Concise Server-Wide Causality Management for Eventually Consistent Data Stores

Ricardo Gonçalves(✉), Paulo Sérgio Almeida,
Carlos Baquero, and Victor Fonte

HASLab, INESC Tec and Universidade do Minho
Portugal, Braga
{tome,psa,cbm,vff}@di.uminho.pt

Abstract. Large scale distributed data stores rely on optimistic replication to scale and remain highly available in the face of network partitions. Managing data without coordination results in eventually consistent data stores that allow for concurrent data updates. These systems often use anti-entropy mechanisms (like *Merkle Trees*) to detect and repair divergent data versions across nodes. However, in practice hash-based data structures are too expensive for large amounts of data and create too many false conflicts.

Another aspect of eventual consistency is detecting write conflicts. Logical clocks are often used to track data causality, necessary to detect causally concurrent writes on the same key. However, there is a non-negligible metadata overhead per key, which also keeps growing with time, proportional with the node churn rate. Another challenge is deleting keys while respecting causality: while the values can be deleted, per-key metadata cannot be permanently removed without coordination.

We introduce a new causality management framework for eventually consistent data stores, that leverages node logical clocks (Bitmapped Version Vectors) and a new key logical clock (Dotted Causal Container) to provides advantages on multiple fronts: 1) a new efficient and lightweight anti-entropy mechanism; 2) greatly reduced per-key causality metadata size; 3) accurate key deletes without permanent metadata.

Keywords: Distributed Systems · Key-Value Stores · Eventual Consistency · Causality · Logical Clocks · Anti-Entropy

1 Introduction

Modern distributed data stores often emphasize high availability and low latency [2,9,7] on geo-replicated settings. Since these properties are at odds with strong consistency [3], these systems allow writing concurrently on different nodes, which avoids the need for global coordination to totally order writes, but creates data divergence. To deal with conflicting versions for the same key, generated by concurrent writes, we can either use the *last-writer-wins* rule [5], which only keeps the "last" version (according to a wall-clock timestamp for example) and lose the other versions, or we can properly track each key causal

A. Bessani and S. Bouchenak (Eds.): DAIS 2015, LNCS 9038, pp. 66–79, 2015.
DOI: 10.1007/978-3-319-19129-4_6

history with logical clocks [10], which track a partial order on all writes for a given key to detect concurrent writes.

Version Vectors [13] – the logical clocks used in Dynamo – are an established technique that provides a compact representation of causal histories [14]. However, Version vectors do not scale well when multiple users concurrently update the same node, as they would require one entry per user. To address this Riak, a commercial Dynamo inspired database, uses a newer mechanism – called *Dotted Version Vectors* [1] – to handle concurrent versions on the same node in addition to the concurrency across nodes. While these developments improved the scalability problem, the logical clock metadata can still be a significant load when tracking updates on lots of small data items.

In this paper, we address the general case in which, for each key, multiple concurrent versions are kept until overwritten by a future version; no updates are arbitrarily dropped. We present a solution that expressively improves the metadata size needed to track per-key causality, while showing how this also benefits anti-entropy mechanisms for node synchronization and add support for accurate distributed deletes. Brief summary of the contributions:

High Savings on Causality Metadata. Building on *Concise Version Vectors* [11], and on *Dotted Version Vectors* [1], we present a new causality management framework that uses a new logical clock per node to summarize which key versions are currently locally stored or have been so in the past. With the node clock, we can greatly reduce the storage footprint of keys' metadata by factoring out the information that the node clock already captures. The smaller footprint makes the overall metadata cost closer to last-write-wins solutions and delivers a better metadata-to-payload ratio for keys storing small values, like integers.

Distributed Key Deletion. Deleting a key in an eventually consistent system while respecting causality is non-trivial when using traditional version vector based mechanisms. If a key is fully removed while keeping no additional metadata, it will re-appear if some node replica didn't receive the delete (by lost messages or network partitions) and still has an old version (the same applies for backup replicas stored offline). Even worse, if a key is deleted and re-created, it risks being silently overwritten by an older version that had a higher version vector (since a new version vector starts again the counters with zeros). This problem will be avoided by using the node logical clock to create monotonically increasing counters for the key's logical clocks.

Anti-Entropy. Eventually consistent data stores rely on anti-entropy mechanisms to repair divergent versions across the key space between nodes. It both detects concurrent versions and allows newer versions to reach all node replicas. Dynamo [2], Riak [7] and Cassandra [9] use Merkle-trees [12] for their anti-entropy mechanism. This is an expensive mechanism, in both space and time, that requires frequent updates of an hash tree and presents a trade-off between hash tree size and risk of false positives. We will show how a highly compact and efficient node clock implementation, using bitmaps and binary logic, can be leveraged to support anti-entropy and dispense the use of Merkle-trees altogether.

2 Architecture Overview and System Model

Consider a *Dynamo-like* [2] distributed key-value store, organized as large number (e.g., millions) of virtual nodes (or simply nodes) mapped over a set of physical nodes (e.g., hundreds). Each key is replicated over a deterministic subset of nodes – called *replica nodes* for that key –, using for example consistent hashing [6]. Nodes that replicate common keys are said to be *peers*. We assume no affinity between clients and server nodes. Nodes also periodically perform an anti-entropy protocol with each other to synchronize and repair data.

2.1 Client API

At a high level, the system API exposes three operations: 1) read: $\mathsf{key} \to \mathcal{P}(\mathsf{value}) \times \mathsf{context}$; 2) write: $\mathsf{key} \times \mathsf{context} \times \mathsf{value} \to ()$; 3) delete: $\mathsf{key} \times \mathsf{context} \to ()$.

This API is motivated by the *read-modify-write* pattern used by clients to preserve data causality: the client first reads a key, updates the value(s) and only then writes it back. Since multiple clients can concurrently update the same key, a read operation can return multiple concurrents values for the client to resolve. By passing the read's context back to the subsequent write, every write request provides the context in which the value was updated by the client. This context is used by the system to remove versions of that key already seen by that client. A write to a non-existing key has an empty context. The delete operation behaves exactly like a normal write, but with an empty value.

2.2 Server-Side Workflow

The data store uses several protocols between nodes, both when serving client requests, and to periodically perform anti-entropy synchronization.

Serving Reads. Any node upon receiving a read request can coordinate it, by asking the respective replica nodes for their local key version. When sufficient replies arrive, the coordinator discards obsolete versions and sends to the client the most recent (concurrent) version(s), w.r.t causality. It also sends the causal context for the value(s). Optionally, the coordinator can send the results back to replica nodes, if they have outdated versions (a process known as *Read Repair*).

Serving Writes/Deletes. Only replica nodes for the key being written can coordinate a write request, while non-replica nodes forward the request to a replica node. A coordinator node: (1) generates a new identifier for this write for the logical clock; (2) discards older versions according to the write's context; (3) adds the new value to the local remaining set of concurrent versions; (4) propagates the result to the other replica nodes; (5) waits for configurable number of *acks* before replying to the client. Deletes are exactly the same, but omit step 3, since there is no new value.

Anti-Entropy. To complement the replication done at write time and to ensure consistency convergence, either because some messages were lost, or some replica node was down for some time, or writes were never sent to all replica nodes to save bandwidth, nodes perform periodically an anti-entropy protocol. The protocol aims to figure out what key versions are missing from which nodes (or must be deleted), propagating them appropriately.

2.3 System Model

All interaction is done via asynchronous message passing: there is no global clock, no bound on the time it takes for a message to arrive, nor bounds on relative processing speeds. Nodes have access to durable storage; nodes can crash but eventually will recover with the content of the durable storage as at the time of the crash. Durable state is written atomically at each state transition. Message sending from a node i to a node j, specified at a state transition of node i by $\mathsf{send}_{i,j}$, is scheduled to happen after the transition, and therefore, after the next state is durably written. Such a send may trigger a $\mathsf{receive}_{i,j}$ action at node j some time in the future. Each node has a globally unique identifier.

2.4 Notation

We use mostly standard notation for sets and maps. A map is a set of (k, v) pairs, where each k is associated with a single v. Given a map m, $m(k)$ returns the value associated with the key k, and $m\{k \mapsto v\}$ updates m, mapping k to v and maintaining everything else equal. The domain and range of a map m is denoted by $\mathsf{dom}(m)$ and $\mathsf{ran}(m)$, respectively. $\mathsf{fst}(t)$ and $\mathsf{snd}(t)$ denote the first and second component of a tuple t, respectively. We use set comprehension of the forms $\{f(x) \mid x \in S\}$ or $\{x \in S \mid \mathit{Pred}(x)\}$. We use $\lhd$ for domain subtraction; $S \lhd M$ is the map obtained by removing from M all pairs (k, v) with $k \in S$. We will use $\mathbb{K}$ for the set of possible keys in the store, $\mathbb{V}$ for the set of values, and $\mathbb{I}$ for the set of node identifiers.

3 Causality Management Framework

Our causality management framework involves two logical clocks: one to be used per node, and one to be used per key in each replica node.

The Node Logical Clock. Each node i has a logical clock that represents all locally known writes to keys that node i replicates, including writes to those keys coordinated by other replica nodes, that arrive at node i via replication or anti-entropy mechanisms;

The Key Logical Clock. For each key stored by a replica node, there is a corresponding logical clock that represents all current and past versions seen (directly or transitively) by this key at this replica node. In addition, we attached to this key logical clock the current concurrent values and their individual causality information.

While this dual-logical clock framework draws upon the work of Concise Version Vectors (CVV) [11], our scope is on distributed key-value stores (KVS) while CVV targets distributed file-systems (DFS). Their differences pose some challenges which prevent a simple reuse of CVV:

- Contrary to DFS where the only source of concurrency are nodes themselves, KVS have external clients making concurrent requests, implying the generation of concurrent versions for the same key, even when a single node is involved. Thus, the key logical clock in a KVS has to possibly manage multiple concurrent values in a way that preserves causality;
- Contrary to DFS, which considers full replication of a set keys over a set of replicas nodes, in a KVS two peer nodes can be replica nodes for two non-equal set of keys. E.g., we can have a key k_1 with the replica nodes $\{a, b\}$, a key k_2 with $\{b, c\}$ and a key k_3 with $\{c, a\}$; although a, b and c are peers (they are replica nodes for common keys), they don't replicated the exact same set of keys. The result is that, in addition to gaps in the causal history for writes not yet replicated by peers, a node logical clock will have many other gaps for writes to key that this node is not replica node of. This increases the need for a compact representation of a node logical clock.

3.1 The Node Logical Clock

A node logical clock represents a set of known writes to keys that this node is replica node of. Since each write is only coordinated by one node and later replicated to other replica nodes, the n^{th} write coordinated by a node a can be represented by the pair (a, n). Henceforth, we'll refer to this pair as a *dot*. Essentially, a dot is a globally unique identifier for every write in the entire distributed system.

A node logical clock could therefore be a simple set of dots. However, the set would be unbound and grow linearly with writes. A more concise implementation would have a version vector to represent the set of consecutive dots since the first write for every peer node id, while keeping the rest of the dots as a separate set. For example, the node logical clock: $\{(a, 1), (a, 2), (a, 3), (a, 5), (a, 6), (b, 1), (b, 2)\}$ could be represented by the pair $([(a, 3), (b, 2)], \{(a, 5), (a, 6)\})$, where the first element is a version vector and the second is the set of the remaining dots. Furthermore, we could map peer ids directly to the pair of the maximum contiguous dot and the set of disjointed dots. Taking our example, we have the map: $\{a \mapsto (3, \{5, 6\}), b \mapsto (2, \{\})\}$.

Crucial to an efficient and compact representation of a node logical clock is the need to have the least amount of gaps between dots as possible. For example, the

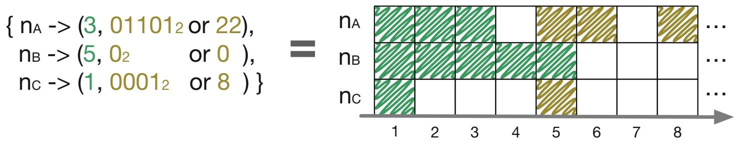

Fig. 1. A bitmapped version vector example and its visual illustration. The bitmap least-significant bit is the first bit from the left.

dots in a node logical clock that are from the local node are always consecutive with no gaps, which means that we only need maximum dot counter mapped to the local node id, while the the set of disjointed dots is empty.

The authors of [11] defined the notion of an *extrinsic* set, which we improve and generalize here as follows (note that an event can be seen as a write made to a particular key):

Definition 1 (Extrinsic). *A set of events E_1 is said to be* **extrinsic** *to another set of events E_2, if the subset of E_1 events involving keys that are also involved in events from E_2, is equal to E_2.*

This definition means that we can inflate our node logical clock to make it easier to compact, if the resulting set of dots is extrinsic to the original set. In other words, we can fill the gaps from a node logical clock, if those gaps correspond to dots pertaining to keys that the local node is not replica node of.

Taking this into consideration, our actual implementation of a node logical clock is called *Bitmapped Version Vector* (BVV), where instead of having the disjointed dots represented as a set of integers like before, we use a bitmap where the least-significant bit represents the dot immediately after the dot in the first element of the pair. A 0 means that dot is missing, while a 1 is the opposite. The actual structure of a BVV uses the integer corresponding to the bitmap to efficiently represent large and sparse sets of dots. Figure 1 gives a simple BVV example and its visual representation.

Functions over Node Logical Clocks. Lets briefly describe the functions necessary for the rest of the paper, involving node logical clocks (we omit the actual definitions due to size limitations):

- $\mathsf{norm}(base, bitmap)$ normalizes the pair $(base, bitmap)$. In other words, it removes dots from the disjointed set if they are contiguous to the base, while incrementing the base by the number of dots removed. Example: $\mathsf{norm}(2, 3) = (4, 0)$;
- $\mathsf{values}(base, bitmap)$ returns the counter values for the all the dots represented by the pair $(base, bitmap)$. Example: $\mathsf{values}(2, 2) = \{1, 2, 4\}$;
- $\mathsf{add}((base, bitmap), m)$ adds a dot with a counter m to the pair $(base, bitmap)$. Example: $\mathsf{add}((2, 2), 3) = (4, 0)$;

- $\mathsf{base}(clock)$ returns a new node logical clock with only the contiguous dots from $clock$, i.e., with the bitmaps set to zero. Example: $\mathsf{base}(\{a \mapsto (2,2), \ldots\}) = \{a \mapsto (2,0), \ldots\}$;
- $\mathsf{event}(c, i)$ takes the node i's logical clock $clock$ and its own node id i, and returns a pair with the new counter for a new write in this node i and the original logical clock c with the new counter added as a dot. Example: $\mathsf{event}(\{a \mapsto (4,0), \ldots\}, a) = (5, \{a \mapsto (5,0), \ldots\})$;

3.2 The Key Logical Clock

A key logical clock using client ids is not realistic in the kind of key-value store under consideration, since the number of clients is virtually unbound. Using simple version vectors with node ids also doesn't accurately capture causality, when a node stores multiple concurrent versions for a single key [1]. One solution is to have a version vector describing the entire causal information (shared amongst concurrent versions), and also associate to each concurrent version their own dot. This way, we can independently reason about each concurrent versions causality, reducing false concurrency. An implementation of this approach can be found in *Dotted Version Vector Sets* (DVVS) [1].

Nevertheless, logical clocks like DVVS are based on per-key information; i.e., each dot generated to tag a write is only unique in the context of the key being written. But with our framework, each dot generated for a write is globally unique in the whole system. One of the main ideas of our framework is to take advantage of having a node logical clock that store these globally unique dots, and use it whenever possible to remove redundant causal information from the key logical clock.

Contrary to version vectors or DVVS, which use per-key counters and thus have contiguous ranges of dots that can have a compact representation, the use of globally unique dots poses some challenges in defining DCC and its operations: even if we only have one version per-key, we still don't necessarily have a contiguous set of dots starting with counter one. Therefore, a compact and accurate implementation of a key logical clock is problematic: using an explicit set of dots is not reasonable as it grows unbounded; neither is using a BVV- like structure, because while a single BVV per node can be afforded, doing so per key is not realistic, as it would result in many low density bitmaps, each as large as the node one. Since there may be millions of keys per node, the size of a key logical clock must be very small.

The solution is to again leverage the notion of extrinsic sets, by filling the gaps in the clock with dots pertaining to other keys, thus not introducing false causal information. The subtlety is that every key logical clock can be inflated to a *contiguous* set of dots, since every gap in the original set was from dots belonging to other keys[1].

[1] The gaps are always from other keys, because a node i coordinating a write to a key k that generates a dot (i, n), is guaranteed to have locally coordinated all other versions of k with dots (i, m), where $m < n$, since local writes are handle sequentially and new dots have monotonically increasing counters.

$$
\begin{aligned}
\mathsf{values}((d,v)) &= \{x \mid (\_,x) \in d\}\\
\mathsf{context}((d,v)) &= v\\
\mathsf{add}(c,(d,v)) &= \mathsf{fold}(\mathsf{add}, c, \mathsf{dom}(d))\\
\mathsf{sync}((d_1,v_1),(d_2,v_2)) &= ((d_1 \cap d_2) \cup \{((i,n),x) \in d_1 \cup d_2 \mid n > \min(v_1(i), v_2(i))\},\\
&\quad\ \mathsf{join}(v_1,v_2))\\
\mathsf{discard}((d,v),v') &= (\{((i,n),x) \in d \mid n > v'(i)\}, \mathsf{join}(v,v'))\\
\mathsf{add}((d,v),(i,n),x) &= (d\{(i,n) \mapsto x\}, v\{i \mapsto n\})\\
\mathsf{strip}((d,v),c) &= (d, \{(i,n) \in v \mid n > \mathsf{fst}(c(i))\})\\
\mathsf{fill}((d,v),c) &= (d, \{i \mapsto \max(v(i), \mathsf{fst}(c(i))) \mid i \in \mathsf{dom}(c)\})
\end{aligned}
$$

Fig. 2. Functions over Dotted Causal Containers (also involving BVV)

Dotted Causal Container. Our key logical clock implementation is called Dotted Causal Container (DCC). A DCC is a container-like data structure, in the spirit of a DVVS, which stores both concurrent versions and causality information for a given key, to be used together with the node logical clock (e.g. a BVV). The extrinsic set of dots is represented as a version vector, while concurrents versions are grouped and tagged with their respective dots.

Definition 2. *A* Dotted Causal Container *(*DCC *for short) is a pair* $(\mathbb{I} \times \mathbb{N} \hookrightarrow \mathbb{V}) \times (\mathbb{I} \hookrightarrow \mathbb{N})$*, where the first component is a map from dots (identifier-integer pairs) to values, representing a set of versions, and the second component is a version vector (map from [replica node] identifiers to integers), representing a set extrinsic to the collective causal past of the set of versions in the first component.*

Functions over Key Logical Clocks. Figure 2 shows the definitions of functions over key logical clocks (DCC) – which also involves node logical clocks (BVV) – necessary for the rest of the paper. Function values returns the values of the concurrent versions in a DCC; $\mathsf{add}(c,(d,v))$ adds all the dots in the DCC (d,v) to the BVV c, using the standard fold higher-order function with the function add defined over BVVs. Function sync merges two DCCs: it discards versions in one DCC made obsolete by the other DCC's causal history, while the version vectors are merged by performing the pointwise maximum. The function context simply returns the version vector of a DCC, which represents the totality of causal history for that DCC (note that the dots of the concurrent versions are also included in the version vector component). Function $\mathsf{discard}((d,v),c)$ discards versions in a DCC (d,v) which are made obsolete by a VV c, and also merges c into v. Function $\mathsf{add}((d,v),(i,n),x)$ adds to versions d a mapping from the dot (i,n) to the value x, and also advances the i component of the VV v to n.

Finally, functions strip and fill are an essential part of our framework. Function $\mathsf{strip}((d,v),c)$ discards all entries from the VV v in a DCC that are covered by the corresponding base component of the BVV c; only entries with greater sequence numbers are kept. The idea is to only store DCCs after stripping the causality information that is already present in the node logical clock. Function fill adds back the dots to a stripped DCC, before performing functions over it.

Note that, the BVV base components may have increased between two consecutive strip $\mapsto$ fill manipulation of a given DCC, but those extra (consecutive) dots to be added to the DCC are necessarily from other keys (otherwise the DCC would have been filled and updated earlier). Thus, the filled DCC still represents an extrinsic set to the causal history of the current concurrent versions in the DCC. Also, when nodes exchange keys: single DCCs are filled before being sent; if sending a group of DCCs, they can be sent in the more compact stripped form together with the BVV from the sender (possibly with null bitmaps), and later filled at the destination, before being used. This causality stripping can lead to significant network traffic savings in addition to the storage savings, when transferring large sets of keys.

4 Server-Side Distributed Algorithm

We now define the distributed algorithm corresponding to the server-side workflow discussed in section 2.2; we define the node state, how to serve updates (writes and deletes); how to serve reads; and how anti-entropy is performed. It is presented in Algorithm 4, by way of clauses, each pertaining to some state transition due to an action (basically receive), defined by pattern-matching over the message structure; there is also a periodically to specify actions which happen periodically, for the anti-entropy. Due to space concerns, and because it is a side issue, read repairs are not addressed.

In addition to the operations over BVVs and DCCs already presented, we make use of: function nodes(k), which returns the replica nodes for the key k; function peers(i), which returns the set of nodes that are peers with node i; function random(s) which returns a random element from set s.

4.1 Node State

The state of each node has five components: g_i is the node logical clock, a BVV; m_i is the proper data store, mapping keys to their respective logical clocks (DCCs); l_i is a map from dot counters to keys, serving as a log holding which key was locally written; v_i is a version vector to track what other peers have seen of the locally generated dots; we use a VV and not a BVV, because we only care for the contiguous set of dots seen by peers, to easily prune older segments from l_i corresponding to keys seen by *all* peers; r_i is an auxiliary map to track incoming responses from other nodes when serving a read request, before replying to the client. It is the only component held in volatile state, which can be lost under node failure. All other four components are held in durable state (that must behave as if atomically written at each state transition).

Algorithm 1. Distributed algorithm for node i

```
durable state:
    $g_i$ : BVV, node logical clock; initially $g_i = \{j \mapsto (0,0) \mid j \in \mathsf{peers}(i)\}$
    $m_i$ : $\mathbb{K} \hookrightarrow$ DCC, mapping from a key to its logical clock; initially $m_i = \{\}$
    $l_i$ : $\mathbb{N} \hookrightarrow \mathbb{K}$, log of keys locally updated; initially $l_i = \{\}$
    $v_i$ : VV; other peers' knowledge; initially $v_i = \{j \mapsto 0 \mid j \in \mathsf{peers}(i)\}$
volatile state:
    $r_i$ : $(\mathbb{I} \times \mathbb{K}) \hookrightarrow (\text{DCC} \times \mathbb{N})$, requests map; initially $r_i = \{\}$
on $\mathsf{receive}_{j,i}(\mathsf{write}, k : \mathbb{K}, v : \mathbb{V}, c : \text{VV})$:
    if $i \notin \mathsf{nodes}(k)$ then
        $u = \mathsf{random}(\mathsf{nodes}(k))$                // pick a random replica node of $k$
        $\mathsf{send}_{i,u}(\mathsf{write}, k, v, c)$                // forward request to node $u$
    else
        $d = \mathsf{discard}(\mathsf{fill}(m_i(k), g_i), c)$        // discard obsolete versions in $k$'s DCC
        $(n, g_i') = \mathsf{event}(g_i, i)$   // increment and get the new max dot from the local BVV
        $d' =$ if $v \neq \mathsf{nil}$ then $\mathsf{add}(d, (i, n), v)$ else $d$     // if it's a write, add version
        $m_i' = m_i\{k \mapsto \mathsf{strip}(d', g_i')\}$                // update DCC entry for $k$
        $l_i' = l_i\{n \mapsto k\}$                              // append key to log
        for $u \in \mathsf{nodes}(k) \setminus \{i\}$ do
            $\mathsf{send}_{i,u}(\mathsf{replicate}, k, d')$        // replicate new DCC to other replica nodes
on $\mathsf{receive}_{j,i}(\mathsf{replicate}, K : \mathbb{K}, d : \text{DCC})$:
    $g_i' = \mathsf{add}(g_i, d)$          // add version dots to node clock $g_i$, ignoring DCC context
    $m_i' = m_i\{k \mapsto \mathsf{strip}(\mathsf{sync}(d, \mathsf{fill}(m_i(k), g_i)), g_i')\}$       // sync with local and strip
on $\mathsf{receive}_{j,i}(\mathsf{read}, K : \mathbb{K}, n : \mathbb{N})$:
    $r_i' = r_i\{(j, k) \mapsto (\{\}, n)\}$                    // initialize the read request metadata
    for $u \in \mathsf{nodes}(k)$ do
        $\mathsf{send}_{i,u}(\mathsf{read\_request}, j, k)$          // request $k$ versions from replica nodes
on $\mathsf{receive}_{j,i}(\mathsf{read\_request}, u : \mathbb{I}, k : \mathbb{K})$:
    $\mathsf{send}_{i,j}(\mathsf{read\_response}, u, k, \mathsf{fill}(m_i(k), g_i))$        // return local versions for $k$
on $\mathsf{receive}_{j,i}(\mathsf{read\_response}, u : \mathbb{I}, k : \mathbb{K}, d : \text{DCC})$:
    if $(u, k) \in \mathsf{dom}(r_i)$ then
        $(d', n) = r_i((u, k))$                          // $d'$ is the current merged DCC
        $d'' = \mathsf{sync}(d, d')$                       // sync received with current DCC
        if $n = 1$ then
            $r_i' = \{(u, k)\} \lhd r_i$                 // remove $(u,k)$ entry from requests map
            $\mathsf{send}_{i,u}(k, \mathsf{values}(d''), \mathsf{context}(d''))$                 // reply to client $u$
        else
            $r_i' = r_i\{(u, k) \mapsto (d'', n - 1)\}$                  // update requests map
periodically:
    $j = \mathsf{random}(\mathsf{peers}(i))$
    $\mathsf{send}_{i,j}(\mathsf{sync\_request}, g_i(j))$
on $\mathsf{receive}_{j,i}(\mathsf{sync\_request}, (n, b) : (\mathbb{N} \times \mathbb{N}))$:
    $e = \mathsf{values}(g_i(i)) \setminus \mathsf{values}((n, b))$           // get the dots from $i$ missing from $j$
    $K = \{l_i(m) \mid m \in e \wedge j \in \mathsf{nodes}(l_i(m))\}$       // remove keys that $j$ isn't replica node of
    $s = \{k \mapsto \mathsf{strip}(m_i(k), g_i) \mid k \in K\}$          // get and strip DCCs with local BVV
    $\mathsf{send}_{i,j}(\mathsf{sync\_response}, \mathsf{base}(g_i), s)$
    $v_i' = v_i\{j \mapsto n\}$                          // update $v_i$ with $j$'s information on $i$
    $M = \{m \in \mathsf{dom}(l_i) \mid m < \min(\mathsf{ran}(v_i'))\}$           // get dots $i$ seen by all peers
    $l_i' = M \lhd l_i$                               // remove those dots from the log
    $m_i' = m_i\{k \mapsto \mathsf{strip}(m_i(k), g_i) \mid m \in M, k \in l_i(m)\}$     // strip the keys removed from the log
on $\mathsf{receive}_{j,i}(\mathsf{sync\_response}, g : \text{BVV}, s : \mathbb{K} \hookrightarrow \text{DCC})$:
    $g_i' = g_i\{j \mapsto g(j)\}$               // update the node logical clock with $j$'s entry
    $m_i' = m_i\{k \mapsto \mathsf{strip}(\mathsf{sync}(\mathsf{fill}(m_i(k), g_i), \mathsf{fill}(d, g)), g_i') \mid (k, d) \in s\}$
```

4.2 Updates

We have managed to integrate both writes and deletes in a unified framework. A $\mathsf{delete}(k, c)$ operation is translated client-side to a $\mathsf{write}(k, \mathsf{nil}, c)$ operation, passing a special nil as the value.

When a node i is serving an update, arriving from the client as a $(\mathsf{write}, k, v, c)$ message (first "on" clause in our algorithm), either i is a replica node for key k or it isn't. If it's not, it forwards the request to a random replica node for k. If it is: (1) it discards obsolete versions according to context c; (2) creates a new dot and adds its counter to the node logical clock; (3) if the operation is not a delete ($v \neq \mathsf{nil}$) it creates a new version, which is added to the DCC for k; (4) it stores the new DCC after stripping unnecessary causal information; (5) appends k to the log of keys update locally; (6) sends a replicate message to other replica nodes of k with the new DCC. When receiving a replicate message, the node adds the dots of the concurrent versions in the DCC (but not the version vector) to the node logical clock and synchronizes with local key's DCC. The result is then stripped before storing.

Deletes. For notational convenience, doing $m_i(k)$ when k isn't in the map, results in the empty DCC: $(\{\}, \{\})$; also, a map update $m\{k \mapsto (\{\}, \{\})\}$ removes the entry for key k. This describes how a delete ends up removing all content from storage for a given key: (1) when there are no current versions in the DCC; (2) and when the causal context becomes older than the node logical clock, resulting in an empty DCC after stripping. If these conditions are not met at the time the delete was first requested, the key will still maintain relevant causal metadata, but when this delete is known by all peers, the anti-entropy mechanism will remove this key from the key-log l_i, and strip its remaining causal history in the DCC, resulting in a automatic and complete key and metadata removal [2].

With traditional logical clocks, nodes either maintained the context of the deleted key stored forever, or they would risk the reappearance of deleted keys or even losing new key-values created after a delete. With our algorithm using node logical clocks, we solve both cases: regarding losing new writes after deletes, updates always have new dots with increasing counters, and therefore cannot be causally in the past of previously deleted updates; in the case of reappearing deletes from anti-entropy with outdated nodes or delayed messages, a node can check if it has already seen that delete's dot in its BVV without storing specific per-key metadata.

4.3 Reads

To serve a read request (third "on" clause), a node requests the corresponding DCC from all replica nodes for that key. To allow flexibility (e.g. requiring a

[2] The key may not be entirely removed if in the meantime, another client has insert back this key, or made a concurrent update to this key. This is the expected behavior when dealing with concurrent writes or new insertions after deletes. Excluding these cases, eventually all keys that received a delete request will be removed.

quorum of nodes or a single reply is enough) the client provides an extra argument: the number of replies that the coordinator must wait for. All responses are synchronized, discarding obsolete versions, before replying to the client with the (concurrent) version(s) and the causal context in the DCC. Component r_i of the state maintains, for each pair client-key, a DCC maintaining the synchronization of the versions received thus far, and how many more replies are needed.

4.4 Anti-entropy

Since node logical clocks already reflect the node's knowledge about current and past versions stored locally, comparing those clocks tells us exactly what updates are missing between two peer nodes. However, only knowing the dots that are missing is not sufficient: we must also know what key a dot refers to. This is the purpose of the l_i component of the state: a log storing the keys of locally coordinated updates, which can be seen as a dynamic array indexed by a contiguous set of counters.

Periodically, a node i starts the synchronization protocol with one of its peers j. It starts by sending j's entry of i's node logical clock to j. Node j receives and compares that entry with its own local entry, to detect which local dots node i hasn't seen. Node j then sends back its own entry in its BVV (we don't care about the bitmap part) and the missing key versions (DCCs) that i is also replica node of. Since we're sending a possibly large set of DCCs, we stripped them of unnecessary causal context before sending, to save bandwidth (they were stripped when they where stored, but the node clock has probably advanced since then, so we strip the context again to possibly have further savings).

Upon reception, node i updates j's entry in its own BVV, to reflect that i has now seen all updates coordinated by j reflected in j's received logical clock. Node i also synchronizes the received DCCs with the local ones: for each key, its fills the received DCC with j's logical clock, it reads and fills the equivalent local DCCs with i's own logical clock, and then synchronizes each pair into a single DCC and finally locally stores the result after striping again with i's logical clock.

Additionally, node j also: (1) updates the i's entry in v_j with the max contiguous dot generated by j that i knows of; (2) if new keys are know known by all peers (i.e. if the minimum counter of v_j has increased), then remove the corresponding keys from the key-log l_i. This is also a good moment to revisit the locally saved DCCs for these keys, and check if we can further strip causality information, given the constant information growth in the node logical clock. As with deletes, if there were no new updates to a key after the one represented by the dot in the key-log, the DCC will be stripped of its entire causal history, which means that we only need one dot per concurrent version in the stored DCC.

5 Evaluation

We ran a small benchmark, comparing a prototype data store based on our framework, against a traditional one based on Merkle Trees and per-key logical clocks. The system was populated with 40000 keys, each key replicated in 3

Table 1. Results from a micro-benchmark run with 10000 writes

	Key/Leaf Ratio	Hit Ratio	Total Metadata	Metadata Per Repair
Merkle Tree	1	60.214 %	449.65 KB	4.30 KB
	10	9.730 %	293.39 KB	2.84 KB
	100	1.061 %	878.40 KB	7.98 KB
	1000	0.126 %	6440.96 KB	63.15 KB
BVV & DCC	–	100 %	3.04 KB	0.019 KB

	Average Entries Per Key L. Clock
VV or DVV	3
DCC	0.231

nodes, and we measured some metrics over the course of 10000 writes, 10% losing a message replicating the write to one replica node. The evaluation aimed to compare metadata size of anti-entropy related messages and the data store causality-related metadata size. We compared against four Merkle Trees sizes to show how its “resolution”, i.e., the ratio of keys-per-leaf impacts results.

Table 1 shows the results of our benchmark. There is always significant overhead with Merkle Trees, worse for larger keys-per-leaf ratios, where there are many false positives. Even for smaller ratios, where the “hit ratio” of relevant-hashes over exchanged-hashes is higher, the tree itself is large, resulting in substantial metadata transferred. In general, the metadata overhead to perform anti-entropy with our scheme is orders of magnitude smaller than any of the Merkle Tree configurations.

Concerning causality-related metadata size, being negligible the cost of node-wide metadata amortized over a large database, the average per-key logical clock metadata overhead is also significantly smaller in our scheme, since most of the time the causality is entirely condensed by the node-wide logical clock. With traditional per-key logical clocks, the number of entries is typically the degree of replication, and can be larger, due to entries for retired nodes that remain in the clock forever, a problem which is also solved by our scheme.

6 Related Work

The paper’s mechanisms and architecture extend the specialized causality mechanisms in [11,1], apply it over a eventually consistent data store. In addition to the already mentioned differences between our mechanism and Concise Version Vectors [11], our key logical clock size is actually bounded by the number of active replica nodes, unlike PVEs (the CVV key logical clock is unbounded).

Our work also builds on concepts of weakly consistent replication present in log-based systems [15,8,4] and data/file synchronization [13]. The assignment of local unique identifiers for each update event is already present in [15], but each node totally orders its local events, while we consider concurrent clients to the same node. The detection of when an event is known in all other replicas nodes – a condition for log pruning – is common to the mentioned log-based systems; however, our log structure (the key log) is only an inverted index that tracks divergent data replicas, and thus is closer to optimistic replicated file-systems. Our design can reduce divergence both as a result of foreground user activity (both on writes, deletes, and read repair) and by periodic background anti-entropy, while using a common causality framework.

Acknowledgments. This work is financed by the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project UID/EEA/50014/2013 and scholarship SFRH/BD/86735/2012; also by the North Portugal Regional Operational Programme (ON.2, O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), within project NORTE-07-0124-FEDER-000058; and by EU FP7 SyncFree project (609551).

References

1. Almeida, P.S., Baquero, C., Gonçalves, R., Preguiça, N., Fonte, V.: Scalable and accurate causality tracking for eventually consistent stores. In: Magoutis, K., Pietzuch, P. (eds.) DAIS 2014. LNCS, vol. 8460, pp. 67–81. Springer, Heidelberg (2014)
2. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: ACM SIGOPS Operating Systems Review, vol. 41, pp. 205–220 (2007)
3. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33(2), 51–59 (2002)
4. Golding, R.A.: Weak-consistency group communication and membership. Ph.D. thesis, University of California Santa Cruz (1992)
5. Johnson, P.R., Thomas, R.H.: The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute (1976)
6. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 654–663. ACM (1997)
7. Klophaus, R.: Riak core: building distributed applications without shared state. In: ACM SIGPLAN Commercial Users of Functional Programming. ACM (2010)
8. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. Comput. Syst. 10(4), 360–391 (1992)
9. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
11. Malkhi, D., Terry, D.: Concise version vectors in winFS. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 339–353. Springer, Heidelberg (2005)
12. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990), http://dl.acm.org/citation.cfm?id=118209.118230
13. Parker Jr., D.S., Popek, G., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D.: Detection of mutual inconsistency in distributed systems. IEEE Transactions on Software Engineering, 240–247 (1983)
14. Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: In search of the holy grail. Distributed Computing 7(3), 149–174 (1994)
15. Wuu, G., Bernstein, A.: Efficient solutions to the replicated log and dictionary problems. In: Symp. on Principles of Dist. Comp. (PODC), pp. 233–242 (1984)