GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks

Jiaqing Du Călin Iorgulescu Amitabha Roy Willy Zwaenepoel
EPFL
{firstname.lastname}@epfl.ch

Abstract

GentleRain is a new causally consistent geo-replicated data store that provides throughput comparable to eventual consistency and superior to current implementations of causal consistency.

GentleRain uses a periodic aggregation protocol to determine whether updates can be made visible in accordance with causal consistency. Unlike current implementations, it does not use explicit dependency check messages, resulting in a major throughput improvement at the expense of a modest increase in update visibility. Furthermore, GentleRain tracks causal consistency by attaching to updates scalar timestamps derived from loosely synchronized physical clocks. Clock skew does not cause violations of causal consistency, but may delay the visibility of updates. By encoding causality in a single scalar timestamp, GentleRain reduces storage and communication overhead for tracking causality.

We evaluate GentleRain using Amazon EC2, and demonstrate that it achieves throughput equal to about 99% of eventual consistency, and 120% better than previous implementations of causal consistency.

Categories and Subject Descriptors C.2.4 [*Computer Systems Organization*]: Distributed Systems

Keywords Causal Consistency, Distributed Consistency, Key Value Stores, Geo-replication

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '14, November 03 - 05 2014, Seattle, WA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3252-1/14/11-1. . . $15.00.
http://dx.doi.org/10.1145/2670979.2670983

1. Introduction

Distributed data stores are a critical infrastructure component of many large-scale online services. To provide performance and availability, these data stores are often geo-replicated. A critical decision in geo-replication is the choice of a consistency model. At one end, strong consistency [16] provides simple semantics, but suffers from long latencies and does not tolerate network partitions. At the other end, eventual consistency provides excellent performance and tolerates partitions [29], but renders the programming model more complicated. Recent work also considers the possibility of multiple consistency models in simultaneous use [21, 28].

Causal consistency [2] is an attractive model for constructing geo-replicated data stores. The causality relation is the transitive closure of the order of events within a single client and the reads-from relation between different clients [20]. A causally consistent store guarantees that an update does not become visible until all its causal dependencies are visible. Causal consistency is attractive, because it avoids the long latencies and partition-intolerance associated with strong consistency, yet it also avoids some of the anomalies possible with eventual consistency.

Large data stores furthermore partition the data at each datacenter in order to scale to very large data sets. Recent papers [12, 23, 24] have shown how to implement causal consistency in a replicated partitioned data store without incurring the serialization bottleneck of going through a single log. Roughly speaking, both the client and the data store track causal dependencies. Updates are replicated asynchronously, and the causal dependencies recorded for an update are sent along with the update replication message. At a remote datacenter, the recipient verifies that all causal dependencies are present by sending dependency check messages to other partitions. Once these checks complete, the new version is installed. Unfortunately, the exchange of potentially many dependency check messages can cause throughput to degrade compared to eventual consistency.

GentleRain presents a different design for a causally consistent key value store with the following features:

- GentleRain eliminates dependency check messages for updates.
- GentleRain uses only a single physical timestamp to track dependencies.

These features distinguish GentleRain from other causally consistent key value stores such as COPS [23], which explicitly tracks individual dependencies, and Orbe [12], which tracks dependencies using dependency matrices. Eliminating dependency check messages altogether improves throughput in comparison to the other two systems. Using only a single timestamp allows for a far more concise representation of dependencies, and therefore a reduction in storage and communication overheads. GentleRain therefore provides a new and hitherto unexplored design point in the construction of causally consistent data stores.

While achieving better throughput and reducing storage and communication overhead, GentleRain incurs longer latencies in making updates remotely visible. Updates become visible immediately at the originating datacenter - as in other systems - but remotely they incur a slightly longer visibility delay.

The contributions of this paper are:

- The design and implementation of a causally consistent data store that has throughput comparable to eventual consistency, with a modest increase in remote update visibility latency
- An implementation of a causally consistent key-value store that encodes dependency information as a single scalar, both in terms of storage and transmission
- The evaluation of these contributions in the context of a key-value store geo-replicated on Amazon EC2
- An exploration of the tradeoff between the throughput and the remote update visibility latency of causally consistent data stores

The outline of the rest of this paper is as follows. Section 2 motivates our approach by showing the origin of the overheads of previous causally consistent data store implementations. Section 3 describes the system model. Section 4 presents GentleRain's core protocol. Section 5 provides a proof sketch and some discussion of various aspects of GentleRain's protocol. We evaluate the performance of GentleRain in Section 6. We discuss related work in Section 7, and conclude in Section 8.

2. Motivation

We demonstrate with a simple experiment the effect of dependency check messages on the throughput of current causal consistency implementations.

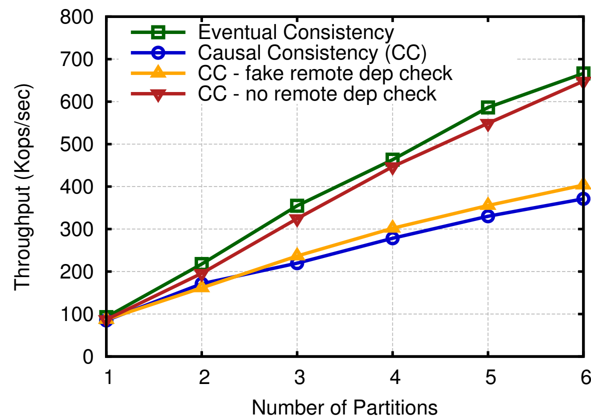

Figure 1. Throughput of a distributed data store with eventual and causal consistency. Each partition is replicated by three replicas. Clients read an item from each partition and update an item at one partition.

To do so, we compare the throughput of causal and eventual consistency by an experiment in a distributed key-value store. The data store provides single-item read and write operations. It implements causal consistency as in COPS [23] and Eiger [24], tracking only the nearest dependencies, because of the transitivity of causality. For eventual consistency, the implementation does not track dependencies and makes updates visible as soon as they arrive. A client reads a random item from each partition and updates one random item at a randomly selected partition. The update is then propagated to replicas at remote datacenters. This workload stretches the causal consistency implementation, since it creates dependencies across all partitions for each update operation. While worst-case, workloads with a large number of dependencies across different partitions may not be rare in real world applications [4]. For instance, the default page of a user of Twitter or Facebook loads at least dozens of or even hundreds of different states. Any subsequent updates via the page, such as commenting on other users' posts, causally depends on all the displayed states.

Figure 1 shows the throughput of causal and eventual consistency. As more partitions are added to the system, the performance gap between the two becomes larger. For causal consistency, dependency check messages are the main reason for the performance degradation. We demonstrate this fact by performing the following additional experiments. In a first experiment, we remove from the code implementing causal consistency any sending or receiving of messages that check dependencies. In a second experiment, we leave the sending and receiving of dependency check messages in place, but we do not perform any computation or waiting as a result of receiving these messages. The resulting throughput is shown in Figure 1 by the curve "no remote dependency check" for the first experiment, and by the curve

"fake remote dependency check" for the second experiment. Not sending or receiving any dependency check messages results in throughput almost identical to eventual consistency. Sending and receiving the messages, but not acting on them, leaves throughput at values comparable to causal consistency. The conclusion is then clear: if causal consistency is to achieve throughput comparable to eventual consistency, we must find a way to implement causal consistency without the exchange of dependency check messages between partitions.

3. Definition and Model

3.1 Causality

Causality is the *happens-before* relationship between two events [2, 20]. We denote causal order by $\rightsquigarrow$. For two operations a and b, if $a \rightsquigarrow b$, we say b depends on a or a is a dependency of b. $a \rightsquigarrow b$ if and only if one of the following three rules holds:

- Thread-of-execution. a and b are operations in a single thread of execution, and a happens before b.
- Reads-from. a is a write operation, b is a read operation, and b reads the value written by a.
- Transitivity. There is some other operation c such that $a \rightsquigarrow c$ and $c \rightsquigarrow b$.

A version X of a data item x is causally dependent on a version Y of data item y if the write of X causally depends on the write of Y. A store is causally consistent if, when a certain version of a data item is visible to a client, then all of its causal dependencies are also visible.

3.2 Architecture and Interface

We assume a distributed key-value store that manages a large set of data items. The system is partitioned into N partitions and each partition is replicated by M replicas. A data item is assigned to a partition based on the hash value of its key. In a typical configuration, as shown in Figure 2, the system runs at M different datacenters. Each datacenter runs N partitions. Hence, a full copy of the data is stored at each datacenter.

We assume a multiversion data store. An update operation creates a new version of an item. In addition to the actual value of the key, each version also stores some metadata, in order to track causality. The system periodically garbage-collects old versions of items.

A server is equipped with a physical clock, which provides monotonically increasing timestamps. Clocks are loosely synchronized by a time synchronization protocol, such as NTP [1]. The correctness of our system does not depend on the synchronization precision.

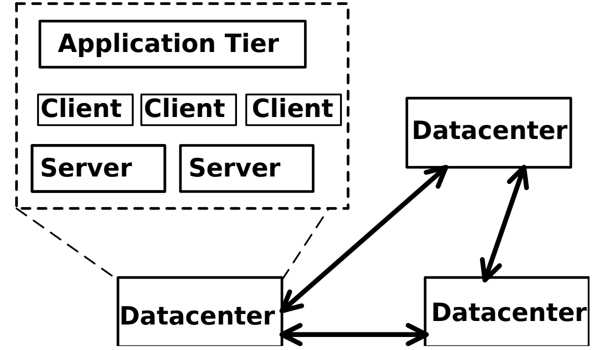

Figure 2. System architecture. The data set is replicated by multiple datacenters. Clients are co-located with the data store in the datacenter and are used by the application tier to access the data store.

Our distributed key-value store provides the following operations to the clients:

- PUT(key, val): A PUT operation assigns value *val* to an item identified by key. If item key does not exist, the system creates a new item with initial value val. If key exists, then a new version storing val is created.
- val $\leftarrow$ GET(key): The GET operation returns the value of the item identified by key.
- $\langle$vals$\rangle$ $\leftarrow$ GET-SNAPSHOT$\langle$keys$\rangle$: This operation returns the values of a group of items from a snapshot of the data store. A *causally consistent snapshot* satisfies the following property: For any two items x and y, if X, a version of x, and Y, a version of y, belong to the same consistent snapshot, then there does not exist X', another version of x, such that 1) X' is created after X, and 2) $X' \rightsquigarrow Y$. The datastore is free to return a snapshot from any point in the past. *It can therefore exclude values that have been already read by the client before executing the snapshot read.*
- $\langle$vals$\rangle$ $\leftarrow$ GET-ROTX$\langle$keys$\rangle$: This operation provides a causally consistent read-only transaction [23, 24]. It has the same semantics as a snapshot but, in addition is constrained to include any values previously read by the client.

While the `GET` and `PUT` operations are relatively standard, the snapshot is a weaker form of the causally consistent read-only transactions provided by systems such as Orbe [12] and COPS [23]. Relaxing the requirement that previously seen updates be part of the snapshot returned by a causally consistent read-only transaction allows the snapshot to be implemented more efficiently than read-only transactions, while continuing to serve many of the same purposes. For example, consider the case of a photo album in a social network. A user might change the album properties to private and then add photos to it. A client, by using a snapshot read

Symbols	Definitions
N	number of partitions
M	number of replicas per partition
DT_c	dependency time at client c
GST_c	global stable time known by client c
p_n^m	server, the mth replica of the nth partition
$Clock_n^m$	current physical clock time at p_n^m
VV_n^m	version vector of p_n^m, with M elements
LST_n^m	local stable time at p_n^m
GST_n^m	global stable time at p_n^m
d	item tuple $\langle k, v, \text{ut}, \text{sr}\rangle$
k	key
v	value
ut	update time
sr	source replica id

Table 1. Definition of symbols.

of the album properties and contents, ensures that it does not read the album as shared and then accesses private photos. This can be achieved both by read-only transactions as well as by read snapshots.

4. GentleRain Protocol

The GentleRain protocol timestamps all updates with the physical clock value of the server where they originate (the source server). Concatenated with their partition and replica identifiers, these timestamps provide a total order on all updates. The protocol guarantees that this total order is consistent with the causal order of events.

We distinguish between, on the one hand, updates that have been received at a server, and, on the other hand, updates that have been made visible to clients. The protocol guarantees that updates are made visible only if that can be done in accordance with causal consistency. *Local updates are always immediately visible*. Updates originating at remote datacenters become visible when their update timestamp is smaller than the global stable time (GST), defined below. The use of physical clock values as update timestamps, rather than logical clock values, is instrumental in making sure that the global stable time makes suitable progress, and remote updates become visible in a short amount of time.

4.1 States

Table 1 provides a summary of the symbols used in the protocol.

Client States. A client c maintains for its session a dependency time DT_c. This value is the maximum update timestamp of all items accessed so far by the client session. A client c also maintains in GST_c the global stable time that it is aware of.

Algorithm 1 Client operations at client c

```
1:  GET(key k)
2:      send ⟨GETREQ k, GST_c⟩ to server
3:      receive ⟨GETREPLY v, ut, gst⟩
4:      DT_c ← max(DT_c, ut)
5:      GST_c ← max(GST_c, gst)
6:      return v
7:  PUT(key k, value v)
8:      send ⟨PUTREQ k, v, DT_c⟩ to server
9:      receive ⟨PUTREPLY ut⟩
10:     DT_c ← max(DT_c, ut)
```

Server States. Each server p_n^m maintains a *version vector* VV_n^m [2, 26], consisting of M physical timestamps from updates seen at that server. $VV_n^m[m]$ is the largest update timestamp of any update originating at p_n^m. Similarly, p_n^m has received all updates with timestamp up to $VV_n^m[i]$ $(i \neq m)$ from p_n^i, replica i of the same partition located at datacenter i.

We define the *local stable time* LST_n^m at a server p_n^m as the minimum element of its VV_n^m. Updates originating at any replica p_n^i of p_n^m with an update timestamp smaller than or equal to LST_n^m have been received at p_n^m.

We define the *global stable time* GST_n^m at a server p_n^m as a lower bound on the minimum LST of all partitions within the same datacenter. Partitions in the same datacenter periodically compute GST, by means of a protocol described in Section 4.5. This quantity can vary across partitions but never exceeds the minimum LST across all partitions in the datacenter.

Item Version. For each item uniquely identified by its key, there exists a chain of one or more item versions. We represent an item version d as a tuple $\langle k, v, ut, sr\rangle$. k is a unique key that identifies the item. v is the value of the item. ut is the *update time*, the creation time of the item at its source server. sr is the *source replica*, the replica id of the item's source server.

4.2 Protocol

We now describe how our protocol executes GET and PUT operations from clients and replicates PUT operations while preserving causality. Algorithms 1 and 2 show the pseudocode of the protocol running at the client and server side, respectively.

GET. A client sends a GET request with an item key and its GST_c to the server that serves the partition containing the item. The server first updates its GST if it is smaller than the client's. The server then obtains the latest version in the version chain of the requested item, which is either created by clients attached to the local datacenter or has an

Algorithm 2 Server operations at server p_n^m

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

5: **upon** receive $\langle$PUTREQ $k, v, dt\rangle$
6: $\quad$ wait until $dt < Clock_n^m$
7: $\quad$ update version vector: $VV_n^m[m] \leftarrow Clock_n^m$
8: $\quad$ create new item d
9: $\quad$ set key: $d.k \leftarrow k$
10: $\quad$ set value: $d.v \leftarrow v$
11: $\quad$ set update time: $d.ut \leftarrow VV_n^m[m]$
12: $\quad$ set source replica: $d.sr \leftarrow m$
13: $\quad$ insert d to version chain of key k
14: $\quad$ send $\langle$PUTREPLY $d.ut\rangle$ to client

15: **upon** new version d created
16: $\quad$ **for** each server p_n^k, $k \in \{0..M-1\}, k \neq m$ **do**
17: $\quad\quad$ send $\langle$REPLICATE $d\rangle$ to p_n^k

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

21: **upon** every Δ time
22: $\quad ct \leftarrow Clock_n^m$
23: $\quad$ **if** $ct \geq VV_n^m[m] + \Delta$ **then**
24: $\quad\quad VV_n^m[m] \leftarrow ct$
25: $\quad\quad$ **for** each server p_n^k, $k \in \{0..M-1\}, k \neq m$ **do**
26: $\quad\quad\quad$ send $\langle$HEARTBEAT $ct\rangle$ to p_n^k

27: **upon** receive $\langle$HEARTBEAT $ct\rangle$ from p_n^k
28: $\quad VV_n^m[k] \leftarrow ct$

29: **upon** every θ time
30: $\quad LST_n^m \leftarrow min(VV_n^m)$
31: $\quad GST_n^m \leftarrow min(\{LST_k^m \mid 0 \leq k \leq N-1\})$

update timestamp no greater than the partition's GST. Hence, a client always reads local updates without any delay and replicated updates from other datacenters once they are globally stable. The partition returns the item value, its update timestamp, and its GST back to the client. Upon receiving the reply, the client sets its dependency time DT_c to the returned item update timestamp if the latter is larger than its current value. It also updates its global stable time GST_c to the returned global stable time if the latter is larger than its current value.

PUT. A client sends a PUT request, $\langle$PUTREQ $k, v, DT_c\rangle$, which includes the item key, the update value, and the client's dependency time, to the server that manages the item. The server then checks that the client's dependency time is smaller than its physical clock time. In the highly unlikely case that it is not, it waits until the condition becomes true.

The server then updates the local element of its version vector (m^{th} element of VV_n^m at server p_n^m) with its physical clock time. It creates a new version of the item by assigning it a tuple that consists of the key, value, update time, and its replica id, and inserts the newly created item version in the version chain of the item. The server sends a reply with the update time of the newly created item version to the client. Upon receiving the reply, the client sets its dependency timestamp to the returned item update timestamp if the latter is larger than its current value.

Update replication. After a server executes a local update, it replicates it asynchronously by sending it in update timestamp order to its replicas at the other datacenters. When a server receives such an update replication message, it inserts the received item version in the corresponding version chain. However, this update is not visible to local clients until the partition's GST becomes larger than its update timestamp.

GST derivation. Each server periodically computes the GST, which is the minimum of the LSTs of all partitions within the same datacenter. We explain how to aggregate the LSTs of all partitions, compute the GST, and distribute it efficiently in Section 4.5.

Heartbeats. If a partition does not receive update requests from clients, GST may not increase. To solve this problem, a partition that does not have frequent local updates periodically (at time interval Δ) broadcasts its latest clock time to its replicas. It does so by piggybacking the clock time in the heartbeat messages used by failure detectors. Heartbeat messages and update replication messages are sent in order of increasing update timestamps and clock values.

4.3 Reads of Multiple Items

GentleRain also supports snapshot reads and causally consistent read-only transactions as defined in Section 3.

GET-SNAPSHOT. Algorithm 3 shows the pseudo-code of snapshot reads. A client sends a snapshot read request, which includes a set of item keys and its GST to a coordinating partition, selected based on some load balancing algorithm. The coordinating partition first updates the partition's GST if it is smaller than the client's. It then initializes the *snapshot timestamp* of the snapshot read using the latest GST.

Using the snapshot timestamp, the snapshot read chooses a proper version of each requested item at the partition that manages the item. The latest item version with an update timestamp smaller than the snapshot timestamp is returned. The item value, update timestamp, and the GST of the accessed partition are sent back to the coordinating partition.

Algorithm 3 Snapshot Reads

```
1:  // at client c
2:  GetSnapshot(keys kset)
3:      send ⟨GETSNAPSHOT ks, GST_c⟩ to server
4:      receive ⟨GETSNAPSHOTREPLY vset, ut, gst⟩
5:      DT_c ← max(DT_c, ut)
6:      GST_c ← max(GST_c, gst)
7:      return vset
8:  // at partition p_n^m
9:  upon receive ⟨GETSNAPSHOT kset, gst⟩
10:     GST_n^m ← max(GST_n^m, gst)
11:     vset ← φ, ut ← 0, gst ← 0
12:     st ← GST_n^m
13:     for each key k ∈ kset do
14:         send ⟨SLICEREQ k, st⟩ to server
15:         receive ⟨SLICEREPLY v, ut', gst'⟩
16:         vset ← vset ∪ {v}
17:         ut ← max(ut, ut')
18:         gst ← max(gst, gst')
19:     send ⟨GETSNAPSHOTREPLY vset, ut, gst⟩ to client
20: // at partition p_n^m
21: upon receive ⟨SLICEREQ k, st⟩
22:     GST_n^m ← max(GST_n^m, st)
23:     obtain latest version d from version chain of key k
        s.t. d.ut ≤ st
24:     send ⟨SLICEREPLY d.v, d.ut, GST_n^m⟩ back
```

Algorithm 4 Causally Consistent Read-Only Transactions

```
1:  // at client c
2:  Get-ROTX(keys kset)
3:      send ⟨GETROTX ks, GST_c, DT_c⟩ to server
4:      receive ⟨GETROTXREPLY vset, ut, gst⟩
5:      DT_c ← max(DT_c, ut)
6:      GST_c ← max(GST_c, gst)
7:      return vset
8:  // at partition p_n^m
9:  upon receive ⟨GETROTX kset, gst, dt⟩
10:     if dt − GST_n^m ≤ α then
11:         wait until dt < GST_n^m
12:         run snapshot reads in Algorithm 3
13:     else
14:         run read-only transaction protocol in Eiger [24]
```

The coordinating partition returns the collected item values with the maximum update timestamp and the maximum GST back to the client. Upon receiving the reply, the client updates its dependency time and GST, as before.

GET-ROTX.

Algorithm 4 shows the pseudo-code of causally consistent read-only transactions. Such a transaction returns values that form a read snapshot, but in addition it must also guarantee that this snapshot includes any values previously seen by the client.

This latter condition may not hold in a situation in which a client reads a version of an item written by a client in the same datacenter, and then performs a snapshot read including that item. If the GST of the server hosting that item at the time of the snapshot is smaller than the update timestamp of that version, then the snapshot would return an earlier version. To avoid this scenario, our protocol for read-only transactions takes one of two approaches. On the one hand, if the dependency time of the client does not exceed the *GST* by more than some threshold α, then we simply block the transaction to allow the GST to advance past the dependency time of the client. We then execute the snapshot read protocol. On the other hand, if the dependency time is larger than the *GST* by an amount that exceeds the threshold α, then we fall back on using the protocol used in Eiger [24] for causally consistent read-only snapshots. Compared with our snapshot read protocol, this protocol may require two rounds to terminate.

4.4 Conflict Detection

A conflict happens when two causally unrelated updates to the same key are done at two different replicas. In GentleRain, we detect such conflicts and handle them by calling up to the application that must then tell GentleRain (*in a consistent manner at all servers*) how to order the conflicting updates. To detect conflicts we use a similar technique to that used in COPS [23]. Each update that needs to be replicated also carries the update time and source replica id of the previous version of the item in the version chain. Before applying a propagated update, a server checks whether the previous version in the chain is the same as the previous version attached to the incoming update. If these are different, then a conflict has occurred and the application must decide whether to retain the incoming version or drop it as it has been superseded by a later version. Conflict detection does not flag causally ordered updates to the same key as conflicting, because the causally previous update must have arrived at *all replicas* before a client can install a causally later version at the local replica.

4.5 Efficient Global Stable Time Derivation

Servers within the same datacenter periodically exchange their LSTs to compute their GST. If the number of partitions is large, exchanging LSTs by simply broadcasting them is too expensive and not scalable. Although, with broadcasting GST is derived and distributed to each partition in one round-trip network latency within a datacenter, the message complexity of broadcasting is $O(N^2)$. If we assume there are a thousand servers and GST is computed every 10ms, each server sends and receives 100K messages per second, which

is not affordable in practice. To efficiently derive the GST at each server, we build a tree over all servers in the same datacenter and compute an aggregate minimum using the tree. This process has been used to solve similar problems such as aggregating state from distributed graph computation [25] or in sensor networks [15].

During initialization, all servers are given the fanout of the tree and a list of servers sorted based on their partition id. A server then finds its parent and children nodes from the list and sets up a TCP connection to each of them. Leaf nodes of the tree periodically push their LSTs to their parent nodes. Once an internal node receives LSTs from all its children, it computes the minimum and pushes it to its parent node. The root node obtains the GST of the round and pushes it down the tree. The message complexity is $O(N)$. Each round of GST computation takes $2 * \log_F N$ round trips in the datacenter, where F is the fanout of the tree.

Using a tree for GST computation brings message counts down to acceptable levels. Continuing the example above, a tree with fanout of five and depth of six can cover almost 20K nodes, well above one thousand servers in the example. Computing the GST every 10ms means that each server only sends and receives one message every 10 ms on every one of its six links in the tree (five children and one parent). This works out to sending and receiving 600 messages per second rather than 100K messages per second.

At the same time, the tree enables computing and distributing the GST in a reasonable amount of time. This is important because a remote update becomes visible after it has been received by the local datacenter and the GST computation takes into account the increase in the LST corresponding to the timestamp of the arrived update. In the same example, if we assume that the link latency within the datacenter is 0.1ms then propagating values from the leaf to the root and back again represents the worst case for latency and works out to 1.2ms. In contrast, the latency between datacenters is usually of the order of about a 100 milliseconds. Therefore GST computation adds only a marginal delay to update propagation.

5. Discussion

5.1 Why Physical Clocks?

The GentleRain protocol continues to provide causal consistency if the loosely synchronized physical clocks are replaced with Lamport clocks (or vectors, such as in [2] or Orbe [12]) that are incremented on receiving updates from clients and replication messages from other servers. This, however, can cause clocks at different servers to move at very different rates if they also receive updates at different rates. Hence, the visibility of propagated updates from replicas can be arbitrarily delayed as GST always tracks the minimum element across all version vectors in the datacenter. Using loosely synchronized physical clocks together with the aggregation protocol allows us to keep GentleRain's ability to scale across partitions and datacenters, while avoiding the problem of large differences between clocks at different partitions.

5.2 Throughput vs. Latency Tradeoff

As we have shown in Section 2, in the current implementations of causal consistency, dependency check messages are a major source of overhead, and explain the difference in throughput between causal and eventual consistency. In GentleRain we have eliminated those dependency messages, and replaced them with a tree aggregation protocol for computing the minimum timestamp and (in the absence of regular updates at a partition) a heartbeat protocol. Both these protocols are parameterized by the time interval at which they are invoked. As we show in Section 6, with reasonable choices of such intervals, throughput is much better than current implementations of causal consistency and approximates that of eventual consistency.

This better throughput comes at the price of an increase in update visibility latency for remote updates. Local updates are visible immediately, both in current implementations and in GentleRain. In current implementations, the latency before an update becomes visible at a remote datacenter is the sum of the network travel time from the origin to the remote datacenter plus the time to exchange the dependency check messages, if any. Since the latter are local to a datacenter, we assume in first approximation that remote update visibility is equal to the travel time to the remote datacenter. In GentleRain, the latency before an update becomes visible at a remote datacenter is the sum of a number of factors. First, there is the network travel time from the furthest removed datacenter, because this is the longest time it takes for an update or a heartbeat with a particular value to arrive. Second, there is the clock skew between those two datacenters. Third, there is the interval at which the aggregation protocol runs, and, finally, in the absence of regular updates, there is the interval at which the heartbeat protocol runs. Assuming expected values for these component times, we can in first approximation conclude that the remote update visibility latency is equal to the network travel time to the furthest removed datacenter.

We argue that this is a reasonable tradeoff for increased throughput because the maximum latency between datacenters is usually under 270ms (for a satellite link) and this latency is tolerable for applications such as social networks. If increased remote update visibility is a consideration, the GentleRain protocol can be modified to, instead of a single scalar, maintain a vector of size the number of datacenters as dependency information. Essentially, rather than computing LST_n^m, the minimum of VV_n^m, we maintain the entire vector,

and use that to compute an element-wise minimum over all partitions in a datacenter. Similarly, a vector of dependencies is maintained by the client and with each data item, and exchanged between clients and servers. With this modified design, the computation and storage overhead increase, but the latency is reduced, in first approximation, to the network travel time to the originator of the update. We have not yet experimented with this modified design.

5.3 Garbage Collection

We briefly describe how to garbage-collect old item versions to keep the storage footprint small. Partitions within the same datacenter periodically exchange global snapshot timestamps of the oldest active snapshot read. If a partition does not have any active snapshot read, it sends out its GST. At each round of garbage collection, a partition chooses the minimum among the received timestamps as the *safe garbage collection timestamp*. With this timestamp, a partition scans the version chain of each item it stores. It only keeps the latest item version created before the safe garbage collection timestamp (if there is one) and the versions created after the timestamp. It removes all the other versions as these are not needed by active and future snapshot reads.

The computation of the safe garbage collection timestamp can be done efficiently using the same techniques as used for computing GST.

5.4 Correctness

We provide an informal proof sketch that GentleRain provides causal consistency. First, local updates can be made visible locally, because they have read and written their dependencies locally, and so they must be visible. Second, for remote updates, we first demonstrate the correctness of GentleRain in the absence of the heartbeat messages. This is done by demonstrating two supporting propositions, which are then used to derive the overall correctness argument. We conclude by showing that the heartbeat messages are correct optimizations.

Proposition 1: If an update $u1$ depends on an update $u2$, then $u2.ut < u1.ut$.

By lines 4 and 8 of Algorithm 1, the client puts in the PUTREQ the largest update timestamp value of any dependency it incurred as a result of a previous GET or PUT. By lines 6, 7, and 11 of Algorithm 2, the update timestamp of an update is always at least as large as the dependency time dt passed in the PUTREQ. It follows that the new update has an update timestamp larger than any of its dependencies.

Proposition 2: When, at some partition p_n^m, GST_n^m has a certain value T, then all partitions p_i^m $(i = 0..N)$ have received all updates with update timestamp less than or equal to T.

This is shown by contradiction. Suppose there is an update u originating in partition p_i^j $(j \neq m)$ with $u.ut < T$, and that update has not been received by partition p_i^m. Since update replication messages arrive in update timestamp order, and by line 20 of Algorithm 2, $\text{VV}_i^m[j] < T$, thus $\text{LST}_i^m < T$, and $\text{GST}_n^m < T$, leading to a contradiction.

Correctness in the absence of heartbeat messages.

We wish to show that the following proposition holds. Assume data item x resides at partition i and data item y resides at partition j. If a client receives as a result of a GET(x) from partition p_i^m a version X of x, and if that version X of x depends on the version Y of y, then if that same client performs a GET(y) to partition p_j^m, it receives in response a version of y created no earlier than Y, and that version is available from p_j^m without blocking.

The phrase "no earlier than" is to be interpreted in terms of the total order imposed on all updates by the update timestamps (physical clock values followed by replica and partition identifiers to break ties).

Assume a client performs a GET(x) on partition p_i^m, and receives as a result a version X of x with update timestamp $X.ut$. Let T be the value of GST_i^m at p_i^m at the time the GET is performed. Then, it follows, by line 3 of Algorithm 2, that $X.ut < T$, and, by line 5 of Algorithm 1, that $T \leq GST_c$.

If that version X of x depends on a version Y of y, then by Proposition 1 it must be that $Y.ut < X.ut$, and therefore also that $Y.ut < T$. Then, by Proposition 2, the version of Y of y must have been received at p_j^m.

Assume finally that the same client performs a GET(y) on partition p_j^m, then by line 2 of Algorithm 2, $GST_c \leq GST_j^m$, and thus also $Y.ut < T \leq GST_j^m$. By line 3 of Algorithm 2, it follows that GET(y) returns a version of y with update timestamp no smaller than $Y.ut$.

Correctness in the presence of heartbeat messages.

The heartbeat messages are an optimization that causes the GST values to move forward when there is no local update in a particular partition. We conclude by showing that this optimization is correct. A heartbeat message from p_i^m to p_n^m informs p_n^m of the fact that there are no updates from p_i^m with update timestamp smaller than the heartbeat value ct. By line 28 of Algorithm 2, $VV_n^m[i] = ct$, and by line 30 of Algorithm 2 $LST_n^m = min(VV_n^m)$, the invariant is maintained that p_n^m has received all updates with update timestamp smaller than LST_n^m.

Correctness of snapshot reads.

All updates across different partitions and replicas are totally ordered by their update timestamps. By Proposition 2, all partitions have received all updates with update timestamp less than or equal to the GST. We assign the latest GST to the

operation as its snapshot timestamp. Therefore, by choosing the item version with the largest timestamp that is not greater than the snapshot timestamp, we provide a snapshot that is causally consistent as defined in Section 3.

Correctness of read-only transactions.

By virtue of the fact that the update timestamp order is a total order that reflects the partial causal order, the value of the snapshot timestamp provides a value that divides all updates causally before and causally after that timestamp. By choosing the version with the largest timestamp value before the snapshot timestamp and delaying the transaction briefly (line 11 of Algorithm 4), we return versions that form a causally consistent snapshot as defined in Section 3.

5.5 Failures and Laggards

GentleRain is designed to continue causally consistent key value service even in the presence of machine and network failures or in the presence of slow machines (laggards). *In particular it never violates causal consistency in response to a query.*

Machine failures, network partitions and machine slowdowns all have a single consequence in GentleRain: *GST* at one or more datacenters does not make adequate progress. By design GentleRain only delays the visibility of remote updates in such a case and not local updates, whose visibility is independent of *GST*. Geo-replicated datastores are usually built under the assumption of locality of traffic to a datacenter to better serve users in a geographical region. GentleRain isolates the exchange of information among clients local to a datacenter from the above mentioned problems.

Further, GentleRain assumes that servers in a datacenter are themselves backed up by strongly consistent replicas within the same datacenter. This means that GST computation can only be stalled on failure of a single server by the amount of time needed to switch over to another local replica.

This leaves network failures both within and across datacenters. Datacenters themselves often contain redundancy in their networks, and this redundancy can be further improved by adequately designing the network topology [22]. This leaves GentleRain only vulnerable to complete partitions between datacenters. If a datacenter gets disconnected from the rest, no remote updates can be visible at *any datacenter* due to GST not making progress. Such datacenter partitions are usually rare and in GentleRain do not impact visibility of local updates within a datacenter. However, we can handle such complete datacenter partitions by excluding the partitioned datacenter from the computation of GST. We also exclude any updates from the disconnected datacenter that have been seen after the GST value where it was deemed to be partitioned, to avoid the situation where a value has arrived without its dependencies. This exclusion is possible since we have a total order on updates given by the single scalar timestamp. This failure handling protocol requires us to maintain a group [17] of connected datacenters. Designing and implementing this protocol to dynamically size the group of datacenters over which GST is computed is a focus for future work in GentleRain.

6. Evaluation

We evaluate GentleRain in terms of local latencies, throughput, and remote update visibility latency, by varying the workloads and the number of partitions. We compare GentleRain to Eventual Consistency (EC) and a current implementation of causal consistency (COPS).

6.1 Implementation and Setup

We implement GentleRain in `C++` and use Google's Protocol Buffers for message serialization. We also implement eventual consistency (EC) and COPS in the same code base for performance comparison. The GentleRain implementation requires about 20k lines of code, compared to 21k lines for EC and COPS. Our implementation of COPS only tracks one-hop nearest dependencies. It does not provide read-only transactions, which requires tracking complete dependencies. In this mode, without support for transactions, the COPS protocol is identical to Eiger [24], which is a later protocol from the same authors. Also, this places COPS in a more favorable position for comparison with GentleRain.

The data store is a key-value store, partitioned over a group of servers using consistent hashing [18]. The data set used in the experiments contains one million key-value pairs per partition, with the key size being eight bytes and the value size 64 bytes. The key-value store keeps all key-value pairs in main memory. A key points to a linked list that contains different versions of the same item. The operation log resides on disk. The system performs group commit to write multiple updates in one disk write. A PUT operation inserts a new version to the version chain of the updated item and adds a record to the operation log.

We run NTP to keep physical clocks synchronized [1]. NTP can be configured to either change the clock value or change the clock frequency to catch up or fall back to a target. We configure it to change clock frequency, so that our physical clocks always move forward, a requirement for correctness in GentleRain.

We run all experiments on Amazon EC2 using `c3.large` instances running Ubuntu 12.04. Each server has two virtual CPU cores, 3.75 GB memory, and 2 x 16GB SSD storage. We replicate each partition at three EC2 datacenters: one on the US east coast (Virginia), one on the US west coast (Oregon), and one in Europe (Ireland). We measure the following average latencies between these datacenters: Virginia to

Op/Bytes	Echo/-	GET/10	PUT/16	PUT/128
GentleRain	65.3	57.8	33.4	30.1
EC	65.3	59.2	33.9	30.4
COPS	65.3	58.4	33.6	30.2

Table 2. Throughput in Kops/sec for client operations on a single partition server without replication.

Oregon 81.2ms, Oregon to Ireland 166.1ms, and Ireland to Virginia 87.5ms. Partitions in a datacenter form a binary tree for the purposes of GST computation. Unless stated otherwise, in all experiments below, the aggregation tree protocol to compute GST is run every 5 milliseconds.

6.2 Microbenchmarks

We first examine the throughput that can be achieved by a single partition server in all three systems. We launch enough clients to saturate the server. GET operations read a randomly selected item, and PUT operations update a randomly selected item. For comparison, we also measure the throughput of Echo operations, in which the server simply returns the arguments to the client.

As shown in Table 2, a GentleRain partition server can process Echo operations at about 65 Kops/sec, GET operations at about 58 Kops/sec, and PUT operations at about 30 Kops/sec. The results are similar for the other two systems.

The throughput of Echo indicates the message processing capability of our hardware. The throughput of GET is within 10% of the throughput of Echo, which indicates the speed of the hardware in terms of message handling. PUT operations are more expensive, primarily because of the cost of creating a new version. As the update value size increases, the throughput drops slightly due to the need for a larger memory copy. In all cases, the CPU is the bottleneck.

6.3 Throughput

We examine the throughput of GentleRain in comparison to EC and COPS, for various workloads while varying the number of partitions.

In the first experiment, each client reads a randomly selected item from every partition and updates a randomly selected item at one partition. The write operation depends on the previous write and the read operations in between. Figure 3 shows the throughput of GentleRain, COPS, and EC. GentleRain performs as well as EC for all number of partitions, and performs significantly better than COPS. At 32 partitions, both EC and GentleRain achieve a throughput of 1670 Kops/sec, while COPS achieves 748 Kops/sec. The difference between GentleRain and COPS stems from the fact that in this experiment, which is a worst case scenario for COPS, it needs to send dependency check messages to all partitions.

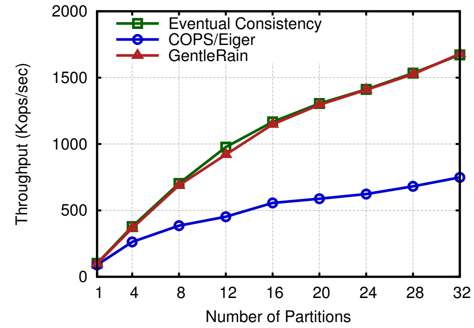

Figure 3. Throughput of 1 to 32 partitions. A client reads an item from every partition and updates an item at one partition.

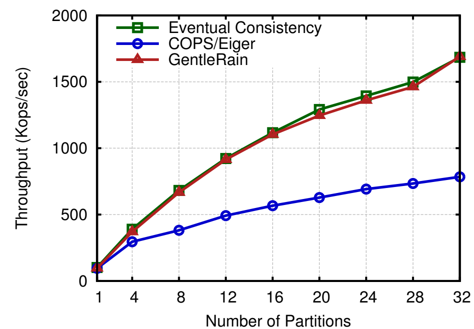

Figure 4. Throughput of 1 to 32 partitions. A client reads an item from every partition and updates an item at one partition with a 1KB value.

Dependency check messages are a source of overhead even with large updates, as Figure 4 shows where we replace the 64 byte updates with 1KB ones.

In the second experiment, each client updates a randomly selected item from each partition in a round-robin fashion. Each write depends only on the previous write. Figure 5 shows the throughput of GentleRain, COPS, and EC. The same general trend shows in these results, but the difference between COPS and GentleRain is smaller because only one dependency check message is required in COPS: at 32 partitions, GentleRain sustains 937 Kops/sec, while COPS sustains 717 Kops/sec.

In the third experiment, a client reads N randomly selected items from randomly selected partitions and writes one randomly selected item to each of M randomly selected partitions. We vary the ratio of N to M. Figure 6 shows the throughput results. Overall, GentleRain is close to the throughput of eventual consistency. GentleRain provides far better throughput than COPS for read-heavy workloads. This performance gap decreases as we move towards the update-heavy end, as COPS no longer needs to track or check as many dependencies for each update.

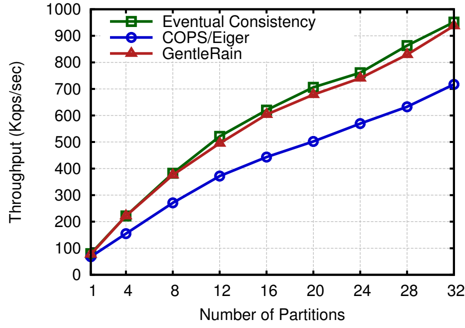

Figure 5. Throughput of 1 to 32 partitions. A client updates an item at each partition in a round-robin fashion.

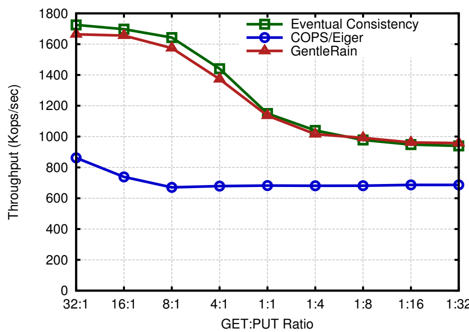

Figure 6. Throughput with different GET:PUT ratios.

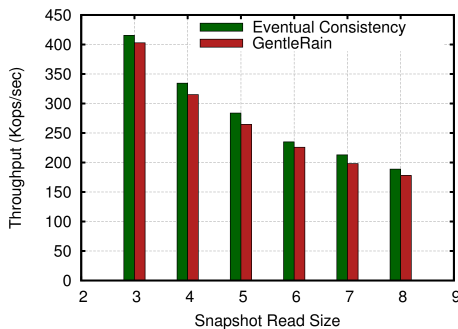

Figure 7. Throughput of snapshot reads.

In the fourth experiment, a client issues snapshot reads. We vary the number of items a snapshot reads. The throughput results for GentleRain and EC (treating the reads as normal reads) are reported in Figure 7. The cost of a causally consistent read-only transaction is almost the same as a snapshot read when the two-round read protocol of Eiger is not selected. Hence, we do not present its throughput numbers in the paper.

In our last throughput experiment, we evaluate the overhead of the GST computation, which requires partitions within the same datacenter to periodically exchange messages. Figure 8 shows throughput as a function of the interval at which the

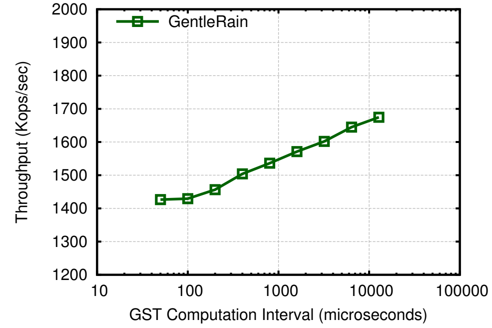

Figure 8. Varying GST computation intervals.

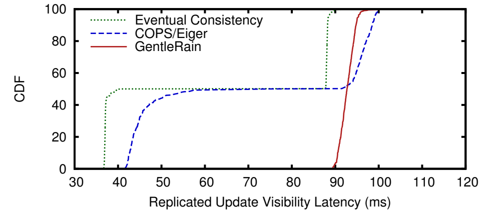

Figure 9. Visibility latency of remote updates replicated from Ireland and Virginia to Oregon

GST computation is carried out for an experiment in which a client reads all partitions and updates one of them. Increasing the value of the interval by 256X causes an increase in throughput of only 1.15X, demonstrating that the throughput is relatively unaffected by the rate at which GST computation messages are exchanged in the same datacenter.

6.4 Update Visibility Latency

We measure the update visibility latency by storing the physical time when an update is installed at its origin (in the value of its key-value pair), and subtracting it from the physical time when the update becomes visible at a remote datacenter. The clock skew between clocks on different servers causes this measurement to be only an approximation of the update visibility latency, but it is a good approximation, because the clock skew is much smaller than the network travel times between datacenters.

Figure 9 shows a cumulative distribution of the latency before updates originating in Ireland and Virginia become visible in Oregon. The results confirm the discussion in Section 5. For EC and COPS the vast majority of the updates become visible at a time equal to the network travel time between the datacenters: about half of the updates originate in Ireland and about half in Virginia, and each half becomes visible in Oregon after the network travel time from their origin to their destination. For GentleRain the update visibility is roughly equal to the longest network travel time, in this case from Ireland to Oregon, plus the GST computation interval.

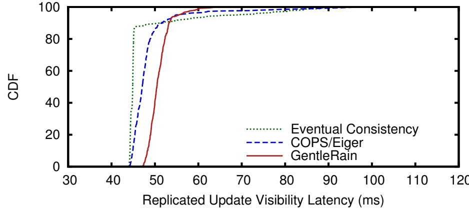

Figure 10. Visibility latency of remote updates replicated from Ireland and Oregon to Virginia

Figure 10 shows a cumulative distribution of the latency before updates originating in Oregon and Ireland become visible in Virginia. The results are less distinct here, because the network travel times from both the other datacenters to Virginia do not differ by much.

7. Related Work

The choice of consistency for replicated systems has been the subject of much research, with models ranging from strong consistency [16] to eventual consistency [29] over various intermediate models such as causal consistency [2, 20], and combinations of various models.

At one end, there are a number of systems that provide only eventual consistency, such as Dynamo [11]. We have demonstrated that we can achieve the stronger semantics of causal consistency, with similar throughput, by allowing a modest increase in remote update visibility. At the other end, there are systems that provide strong consistency, such Spanner [10] and MegaStore [5]. While providing a simple programming model, such systems provably must exhibit much longer latencies [9, 14]. Some recent systems employ multiple consistency models within a single system [21, 28]. The system chooses which consistency to use for what data item based on SLAs or ordering constraints imposed by the user.

The concept of causality in distributed systems was introduced by Lamport [20]. Lamport also discussed the use of physical clocks instead of logical Lamport clocks. However, the physical clocks there were used as a mechanism to incorporate out-of-band causality beyond the causality propagated through message exchange. Consequently, those clocks required a tight enough synchronization bound to subsume any such causality. In contrast, physical clocks in GentleRain do not require any synchronization bounds to offer causal consistency. Rather, they are more similar to *logical Lamport Clocks* with the coupling to physical time made to ensure that clocks on different servers progress at a reasonably similar rate to aid update visibility.

Bayou [27], lazy replication [19], ISIS [8], causal memory [2], and PRACTI [6] implement causal consistency, but all assume single-machine replicas and do not consider partitions. COPS was the first system to implement causal consistency in a partitioned replicated data store [23]. The authors also introduce the concept of a causally consistent read-only transaction, and in a later system, Eiger [24], the concept of a causally consistent write-only transaction. Later systems along the same lines include ChainReaction [3] and Orbe [12]. In terms of maintaining causal consistency, all four systems (COPS, Eiger, ChainReaction, and Orbe) rely on maintaining detailed dependency information and explicit dependency check messages to verify that an update can be installed according to the rules of causal consistency. They also employ various optimizations to reduce the number of dependencies and the number of dependency check messages, by relying on the transitivity of causality and only tracking nearest dependencies [23, 24] or by using a sparse representation of a dependency matrix [12]. Nonetheless, their worst-case behavior remains linear in the number of partitions. In contrast, GentleRain needs only a single scalar to track dependencies, independent of any workload characteristics.

Our work has in part been inspired by the use of physical clocks to implement causal read-only transactions in Orbe [12]. Physical clocks have also been used in many other distributed systems. In recent examples, Spanner implements serializable transactions with external consistency [16] in a geographically replicated and partitioned data store [10]. Unlike GentleRain, Spanner uses synchronized clocks with bounded uncertainty, called TrueTime, requiring access to GPS and atomic clocks. Clock-SI [13] uses loosely synchronized clocks to provide snapshot isolation [7] in a partitioned data store.

GentleRain supports only a mapping from variable length keys to values, and does not explicitly support more complex schemas such as column families [24]. One could add support for columns and column families, if desired, by treating the key as multidimensional with one dimension per-column.

8. Conclusion

We have presented a new protocol for implementing causal consistency for geo-replicated partitioned data stores. The protocol trades remote update visibility latency for improved throughput. We have demonstrated by means of measurements that for a variety of workloads GentleRain indeed provides very good throughput, close to eventual consistency, and considerably better than existing solutions.

Acknowledgments: We would like to thank our shepherd Michael Kaminsky and the anonymous reviewers for their feedback that helped make the paper better.

References

[1] The network time protocol. `http://www.ntp.org`, 2014.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] S. Almeida, J. Leitão, and L. Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the European Conference on Computer Systems*, pages 85–98. ACM, 2013.

[4] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the Conference on Management of Data*, pages 761–772. ACM, 2013.

[5] J. Baker, C. Bond, J. Corbett, et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[6] N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, 2006.

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the Conference on Management of Data*, pages 1–10. ACM, 1995.

[8] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *Transactions on Computer Systems*, 5(1):47–76, 1987.

[9] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Symposium on Principles of Distributed Computing*, page 7. ACM, 2000.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally-distributed database. In *Proceedings of the Conference on Operating Systems Design and Implementation*, pages 251–264. USENIX Association, 2012.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[12] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the Symposium on Cloud Computing*, pages 1–14. ACM, 2013.

[13] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 173–184. IEEE, 2013.

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

[15] I. Gupta, R. van Renesse, and K. P. Birman. Scalable fault-tolerant aggregation in large process groups. In *Proceedings of the Conference on Dependable Systems and Networks*, pages 433–442. IEEE, 2001.

[16] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[17] F. Jahanian, S. Fakhouri, and R. Rajkumar. Processor group membership protocols: specification, design and implementation. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 2–11. IEEE, 1993.

[18] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Symposium on Theory of Computing*, pages 654–663. ACM, 1997.

[19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[21] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the Conference on Operating Systems Design and Implementation*, 2012.

[22] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the Conference on Networked Systems Design and Implementation*, pages 399–412. USENIX Association, 2013.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the Conference on Networked Systems Design and Implementation*, pages 313–328. USENIX Association, 2013.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the Conference on Management of Data*, pages 135–146. ACM, 2010.

[26] D. S. Parker Jr, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, (3): 240–247, 1983.

[27] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Operating Systems Review*, volume 31, pages 288–301. ACM, 1997.

[28] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

[29] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.