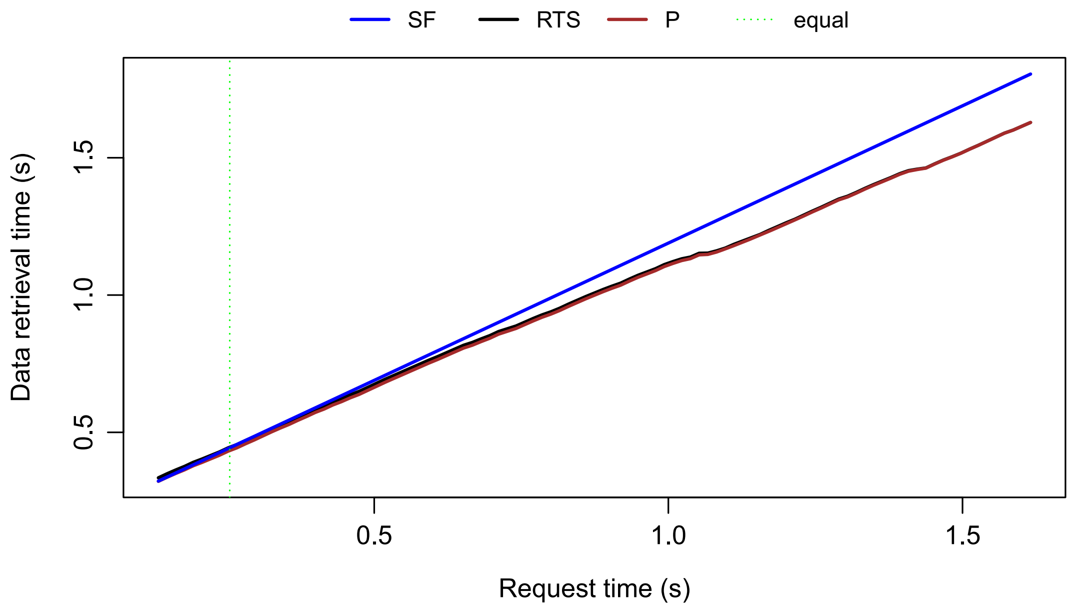

Fig. 10 DynamoDB retrieval times

only delay the retrieval without any compensation in the number of requests, thus the vertical ending of the plot.

4.6 MongoDB Results

As we illustrate in Fig. 13, the concentration of the replication times for MongoDB makes a high peak (in this case we seem to have two very close ones). Hence, unlike the cases of S3 and DynamoDB, (4) has two solutions, corresponding to the best and worst times of the DRTS strategy. The worst time is necessarily a local maximum because one can make the retrieval time as bad as one wants, by simply delaying the request time (in Fig. 14). The DP strategy is also worthwhile, especially once going straight to the

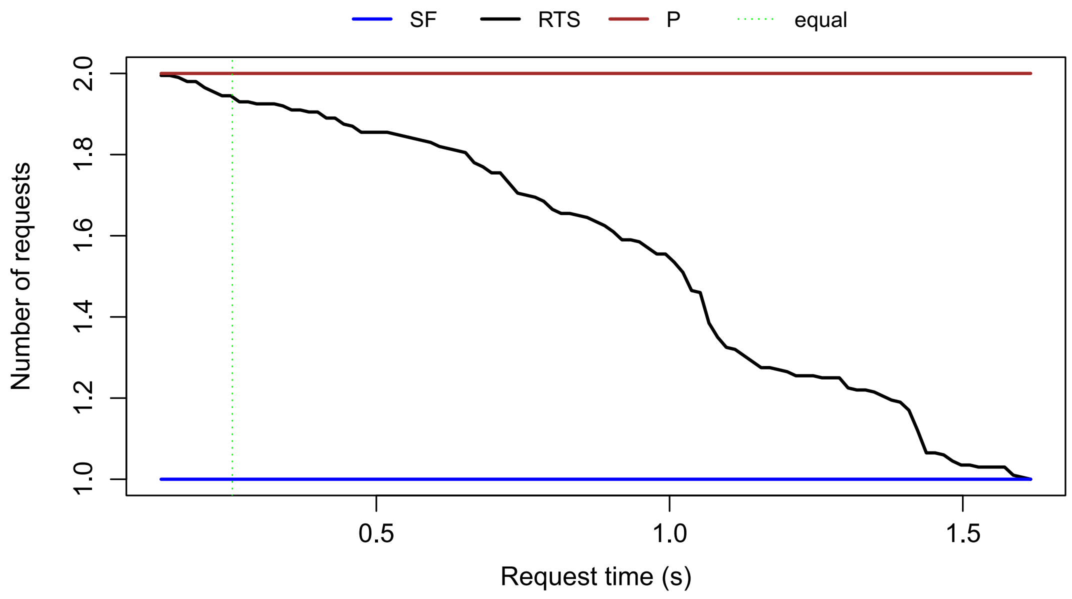

Fig. 11 DynamoDB number of retrieval requests

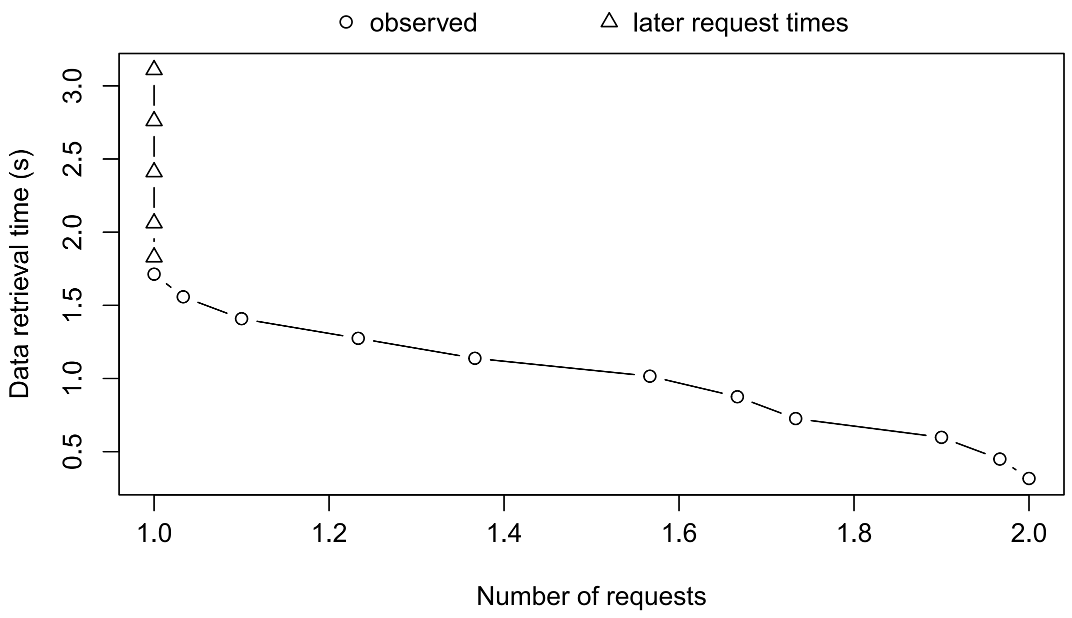

Fig. 12 DynamoDB retrieval time versus the number of requests

source ceases to be competitive and before the best time of DRTS. After that point, one must use Parallel instead of DP.

As a consequence of the CDF shape, the foreseen number of requests for DRTS, in Fig. 15, sharply decreases between the lines of the worst and best times. Interestingly, in this case, we can make the process both cheaper and faster. We can see this effect in Fig. 16, where cost and time evolve to their lowest bound almost in synchrony. In the very short range of the x-axis, data retrieval times decrease in pace with the number of reading requests to S, thus making

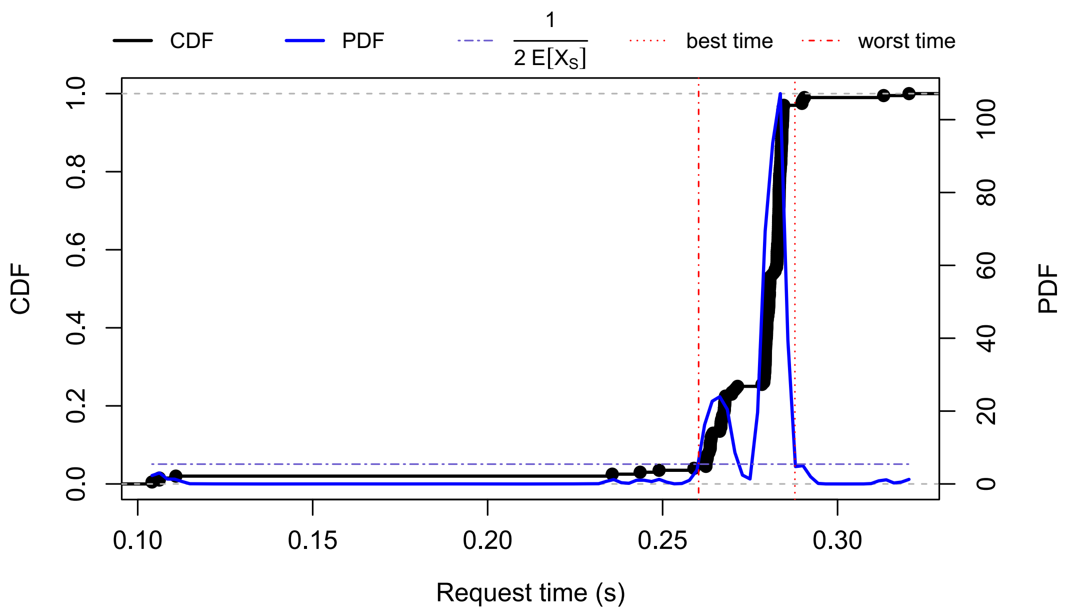

Fig. 13 MongoDB CDF and PDF

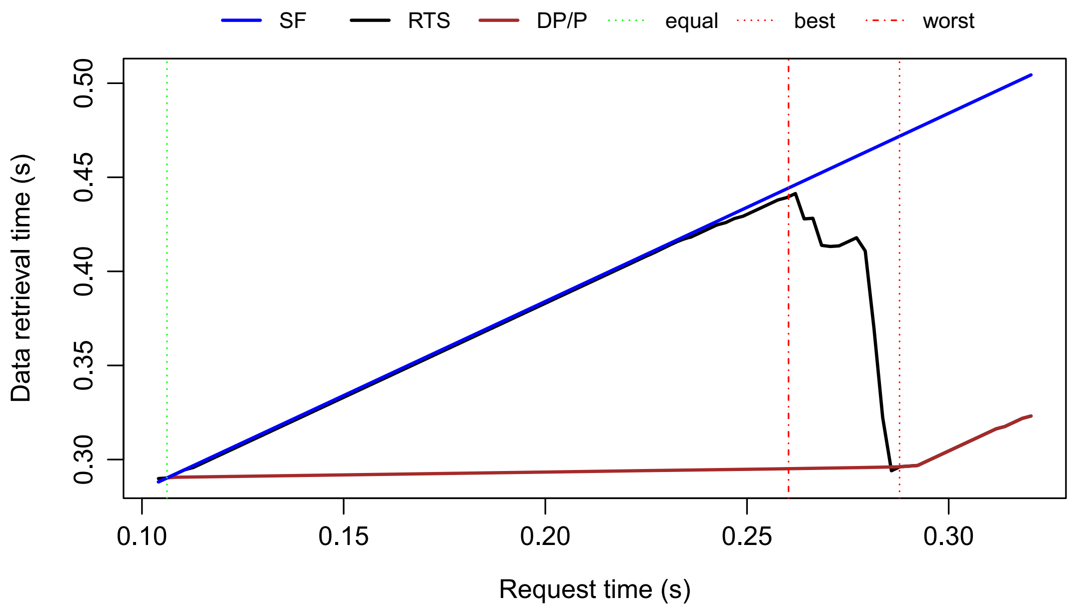

Fig. 14 MongoDB retrieval times

them seem proportional (which they are not). Finally, in Fig. 17, we show the same data but plotted in terms of data retrieval time versus the number of requests. In this case, low start times (not to be confused with the retrieval times) are in the top right corner, they grow to the bottom left and continue up from that point. The decision for system designers is unambiguous: the best point is the lowest left-most one, saving almost one retrieval request against DP.

4.7 Discussion

Table 2 shows that, despite the lack of guarantees, replication times tend to be contained within very tight

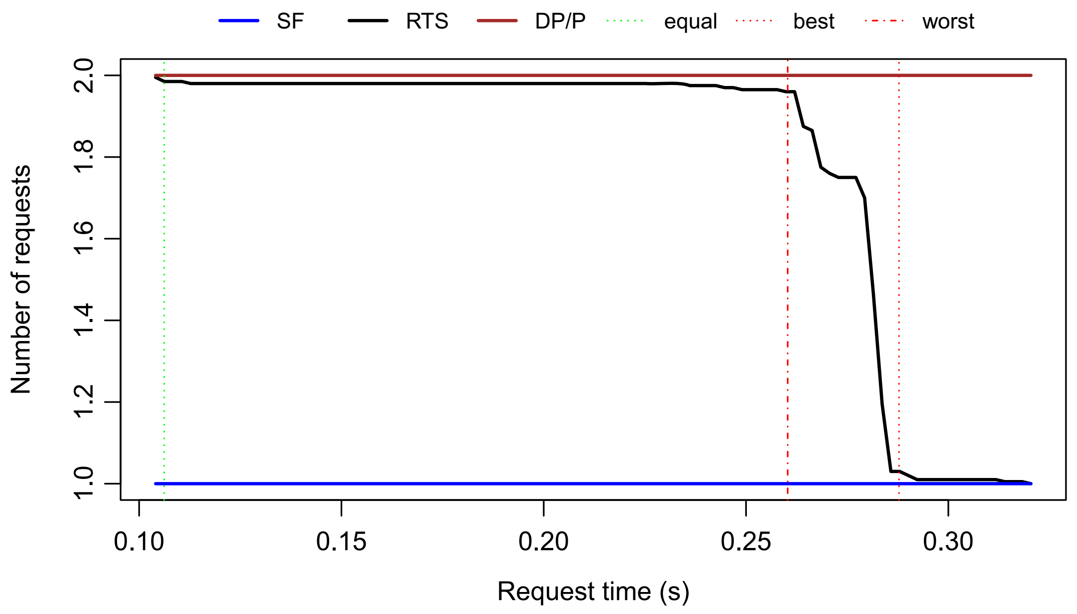

Fig. 15 MongoDB number of retrieval requests

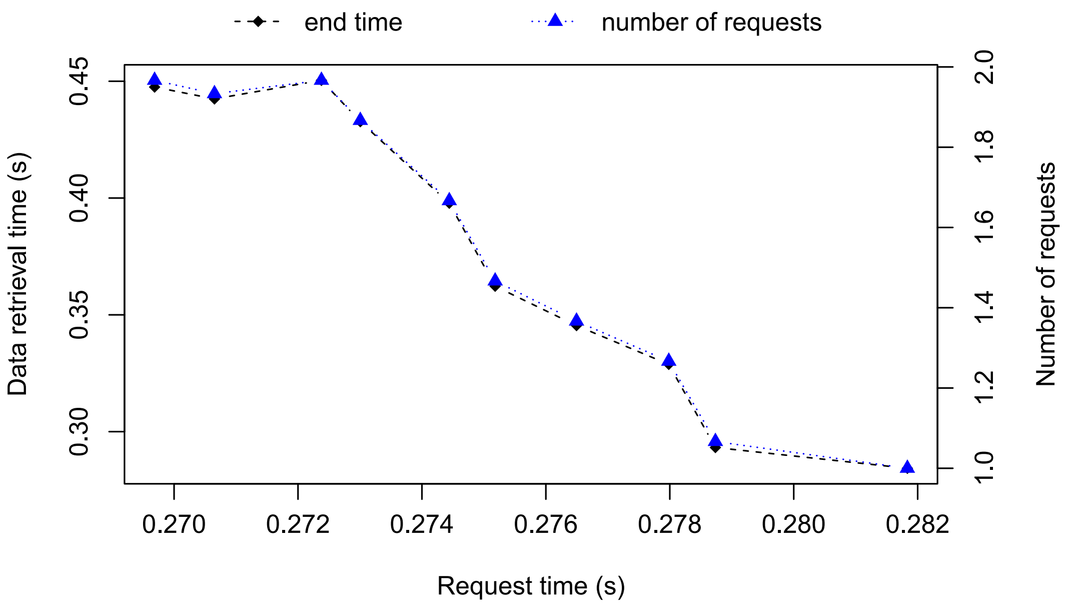

Fig. 16 MongoDB number of retrieval requests for DRTS

bounds. Even at the S3 time scale, the systems we observed tend to naturally ensure causality whenever human interaction is involved, e.g., in conversations on social media. An application that reads some data and shows it to a user, who replies or takes some action, followed by a write, will most likely take enough time to let the original content spread to all replicas. Other applications may involve automated responses, e.g., to log readings [19] or alarms. In this case, a violation of causal order is possible, especially with out-of-band communication mechanisms. Our approach aims precisely to improve this case.

We also made an experiment to understand if we could take advantage of past values of replication times. For MongoDB and DynamoDB, the autocorrelation of the list of replication times rarely goes above

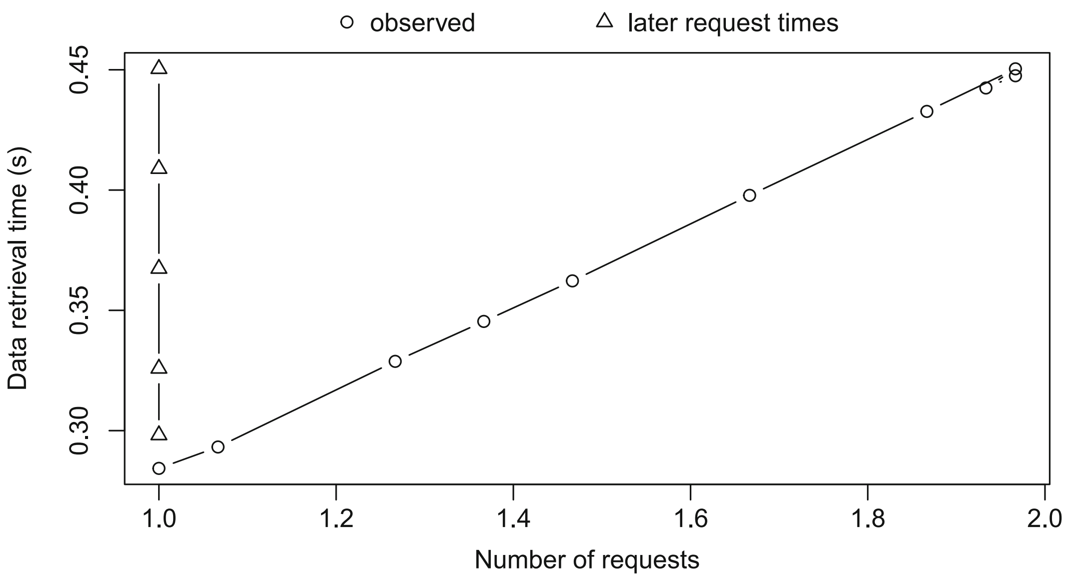

Fig. 17 MongoDB retrieval time versus number of requests for DRTS

Table 4 Access strategies

Relative times	S3	DynamoDB	MongoDB
$t_0 = t_m < t_e$	X	X	X
$t_0 < t_m < t_e$			
$t_0 < t_e < t_m$			X
$t_e < t_0 = t_m$	X	X	X
$t_e < t_0 < t_m$			X
$t_e < t_m < t_0$		–	
$t_m < t_0 < t_e$		–	
$t_m < t_e < t_0$		–	

Evaluation occurs at time $t = t_0$ ("now"). Time t_m is the minimum of (1), such that $t_m \in [t_0, \infty)$. Time t_e is the equilibrium time as in (2). "X" stands for a storage system where the condition may hold

0.2, and stays mostly around 0, thus suggesting that successive replication times are unrelated.

Finally, in Table 4, we show which cases exist in the storage systems we observed. We use the information in this table, together with Table 1, to produce Table 5, which matches algorithms to storage systems. MongoDB is the most interesting case. Assuming that C_2 receives the notification at once, it should go straight to the source using SO, or in parallel using Parallel. At some point, waiting for the local minimum near the 0.3 seconds will become worthwhile, either with DRTS or DP. After this minimum, delaying is no longer advantageous and C_2 should use RTS or Parallel.

5 Related Work

Fault tolerance in distributed systems requires replication [31]. Geo-replicated storage systems aim at ensuring available, low-latency access to data even under server crashes and network partitions. According to the CAP theorem [26], however, when there is a partition (the "P"), a data store might either be Consistent (the "C") or Available ("A"), but not both. The PACELC theorem [1, 27] extends the idea, by adding an *else* clause (the "E"), which stands for the no partition case. Under normal operation, the system may either go for Consistency ("C") or Latency ("L"). Systems tend to either be consistent all the time to the detriment of availability (under partition) or latency (under normal operation) or the other way around, sacrificing consistency in both cases (partition or no partition). As predicted in this theorem, our preference for consistency thus comes at the expense of latency and partition tolerance, as the server or one of its internal replicas may be inaccessible.

Table 5 Access strategies for each storage system

Storage system	SO	RTS	DRTS	Par.	DP
AWS S3	P	P		P	
DynamoDB	P	P		P	
MongoDB	P	P	P	P	X

"P" means that the access strategy is in the Pareto front for the storage system, and "X" means that the solution is non-dominated in this set

We can see this tradeoff at play in many systems, for example, in Azure's CosmosDB, which offers multiple storage products with five different consistency levels, ranging from eventual to strong [51]. Many cloud storage systems supporting large-scale applications, like AWS S3 [2], initially opted for scalability, offering always-on, low-latency, low-consistency services, with a promise of *eventual* consistency. S3 has now evolved to "read-after-write" consistency within a single region but still offers no guarantees if one replicates a bucket across regions. Other more expensive systems, like the DynamoDB [4, 20] or MongoDB [8] NoSQL databases have multiple consistency options, including consistent writes with short consistency timelines, even across regions. Such guarantees, however, hinder performance.

Some systems use quorums to ensure that clients access a consistent version of data. For example, in Cassandra [34], the client takes the initiative to talk to multiple replicas. In Kafka [40], the writer may be forced to wait for synchronization although asynchronous writing options also exist. In other systems, like SLOG [53], the reads need to go to the source, an approach somewhat similar to our own. These serve as examples of why consistency slows down systems, as foreseen by PACELC. The limitations of PACELC impact cloud storage systems design, like AWS S3 [2], Google Cloud [29], Microsoft Azure [50] or IBM Cloud Storage [37], which tend to drop consistency in favor of latency and availability, while also gaining scalability with their option. These systems tend to offer eventual consistency while being always on.

In academia, some authors have proposed replicated systems that offer Causal+ consistency [46]. In addition to the standard causal order, Causal+ *eventually* produces the same output even if two applications concurrently write different values to the same item at different replicas. This "eventual" property also used to exist in S3, with the last timestamp winning (note, however, that S3 was *eventually consistent*, but not *causally consistent*). In the case of COPS [46], replicas are clusters of machines that are partitioned by key (i.e., each replica is its datacenter). Each client accesses the storage system via its local replica, which takes care of replicating the writes and dependencies of the client to other replicas.

Mahajan et al. [47] demonstrate the interesting result that causal consistency, more precisely real-time causal consistency is the strongest consistency guarantee that is available in the presence of partitions. Any stronger consistency model can cause clients to block, should some server replica become unavailable. This promise, however, entails a drawback, as this theorem makes no promises regarding the novelty of the data being read. A system where replicas do not exchange data is non-blocking and causally consistent, but not particularly useful. For this reason, the notion of staleness is also important [10, 58], i.e., how old is the data that the reader is consuming. Pointers in the causal DAG rule out the staleness problem from our approach, at the expense of blocking under network partitions or source region unavailability.

Non-blocking operation at the cost of staleness is also the option in Bolt-on [7], which uses local storage (shim) and the notion of causal cut to keep causally consistent data always available to the client. The causal cut [7] essentially keeps file dependencies ready for delivery. Bolt-on does not deliver a file to the application before having all the necessary files in the cut, or otherwise, the application could block while looking for a previous dependency file. While powerful, this system is not very simple, as the shim layer must keep track of causal dependencies on behalf of the application, while keeping local replicas of files. Bolt-on delivers the promise of the Mahajan et al.'s result [47], but again, the staleness problem persists.

In [6], Bailis et al. identify a critical trade-off between staleness ("visibility latency") and write throughput. As the utilization of throughput across replicas increases, thus creating longer queues, the new data will take longer to arrive. Furthermore, the authors point out the huge potential causal histories of dependencies that impose additional delays on the replicas, as these need to wait for other write operations on different servers to arrive. Fortunately, in practice, dependencies are usually much shorter. The actual dependency of a Facebook reply is usually the single post to where it belongs. This supports the option of letting programs determine data dependencies, as we do in this work.

Ladin et al. [43] proposed one of the first replicated systems to offer causal order. Their approach is quite interesting for us, because, as in [6], they allow applications to specify the ordering they want (among "causal", "immediate" and "forced"); and their replica update scheme is lazy because the authors use gossip to optimize the replication mechanisms. While this ensures causality, when applications require so, the lazy approach causes stale reads. This scheme is thus ideally suited for our approach, as we let clients pick data from other replicas. Ladin et al. [43] also provide a quite interesting list of use cases for causal consistency, like distributed garbage collection [42], deadlock detection [24], orphan detection [45], mobile object location in a distributed system [36], and deletion of unused versions in a hybrid concurrency control scheme [60]. Indeed, ensuring causal ordering in distributed systems, namely among multicast groups dates back a few decades already, e.g., the ISIS implementation [12] or [41]. In these settings, a client will read data in causal order, as long as it keeps reading from the same replica in the group. Some authors also consider partial instead of full replication [35]. Since clients know the source region of an object, at least some of the replicas can delete a copy of the object if they need to.

Our approach also bears significant similarities to caches, if we see data present at the local replica as a cache hit. Programs will often use caches, like Amazon ElastiCache [3], Redis [16], and Memcached [38] as fast access memory for common or recent results. Nonetheless, since they can also behave as fast in-memory geo-replicated storage, the line between cache and storage is somewhat fuzzy. On the web, this is usually transparent for the browser, through Content Delivery Networks (CDNs). We propose treating local storage replicas as caches and reacting to cache misses when reading absent nodes in a causal DAG.

We also observe that our problem bears some similarities to Global Server Load Balancing (GSLB) products, such as Route 53 [55]. In these products, the service provider has multiple regional replicas and uses the name service to point clients to the best replica, using one of many possible policies, including geographical location, latency, and others. Similarly to our case, these products need to compute the best replica to access.

6 Conclusion

In this paper, we considered the problem of causally accessing objects on multiple geo-replicated storage systems. Since cloud providers usually charge for each read access, we aim at minimizing the number of client requests, while also minimizing latency.

We propose five non-dominated access strategies, of which four are Pareto optimal. Our access strategies proved to be worth using in real storage systems. To make this possible, clients must know metrics like the age of the object they are looking for and the CDFs of object arrival times at their local replica, for each source. To take this burden off clients, local replicas can take care of the most difficult figure to collect, the CDFs.

Judicious utilization of access strategy and time can, in some real cases, minimize latency, while cutting costs by nearly 50%. This is a considerable improvement for all software interacting with replicated data, like JavaScript on a web page accessing geo-replicated storage, or to Internet of Things (IoT) devices interacting with edge servers.

One of the conclusions we take from this work is that cooperation from service providers could have a positive impact on latency and costs if they let clients pick the replica to access while providing blocking read operations. They would be a good option for clients if they cost less than the average cost of RTS or DRTS and take less time than SO. Enriching cloud storage services with a blocking read primitive would thus provide value for the client and the provider.

A natural continuation for this work would be to consider more than a single pair of replicas and extend this analysis to an arbitrary set of geo-replicated servers.

Acknowledgements The authors would like to thank the Centre for Informatics and Systems of the University of Coimbra (CISUC) for providing the conditions for this work.

Author Contributions Stanley Lima and Filipe Araujo started the work on the paper and designed the first experiments. Filipe Araujo later reformulated the paper with the help of Miguel Guerreiro. While both have contributed to the design and execution of the experiments, Andre Bento and Jaime Correia participated in discussions regarding the objectives and the evaluation of results. Stanley Lima, Filipe Araujo, and Miguel Guerreiro wrote most of the paper. Raul Barbosa was involved in the review process. All authors approved the final manuscript.

Funding Open access funding provided by FCT—FCCN (b-on). This work is funded by national funds through the FCT – Foundation for Science and Technology, I.P., within the scope of the project CISUC – UID/CEC/00326/2020, by the European Social Fund, through the Regional Operational Program Centro 2020 and by research grants of the programs: Science Without Borders (Ciências sem Fronteiras - CsF), Brazilian Space Agency (Agência Espacial Brasileira - AEB).

Data Availability The datasets generated during the current study are available from the corresponding author upon reasonable request.

Compliance with Ethical Standards

Consent for Publication All authors gave their consent for this publication.

Competing interests The authors are not aware of any competing interests regarding this work.

References

1. Abadi, D.: Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. IEEE Comput. **45**(2), 37–42 (2012). https://doi.org/10.1109/MC.2012.33
2. Amazon: Cloud Object Storage - Amazon S3 - Amazon Web Services. https://aws.amazon.com/s3/?nc1=h_ls. Accessed 21 May 2022 (2006)

3. Amazon Web Services, Inc. or affiliates. Amazon ElastiCache- In-memory data store and cache. https://aws.amazon.com/elasticache/. Accessed 13 July 2022 (2011)
4. Amazon Web Services, Inc. or affiliates, Amazon DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html. Accessed 13 July 2022 (2012)
5. Attiya, H., Welch, J.L.: Sequential consistency versus linearizability. ACM Trans. Comput. Syst. (TOCS) **12**(2), 91–122 (1994)
6. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: The potential dangers of causal consistency and an explicit solution. In: Proceedings of the 3rd ACM Symposium on Cloud Computing, SoCC '12. Association for Computing Machinery, New York (2012). ISBN 9781450317610. https://doi.org/10.1145/2391229.2391251
7. Bailis, P., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Bolt-on causal consistency. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, pp. 761–772. Association for Computing Machinery, New York (2013). ISBN 9781450320375. https://doi.org/10.1145/2463676.2465279
8. Banker, K., Garrett, D., Bakkum, P., Verch, S.: MongoDB in action: covers MongoDB version 3.0. Manning, Shelter Island, NY. ISBN 978-1617291609 (2016)
9. Barbara, L., Ladin, R.: Highly-available distributed services fault-tolerant distributed garbage collection. In: Proceedings of the 5th Symposium on the Principles of Distributed Computing, pp. 29–39. ACM, Canada (1986)
10. Bermbach, D., Kuhlenkamp, J.: Consistency in distributed storage systems. In: Gramoli, V., Guerraoui, R. (eds.) Networked Systems, pp. 175–189. Springer, Berlin (2013). ISBN 978-3-642-40148-0
11. Bermbach, D., Tai, S.: Eventual consistency: How soon is eventual? An evaluation of amazon S3's consistency behavior. In: Proceedings of the 6th Workshop on Middleware for Service Oriented Computing, MW4SOC 2011, Lisbon, Portugal, December 12-16, 2011, p. 1 (2011). https://doi.org/10.1145/2093185.2093186
12. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Trans. Comput. Syst. **5**(1), 47–76 (1987). ISSN 0734-2071. https://doi.org/10.1145/7351.7478
13. Bravo, M., Rodrigues, L.E.T., Van Roy, P.: Saturn: a distributed metadata service for causal consistency. In: Proceedings of the 12th European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017, pp. 111–126 (2017). https://doi.org/10.1145/3064176.3064210
14. Brzezinski, J., Sobaniec, C., Wawrzyniak, D.: Session guarantees to achieve PRAM consistency of replicated shared objects. In: International Conference on Parallel Processing and Applied Mathematics, pp. 1–8. Springer (2003)
15. Brzezinski, J., Sobaniec, C., Wawrzyniak, D.: From session causality to causal consistency. In: 12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2004), 11–13 February 2004, A Coruna, Spain, pp. 152–158 (2004). https://doi.org/10.1109/EMPDP.2004.1271440
16. Carlson, J.: Redis in action. Manning, Shelter Island, NY. ISBN 978-1617290855 (2013)
17. Cowling, J.: Inside the MagicPocket - Dropbox. https://dropbox.tech/infrastructure/inside-the-magic-pocket. Accessed 14 July 2022 (2016)
18. Cristian, F.: Probabilistic clock synchronization. Distrib. Comput. **3**(3), 146–158 (1989). ISSN 0178-2770. https://doi.org/10.1007/BF01784024
19. Weeks, D.C.: S3mper: Consistency in the cloud. https://netflixtechblog.com/s3mper-consistency-in-the-cloud-b6a1076aa4f8, Accessed 13 Aug 2022 (2014)
20. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. ACM SIGOPS Oper. Syst. Rev. **41**(6), 205–220 (2007)
21. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. **36**(4), 372–421 (2004). ISSN 0360-0300. https://doi.org/10.1145/1041680.1041682
22. Du, J., Iorgulescu, C., Roy, A., Zwaenepoel, W.: Gentlerain: Cheap and scalable causal consistency with physical clocks. In: Proceedings of the ACM Symposium on Cloud Computing, SOCC '14, pp. 1–13. Association for Computing Machinery, New York (2014). ISBN 9781450332521. https://doi.org/10.1145/2670979.2670983
23. Duan, Y., Koufaty, D., Torrellas, J.: SCsafe: Logging sequential consistency violations continuously and precisely. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 249–260. IEEE (2016)
24. Farrell, A.K.: A deadlock detection scheme for Argus. Bachelor's Thesis July 1989 MIT Dept. of Electrical Engineering and Computer Science
25. Python Software Foundation: pymongo · pypi. https://pypi.org/project/pymongo/. Accessed 14 July 2022 (2009)
26. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002). ISSN 0163-5700. https://doi.org/10.1145/564585.564601
27. Golab, W.: Proving PACELC. SIGACT News **49**(1), 73–81 (2018). ISSN 0163-5700. https://doi.org/10.1145/3197406.3197420
28. Golab, W., Li, X., Shah, M.A.: Analyzing consistency properties for fun and profit. In: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 197–206 (2011)
29. Google: Cloud Computing Services — Google Cloud. https://cloud.google.com/. Accessed 14 July 2022 (2008)
30. Google: Datastore – Google Cloud. https://cloud.google.com/datastore/. Accessed 14 July 2022 (2013)
31. Guerraoui, R., Schiper, A.: Fault-tolerance by replication in distributed systems. In: International Conference on Reliable Software Technologies, pp. 38–57. Springer (1996)
32. Hasan, R., Tucek, J., Stanton, P., Yurcik, W., Brumbaugh, L., Rosendale, J., Boonstra, R.: The techniques and challenges of immutable storage with applications in multimedia. In: Lienhart, R.W., Babaguchi, N., Chang, E.Y. (eds.) Storage and Retrieval Methods and Applications for Multimedia 2005, vol. 5682, pp. 41–52. International Society for Optics and Photonics, SPIE, Bellingham (2005). https://doi.org/10.1117/12.588103

33. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). ISSN 0164-0925. https://doi.org/10.1145/78969.78972
34. Hewitt, E.: Cassandra: The Definitive Guide, (Revised) 3rd edn.: Distributed Data at Web Scale. O'Reilly Media, Inc., Sebastopol (2022). ISBN 978-1492097143
35. Hsu, T.-Y., Kshemkalyani, A.D., Shen, M.: Causal consistency algorithms for partially replicated and fully replicated systems. Future Gener. Comput. Syst. **86**, 1118–1133 (2018). ISSN 0167-739X. https://doi.org/10.1016/j.future.2017.04.044. https://www.sciencedirect.com/science/article/pii/S0167739X17308166
36. Hwang, D.J.-H.: Constructing a highly-available location service for a distributed environment. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, Cambridge, Massachusetts, USA (1987)
37. IBM: IBM Cloud Storage. https://www.ibm.com/cloud/storage. Accessed 14 July 2022 (2013)
38. Jose, J., Subramoni, H., Luo, M., Zhang, M., Huang, J., Wasi-ur Rahman, M., Islam, N.S., Ouyang, X., Wang, H., Sur, S., Panda, D.K.: Memcached design on high performance RDMA capable interconnects. In: 2011 International Conference on Parallel Processing, pp. 743–752 (2011). https://doi.org/10.1109/ICPP.2011.37
39. Kermarrec, A.-M., Kuz, I., van Steen, M., Tanenbaum, A.S.: A framework for consistent, replicated Web objects. In: The 18th International Conference on Distributed Computing Systems, pp. 276–291. IEEE Computer Society, Amsterdam (1998). https://doi.org/10.1109/ICDCS.1998.679725
40. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB, vol. 11, pp. 1–7 (2011)
41. Kshemkalyani, A.D., Singhal, M.: Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. Distrib. Comput. **11**(2), 91–111 (1998). ISSN 0178-2770. https://doi.org/10.1007/s004460050044
42. Ladin, R.: A method for constructing highly available services and a technique for distributed garbage collection. PhD thesis, MIT Dept. of Electrical Engineering and Computer Science (1989)
43. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. Comput. Syst. **10**(4), 360–391 (1992). https://doi.org/10.1145/138873.138877
44. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
45. Liskov, B., Scheifler, R., Walker, E., Weihl, W.: Orphan detection (extended abstract). In: Proceedings of the 17th International Symposium on Fault-Tolerant Computing, Pittsburgh (1987)
46. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal October 23–26 2011, pp. 401–416 (2011). https://doi.org/10.1145/2043556.2043593
47. Prince, M., Lorenzo, A., Dahlin, M.: Consistency, availability, and convergenc. Technical Report UTCS TR-11-22, Department of Computer Science, The University of Texas at Austin (2011)
48. McCoy, K.: VMS File System Internals (VAX - VMS Series). Digital Press, Maynard, Massachusetts USA. ISBN 1555580564 (1990)
49. Microsoft: Azure Blob Storage documentation. https://docs.microsoft.com/en-us/azure/storage/blobs/. Accessed 14 july 2022 (2008)
50. Microsoft: Microsoft Azure. https://azure.microsoft.com/services/storage. Accessed 14 July 2022 (2008)
51. Microsoft: Consistency levels in Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels. Accessed 14 July 2022 (2022)
52. Evaggelia, P., Bharat, B.: Maintaining consistency of data in mobile distributed environments. In: Proceedings of 15th International Conference on Distributed Computing Systems, pp. 404–413. IEEE (1995)
53. Ren, K., Li, D., Abadi, D.J.: Slog: serializable, low-latency, geo-replicated transactions. Proc. VLDB Endowment **12**(11), 1747–1761 (2019)
54. Roohitavaf, M., Demirbas, M., Kulkarni, S.: Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks. In: 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS), pp. 184–193 (2017). https://doi.org/10.1109/SRDS.2017.27
55. Soni, M.: Practical AWS Networking: Build and manage complex networks using services such as Amazon VPC, Elastic Load Balancing, Direct Connect, and Amazon Route 53. ISBN 978-1788398299 (2018)
56. Tanenbaum, A.S., van Steen, M. Distributed systems - principles and paradigms, 2nd edn. Pearson Education, Upper Saddle River (2007). ISBN 978-0-13-239227-3
57. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M., Theimer, M., Welch, B.B.: Session guarantees for weakly consistent replicated data. In: Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28–30, 1994, pp. 140–149 (1994). https://doi.org/10.1109/PDIS.1994.331722
58. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A.: Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective. In: 5th Biennial Conference on Innovative Data Systems Research (CIDR 2011) (2011)
59. Wasabi Technologies Inc. Cloud Object Storage by Wasabi — 1/5th the Price — Wasabi. https://wasabi.com. Accessed 14 July 2022 (2017)
60. Weihl, W.E.: Distributed version management for read-only actions. IEEE Trans. Softw. Eng. **E-13**(1), 55–64 (1987)