On the use of Clocks to Enforce Consistency in the Cloud

Manuel Bravo[*†], Nuno Diegues[*], Jingna Zeng[*‡], Paolo Romano[*], Luís Rodrigues[*]

[*]Instituto Superior Técnico, Universidade de Lisboa, Portugal
[†]Université Catholic de Louvain, Belgium
[‡]KTH/Royal Institute of Technology, Sweden

Abstract

It is well known that the ability to timestamp events, to keep track of the order in which they occur, or to make sure they are processed in a way that is meaningful to the application, is of paramount importance in a distributed system. The seminal work by Leslie Lamport[30] has laid the foundations to understand the trade-offs associated with the use of physically and logically synchronized clocks for establishing the order of events. Lamport's original work considered scalar clocks and has later been extended to consider richer forms of logical clocks such as vector clocks and matrix clocks. The purpose of this paper is to revisit these concepts in the concrete setting of developing distributed data store systems for the cloud. We look at how different clock mechanisms have been applied in practice to build different cloud data stores, with different consistency/performance trade-offs. Our goal is to gain a better understating of what are the concrete consequences of using different techniques to capture the passage of time, and timestamp events, and how these choices impact not only the performance of cloud data stores but also the consistency guarantees that these systems are able to provide. In this process we make a number of observations that may offer opportunities to enhance the next generations of cloud data stores.

1 Introduction

With the maturing of cloud computing, cloud data stores have become an indispensable building block of existing cloud services. The ideal cloud data store would offer low latency to clients operations (effectively hiding distribution), consistency (hiding concurrency), and high availability (masking node failures and network partitions). Unfortunately, it is today well understood that such an idealized data store is impossible to implement; a fact that has been captured by the following observation: in a distributed system subject to faults it is impossible to ensure simultaneously, and at all times, Consistency, Availability, and Partition tolerance, a fact that became known as the CAP theorem[22, 9]. In face of this impossibility, several different cloud data stores have been designed, targeting different regions of the design space in an attempt to find the sweetest spot among consistency, availability, and performance.

On one extreme, some systems have favoured concurrency and availability over consistency. For instance, Dynamo[12] allows for concurrent, possibly conflicting, operations to be executed in parallel on different nodes.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

It then provides the tools to help the application developers to eventually merge in a coherent manner the updates performed independently, which is known as *eventual consistency*. Practice has shown that reconciling conflicting updates can be extremely hard, and that it poses a heavy burden on programmers, favouring ad hoc solutions that are hard to compose and maintain[6]. For this reason, some people claim that eventually consistency is, in practice, no consistency[33].

On the other extreme, there are also examples of systems that favour strong consistency, at the cost of lower availability and performance[11, 38]. Although, as stated by the CAP theorem, strong consistency prevents unconditioned availability, in practice, if enough replicas are used, such that it is unlikely that a majority becomes failed or disconnected, the availability concerns can be mitigated. On the other hand, the coordination required to ensure the consistent processing of concurrently submitted operations introduces delays that are hard to overcome and that become visible to the user experience.

Between these two extremes of the spectrum, other systems have proposed to use weaker forms of consistency, such as causal consistency, to allow for a reasonable degree of concurrency while still providing clear and meaningful semantics to the application[33]. In fact, causal consistency is often regarded as the strongest consistency level that a system can provide without compromising availability and incurring high latencies [34].

Not surprisingly, regardless of the type of consistency guarantees that all these systems provide, they still need to track the relative order of different read and write requests in order to ensure that the state of the data store remains consistent with the user intents and that users observe updates in an expectable order. Key to correctly tracking such ordering relations is to use some form of clock in order to timestamp events of interest. These clocks can be either logical (i.e., just based on the number of events observed by the nodes) or physical (i.e, synchronized with some external source of universal time). In a seminal paper[30], Leslie Lamport has laid the foundations to understand the trade-offs among the use of physically synchronized clocks and the use of logically synchronized clocks for establishing the order of events. Lamport's original work considered scalar clocks and has later been extended to consider richer forms of logical clocks such as vector clocks[21, 35] and matrix clocks[42, 49]. As their name implies, vector and matrix clocks do not use a single scalar value; instead, they use multiple scalar values to capture more precisely the causal dependencies among events. Therefore, vector and matrix clocks require to maintain, and exchange, more metadata. A non-informed outsider may assume that systems providing stronger consistency are required to maintain more sophisticated forms of clocks and that systems providing weak consistency can be implemented with cheaper scalar clocks. Interestingly, this is not the case and the trade-offs involved are subtler than may appear at first sight.

In this paper, we compare recent cloud data store implementations, focusing on this crucial aspect that is at the core of their design, i.e., the choice of the mechanisms used to capture the passage of time and track cause-effect relations among data manipulations. In particular, we analyze a set of recent cloud data store systems supporting both strong and weak consistency, and which use a variety of approaches based on physical clocks (either loosely [17] or tightly synchronized [5]), various forms of logical clocks (scalar [40], vector [38] and matrix [18]), as well as hybrid solutions combining physical and logical clocks [1]. Our study encompasses a variety of cloud data stores, ranging from academic solutions, such as COPS [33] and ClockSI [17], to industry systems, such as Spanner [5] and Cockroach [1]. We provide a comparative analysis of alternative clock implementations, and identify a set of trade-offs that emerge when using them to enforce different consistency criteria in large-scale cloud data stores. Our aim is to derive insights on the implications associated with the design choices of existing solutions that may help cloud service builders to choose which types of stores are more suitable for their customers and services.

The remainder of this paper is structured as follows. We start by providing a generic reference architecture of a cloud service, that we will use to illustrate some of our points. Then, we review some fundamental notions and techniques for tracking time and establishing the order of events in a distributed system. These techniques represent crucial building blocks of the mechanisms that are used to enforce consistency in existing cloud data stores. In the following two sections, we focus on analyzing the design of recent distributed data stores providing weak (i.e., some form of causality) and strong (i.e., general purpose transactions) consistency. Subsequently, we

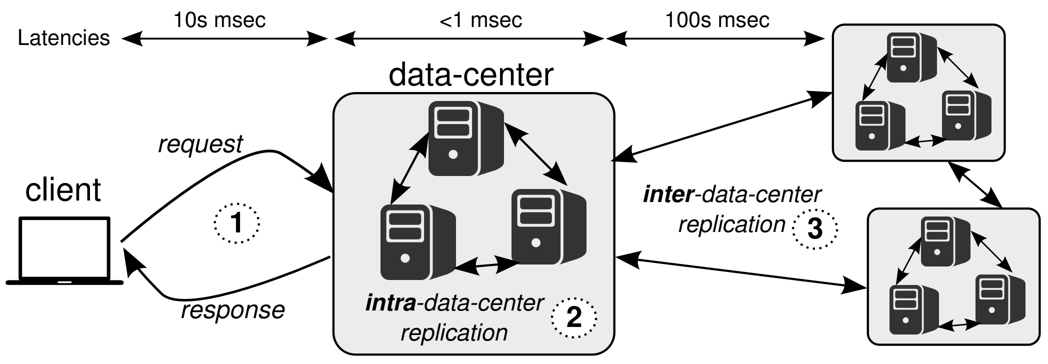

Figure 1: Generic Cloud Data Store Architecture.

draw a number of observations regarding the use of clocks for both weakly and strongly consistent systems. The last section offers some concluding remarks.

2 Architecture of a Cloud Data Store

We start by presenting a high level architecture of a cloud data store service, which we shall use as a generic framework for our subsequent discussion. An illustration of the architecture is presented in Figure 1. Typically, for fault-tolerance and locality, a data item is replicated in multiple nodes of the cloud data store. It is possible to replicate data just in a single data center, but this makes the service vulnerable to unavailability and data loss due to power outages, network problems, or natural catastrophes. Therefore, data is usually replicated in multiple, geographically replicated, data centers, which can also potentially reduce latencies.

In this setting, a client request is redirected to some data center, where it is served by some proxy node that abstracts the entire set of replicas. The role of the proxy is to initiate the required coordination to ensure that enough local and remote replicas are involved to provide the desired quality of service to the client. In theory, it is possible to implement a flat coordination scheme that treats all replicas alike, regardless of their location; for instance, one could use an algorithm that would apply an update by storing the data in a majority of all replicas (including replicas at local and remote data centers). However, the delays involved in contacting replicas within the same data center are several orders of magnitude smaller than the delays involved in the communication among different data centers. As a result, many geo-replicated data stores use some form of hierarchical coordination mechanism, where the algorithms used to maintain intra-data center replica consistency are different from the algorithms used to maintain inter-data center replica consistency.

Needless to say, it is cheaper to enforce strong consistency within a single data center than across multiple data centers. Therefore, a common pattern in many implementations (but not all) is that, when a client contacts a given data center, some local protocol is immediately executed to ensure strong consistency among the replicas within that data center. This means that, as long as the client keeps contacting the same data center, it can observe consistent results without incurring significant delays. On the other hand, synchronizations among data centers is often performed asynchronously. This means that updates performed in one data center may be observed with noticeable staleness in remote data centers. This also means that if a client is redirected to another data center it may either observe inconsistent results or be forced to wait for pending updates to be propagated. In the later case, clocks play a fundamental role to understand when consistency can be ensured.

Finally, different clients may try to update shared data concurrently at different data centers. To coordinate such updates is hard. In the general case, to avoid conflicting updates, coordination among data servers must be performed synchronously, i.e., finalized before replying to the client. However, as previously stated, many systems avoid this approach due to the potentially long delays involved in the process. This choice is supported by studies that have demonstrated the sensitiveness of users to even very small latency increases [44, 26, 32]. The consequence is that, in many systems, conflicting concurrent updates are accepted, and the resulting conflicts need to be resolved in background. In any case, either to perform eager coordination (for instance to run consensus among the data centers[37, 31, 5]), or just to order conflicting updates in a meaningful way, clocks play a role of paramount importance.

In this paper, we discuss the role of clocks in both the intra and inter data center coordination and also for both strong and weak consistency protocols.

3 Tracking the Ordering of Events in Distributed Systems

For self-containment, we provide a brief summary of different techniques that have been proposed to keep track of the ordering of events in distributed systems.

We assume that the nodes of the system are fully connected and that each node keeps one or more values to track the passage of time and to assign timestamps to events; these timestamps can be later used to order the corresponding events. We call this set of values a *clock*. If the clock maintains a single value we say that the system uses a *scalar clock*. If the clock maintains one value per node in the system we say that the system uses a *vector clock*[21, 35]. Finally, a clock that maintains a value for each link between pairs of nodes in the system is denoted a *matrix clock*[42, 49].

Orthogonal to the number of values in the clock is whether the values are supposed to capture the real time instant at which the events occurred or if the values are just logical values that count how many relevant events have been observed. In the first case we say the system uses *physical* values and in the second case *logical* values. More recently, some works have proposed to consider values that are actually a tuple, containing both a physical clock and a logical clock[28]. In this later case, we say that the system uses *hybrid* values[27].

When the two dimensions above are combined, we obtain data structures that have been used in different systems. For instance, if a system uses a scalar clock with physical values we say that it uses (scalar) physical clocks. Conversely, if it uses a scalar clock with logical values we say that it uses (scalar) logical clocks, also known as Lamport's clocks [30]. Finally, a system that uses a scalar clock with hybrid values is said to use a (scalar) hybrid clock. Similar combinations can be obtained for vector and matrix clocks.

In abstract, clocks based on physical values can be considered as potentially superior to logical clocks in the sense that, in addition to tracking potential cause-effect relations, they can allow to correlate the timestamp of events with the actual (real-time) instant at which a given event occurred. Unfortunately, in practice, physical clocks are subject to unavoidable drifts and are, as such, never perfectly synchronized. One way to address this issue is to rely on distributed clock synchronization algorithms, such as the Network Time Protocol (NTP) [36]: nonetheless, the quality of the synchronization achievable via these algorithms may vary significantly (e.g., depending on the network load), creating uncertainty intervals that can vary from only a few to hundreds of milliseconds. An alternative to address the synchronization problems is to rely on specialized hardware devices (such as GPS and atomic clocks)[5]: these mechanisms do allow for ensuring small and bounded uncertainty intervals (typically less than 10 msec), but these devices have a non-negligible cost and are not commonly available in public cloud infrastructures.

It is also obvious that scalar clocks are more space efficient than vector or matrix clocks. For instance, a system that uses vector clocks has to maintain metadata whose size is linear with the number of entities (namely, nodes across all the data centers) while a system that maintains scalar clocks maintains metadata whose size is constant with the number of entities. On the other hand, with a scalar clock it is, in general, impossible to assess if two events are concurrent or causally related. For instance consider events e and e' with timespamps t and t' respectively, where $\mathit{value}(t) < \mathit{value}(t')$. With scalar clocks we know that e' can causally depend on e or that e and e' are concurrent (but that, for sure, e does not causally depend on e'). With a vector (and hence a matrix) clock it is always possible to be sure if e and e' are causally dependent or concurrent.

4 Clocks for Weak Consistency

We now discuss the use of clocks in systems that trade consistency for availability; we mainly focus on systems that provide causal consistency. The common features of these systems are: (i) no support for general purpose update transactions (i.e., update operations address a single object) and sometimes the availability of read-only transactions (i.e., transaction that read multiple objects observe a consistent snapshot of the system) (ii) geo-replication across multiple data centers, (iii) asynchronous propagation of updates, (iv) causal consistency for operations that may access different objects, and (v) client support for ensuring causality by tracking dependences via dedicated metadata.

Dynamo Probably one of the most representative systems which rests on the extreme of the spectrum is Dynamo[12]. This system borrows multiple ideas and techniques previously proposed in the literature to build a cloud data store designed to be highly available in the presence of both temporary and permanent failures. This is achieved by relying on a weak form of quorums, namely sloppy quorums and a hinted hand-off protocol. Sloppy quorums allow the system to make progress on write operations even in the presence of failures in the replication group. Thus, if a client issues a write operation but there are not sufficient available replicas to complete the operation, nodes that do not replicate the data being updated are used to complete the operation. Eventually, Dynamo will move the data from the non-responsible nodes to the actual replicas using a hinted hand-off protocol. The Dynamo replication scheme is flat. Dynamo uses vector clocks to track causal dependencies within the replication group of each key. A vector clock contains one entry for each replica (thus the size of clocks grows linearly with the number of replicas). This metadata is kept in the data store, close to the data object, but is never transmitted to the clients. Therefore, when a client contacts a replica it has no way to assess if the replica has a state that is consistent with its past observations. Thus, it is possible for a client to issue a read operation and obtain a version that does not include the client's previous write operations (thus lacking session guarantees [48]). The purpose of the metadata is to detect conflicting updates and to be used by programmers in coding the reconciliation function, using some ad-hoc, application specific, approach. This very weak form of consistency, known as *eventual consistency*, makes it difficult to reason about and can lead to some known anomalies, such as the Amazon's shopping cart anomalies[12]. Nevertheless, its usability has been widely proven for some applications, namely through several commercial systems, such as Riak[2], which have successfully created a business around Dynamo's ideas.

COPS Due to the difficulty of building applications on top of eventually consistent data stores, the community has tried to achieve stronger consistency guarantees, such as session guarantees [48] or causal consistency[41], without hindering availability. More recently, in the context of geo-replicated cloud data stores, there has been a noticeable effort on providing causal consistency, which is widely regarded as the strongest consistency level that a system can provide without compromising availability and incurring high latencies[34]. COPS[33], a scalable causally consistent geo-replicated data store, revived the interest of the community for causally consistent cloud data stores (in fact, COPS offers a slightly stronger consistency criteria, dubbed causal+ consistency but this difference is not critical to our discussion; the reader may refer to [33] for a rigorous definition). COPS uses a hierarchical approach to manage replication: it assumes that some intra data center replication scheme ensures linearizability, such that replication within the data center can be completely masked from the inter data center protocol; COPS addresses the latter. To avoid the inconsistency issues of Dynamo, COPS requires clients to keep metadata and to exchange this metadata every time they access the cloud store. For this purpose, COPS assigns a scalar clock to each object. Clients are required to maintain in their metadata the last clock value of all objects read in the causal past. This metadata, representing their causal history, is purged after every update operation, due to the transitivity property of the happens-before relationship [30]. Thus, the dependencies of an update operation are those immediately previous to the update operation just issued, and the subsequent reads up to the newly issued update operation. This set of dependencies is known as *nearest dependencies*. As long as the client sticks to the same data center, its causal dependencies are automatically satisfied. Nevertheless, updates piggyback their dependencies when being propagated to other data centers, similarly to lazy replication[29]. When a data center receives an update propagated by another data center, it only makes it visible when its dependencies are satisfied. In addition, COPS provides a partial form of transactions called causally consistent read-only transactions. As its name clearly suggests, these are transactions with only read accesses that return versions of the read objects that belong to a causally consistent snapshot. To implement this construct, the authors propose a two-round protocol. In the first round, the client issues a concurrent read operation per key in the read set. Each retrieved object version has a list of dependencies associated. In case any conflicting version is returned (whose dependencies are not satisfied consistently), a second round of read operations is used to retrieve non-conflicting versions. Two object versions are free of conflicts if the objects do not depend on one another, or, if they do so, the version retrieved is equal or greater than the one in the dependencies list. Notice that the conflicting versions may appear only due to concurrent updates to the read-only transaction. The authors argue that in order to correctly identify conflicting versions, the *nearest dependencies* are not sufficient. Therefore, the system needs to store the full list of dependencies and piggyback them when clients issue reads within transactions. This clearly slows down the system and may even make the system unusable due to high latencies in processing and sending large chunks of dependencies over the network.

Chain Reaction In COPS, the fact that causal consistency is provided to the clients is not exploited to increase concurrency within a data center, given that the protocols assume linearizability. ChainReaction [4] exploits this opportunity by proposing a modified chain replication technique that allows for concurrent causally consistent reads within a data center. For this purpose ChainReaction slightly augments the metadata maintained by clients, with two logical clock values per data object: the first value is a global Lamport's clock and the second value is a hint on which local replicas can provide consistent reads. ChainReaction also explores an interesting trade-off when offering causally consistent read-only transactions: it introduces a logical serialization point in each data center (slightly reducing the concurrency of updates) to make it easier to identify consistent snapshots and avoid the need for multiple read-phases when implementing read-transactions.

GentleRain A recently proposed system that uses physical clocks in its design is GentleRain [19]. The main contributions of GentleRain are to reduce the metadata piggybacked on updates propagation to almost zero and to eliminate dependency checking procedures. The idea is to only allow a data center to make a remote update visible once all partitions (whithin the data center) have seen all updates up to the remote update time stamp. Thus, a client that reads a version is automatically ensured to read causally consistent versions in subsequent reads without the need of explicitly checking dependencies or being forced to wait until a causally consistent version is ready.

GentleRain uses physical clocks in its implementation for several purposes: (i) timestamp updates, (ii) detect conflicting versions, (iii) generate read snapshot time for read-only transactions, and (iv) compute the global stable clock. Clients only need to keep track of the latest global stable clock seen in order to ensure across-objects causality. The global stable clock is the minimum physical clock known among all the partitions and data centers, and it is local to each partition. Since, two partitions may have different global stable clocks, GentleRain relies on the clients to convey sufficient information to avoid causality violations. GentleRain is able to achieve performance results similar to eventually consistent systems. Notice that physical clocks could be substituted by simple scalar logical clocks without compromising correctness. Nevertheless, this could cause clocks to advance at very different rates in different partitions, causing the global stable clock to move slowly. This would increase the latency for the visibility of updates to data. Furthermore, it would generate staler read snapshot times for read-only transactions

ORBE ORBE [18] is a partitioned geo-replicated data store that proposes a solution based on the joint use of physical and logical clocks. ORBE uses vector clocks, organized as a matrix, to represent dependencies. The vector clock has an entry per partition and data center. Thus, multiple dependencies on data items of the same partition and data center are represented by only one scalar, which allows for reducing the size of the metadata piggybacked to messages. In addition, similarly to ChainReaction, ORBE is able to optimize the read-only transactional protocol to complete in only one round by relying on physical clocks. Physical clocks are used to generate read snapshot times. Thus, upon receiving a transaction request, the coordinator of the transaction sets the transaction's snapshot time to its physical clock. In consequence, the reads of the transaction have a version upper bound, the snapshot time, which is used to avoid retrieving versions that would violate causality due to concurrent update operations, as in COPS.

5 Clocks for Strong Consistency

In this section, we address systems that provide strong semantics with general purpose transactions, supporting read and write operations to arbitrary objects in the data store, with the guarantee that they will be accessed atomically and isolated from concurrent accesses.

GMU and SCORe The ability of scaling out data stores in large cloud deployments is highly affected by the cost of synchronizing data among the different replicas. This is particularly true for the case of strongly consistent data stores, which require synchronous interactions among all the replicas maintaining data manipulated by a transaction. In order to reduce this cost, a widely adopted strategy is to use partial replication techniques, i.e., to limit the nodes r that replicate each data item to a subset of all the nodes m (with $r \ll m$). One additional characteristic that is particularly desirable in large-scale systems is that of Genuine Partial Replication (GPR) [43], i.e., ensuring that a transaction involves coordination of only the nodes s that replicate the data items accessed (where typically $s \ll m$). We note that the GPR property rules out non-scalable designs that rely on a centralized time source[7], or that force the synchronous advancement of clocks on all nodes of the system upon the commit of any (update) transaction [46]. Two recent examples of GPR data stores that rely on logical clocks to ensure strong consistency are GMU [38] and SCORe [40]. Both systems implement a fully-decentralized multiversion concurrency control algorithms, which spares read-only transactions from the cost of remote validations and from the possibility of aborting. As for update transactions, GMU and SCORe can be characterized as Deferred Update Replication [45] protocols, since writes are buffered at the transaction's coordinator and sent out during the commit operation to be applied and made visible to other transactions in an atomic step.

The two systems differ, however, in the way they reify time. GMU uses a vector clock of size m (one entry per node) whereas SCORe uses a single scalar clock. These clocks are used to assign the read and commit timestamps to transactions, as well as to tag the various versions of each data item. In both systems, the read timestamp of a transaction is used to establish a consistent snapshot (i.e., the set of visible data item versions) over the partially replicated data set. In SCORe, the start timestamp is assigned to a transaction upon its first read operation in some node. From that moment on, any subsequent read operation is allowed to observe the most recent committed version of the requested datum having timestamp less than or equal to the start timestamp (as in classical multiversion concurrency control algorithms). In contrast, for GMU, whenever a transaction T reads for the *first* time on a node, it tries to advance the start timestamp (a vector clock) to extend the visible snapshot for the transaction. This can allow, for instance, to include in T's snapshot the versions created by a transaction T' that have involved a node n not contacted so far by T, and whose commit events would not be visible using T's initial read-timestamp (as the n-th entry of the commit timestamp of T' is higher than the corresponding entry of T's read-timestamp).

Another relevant difference between these two systems lies in the consistency guarantees that they provide: SCORe provides 1-Copy Serializability (1CS), whereas GMU only Update Serializability (US) [3], a consistency criterion that allows read-only transactions to serialize the commit events of non-conflicting update transactions in different orders — an anomaly only noticeable by users of the systems if they communicate using some external channel, which still makes US strictly weaker than classic 1CS. In fact, both protocols guarantee also the serializability of the snapshot observed by transactions that have to be aborted (because their updates are not reconcilable with those of already committed transactions). This additional property is deemed as important in scenarios in which applications process the data retrieved by the store in an non-sandboxed environment (e.g., in a Transactional Memory [15]), in which exposing data anomalies to applications may lead to unpredictable behaviors or faults.

Clock-SI Clock-SI[17] assumes loosely synchronized clocks that only move forward, but that can be subject to possibly arbitrary skews. Clock-SI[17] provides the so-called Snapshot Isolation[8] consistency criterion, in which read-only transactions read from a consistent (possibly versioned) snapshot, and other transactions commit if no object written by them was also written concurrently. Each transaction identifies its read timestamp simply by using its local physical clock. In order to ensure safety in spite of clocks skews, Clock-SI resorts to delaying read operations in two scenarios: i) the read operation is originated by a remote node whose clock is in the future with respect to the physical clock of the node that processes the read request; ii) the read operation targets a data item for which a pending pre-committed version exists and whose commit timestamp may be smaller than the timestamp of the read operation.

Spanner Spanner [5] provides a stronger consistency guarantee than the data stores discussed so far, namely external consistency (also known as strict serializability). The key enabler behind Spanner is the TrueTime API. Briefly, clock skew is exposed explicitly in TrueTime API by representing the current time as an interval. This interval is guaranteed to contain the absolute time at which the clock request was invoked. In fact, multiple clock references (GPS and atomic clocks) are used to ensure strict, and generally small, bounds on clock uncertainty (less than 10 ms).

Spanner assigns a timestamp s to (possibly read-only) transactions by reading the local physical clock. Similarly, it also assigns a commit timestamp c to update transactions once that they acquire all the locks over the write-set w, which is then used to version the new data items. Spanner blocks read operations associated with a physical timestamp t in case it cannot yet be guaranteed that no new update transaction will be able to commit with a timestamp $t' < t$. This is achieved by determining a, so called, *safe time* t_{safe} that is lower bounded by the physical timestamp of the earliest prepared transaction at that node. Unlike Clock-SI, though, Spanner also introduces delays in commit operations: the write-set w of a transaction with commit timestamp c is made visible only after having waited out the uncertainty window associated with c. This is sufficient to ensure that the side-effects of a transaction are visible solely in case timestamp c is definitely in the past, and that any transaction that reads or over-writes data in w is guaranteed to be assigned a timestamp greater than c.

CockroachDB The waiting phases needed by the data stores based on physical clocks can have a detrimental effect on performance, especially in the presence of large skews/uncertainties in the clock synchronization. This has motivated the investigation of techniques that use in conjunction both logical and physical clocks, with the aim to allow the correct ordering of causally related transactions without the need of injecting delays for compensating possible clock skews. Augmented-Time (AT) [13], for instance, uses a vector clock that tracks the latest known physical clock values of any node with which communication took place (even transitively) in the current uncertainty window. The assumption on the existence of a bounded skew among physical clocks allows to trim down the size of the vector clocks stored and disseminated by nodes. Also, by relying on vector clocks to track causality relations within the uncertainty interval, AT ensures external consistency for causally related transactions (e.g,. if a transaction T2 reads data updated by transaction T1, T2's commit timestamp will be greater than T1's) without the need for incurring Spanner's commit wait phases.

Real-time order (or external consistency) cannot however be reliably enforced between transactions whose uncertainty intervals overlap and which do not develop any (possibly transitive) causal dependency. In such cases, waiting out the uncertainty interval (provided that reasonable bounds can be assumed for it) remains necessary to preserve external consistency. Hybrid Logical Clocks (HLC) [28] represent a recent evolution of the same idea, in which physical clocks are coupled with a scalar logical clock with the similar goal of efficiently ordering causally related transactions whose uncertainty intervals overlap. Roughly speaking[1], with HLC, whenever a node n receives a message timestamped with a physical clock value, say pt', greater than the maximum currently heard of among the nodes, say pt, n sets $pt = pt'$ and uses the updated value of pt as if it were a classic scalar logical clock in order to propagate causality information among transactions.

HLC is being integrated in CockroachDB[1], an open-source cloud data store (still under development at the time of writing) that, similarly to Spanner, relies on physical clocks to serialize transactions (ensuring either Snapshot Isolation or Serializable Snapshot Isolation[20]). Unlike Spanner, however, CockroachDB does not assume the availability of specialized hardware to ensure narrow bounds on clock synchronization, but relies on conventional NTP-based clock synchronization that frequently imposes clock skews of several tens of milliseconds. HLC is hence particularly beneficial in this case, at it allows for ensuring external consistency across causally related transactions while sparing from the costs of commit waits.

6 Discussion

We now draw some observations, based on the systems analyzed in the previous sections.

6.1 It is Unclear How Well External Consistency May Scale

Ensuring external consistency in a distributed data store without sacrificing scalability (e.g., using a centralized time source, or tracking the advancement of logical time at all nodes whenever a transaction commits) is a challenging problem. GPR systems based on logical clocks, like SCORe or GMU, can accurately track real-time order relations *only* between causally related transactions — a property that goes also under the name of witnessable real-time order [39]). Analogous considerations apply to system, like Clock-SI, that rely on physical clocks and assume unbounded clock-skews (and monotonic clock updates). The only GPR system we know of that guarantees external consistency is Spanner, but this ability comes with two non-negligible costs: 1) the reliance on specialized hardware to ensure tight bounds on clock uncertainty; and 2) the introduction of delays upon the critical path of execution of transactions, which can have a detrimental impact on performance. As already mentioned, the usage of hybrid clocks, like HLC or AT, can be beneficial to reduce the likelihood of blocking due to clock uncertainty. However, we are not aware of any GPR data store that uses hybrid clocks and loosely-synchronized physical clocks (i.e., subject to unbounded skew), which can consistently capture real-time order among non-causally related transactions, e.g., two transactions that execute sequentially on different nodes and update disjoint data items. In the light of these considerations, there appears to exist a trade-off between the strictness of the real-time ordering guarantees that can be ensured by a scalable transactional data store and the necessity of introducing strong assumptions on clock synchronization.

More in general, the analysis of these systems raises interesting theoretical questions such as: which forms of real-time order can be guaranteed by scalable (i.e., GPR) cloud data store implementations that use exclusively logical clocks? Does the use of loosely synchronized physical clocks, possibly complemented by logical

[1]A more sophisticated algorithm using two scalar logical clocks is also proposed in order to guarantee the boundedness of the information stored in the logical timestamps and that the divergence between physical and logical clocks is bounded — which allows for using logical clocks as a surrogate of physical clocks, e.g., to use logical clocks to obtain consistent snapshots at "close" physical times.

clocks, allow for providing stronger real-time guarantees? Would relaxing the progress guarantees for read-only transactions (e.g., lock-freedom in systems like Spanner or GMU) allow for strengthening the real-time ordering guarantees that a GPR system can guarantee? One may argue that the above questions appear related to some of the theoretical problems studied by the parallel/concurrent computing literature, which investigated the possibility of building scalable Transactional Memory implementations [23, 47, 14, 24] that guarantee a property akin to GPR, namely Disjoint Access Parallelism (DAP) [25, 5, 10, 39]: roughly speaking, in a DAP system two transactions that access disjoint data items cannot contend for any shared object (like logical clocks). However these works considered a different system model (shared-memory systems) and neglected the usage of physical and hybrid clocks. Understanding whether those results apply to the case of distributed data stores (i.e., message-passing model) using different clock types remains an interesting open research problem.

6.2 The Trade-offs Between Logical and Physical Clocks Are Not Completely Understood

A key trade-off that emerges when using logical clocks in a strongly consistent GPR cloud data store is that the advancement of clocks across the various nodes of the system is dependent on the communication patterns that are induced by the data accesses issued by the user-level applications. As such, systems based on logical clocks may be subject to pathological scenarios in which the logical clocks of a subset s of the platform's nodes can lag significantly behind with respect to the clocks of other nodes that commit update transactions without accessing data maintained on s. In this case, the transactions originated on the nodes in s are likely to observe stale data and incur higher abort rates. The mechanism typically suggested to tackle this issue is to rely on some asynchronous, epidemic dissemination of updates to the logical clocks. Such techniques can at least partially address the issue, but they introduce additional complexity (as the optimal frequency of dissemination is a non-trivial function of the platform's scale and workload) and consume additional network bandwidth.

Clearly, physical clocks — which advance spontaneously — avoid this issue. On the other hand, in order to compensate for clock skews, data stores that rely on physical clocks have to inject delays proportional to the uncertainty intervals in critical phases of the transactions' execution. As already discussed in various works[5, 1, 28, 13], these delays can severely hamper performance unless specialized, expensive hardware is used to ensure tight bounds on the uncertainty intervals (as in Spanner). In this sense, the idea at the basis of hybrid clocks, and in particular of approaches like HLC [28], appears quite interesting, since it attempts to combine the best of both worlds. On the other hand, research in this area is still at its infancy and the only cloud data store we are aware of that is using hybrid clocks is CockroachDB, which is still in its early development stages. It is hence unclear to what extent hybrid clocks can alleviate the need for injecting artificial delays in solutions based on physical clocks. Indeed, we advocate the need for conducting a rigorous and systematic experimental analysis aimed at identifying under which conditions (e.g., platform's scale, maximum clock synchronization skew, workloads) each of the various approaches provides optimal efficiency. The results of such a study would be valuable to guide practitioners in the selection of the data store best suited to their application needs. They may also provide motivations to foster research in the design of autonomic systems aimed at adapting the type of clock implementation to pursue optimal efficiency in face of dynamic workload conditions and/or platform's scale (e.g., following an elastic re-scaling of the platform).

6.3 Time is a Resource That Can be Traded for Other Resources

As we have noted earlier, the end-user experience can be negatively affected by the latency of the cloud service. Therefore, solutions that require waiting, as some of the previously described protocols based on the use of physical clocks, have to be used carefully. On the other hand, time is just one of the many resources that can be a source of bottlenecks in the operation of a cloud data store. Large metadata structures consume disk space and increase network bandwidth utilization, within the data center, across data centers, and between the data center and the client. In turn, heavy disk and network utilization are also a major source of delays and instability.

Since solutions based on physical clock often allow for significant metadata savings, the overall performance of a given algorithm is hard to predict, and strongly depends on which resources the system is more constrained.

For instance, although GentleRain is able to reduce dependencies metadata to almost zero, it also has a disadvantage: clients are likely to read stale data. Since remote updates visibility is delayed, we may end up with latencies of up to seconds (depending on the geo-replicated deployment and network conditions) before an update is visible to other clients. On the other hand, the system will have performance almost similar to an eventually consistent data store since the metadata is almost negligible and no dependence checking mechanism is required. Thus, depending on application requirements, one could give preference to freshness of data or performance.

Another example is how ORBE uses time to avoid multiple rounds of messages to implement read-only transactions. In this case, loosely synchronized physical clocks are used as a decentralized solution for implementing one round causally consistent read-only transactions. However, this may affect negatively the latency of operations. A partition that receives a read operation with snapshot time st has to first wait until its physical clock catches up with st, and also to make sure that updates up to that time from other replicas have been received.

GMU and SCORe also exemplify this trade-off. By using vector clocks, GMU can advance the transaction's read snapshot more frequently than SCORe, which relies on a scalar clock. The intuition is that, since SCORe needs to capture causal dependencies among transactions via a single scalar clock, it is sometimes forced to over-approximate such relations, which leads to observing obsolete snapshots. This impacts the freshness of the data snapshots observed by transactions, an issue that is particularly relevant for update transactions, which can be subject to spurious aborts as a consequence of unnecessarily reading obsolete data versions. On the other side of the coin, in both systems, logical timestamps are piggybacked to every message, both for read operations (to identify the read timestamp), as well as during the distributed commit phase (to obtain the final commit timestamp). As such, SCORe naturally incurs less overheads by sending a single scalar clock with each message, in contrast with GMU's vector clocks that grow linearly with the number of nodes in the deployment. Experiments comparing both systems in a similar infra-structure (both in terms of software as well as in hardware) confirmed that, at least in low conflict workloads, SCORe tends to perform even slightly better than GMU [40], despite providing a stronger consistency criterion — we note that their scalability trends are similar, as expected in workloads with low contention to data. Other workloads may be found on another work relying on SCORe [16].

6.4 Total Order is Expensive but Concurrency Also Comes with a Cost

Most strong consistency models, such as serializability, require concurrent requests to be serialized and this, in practice, means that some form of total order/consensus protocol needs to be implemented in the system. By definition, such protocols materialize a logic serialization point for the system, which can easily turn into a bottleneck. The use of weaker consistency models, which allow for more concurrency, has therefore great appeal. We have already discussed that weak consistency imposes a burden on the programmer of applications that use the cloud data store. Maybe not so obvious is the fact that weak consistency also imposes a metadata burden on the protocols that support the additional level of concurrency. For instance, consider an idealized system where all operations, for all objects, are executed by a single "primary" node, replicated using Paxos [31]. The entire state of the system at a given point can be referred to by a single scalar, the number of the last operation executed by the primary node. On the one hand, the use of a single primary node to serialize all operations is inherently non-scalable, and hence undesirable in a large-scale system. On the other hand, an idealized fully concurrent system supporting partial replication, with no constraints on how updates are propagated among replicas, requires to maintain a matrix clock of size m^2 where m is the total number of nodes in the entire system. Such timestamps would be impossible to maintain and exchange in large scale.

The discussion above shows that fully decentralized (and inherently fully concurrent) systems may be as little scalable as fully centralized (and inherently fully sequential) systems. It is therefore no surprise that, as with the use of different clock implementations, the right trade-offs may heavily depend on concrete factors such as the workload characteristics, the hardware deployment, etc. In consequence, it is hard to define, in general, when one approach outperforms the other.

Consider for instance the different approaches taken by COPS and ChainReaction for different aspects of the system. ChainReaction is geared towards efficient read operations while COPS aims at more balanced workloads. For instance, ChainReaction is willing to slightly increase the metadata size to allow for additional concurrency while reading objects within a data center, contrary to COPS that enforces linearizabilty at each data center. Also ChainReaction serializes updates within the data center to optimize read-only transactions while COPS does not serialize updates to distinct objects but may pay the cost of multi-phase read-only transactions. Similarly, ORBE implements a number of metadata reduction techniques, that may create false dependencies among operations, thus, achieving metadata savings at the cost of concurrency loss.

7 Summary

In this work we focused on one common aspect that is crucial to any distributed cloud data store, i.e., the mechanisms employed to capture the passage of time and to keep track of the cause-effect relations among different data manipulations.

We analyzed existing literature in the area of cloud data stores from two antagonistic perspectives, those of low latency intra data center communication and geo-replicated systems. These two complementary deployments of machines create the opportunity to explore both strongly consistent and weakly consistent systems. This is, in part, a consequence of the design of modern data centers that allow strong consistent systems to perform fast, whereas for inter data center replication we can effectively use weak consistent systems (in particular with causal consistency) while still providing meaningful semantics and guarantees to the programmer.

Our analysis allowed us to derive some observations on the implications of various design choices at the basis of existing cloud data stores. These insights may not only help system architects to select the solutions that best fit their needs, but also identify open research questions that may offer opportunities to enhance the next generations of cloud data stores.

References

[1] Cockroach. https://github.com/cockroachdb/cockroach.

[2] Riak. http://basho.com/riak/.

[3] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999. AAI0800775.

[4] 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*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[5] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4):698–719, 2011.

[6] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, March 2013.

[7] Roberto Baldoni, Carlo Marchetti, and Sara Tucci Piergiovanni. A fault-tolerant sequencer for timed asynchronous systems. In *Euro-Par 2002 Parallel Processing*, pages 578–588. Springer, 2002.

[8] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

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

[10] Victor Bushkov, Dmytro Dziuma, Panagiota Fatourou, and Rachid Guerraoui. Snapshot isolation does not scale either. Technical report, Technical Report TR-437, Foundation of Research and Technology–Hellas (FORTH), 2013.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[12] 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.

[13] Murat Demirbas and Sandeep Kulkarni. Beyond truetime: Using augmented time for improving Google Spanner, 2013.

[14] Nuno Diegues and Paolo Romano. Time-warp: lightweight abort minimization in transactional memory. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*,167–178. 2014.

[15] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 3–14, 2014.

[16] Nuno Lourenço Diegues and Paolo Romano. Bumper: Sheltering transactions from conflicts. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 185–194, Washington, DC, USA, 2013. IEEE Computer Society.

[17] Jiaqing Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 173–184, Sept 2013.

[18] 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. ACM.

[19] 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. ACM.

[20] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[21] Colin J Fidge. *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science, 1987.

[22] 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.

[23] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[24] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[25] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, 151–160. ACM, 1994.

[26] Caroline Jay, Mashhuda Glencross, and Roger Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Trans. Comput.-Hum. Interact.*, 14(2), August 2007.

[27] Sandeep S Kulkarni et al. Stabilizing causal deterministic merge. In *Self-Stabilizing Systems*, pages 183–199. Springer, 2001.

[28] SandeepS. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In MarcosK. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems*, volume 8878 of *Lecture Notes in Computer Science*, pages 17–32. Springer International Publishing, 2014.

[29] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, November 1992.

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

[31] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[32] Greg Linden. Make data useful. *Presentation, Amazon, November*, 2006.

[33] 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. ACM.

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

[35] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[36] David L Mills. A brief history of ntp time: Memoirs of an internet timekeeper. *ACM SIGCOMM Computer Communication Review*, 33(2):9–21, 2003.

[37] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[38] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 455–465, June 2012.

[39] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. Brief announcement: Breaching the wall of impossibility results on disjoint-access parallel tm. *Distributed*, page 548.

[40] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 456–475, New York, NY, USA, 2012. Springer-Verlag New York, Inc.

[41] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 288–301, New York, NY, USA, 1997. ACM.

[42] Sunil K. Sarin and Nancy A. Lynch. Discarding obsolete information in a replicated database system. *Software Engineering, IEEE Transactions on*, (1):39–47, 1987.

[43] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, pages 214–224, Washington, DC, USA, 2010. IEEE Computer Society.

[44] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*, June 2009.

[45] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable deferred update replication. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[46] Damián Serrano, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 290–297. IEEE, 2007.

[47] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[48] 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.

[49] Gene TJ Wuu and Arthur J Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 233–242. ACM, 1984.