CausalSpartan: Causal Consistency for Distributed Data Stores using Hybrid Logical Clocks

Mohammad Roohitavaf*, Murat Demirbas†, and Sandeep Kulkarni*
* Department of Computer Science and Engineering
Michigan State University
East Lansing, MI
Email: {roohitav, sandeep}@cse.msu.edu
† Department of Computer Science and Engineering
University of Buffalo, SUNY
Buffalo, NY
Email: demirbas@buffalo.edu

***Abstract*—Causal consistency is an intermediate consistency model that can be achieved together with high availability and high-performance requirements even in presence of network partitions. In the context of partitioned data stores, it has been shown that implicit dependency tracking using clocks is more efficient than explicit dependency tracking by sending dependency check messages. Existing clock-based solutions depend on monotonic psychical clocks that are closely synchronized. These requirements make current protocols vulnerable to clock anomalies. In this paper, we propose a new clock-based algorithm, CausalSpartan, that instead of physical clocks, utilizes Hybrid Logical Clocks (HLCs). We show that using HLCs, without any overhead, we make the system robust on physical clock anomalies. This improvement is more significant in the context of query amplification, where a single query results in multiple GET/PUT operations. We also show that CausalSpartan decreases the visibility latency for a given data item comparing to existing clock-based approaches. In turn, this reduces the completion time of collaborative applications where two clients accessing two different replicas edit same items of the data store.**

Like previous protocols, CausalSpartan assumes that a given client does not access more than one replica. We show that in presence of network partitions, this assumption (made in several other works) is essential if one were to provide causal consistency as well as immediate availability to local updates.

Keywords*—*Causal Consistency, Hybrid Logical Clocks, Distributed Data Stores, Key-value Stores, Geo-replication

I. Introduction

Geo-replicated data stores are one of the integral parts of today's Internet services. Service providers usually replicate their data on different nodes worldwide to achieve higher performance and durability. However, when we use this approach, the consistency among replicas becomes a concern. In an ideal situation, any update to any data item instantaneously becomes visible in all replicas. This model of consistency is called *strong consistency*. Unfortunately, it is impossible to achieve strong consistency without sacrificing the availability when we have network partitions. The CAP theorem [16] implies that in presence of network partitions, we cannot have strong consistency and availability together. Even in the absence of network partitions, strong consistency comes with its performance overhead [8].

Due to performance and availability overhead of the strong consistency, many systems use *eventual consistency [26]*. In this consistency model, as the name suggests, the only guarantee is that replicas become consistent "eventually". We can implement always-available services under this consistency model. However, it may expose clients to anomalies, and application programmers need to consider such anomalies. To understand how eventual consistency may lead to an anomaly, consider the following example from [20]: Suppose in a social network, Alice uploads a photo and then adds it to an album. Under eventual consistency model, a remote replica may update the album before writing the photo. That scenario is not desirable, as the album is pointing to a photo which is not visible to clients. Despite such anomalies, because of the availability and performance benefits, some distributed data stores (e.g., Dynamo [12]) use eventual consistency.

Causal consistency is an intermediate consistency model. Causal consistency requires that the effect of an event can be visible only when the effect of its causal dependencies is visible. The causal dependency captures the notion of happens-before relation define in [19]. Under this relation, any event by a client depends on all previous events by that client. Thus, in our example, adding reference to the album depends on the event of uploading the photo. Thus, no replica can update the album before writing the photo. Causal consistency is achievable with availability even in the presence of networks partitions.

To guarantee causal consistency, we need some mechanism to track causal dependencies. Specifically, we have to keep a version invisible to clients if some of its causal dependencies are invisible. Checking dependencies is especially challenging for partitioned systems where each replica consists of several machines. Existing work on causal consistency for replicated and partitioned data stores can be classified into those where partitions need to communicate with each other to check causal dependencies (e.g., COPS [20] and Orbe [14]), and those that rely on synchronized clocks (e.g., GentleRain). The former suffer from high overhead resulted from high message complexity. GentleRain [15], on the other hand, uses *an implicit dependency check* to reduce the overhead. It guarantees causal consistency by using physical clocks. Partitions communicate with each other only *periodically* rather than for every new update. When compared with COPS and Orbe, this reduces the message complexity of GentleRain significantly thereby providing higher throughput.

Although GentleRain reduces the overhead of tracking dependencies, it relies on synchronized and monotonic physical clocks for both its correctness and performance. Specifically, it requires that clocks are strictly increasing. This may be hard to guarantee if the underlying service such as NTP causes non-monotonic updates to POSIX time [7] or suffers from leap

seconds [2], [4]. In addition, as we will see, the clock skew between physical clocks of partitions may lead to cases where GentleRain must intentionally delay write operations.

The issue of clock anomalies is intensified in the context of query amplification, where a single query (e.g., an update on a Facebook page) results in many (possibly 100s to 1000s) GET/PUT operations [10]. In this case, the delays involved in each of these operations contribute to the total delay of the operation, and can substantially increase the response time for the clients.

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

To achieve this goal, we develop CausalSpartan that is based on the structure of GentleRain but utilizes Hybrid Logical Clocks (HLCs) [17]. HLCs combine the logical clocks [19] and physical clocks. In particular, these clocks assign a timestamp hlc for event e such that if e happened before f (as defined in [19]), then $hlc.e < hlc.f$. Furthermore, the value of hlc is very close to the physical clock and is backward compatible with NTP clocks [6].

Similar to [14], [15], [20], [21], we assume that a client only accesses one replica during its execution. Since this assumption is standard in the literature, we investigate its necessity. We observe that this assumption is essential if we want to provide causal consistency while ensuring that all local updates are visible immediately. In other words, we show that if the client could access multiple replicas and a replica makes local updates visible immediately then it is impossible to provide causal consistency.

Contributions of the paper.

- We show that in the presence of clock anomalies, CausalSpartan reduces the latency of PUT operations compared with that of GentleRain. Moreover, the performance or correctness of CausalSpartan is unaffected by clock anomalies.
- We demonstrate that CausalSpartan is especially effective to deal with delays associated with query amplification.
- We demonstrate that CausalSpartan reduces the update visibility latency. This is especially important in collaborative applications associated with a data store. For example, in an application where two clients update a common variable (for example, bid price for an auction) based on the update of the other client, CausalSpartan reduces the execution time substantially.
- We show that using HLC instead of physical clocks does not have any overhead.
- We demonstrate the efficiency provided by our approach by performing experiments on cloud services provided by Amazon Web Services [1].
- We provide an impossibility result that shows locality of traffic is necessary for a causally consistent data store that immediately makes local updates visible.

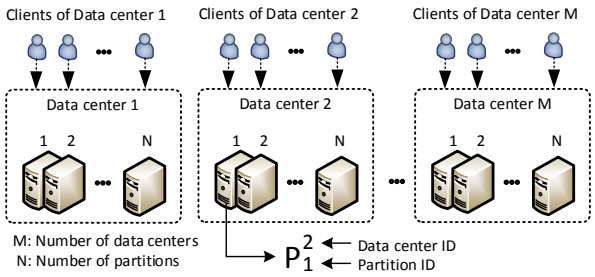


Fig. 1. A system consisting of M data centers (replicas) each of which consists of N partitions. p_n^m denotes nth partition in mth data center.

Organization of the paper. In Section II, we define our system architecture and the notion of causal consistency. In Section III, we discuss, in detail, the issues of clock anomalies that we want to address. In Section IV, we provide a brief overview of HLCs from [17]. Section V provides our CausalSpartan and Section VI provides our experimental results. We provide an impossibility result in Section VII. In Section VIII we discuss related work. Finally, Section IX concludes the paper.

II. Background

A. Architecture

In this section, we focus on the system architecture and assumptions that are the same as those assumed in [14], [15], [20], [21]. We consider a data store whose data is fully replicated into M data centers (i.e., replicas) where each data center is partitioned into N partitions (see Figure 1). Partitioning the data means that the data is distributed across multiple machines. Partitioning increases the scalability of a data store, as the size of the data store does not depend upon the capacity of a machine. Replication, on the other hand, increases the performance and durability. Specifically, by replicating data in geographically different locations, we let the clients query their local data center for reading or writing data thereby reducing the response time for client operations. In addition, by replicating data, we increase the durability of our data in case of failures at one data center. Like [14], [15], [20], [21], we assume that a client does not access more than one data center. We prove the necessity of this assumption in Section VII. There might be network failure *between* data centers that causes network partitions. We assume network failure do not happen *inside* data centers. Thus, partitions inside a data center can always communicate with each other.

We assume multi-version key-value stores that store several versions for each key. A key-value store has two basic operations: $PUT(k, val)$ and $GET(k)$, where $PUT(k, val)$ writes new version with value val for item with key k, and $GET(k)$ reads the value of an item with key k.

B. Causal Consistency

Causal consistency is defined based on the happens-before relation between events [19]. In the context of key-value stores, we define happens-before relation as follows:

***Definition** 1 (Happens-before):* Let a and b be two events. We say a happens before b, and denote it as $a \rightarrow b$ iff:

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

Now, we define causal dependency as follows:

***Definition** 2 (Causal Dependency):* Let v_1 be a version of key k_1, and v_2 be a version of key k_2. We say v_1 causally depends on v_2, and denote it as v_1 dep v_2 iff $PUT(k_2, v_2) \rightarrow PUT(k_1, v_1)$.

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

***Definition** 3 (Visibility):* We say version v of key k is visible to client c iff $GET(k)$ performed by client c returns v' such that $v' = v$ or $\neg(v$ dep $v')$.

Now, we define causal consistency as follows:

***Definition** 4 (Causal Consistency):* Let k_1 and k_2 be any two arbitrary keys in the store. Let v_1 be a version of key k_1, and v_2 be a version of key k_2 such that v_1 dep v_2. The store is causally consistent if for any client c that has read v_1, v_2 is visible to client c.

In the above definitions, we ignore the possibility of conflicts in writes. Conflicts occur when we have two writes on the same key such that there is no causal dependency relation between them. Specifically,

***Definition** 5 (Conflict):* Let v_1 and v_2 be two versions for key k. We call v_1 and v_2 are conflicting iff $\neg(v_1$ dep $v_2)$ and $\neg(v_2$ dep $v_1)$. (i.e., none of them depends on the other.)

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

***Definition** 6 (Visibility+):* We say version v of key k is visible+ for conflict resolution function f to client c iff $GET(k)$ performed by client c returns v' such that $v' = v$ or $v' = f(v, v')$.

Causal consistency with convergent conflict handling is called causal+ [20]. We formalize causal+ consistency as follows:

***Definition** 7 (Causal+ Consistency):* Let k_1 and k_2 be any two arbitrary keys in the store. Let v_1 be a version of key k_1, and v_2 be a version of key k_2 such that v_1 dep v_2. The store is causal+ consistent for conflict resolution function f if for any client c that has read v_1, v_2 is visible+ for f to client c.

A trivial solution for a causally consistent data store is a data store that always returns the initial value for each key. Of course, such data store is not desirable. Thus, we define causal++ consistency that requires the data store to make all local updates visible to clients immediately. Specifically,

***Definition** 8 (Causal++ Consistency):* A store is causal++ consistent for conflict resolution function f if

- it is causal+ consistent for f, and
- any version v written in data center r is *immediately* visible+ for f to any client accessing r.

In practice, in addition to the consistency, we want all data centers to eventually converge to the same data. In other words, we want a write in a data center to be reflected in other connected data centers as well. Two data centers are called connected, if there is no network partition that prevents them from communication. Thus, we define convergence as follows:

***Definition** 9 (Convergence):* Let v_1 be a version for key k written in data center r.

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

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

Note that, current proposals for causally consistent data stores like [14], [15], [20], [21], are in fact convergent and causal++ consistent for conflict resolution function last-writer-wins that breaks ties by data center IDs. Specifically, each version is assigned a timestamp. If two versions are conflicting, the winner version is the one with higher timestamp. If two timestamps are equal, the version written in the data center with higher ID is the winner. For brevity of presentation, we simply use the word visible, and causal consistency instead of visible+, and causal++ consistency. Also, we assume conflict resolution function last-writer-wins that breaks ties by data center IDs.

III. Effect of Clock Anomalies

In this section, we identify the issues caused by clock anomalies in providing causal consistency in protocols such as GentleRain [15]. First, in Section III-A, we identify how causal consistency is achieved in [15] with synchronized clocks. Next, in Section III-B, we identify why delays have to be introduced in PUT operations to satisfy causal consistency. In Section III-C, we identify why the effect of latency introduced in PUT operations causes a significant problem to queries that result in multiple GET/PUT operations for a given query.

A. Using Physical Clocks to Achieve Causal Consistency

In this section, we review the basic principles used by GentleRain to achieve causal consistency with the help of physical clocks. GentleRain assigns each version a timestamp equal to the physical clock of the partition where the write of the version occurs. We denote the timestamp assigned to version X by $X.t$. GentleRain assigns timestamps such that following condition is satisfied:

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

Also, each node in the data center periodically computes a variable called Global Stable Time (GST) (through communication with other partitions) such that following condition is satisfied:

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

When a client performs $GET(k)$, the partition storing k, returns the newest version v of k which is either created locally, or has a timestamp no greater than GST. According to conditions $C1$ and $C2$ defined above, any version which is a dependency of v is visible in the local data center and causal consistency is satisfied.

B. Sensitivity on physical clock and clock synchronization

To satisfy condition $C1$, in some cases, it may be necessary to wait before creating a new version. Specifically, if a client has read/written a key with timesstamp t, then any future PUT operation the client invokes must have a timestamp higher than t. Hence, the client sends the timestamp t of the last version that it has read/written together with a PUT operation. The partition receiving this request first waits until its physical clock is higher than t before creating the new version. This wait time, as we observed in our experiments, is proportional to the clock skew between servers. In other words, as the physical clocks of servers drift from each other, the incidence and the amount of this wait period increases.

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

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

C. Query Amplification

The sensitivity issue identified in Section III-B is made worse in practice, because a single end user request usually translates to so many possibly causally dependent internal queries. This phenomenon is known as *query amplification* [10]. In a system like Facebook, query amplification may result in up to thousands of internal queries for a single end user request [10]. In such a system, an end user submits the request to a web server. The web server performs necessary internal queries, and then responds to the end user. This implies that the web server needs to wait for all internal queries before responding the end user request. Thus, any delay in any of internal queries will affect the end user experience [10].

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

IV. Hybrid Logical Clocks

In this section, we recall HLCs from [17]. HLC combines logical and physical clocks to leverage key benefits of both. The HLC timestamp of event e, denoted as $hlc.e$, is a tuple $\langle l.e, c.e\rangle$. The first component, $l.e$, is the value of the physical clock, and represents our best approximation of the global time when e occurs. The second component, $c.e$, is a *bounded* counter that is used to capture causality whenever $l.e$ is not enough to capture causality. Specifically, if we have two events e and f such that e happens-before f (see Definition 1), and $l.e = l.f$, to capture causality between e and f, we set $c.e$ to a value higher than $c.f$. Although we increase c, as it is proved in [17], the theoretical maximum value of c is $O(n)$ where n is the number of processes. In practice, this value remains very small. In addition to HLC timestamps, each process a maintains an HLC clock $\langle l.a, c.a\rangle$. For completeness, we recall algorithm of HLC from [17] below.

Algorithm 1 HLC algorithm from [17]

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

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

HLC satisfies logical clock property that allows us to capture (one-way) causality between two events. Specifically, if e happens-before f, then $hlc.e < hlc.f$[1]. This implies that if $hlc.e = hlc.f$, then e and f are (causally) concurrent. At the same time, just like physical clock, HLC increases spontaneously, and it is close to the physical clock. Thus, it can be used to take snapshot at a given physical time.

V. CausalSpartan Protocol

One way to get around the issue of PUT latency identified in Section III is as follows: Suppose that a client has read a value written at time t and it wants to perform a new PUT operation on a server whose time is less than t. To satisfy $C1$, in [15], PUT operation is delayed until the clock of the server was increased beyond t. Another option is to change the clock of the server to be $t + 1$. However, changing the physical clock is undesirable; it would lead to violation of clock synchronization achieved by protocols such as NTP.

[1] $hlc.e < hlc.e$ iff $l.e < l.f \vee (l.e = l.f \wedge c.e < c.f)$.

It would also have unintended consequences for applications using that clock.

Using HLC in this problem solves several problems associated with changing the physical clock. Specifically, HLC is a logical clock and can be changed if needed. In the scenario described in previous paragraph, this would be achieved by increasing the c value which is still guaranteed to stay bounded [17]. At the same time, HLC is guaranteed to be close to the physical clock. Hence, it can continue to be used in place of physical clock. Also, HLC uses the physical clock as a read-only variable thereby ensuring that it does not affect protocols such as NTP. For these reasons, CausalSpartan uses HLC. In particular, CausalSpartan utilizes GentleRain [15] as a starting point and makes it faster and resilient on clock anomalies.

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

Next, we focus on different parts of the CausalSpartan protocol.

A. Client Side

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

Algorithm 2, shows the algorithm for the client operations. For a GET operation, the client sends the key that it wants to read together with its DSV_c by sending $\langle \text{GETREQ } k, DSV_c\rangle$ message to the server where key k resides. In the response, the server sends the value of the requested key together with a list of dependencies of the returned value, ds, and the DSV in the server, dsv. The client, then, first updates its DSV, and next update its DS_c by calling $updateDS$ for each member of ds as follows: for each $\langle i, h\rangle \in ds$ if currently there is an entry $\langle i, h'\rangle$ in DS_c, it replaces h' with the maximum of h and h', otherwise it adds $\langle i, h\rangle$ to the DS_c.

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

[2]This could be maintained as part of client library as in [20] so that the effective interface of the client does not have to explicitly maintain this information. Alternatively, this could also be maintained by the server for each client. For sake of simplicity, in our discussion, we assume that this information is provided by the client.

Algorithm 2 Client operations at client c

```
1:  GET (key k)
2:      send ⟨GETREQ k, DSV_c⟩ to server
3:      receive ⟨GETREPLY v, ds, dsv⟩
4:      DSV_c ← max(DSV_c, dsv)
5:      for each ⟨i, h⟩ ∈ ds
6:          DS_c ← updateDS(i, h, DS_c)
7:  return v

8:  PUT (key k, value v)
9:      send ⟨PUTREQ k, v, DS_c⟩ to server
10:     receive ⟨PUTREPLY ut, sr⟩
11:     DS_c ← updateDS(sr, ut, DS_c)

12: updateDS (data center id i, timestamp h, dependency set ds)
13:     if ∃⟨i, h'⟩ ∈ ds
14:         ds ← ds − ⟨i, h'⟩
15:         ds ← ⟨i, max(h, h')⟩
16:     else
17:         ds ← ds ∪ {⟨i, h⟩}
18: return ds
```

B. Server Side

In this section, we focus on the server side of the protocol. We have M data centers (i.e., replicas) each of which with N partitions (i.e., servers). We denote the nth partition in mth replica by p_n^m (see Figure 1). We denote the physical clock at partition p_n^m by PC_n^m. Each partition p_n^m stores a vector of size M (one entry for each data center) of (HLC) timestamps denoted by VV_n^m. For $k \neq m$, $VV_n^m[k]$ is the latest timestamp received from server p_n^k. $VV_n^m[m]$ is the highest timestamp assigned to a version written in partition p_n^m. Partitions inside a data center, periodically share their VVs with each other, and compute DSV as the entry-wise minimum of VVs. DSV_n^m is the DSV computed in server p_n^m.

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

Algorithm 3 shows the algorithm for PUT and GET operations at the server side. Upon receiving a GET request (GETREQ), the server first updates its DSV if necessary using DSV value received from the client (see Line 2 of Algorithm 3). After updating DSV, the server finds the latest version of the requested key that is either written in the local data center, or all of its dependencies are visible in the data center. To check this, the server compares the DS of the key with its DSV. Note that to find the *latest* version, the server uses the last-writer-wins conflict resolution function that breaks ties by data center IDs as explained in Section II-B. After finding the proper value, the server returns the value together with the list of dependencies of the value, and its DSV in a GETREPLY message. The server also includes the version being returned in the dependency list in Line 4 of Algorithm 3 by calling the same $updateDS$ function as defined in Algorithm 2.

A major improvement in CausalSpartan over GentleRain is providing wait-free PUT operations. Once server p_n^m receives a PUT request, the server updates the version vector $VV_n^m[m]$ to reflect the newly learned clock information. Next, the server creates a new version for the key specified by the client and uses the current $VV_n^m[m]$ value for its timestamp. The server sends back the assigned timestamp $d.ut$ and data center ID m to the client in a PUTREPLY message.

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

Algorithm 4 shows the algorithm for updating DSVs. As mentioned before, partitions inside a data center periodically update their DSV values. Specifically, every θ time, partitions share their VVs with each other and compute DSV as the entry-wise minimum of all VVs (see Line 2 of Algorithm 4). Broadcasting VVs has a high overhead. Instead, we efficiently compute DSV over a tree like the way GST is computed in [15]. Specifically, each node upon receiving VVs of its children computes entry-wise minimum of the VVs and forwards the result to its parent. The root server computes the final DSV, and pushes it back through the tree. Each node, then, updates its DSV upon receiving DSV from its parent. Algorithm 4 also shows the algorithm for the heartbeat mechanism. Heartbeat messages are sent by a server, if the server has not sent any replicate message for a certain time Δ. The goal of heartbeat messages is updating the knowledge of the peers of a partition in other data centers (i.e., updating VVs).

A data store running CausalSpartan protocol presented in this section is convergent and causal++ for conflict resolution function last-writer-wins that breaks ties by data center IDs. For sake of space, the proof of correctness is presented in the Appendix.

VI. EXPERIMENTAL RESULTS

We have implemented CausalSpartan protocol in a distributed kay-value store called MSU-DB. MSU-DB is written in Java, and it can be downloaded from [5]. MSU-DB uses Berkeley DB [3] in each server for data storage and retrieval. For comparison purposes, we have implemented GentleRian in the same code base. We run all of our experiments on AWS [1] on `c3.large` instances running Ubuntu 14.04. The specification of servers is as follows: 7 ECUs, 2 vCPUs, 2.8 GHz, Intel Xeon E5-2680v2, 3.75 GiB memory, 2 x 16 GiB Storage Capacity.

First, in Section VI-A, we investigate the effect of clock skew on PUT latency. Next, in Section VI-B, we evaluate the effect of this increased PUT latency along with query amplification. We analyze the effectiveness of CausalSpartan in reducing update visibility latency by analysis of a typical collaborative applications in Section VI-C. Finally, we evaluate the overhead of CausalSpartan by comparing the throughput of CausalSpartan and GentleRain in Section VI-D in cases where clocks are perfectly synchronized.

Algorithm 3 PUT and GET operations at server p_n^m

1: **Upon** receive $\langle$GETREQ $k, dsv\rangle$
2: $DSV_n^m \leftarrow max(DSV_n^m, dsv)$
3: obtain latest version d from version chain of key k s.t.
- $d.sr = m$, or
- for any member $\langle i, h\rangle$ in $d.ds$, $h \leq DSV_n^m[i]$

4: $ds \leftarrow$ updateDS$(d.sr, d.ut, d.ds)$
5: send $\langle$GETREPLY $d.v, ds, DSV_n^m\rangle$ to client

6: **Upon** receive $\langle$PUTREQ $k, v, ds\rangle$
7: $dt \leftarrow$ maximum value in ds
8: $updateHCL(dt)$
9: Create new item d
10: $d.k \leftarrow k$
11: $d.v \leftarrow v$
12: $d.ut \leftarrow VV_n^m[m]$
13: $d.sr \leftarrow m$
14: $d.ds \leftarrow ds$
15: insert d to version chain of k
16: send $\langle$PUTREPLY $d.ut, m\rangle$ to client
17: **for** each server $p_n^k, k \in \{0 \dots M-1\}, k \neq m$ **do**
18: send $\langle$REPLICATE $d\rangle$ to p_n^k

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

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

Algorithm 4 HEARTBEAT and DSV computation operations at server p_n^m

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

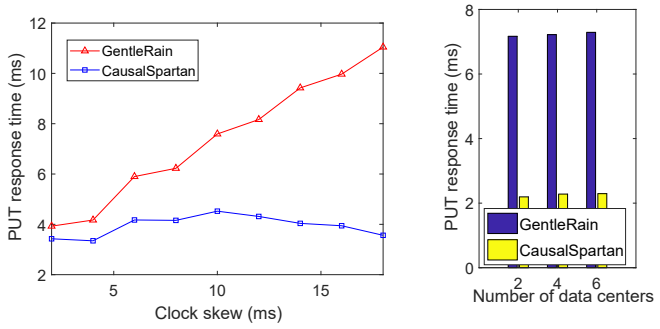
3: **Upon** every Δ time
4: **if** there has not been any replicate message in the past Δ time
5: $updateHCL()$
6: **for** each server $p_n^k, k \in \{0 \dots M-1\}, k \neq m$ **do**
7: send $\langle$HEARTBEAT $HLC_n^m\rangle$ to p_n^k

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

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

A. Response Time of PUT Operations

To study the effect of clock skew on the response time accurately, we need to have a precise clock skew between servers. However, the clock skew between two different machines depends on many factors out of our control. To have a more accurate experiment, we consolidate two virtual servers on a single machine and impose an artificial clock skew between them. Then, we change the value of the clock skew and observe its effect on the response time for PUT operations. A client sends PUT requests to the servers in a round robin fashion. Since the physical clock of one server is behind the physical clock of the other server, half of the PUT operations will be delayed by the GentleRain. On the other hand, CausalSpartan does not delay any PUT operation, and processes them immediately. We compute the average response time for PUT operations with value size 1K. Figure 2-(a) shows that average response time for PUT operation in GentleRain grows as the clock skew grows, while the average response time in CausalSpartan is independent of clock skew.



(a) (b)

Fig. 2. The effect of clock skew on PUT response time: a) with accurate artificial clock skew when servers are running on the same physical machine b) without any artificial clock skew when servers are running on different physical machines synchronized with NTP.

Next, we do the same experiment when the servers are running on two different machines without introducing any artificial clock skew. We run NTP [6] on servers to synchronize physical clocks. In other words, this simulates the exact condition that is expected to happen in an ideal scenario where we have two partitions within the same physical location. In this setting, the client sends PUT requests to these servers in a round robin manner. Figure 2-(b) shows average delay of PUT operations in GentleRain and CausalSpartan. We observe that in this case, the effect of PUT latency is visible even though the servers are physically collocated and have clocks that are synchronized with NTP.

B. Query Amplification

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

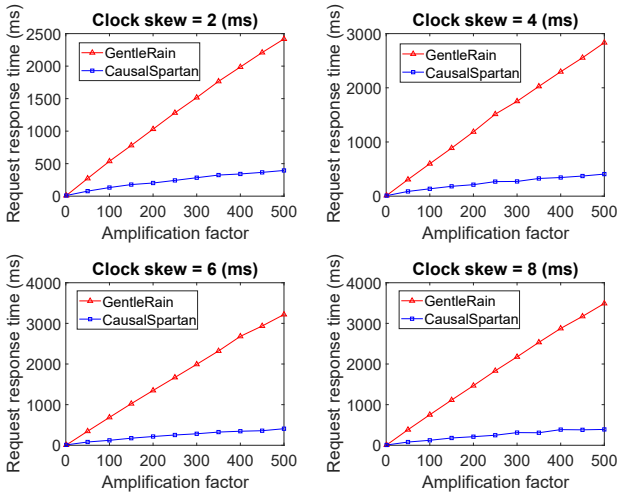


Fig. 3. The effect of different values of clock skew on request response time for different query amplification factor in GentleRain and CausalSpartan.

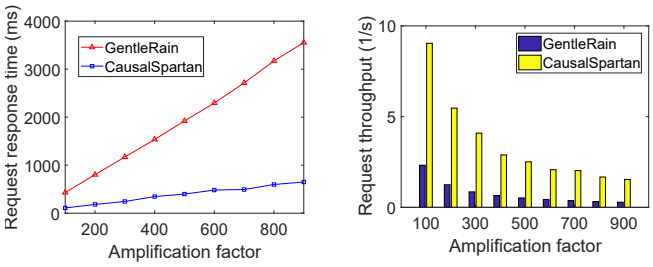
Now, we want to study how the average response time changes as the query amplification factor changes. We simulate the scenario where the user sends requests to a web server, and each request generates multiple internal PUT operations. The web server sends PUT operations to partitions in a round robin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

Figure 3 shows average response time versus query amplification factor, when we have two partitions for different clock skews. As query amplification factor increases, the response time in both GentleRain and CausalSpartan increases. This is expected since each request now contains more work to be done. However, the rate of growth in CausalSpartan is significantly slower. For example, for only 2 (ms) clock skew, the response time of a request with amplification factor 100 in GentleRain is 4 time higher than that in CausalSpartan. Note that in practical systems higher clock skews are possible [22], [23]. For example, clock skew upto 100 (ms) is possible when the underlying netwrok suffers from asymmetric links [23]. In this case, for a query amplification factor of 100, the response time of GentleRain is 35 times higher than CausalSpartan.

For results shown in Figure 3, we used a controlled artificial clock skew to study the effect of clock skew accurately. Figure 4-(a) shows effect of amplification factor on request response time when there is no artificial clock skew, and servers are synchronized with NTP. It shows how real clock skew between synchronized servers that use NTP affects request response time. For instance, for query amplification factor 100, our experiments show that the response time of GentleRain is 3.89 times higher than that of CausalSpartan. Figure 4-(b) also shows the client request throughput in GentleRain and CausalSpartan for different query amplification factor.

C. Update Visibility Latency

In this section, we want to focus on update visibility latency which is another important aspect of a distributed data store. Update visibility latency is the delay before an



(a) Request response time (b) Request throughput

Fig. 4. The effect of amplification factor on client request response time and throughput when we have 8 partitions and 6 data centers, and all partitions are synchronized by NTP without any artificial clock skew.

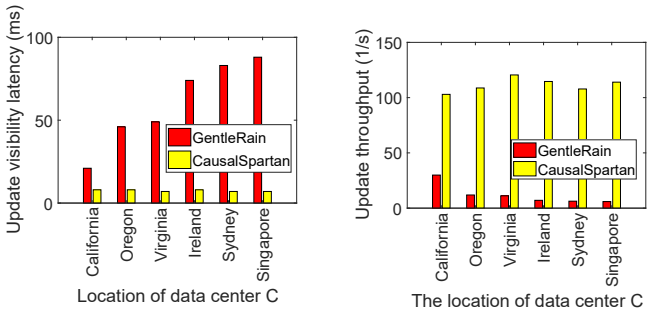
update becomes visible in a remote replica. Update visibility latency is ultimately important for today's cloud services, as even few milliseconds matters for many businesses [10]. In GentleRain, only one slow replica adversely affects the whole communication in the system by increasing the update visibility latency. In CausalSpartan, we use a vector (DSV) with one entry for each data center instead of a single scalar (GST) as used in GentleRain. As a result, a long network latency of a data center only affects the communication with that specific data center, and does not affect independent communication between other data centers.

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

TABLE I. ROUND TRIP TIMES.

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

We measure the update visibility latency as the time elapsed between writing a new update by a client and reading it by another client. We use the timestamp of updates to compute the update visibility latency. Thus, because of clock skew the values we compute are an estimation of actual update visibility latency. Figure 5-(a) shows the update visibility latency is lower in CausalSpartan than that in GentleRain. Also, the update visibility latency in GentleRain increases as the the network delay between data center C and A/B increases. For example, when data center C is in Oregon the update visibility latency in CausalSpartan is 83% lower than that in GentleRain. This value increases to 92% when data center C is in Singapore. Figure 5-(b) shows the throughput of communication between clients as the number of updates by a client per second. The location of data center C affects the throughput of GentleRain, while the throughput of CausalSpartan is unaffected.



(a) Update visibility latency (b) Update throughput

Fig. 5. How the location of an irrelevant data center adversely affects a collaborative communication in GentleRain, while CausalSpartan is unaffected.

D. Throughput Analysis and Overhead of CausalSpartan

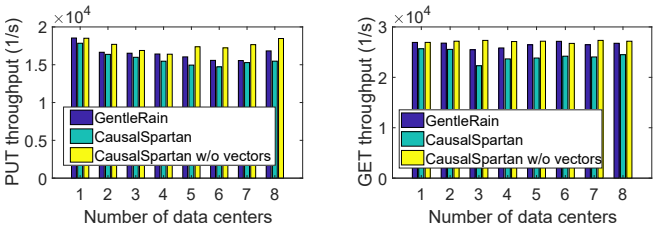
CausalSpartan utilizes HLC to eliminates the PUT latency, and utilizes DSV to improve update visibility latency. In this section, we analyze the overhead of these features in the absence of clock skew, query amplification or collaborative nature of the application. In particular, we analyze the throughput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

Since the two features of CausalSpartan, the use of HLC and the use of DSV are independent, we analyze the throughput with just the use of HLC and with both features. Figure 6 demonstrates the throughput of GET and PUT operations. We observe that when GET/PUT operations are independent, then throughout of CausalSpartan is 5% lower than GentleRain. However, the throughput of CausalSpartan with just HLC (and not DSV) is virtually identical to that of GentleRain. We note that additional experiments comparing CausalSpartan with just using HLC is available in [25]. They show that just using HLC does not add to the overhead of CausalSpartan.

Even though there is a small overhead of CausalSpartan when GET/PUT operations are unrelated, we observe that with query amplification (that causes some PUT operations to be delayed in GentleRain), the request throughput of CausalSpartan is higher (cf. Figure 4). Thus, while the throughput of basic PUT/GET operations is slightly higher in GentleRain, the throughput of actual requests issued by the end users is expected to be higher in CausalSpartan.

VII. IMPOSSIBILITY RESULTS

In this section, we want to focus on the locality of traffic assumption made in our protocol as well as other proposals [14], [15], [20], [21]. The CAP theorem [16] proves that achieving strong consistency, together with availability is impossible under the asynchronous network model. Availability means any client operation must be answered in a finite time. Asynchronous network model means that the communication



(a) PUT (b) GET

Fig. 6. The basic PUT/GET operations throughput in GentleRain and CausalSpartan.

among servers can be arbitrary delayed, and the network can be partitioned for an unbounded duration.

Although achieving strong consistency together with availability is impossible in asynchronous network model, there are several proposals in the literature to guarantee causal++ consistency together with availability under asynchronous network model. However, they assume the locality of traffic. In other words, they assume that a client does not access more than one replica. In this section, we consider the *moving client model* where clients can access different keys from different replicas.

Next, we claim that the locality of traffic assumption is essential for causal++ consistency, as we have the following impossibility result:

***Theorem** 1:* In an asynchronous network with moving client model, it is impossible to implement a replicated data store that guarantees following properties:

- Availability
- Causal++ consistency

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

- There is a partition between r and r', i.e., all future messages between them will be lost.
- c performs $GET(k_1)$ on replica r, and reads v_1^0.
- c performs $PUT(k_1, v_1^1)$ on replica r.
- c performs $PUT(k_2, v_2^1)$ on replica r.
- c' performs $GET(k_2)$ on replica r. Since v_2^1 is a local update, and in causal++ consistency, local updates have to be immediately visible, the value returned is v_2^1.
- c' moves to replica r'.
- c' performed $GET(k_1)$ on replica r'. Because of the network partition, there is no way for replica r' to learn v_1^1. Thus, the value returned is v_1^0.

Since v_1^1 dep v_1^0, for any conflict resolution function f, $f(v_1^0, v_1^1) = v_1^1$. Thus, according to Definition 6, v_1^1 is not visible+ for f to client c'. It is a contradiction to causal++ consistency, as c' has read v_2^1, but its causal consistency v_1^1 is not visible+ to c'. ■

VIII. Related Work

There are several proposals for causally consistent replicated data stores such as [9], [11], [18], [24] where each replicas consists of only one machine. Such assumption is a serious limitation for the scalability of the system, as the whole data must fit in a single machine.

To solve this scalability issue, we can partition the data inside each replica. When we have more than one machine in each replica, an important issue is how we want to make sure all causal dependencies are visible in the replica if some of them reside in other partitions. Some exisiting protocols such as [14], [20], [21] explicitly check dependencies by sending messages to other partitions. This approach suffers from high message complexity.

To eliminate the need for dependency check messages, GentleRain [15] relies on physical clocks of partitions. However, as we discussed in Section III, clock anomalies such as clock skew among servers can adversely affect the performance of systems that rely on synchronized physical clock. CausalSpartan solves this problem by using HLCs instead of physical clocks. The idea of using HLC for providing causal consistency for the first time proposed in our technical report [25]. This report provides the version of CausalSpartan without the DSVs, and provides additional details about the effect of using HLC to provide causal consistency.

Another issue in GentleRain is that the quality of communication between replicas directly affects the update visibility latency even when a replica is irrelevant in a communication. Thus, even one slow replica adversely affects the update visibility latency in the whole system. CausalSpartan solves this problem by keeping track of dependencies written in different data center separately. This idea is also used in Okapi [13]. However, Okapi also suffers from high update visibility latency, as in Okapi an update is not visible until it has been received by all replicas in the system. This constraint makes the update visibility latency of Okapi even higher than that of GentleRain [13].

IX. Conclusion

In this paper, we presented CausalSpartan, a protocol for providing causal consistency for replicated and partitioned key-value data stores. Our protocol is robust to clock anomalies. In particular, CausalSpartan, which is based on GentleRain, ensures that no delays are introduced in GET/PUT operations due to clock synchronization errors. One of the important effects of eliminating this delay occurs in time for query processing when that query results in multiple GET/PUT operations on the key-value store. CausalSpartan guarantees that the response time is unaffected by such clock skew. For example, when the clock skew is 10)ms(, the average response time for PUT operations of CausalSpartan was 4.5 (ms) whereas the average response time of GentleRain was 7.6 (ms). Also, correctness of CausalSpartan is unaffected by

NTP kinks such as leap seconds, non-monotonic clock updates and so on.

This reduction in response time is especially important for federated data centers, virtual data centers, and multi-cloud environment. In federated data centers, the data is created by different entities, and all of partitions may not be physically collocated. Likewise, a virtual data center can be created by utilizing available resources from different cloud service providers to minimize cost. In multi-cloud environment, the data is split intentionally in such a way that no provider has access to the data but the actual data can be accessed only by clients that have access (with proper credentials) to all providers simultaneously. Multi-cloud environments are desired to avoid vendor lock-in. A key characteristic of such data centers is that the partitions in a data center may be geographically distributed. In this case, one has to rely on NTP protocol for clock synchronization. Typical NTP synchronization provides clock synchronization to be within $10ms$. Even at this level of synchronization, CausalSpartan reduces the average response time from 814 (ms) to 124 (ms) (for a query that consists of 100 operations) and from 3800 (ms) to 407 (ms) (for a query that consists of 500 operations). Moreover, NTP clock synchronization errors may be even large (e.g., 100 (ms)). In the case of 100 (ms) clock skew, CausalSpartan reduces the average response time from 4540 (ms) to 124 (ms) (for a query that consists of 100 operations).

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

Finally, we provided an impossibility result which states in presence of network partitions the locality of traffic is necessary, to have an always available causally consistent data store that immediately makes local updates visible. We note that the assumption about client accessing only the local data center is made in several works in the literature [14], [15], [20], [21]. Our argument shows that this assumption is essential in them.

REFERENCES

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

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

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

[4] The future of leap seconds. http://www.ucolick.org/œsla/leapsecs/onlinebib.html.

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

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

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

[8] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] 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.

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

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

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

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

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

[26] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.