Scalable and Accurate Causality Tracking for Eventually Consistent Stores

Paulo Sérgio Almeida[1], Carlos Baquero[1],
Ricardo Gonçalves[1], Nuno Preguiça[2], and Victor Fonte[1]

[1] HASLab, INESC Tec & Universidade do Minho
{psa,cbm,tome,vff}@di.uminho.pt
[2] CITI/DI, FCT, Universidade Nova de Lisboa
nuno.preguica@fct.unl.pt

Abstract. In cloud computing environments, data storage systems often rely on optimistic replication to provide good performance and availability even in the presence of failures or network partitions. In this scenario, it is important to be able to accurately and efficiently identify updates executed concurrently. Current approaches to causality tracking in optimistic replication have problems with concurrent updates: they either (1) do not scale, as they require replicas to maintain information that grows linearly with the number of writes or unique clients; (2) lose information about causality, either by removing entries from client-id based version vectors or using server-id based version vectors, which cause false conflicts. We propose a new logical clock mechanism and a logical clock framework that together support a traditional key-value store API, while capturing causality in an accurate and scalable way, avoiding false conflicts. It maintains concise information per data replica, only linear on the number of replica servers, and allows data replicas to be compared and merged linear with the number of replica servers and versions.

1 Introduction

Amazon's Dynamo system [5] was an important influence to a new generation of databases, such as Cassandra [10] and Riak [9], focusing on partition tolerance, write availability and eventual consistency. The underlying rationale to these systems stems from the observation that when faced with the three concurrent goals of *consistency*, *availability* and *partition-tolerance* only two of those can be achievable in the same system [3,6]. Facing geo-replication operation environments where partitions cannot be ruled out, consistency requirements are inevitably relaxed in order to achieve high availability.

These systems follow a design where the data store is always writable: replicas of the same data item are allowed to temporarily diverge and to be repaired later on. A simple repair approach followed in Cassandra, is to use wall-clock timestamps to know which concurrent updates should prevail. This last writer wins (lww) policy may lead to lost updates. An approach which avoids this, must be able to represent and maintain causally concurrent updates until they can be reconciled.

Accurate tracking of concurrent data updates can be achieved by a careful use of well established causality tracking mechanisms [11,14,20,19,2]. In particular, for data

K. Magoutis and P. Pietzuch (Eds.): DAIS 2014, LNCS 8460, pp. 67–81, 2014.

storage systems, version vectors (vv) [14] enable the system to compare any pair of replica versions and detect if they are equivalent, concurrent or if one makes the other obsolete. However, as we will discuss in Section 3, vv lack the ability to accurately represent concurrent values when used with server ids, or are not scalable when used with client ids.

We present a new and simple causality tracking solution, Dotted Version Vectors (briefly introduced in [16]), that overcomes these limitations allowing both scalable (using server ids) and fully accurate (representing same server concurrent writes) causality tracking. It achieves this by explicitly separating a new write event identifier from its causal past, which has the additional benefit of allowing causality checks between two clocks in constant time (instead of linear with the size of version vectors).

Besides fully describing Dotted Version Vectors (dvv), in this paper we make two novel contributions. First, we propose a new container (DVV Sets or dvvs) that efficiently compacts a set of concurrent dvv's in a single data structure, improving on two dvv limitations: (1) dvvs representation is independent of the number of concurrent values, instead of linear; (2) comparing and synchronizing two replica servers w.r.t. a single key is linear with the number of concurrent values, instead of quadratic.

Our final contribution is a general framework that clearly defines a set of functions that logical clocks need to implement to correctly track causality in eventually consistent systems. We implement both dvv and dvvs using this framework.

The rest of this paper is organized as follows. Section 2 presents the system model for the remaining paper. We survey and compare current mechanisms for causality tracking in Section 3. In Section 4, we present our mechanism dvv, followed by its compact version dvvs, in Section 5. We then propose in Section 6 a general framework for logical clocks and its implementation with both dvv and dvvs. In Section 7 we present the asymptotic complexities for both the current and proposed mechanisms, as well as an evaluation of dvvs. Additional techniques are briefly discussed in Section 8. We conclude in Section 9.

2 System Model and Data Store API

We consider a standard Dynamo-like key-value store interface that exposes two operations: get(key) and put$(key, value, context)$. get returns a pair $(value(s), context)$, i.e., a value or set of causally concurrent values, and an opaque context that encodes the causal knowledge in the value(s). put submits a single value that supersedes all values associated to the supplied context. This context is either empty if we are writing a new value, or some opaque data structure returned to the client by a previous get, if we are updating a value. This context encodes causal information, and its use in the API serves to generate a *happens-before* [20] relation between a get and a subsequent put.

We assume a distributed system where nodes communicate by asynchronous message passing, with no shared memory. The system is composed by possibly many (e.g., thousands) clients which make concurrent get and put requests to server nodes (in the order of, e.g., hundreds). Each key is replicated in a typically small subset of the server nodes (e.g., 3 nodes), which we call the replica nodes for that key. These different orders of magnitude of clients, servers and replicas play an important role in the design of a scalable causality tracking mechanism.

We assume: no global distributed coordination mechanism, only that nodes can perform internal concurrency control to obtain atomic blocks; no sessions or any form of client-server affinity, so clients are free to read from a replica server node and then write to a different one; no byzantine failures; server nodes have stable storage; nodes can fail without warning and later recover with their last state in the stable storage.

As we do not aim to track causality between different keys, in the remainder we will focus on operations over a single key, which we leave implicit; namely, all data structures in servers that we will describe are per key. Techniques as in [13] can be applied when considering groups of keys and could introduce additional savings; this we leave for future work.

3 Current Approaches

To simplify comparisons between different mechanisms, we will introduce a simple execution example between clients Mary and Peter, and a single replica node. In this example, presented in Figure 1, Peter starts by writing a new object version $\mathbf{v_1}$, with an empty context, which results in some server state A. He then reads server state A, returning current version $\mathbf{v_1}$ and context ctx_A. Meanwhile, Mary writes a new version $\mathbf{v_2}$, with an empty context, resulting in some server state B. Since Mary wrote $\mathbf{v_2}$ without reading state A, state B should contain both $\mathbf{v_1}$ and $\mathbf{v_2}$ as concurrent versions, if causality is tracked. Finally, Peter updates version $\mathbf{v_1}$ with $\mathbf{v_3}$, using the previous context ctx_A, resulting in some state C. If causal relations are correctly represented, state C we should only have $\mathbf{v_2}$ and $\mathbf{v_3}$, since $\mathbf{v_1}$ was superseded by $\mathbf{v_3}$ and $\mathbf{v_2}$ is concurrent with $\mathbf{v_3}$. We now discuss how different causality tracking approaches address this example, which are summarized in Table 1.

Last Writer Wins (lww). In systems that enforce a lww policy, such as Cassandra, concurrent updates are not represented in the stored state and only the last update prevails. Under lww, our example would result in the loss of $\mathbf{v_2}$. Although some specific application semantics are compatible with a lww policy, this simplistic approach is not adequate for many other application semantics. In general, a correct tracking of concurrent updates is essential to allow all updates to be considered for conflict resolution.

Causal Histories (ch). Causal Histories [20] are simply described by sets of unique write identifiers. These identifiers can be generated with a unique identifier and a monotonic counter. In our example, we used server identifiers r, but client identifiers could be used as well. The crucial point is that identifiers have to be globally unique to correctly represent causality. Let id_n be the notation for the n^{th} event of the entity represented by id. The partial order of causality can be precisely tracked by comparing these sets under set inclusion. Two ch are concurrent if neither includes the other: $A \parallel B$ iff $A \not\subseteq B$ and $B \not\subseteq A$. ch correctly track causality relations, as can be seen in our example, but have a major drawback: they grow linearly with the number of writes.

Version Vectors (vv). Version Vectors are an efficient representation of ch, provided that the ch has no gaps in each id's event sequence. A vv is a mapping from identifiers

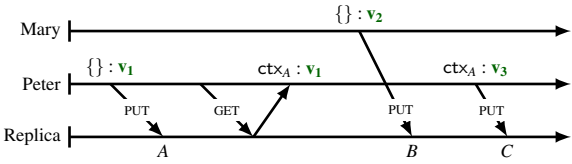

Fig. 1. Example execution for one key: Peter writes a new value $\mathbf{v_1}$ (A), then reads from Replica (ctx_A). Next, Mary writes a new value $\mathbf{v_2}$ (B) and finally Peter updates $\mathbf{v_1}$ with $\mathbf{v_3}$ (C).

Table 1. The table shows the replica (r) state after write from Peter (p) and Mary (m), and the context returned by Peter's read. We use the $metadata : value(s)$ notation, except for dvvs which has its own internal structure.

	lww	ch	$\mathsf{vv}_{\mathsf{client}}$	$\mathsf{vv}_{\mathsf{server}}$	dvv	dvvs
A	17h00 : $\mathbf{v_1}$	$\{r_1\} : \mathbf{v_1}$	$\{(p,1)\} : \mathbf{v_1}$	$\{(r,1)\} : \{\mathbf{v_1}\}$	$((r,1),\{\}) : \mathbf{v_1}$	$\{(r,1,[\mathbf{v_1}])\}$
ctx_A	$\{\}$	$\{r_1\}$	$\{(p,1)\}$	$\{(r,1)\}$	$\{(r,1)\}$	$\{(r,1)\}$
B	17h03 : $\mathbf{v_2}$	$\{r_1\} : \mathbf{v_1}$ $\{r_2\} : \mathbf{v_2}$	$\{(p,1)\} : \mathbf{v_1}$ $\{(m,1)\} : \mathbf{v_2}$	$\{(r,2)\} :$ $\{\mathbf{v_1},\mathbf{v_2}\}$	$((r,1),\{\}) : \mathbf{v_1}$ $((r,2),\{\}) : \mathbf{v_2}$	$\{(r,2,[\mathbf{v_2},\mathbf{v_1}])\}$
C	17h07 : $\mathbf{v_3}$	$\{r_2\} : \mathbf{v_2}$ $\{r_1,r_3\} : \mathbf{v_3}$	$\{(m,1)\} : \mathbf{v_2}$ $\{(p,2)\} : \mathbf{v_3}$	$\{(r,3)\} :$ $\{\mathbf{v_1},\mathbf{v_2},\mathbf{v_3}\}$	$((r,2),\{\}) : \mathbf{v_2}$ $((r,3),\{(r,1)\}) : \mathbf{v_3}$	$\{(r,3,[\mathbf{v_3},\mathbf{v_2}])\}$

to counters, and can be written as a set of pairs $(id, counter)$; each pair represents a set of ch events for that id: $\{id_n \mid 0 < n \leq counter\}$. In terms of partial order, $A \leq B$ iff $\forall (i,c_a) \in A \cdot \exists (i,c_b) \in B \cdot c_a \leq c_b$. Again, $A \parallel B$ iff $A \not\leq B$ and $B \not\leq A$. Whether client or server identifiers are used in vv has major consequences, as we'll see next.

Version Vectors with Id-per-Client ($\mathsf{vv}_{\mathsf{client}}$). This approach uses vv with clients as unique identifiers. An update is registered in a server by using the client identification issued in a put. This provides enough information to accurately encode the concurrency and causality in the system, since concurrent client writes are represented in the $\mathsf{vv}_{\mathsf{client}}$ with different ids. However, it sacrifices scalability, since $\mathsf{vv}_{\mathsf{client}}$ will end up storing the ids of all the clients that ever issued writes to that key. Systems like Dynamo try to compensate this by *pruning* entries in $\mathsf{vv}_{\mathsf{client}}$ at a specific threshold, but it typically leads to false concurrency and further need for reconciliation. The higher the degree of pruning, the higher is the degree of false concurrency in the system.

Version Vectors with Id-per-Server ($\mathsf{vv}_{\mathsf{server}}$). If causality is tracked with $\mathsf{vv}_{\mathsf{server}}$, i.e., using vv with server identifiers, it is possible to correctly detect concurrent updates that are handled by different server nodes. However, if concurrent updates are handled by the same server, there is no way to *express* the concurrent values — *siblings* — separately. To avoid overwriting siblings and losing information (as in lww), a popular solution to this, is to group all siblings under the same $\mathsf{vv}_{\mathsf{server}}$, losing individual causality information. This can easily lead to false concurrency: either a write's context causally dominates the server $\mathsf{vv}_{\mathsf{server}}$, in which case all siblings are deemed obsolete and replaced by the new value; or this new value must be added to the current siblings, even if some of them were in its causal past.

Using our example, we finish the execution with all three values $\{\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}\}$, when in fact $\mathbf{v_3}$ should have obsoleted $\mathbf{v_1}$, like the other causally correct mechanisms in Table 2 (expect for lww).

With $\mathsf{vv}_{\mathsf{server}}$, false concurrency can arise whenever a client *read-write cycle* is interleaved with another concurrent write on the same server. This can become especially problematic under heavy load with many clients concurrently writing: under high latency, if a read-write cycle cannot be completed without interleaving with another concurrent write, the set of siblings will keep on growing. This will make messages grow larger, the server load heavier, resulting in a positive feedback loop, in what can be called a *sibling explosion*.

4 Dotted Version Vectors

We now present an accurate mechanism that can be used as a substitute for classic version vectors (vv) in eventually consistent stores, while still using only one *Id* per replica node. The basic idea of *Dotted Version Vectors* (dvv) is to take a vv and add the possibility of representing an individual causal event — *dot* — separate from the rest of the contiguous events. The dot is kept separate from the causal past and it globally and uniquely identifies a write. This allows representing concurrent writes, on the same server, by having different dots.

In our example from Figure 1, we can see that state B is represented with a unique dot for both $\mathbf{v_1}$ and $\mathbf{v_2}$, even-though they both were written with an equally empty context. This distinction in their dots is what enables the final write by Peter to correctly overwrite $\mathbf{v_1}$, since the context supersedes its dot (and dvv), while maintaining $\mathbf{v_2}$ which has a newer dot than the context. In contrast, $\mathsf{vv}_{\mathsf{server}}$ loses this distinction gained by separating dots by grouping every sibling in one vv and thus cannot know that $\mathbf{v_1}$ is outdated by $\mathbf{v_3}$.

4.1 Definition

A dvv consists in a pair (d, v), where v is a traditional vv and the dot d is a pair (i, n), with i as a node identifier and n as an integer. The dot uniquely represents a write and its associated version, while the vv represents the causal past (i.e. its context). The causal events (or dots) represented by a dvv can be generated by a function toch that translates logical clocks to causal histories (ch can be viewed as sets of dots):

$$\begin{aligned} \mathsf{toch}(((i,n),v)) &= \{i_n\} \cup \mathsf{toch}(v), \\ \mathsf{toch}(v) &= \bigcup_{(i,n)\in v} \{i_m \mid 1 \leq m \leq n\}, \end{aligned}$$

where i_n denotes the n^{th} dot generated by node i, and $\mathsf{toch}(v)$ is the same function but for traditional vv. With this definition, the ch $\{a_1, b_1, b_2, c_1, c_2, c_4\}$ that cannot be represented by vv, can now be represented by the dvv $((c,4),\{(a,1),(b,2),(c,2)\})$.

4.2 Partial Order

The partial order on dvv can be defined in terms of inclusion of ch; i.e.:

$$X \leq Y \iff \mathsf{toch}(X) \subseteq \mathsf{toch}(Y),$$

Given that each dot is generated as a globally unique event — using the notational convenience $v[i] = n$, for $(i,n) \in v$ and $v[i] = 0$ for any non mapped id — the partial order on possible dvv values becomes:

$$((i,n),u) < ((j,m),v) \iff n \leq v[i] \wedge u \leq v,$$

where the traditional point-wise comparison of vv is used: $u \leq v \iff \forall_{(i,n)\in u}.\, n \leq v[i]$.

An important consequence of keeping the dot separate from the causal past is that, if the dot in X is contained in the causal past of Y, it means that Y was generated causally after X, thus Y also contains the causal past of X. This means that there is no need for the comparison of the vv component and the order can be computed as an $O(1)$ operation (assuming access to a map data structure in effectively constant time), simply as:

$$((i,n),u) < ((j,m),v) \iff n \leq v[i].$$

5 Dotted Version Vector Sets

Dotted Version Vectors (dvv), as presented in the previous section, allow an accurate representation of causality using server-based ids. Still, a dvv is kept for each concurrent version: $\{(dvv_1, v_1), (dvv_2, v_2), \ldots\}$. We can go further in exploring the fact that operations will mostly handle sets of dvv, and not single instances.

We propose now that the set of $(dvv, version)$ for a given key in a replica node is represented by a single instance of a container data type, a *Dotted Version Vector Set* (dvvs), which describes causality for the whole set. dvvs factorizes out common knowledge for the set of dvv described, and keeps only the strictly relevant information in a single data structure. This results in not only a very succinct representation, but also in reduced time complexity of operations: the concurrent values will be indexed and ordered in the data structure, and traversal will be efficient.

5.1 From a Set of Clocks to a Clock for Sets

To obtain a logical clock for a set of versions, we will explore the fact that at each node, the set of dvv as a whole can be represented with a compact vv. Formally this invariant means that, for any set of dvv S, for each node id i, *all* dots for i in S form a contiguous range up to some dot. Note that we can only assume to have this invariant, if we follow some protocol rules enforced by our framework, described in detail in section 6.3.

Assuming this invariant, we obtain a logical clock for a set of $(dvv, version)$ by performing a two-step transformation of the sets of versions. In the *first step*, we compute a single vv for the whole set — the *top vector* — by the pointwise maximum of the dots and vv in the dvv's; additionally, for each dvv in the set, we discard the vv component. As an example, the following set:

$$\{(((r,4),\{(r,3),(s,5)\}),v_1),(((r,5),\{(r,2),(s,3)\}),v_2),(((s,7),\{(r,2),(s,6)\}),v_3)\},$$

generates the top vector $\{(r,5),(s,7)\}$ and is transformed to a set of *(dot, version)*:

$$\{((r,4),v_1),((r,5),v_2),((s,7),v_3)\}.$$

This first transformation has incurred in a loss of knowledge: the specific causal past of each version. This knowledge is not, however, needed for our purposes. The insight is that, to know whether to discard or not a pair (dot,version) (d,v) from some set when comparing with another set of versions S, we do not need to know exactly *which* version in S dominates d, but only that *some* version does; if version v is not present in S, but its dot d is included in the causal information of the whole S (which is now represented by the top vector), then we know that v was obsolete and can be removed.

In the *second step*, we use the knowledge that all dots for each server id, form a contiguous sequence up to the corresponding top vector entry. Therefore, we can associate a list of versions (siblings) to each entry in the top vector, where each dot is implicitly derived by the corresponding version position in the list. In our example, the whole set is then simply described as:

$$\{(r,5,[v_2,v_1]),(s,7,[v_3])\},$$

where the head of each list corresponds to the more recently generated version at the corresponding node. The first version has the dot corresponding to the maximum of the top vector for that entry, the second version has the maximum minus one, and so on.

5.2 Definition

A dvvs is a set of triples (i,n,l), each containing a server id, an integer, and a list of concurrent versions. It describes a set of versions and their dots, implicitly given by the position in the list. It also describes only the knowledge about the collective causal history, as given by the vv derived from the pairs (i,n).

6 Using dvv and dvvs in Distributed Key-Value Stores

In this section we show how to use logical clocks — in particular dvv and dvvs— in modern distributed key-value stores, to accurate and efficiently track causality among writes in each key. Our solution consists in a general workflow that a database must use to serve get and put requests. Towards this, we define a kernel of operations over logical clocks, on top of which the workflow is defined. We then instantiate these operations over the logical clocks that we propose, first dvv and then dvvs.

We support both get and put operations, performing possibly several steps, as sketched in Figure 2. Lets first define our kernel operations.

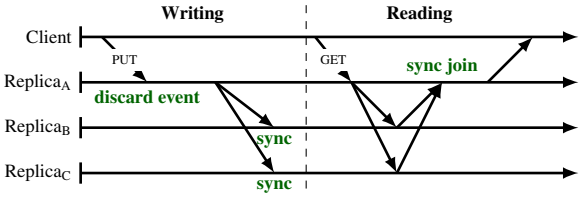

Fig. 2. Generic execution paths for operations get and put

Function sync. The function sync takes two sets of clocks, each describing a set of siblings, and returns the set of clocks for the siblings that remain after removing obsolete ones. It can have a general definition only in terms of the partial order on clocks, regardless of their actual representation: Equation 1.

Function join. The join function takes a set of clocks and returns a single clock that describes the collective causal past of all siblings in the set received. An actual implementation of join is any function that corresponds to performing the union of all the events (dots) in the ch corresponding to the set, i.e., that satisfies Equation 2.

Function discard. The discard function takes a set of clocks S (representing siblings) and a clock C (representing the context), and discards from S all siblings that are obsolete because they are included in the context C. Similar to sync, discard has a simple general definition only in terms of the partial order on clocks: Equation 3.

Function event. The event function takes a set of clocks S (representing siblings) and a clock C (representing the context) and a replica node identifier r; it returns a new clock to represent a new version, given by a new unique event (dot) generated at r, and having C in the causal past. An implementation must respect Equation 4.

$$\mathsf{sync}(S_1,S_2) = \{x \in S_1 \mid \nexists y \in S_2 . x < y\} \cup \{x \in S_2 \mid \nexists y \in S_1 . x < y\}. \quad (1)$$

$$\mathsf{toch}(\mathsf{join}(S)) = \bigcup \{\mathsf{toch}(x) \mid x \in S\}. \quad (2)$$

$$\mathsf{discard}(S,C) = \{x \in S \mid x \not\leq C\}. \quad (3)$$

$$\mathsf{toch}(\mathsf{event}(C,S,r)) = \mathsf{toch}(C) \cup \{\mathsf{next}(C,S,r)\}, \quad (4)$$

where next denotes the next new unique event (dot) generated with r, which can be deterministically defined given C, S and r.

6.1 Serving a get

Functions sync and join are used to define the get operation: when a server receives a get request, it may ask to a subset of replica nodes for their set of versions and clocks for that key, to be then "merged" by applying sync pairwise; however, the server can skip this phase if it deems it unnecessary for a successful response. Having the necessary information ready, it is returned to the client both the values stripped from causality

information and the context as a result of applying join to the clocks. sync can also be used at other times, such as anti-entropy synchronization between replica nodes.

6.2 Serving a put

When a put request is received, the server forwards the request to a replica node for the given key, unless the server is itself a replica node. A non-replica node for the key being written can coordinate a put request using $\mathsf{vv}_{\mathsf{client}}$ for example, because it can use the *client Id* to update the clock and then propagate the result to the replica nodes. However, clocks using *server Ids* like $\mathsf{vv}_{\mathsf{server}}$, dvv and dvvs need the coordinating node to generate an unique event in the clock, using its own *Id*. Not forwarding the request to replica node, would mean that non-replica nodes Ids would be added to clocks, making them linear with the total number of servers (e.g. hundreds) instead of only the replica nodes (e.g. three).

When a replica node r, containing the set of clocks S_r for the given key, receives a put request, it starts by removing obsolete versions from S_r, using function discard, resulting in S'_r; it also generates a new clock u for the new version with event; finally, u is added to the set of non- obsolete versions S'_r, resulting in S''_r.

The server can then save S''_r locally, propagate it to other replica nodes and successfully inform the client. The order of these three steps depends on the system's durability and replication parameters. Each replica node that receives S''_r, uses function sync to apply it against its own local versions.

For each key, the steps at the coordinator (discarding versions, generating a new one and adding it to the non-obsolete set of versions) must be performed atomically when serving a given put. This can be trivially obtained by local concurrency control, and does not prevent full concurrency between local operations on different keys. For operations over the same key, a replica can pipeline the steps of consecutive put for maximizing throughput (note that some steps already need to be serialized, such as writing versions to stable storage).

6.3 Maintaining Local Conciseness

As previously stated, both dvv and dvvs have an crucial invariant that servers must maintain, in order to preserve their correctness and conciseness:

Invariant 1 (Local Clock Conciseness). *Every key at any server has locally associated with it a set of version(s) and clock(s), that collectively can be logically represented by a contiguous set of causal events (e.g. represented as a vv).*

To enforce this invariant, we made two design choices: *(rule 1)* a server cannot respond to a get with a subset of the versions obtained locally and/or remotely, only the entire set should be sent; *(rule 2)* a coordinator cannot replicate the new version to remote nodes, without also sending all local concurrent versions (siblings).

Without the first rule, clients could update a key by reading and writing back a new value with a context containing arbitrary gaps in its causal history. Neither dvv nor dvvs would be expressive enough to support this, since dvv only supports one gap (between the contiguous past and the dot) and dvvs does not support any.

Without the second rule, dvvs would clearly not work, since writes can create siblings, which cannot be expressed separately with this clock. It could work with dvv, however it would eventually result in some server not having a local concise representation for a key (e.g. the network lost a previous sibling), which in turn would make this server unable to respond to get without contacting other servers (see *rule 1*); it would degrade latency and in case of partitions, availability could also suffer.

6.4 Dotted Version Vectors

Functions sync and discard for dvv can be trivially implemented according to their general definitions, by using the partial order for dvv, already defined in Section 4.2.

We will make use of some two functions: function ids returns the set of identifiers of a pair from a vv, a dvv or a set of dvv; the maxdot function takes a dvv or set of dvv and a server id and returns the maximum sequence number of the events from that server:

$$\begin{aligned}
\mathsf{ids}((i,_)) &= \{i\},\\
\mathsf{ids}(((i,_),v)) &= \{i\} \cup \mathsf{ids}(v),\\
\mathsf{ids}(S) &= \bigcup_{s \in S} \mathsf{ids}(s).\\
\mathsf{maxdot}(r,((i,n),v)) &= \max(\{n \mid i = r\} \cup \{v[r]\}),\\
\mathsf{maxdot}(r,S) &= \max(\{0\} \cup \{\mathsf{maxdot}(r,s) \mid s \in S\}).
\end{aligned}$$

Function join returns a simple vv, which is enough to accurately express the causal information. Function event can be defined as simply generating a new dot and using the context C, which is already a vv, for the causal past.

$$\begin{aligned}
\mathsf{join}(S) &= \{(i, \mathsf{maxdot}(i,S)) \mid i \in \mathsf{ids}(S)\}.\\
\mathsf{event}(C,S,r) &= ((r, \max(\mathsf{maxdot}(r,S), C[r]) + 1), C).
\end{aligned}$$

6.5 Dotted Version Vector Sets

With dvvs, we need to make slight interface changes: functions now receive a single dvvs, instead of a set of clocks; and event now inserts the newly generated version directly in the dvvs.

For clarity and conciseness, we will assume R to be the complete set of replica nodes ids, and any absent id i in a dvvs, is promoted implicitly to the element $(i,0,[])$. We will make use of the functions: $\mathsf{first}(n,l)$, that returns the first n elements of list l (or the whole list if it has less than n elements, or an empty list for non-positive n); $|l|$ for the number of elements in l, $[x \mid l]$ to append x at the head of list l; and function merge:

$$\mathsf{merge}(n,l,n',l') = \begin{cases} \mathsf{first}(n - n' + |l'|, l), & \text{if } n \geq n',\\ \mathsf{first}(n' - n + |l|, l'), & \text{otherwise.} \end{cases}$$

Table 2. Space and time complexity, for different causality tracking mechanisms. U: updates; C: writing clients; R: replica servers; V: (concurrent) versions; S_r and S_w: number of servers involved in a GET and PUT, respectively.

		lww	ch	vv$_{\text{client}}$	vv$_{\text{server}}$	dvv	dvvs
Space		$\tilde{O}(1)$	$\tilde{O}(U)$	$\tilde{O}(C\times V)$	$\tilde{O}(R+V)$	$\tilde{O}(R\times V)$	$\tilde{O}(R+V)$
Time	**event**	–	$\tilde{O}(1)$	$\tilde{O}(1)$	$\tilde{O}(1)$	$\tilde{O}(V)$	$\tilde{O}(R)$
	join	–	$\tilde{O}(U\times V)$	$\tilde{O}(C\times V)$	$\tilde{O}(1)$	$\tilde{O}(R\times V)$	$\tilde{O}(R)$
	discard	–	$\tilde{O}(U\times V)$	$\tilde{O}(C\times V)$	$\tilde{O}(R)$	$\tilde{O}(V)$	$\tilde{O}(R+V)$
	sync	–	$\tilde{O}(U\times V^2)$	$\tilde{O}(C\times V^2)$	$\tilde{O}(R+V)$	$\tilde{O}(V^2)$	$\tilde{O}(R+V)$
	PUT	$\tilde{O}(1)$	$\tilde{O}(S_w\times U\times V^2)$	$\tilde{O}(S_w\times C\times V^2)$	$\tilde{O}(S_w\times(R+V))$	$\tilde{O}(S_w\times V^2)$	$\tilde{O}(S_w\times(R+V))$
	GET	$\tilde{O}(1)$	$\tilde{O}(S_r\times U\times V^2)$	$\tilde{O}(S_r\times C\times V^2)$	$\tilde{O}(S_r\times(R+V))$	$\tilde{O}(R\times V+S_r\times V^2)$	$\tilde{O}(S_r\times(R+V))$
Causally Correct		✗	✓	✓	✗	✓	✓

Function discard takes a dvvs S and a vv C, and discards values in S obsoleted by C. Similarly, sync takes two dvvs and removes obsolete values. Function join simply returns the top vector, discarding the lists. Function event is now adapted to not only produce a new event, but also to insert the new value, explicitly passed as parameter, in the dvvs. It returns a new dvvs that contains the new value v, represented by a new event performed by r and, therefore, appended at the head of the list for r. The context is only used to propagate causal information to the top vector, as we no longer keep it per version.

$$\begin{aligned}
\mathsf{sync}(S,S') &= \{(r,\max(n,n'),\mathsf{merge}(n,l,n',l')) \mid r\in R,(r,n,l)\in S,(r,n',l')\in S'\},\\
\mathsf{join}(C) &= \{(r,n)\mid (r,n,l)\in C\},\\
\mathsf{discard}(S,C) &= \{(r,n,\mathsf{first}(n-C(r),l))\mid (r,n,l)\in S\},\\
\mathsf{event}(C,S,r,v) &= \{(i,n+1,[v\mid l])\mid (i,n,l)\in S\mid i=r\}\cup\\
&\quad\ \{(i,\max(n,C(i)),l)\mid (i,n,l)\in S\mid i\neq r\}
\end{aligned}$$

7 Complexity and Evaluation

Table 2 shows space and time complexities of each causality tracking mechanism, for a single key. Lets consider U the number of updates (writes), C the number of writing clients, R the number of replica servers, V the number of concurrent versions (siblings) and S_w and S_r the number of replicas nodes involved in a put and get, respectively. Note that U and C are generally several orders of magnitude larger than R and V. The complexity measures presented assume effectively constant time in accessing or updating maps and sets. We also assume ordered maps/sets that allow a pairwise traversal linear on the number of entries.

lww is constant both in time and space, since it does not track causality and ignores siblings. Space-wise, ch and vv$_{\text{client}}$ do not scale well, because they grow linearly with writes and clients, respectively. dvv scales well given that typically there is little

concurrency per key, but it still needs a dvv per sibling. From the considered clocks, dvvs and $\text{vv}_{\text{server}}$ have the best space complexity, but the latter is not causally accurate.

Following our framework (Section 6), the time complexities are[1]:

- put is $\tilde{O}(discard + event + S_w \times sync)$ and get is $\tilde{O}(join + S_r \times sync)$;
- *event* is effectively $\tilde{O}(1)$ for ch, $\text{vv}_{\text{client}}$ and $\text{vv}_{\text{server}}$; is linear with V for dvv, because it has to check each value's clock; and is $\tilde{O}(R)$ for dvvs because it also merges the context to the local clock;
- *join* is constant for $\text{vv}_{\text{server}}$, since there is already only one clock; for ch, $\text{vv}_{\text{client}}$ and dvv it amounts to merging all their clocks into one; for dvvs, *join* simply extracts the top vector from the clock;
- *discard* is only linear with V in dvv, because it can check the partial order of two clocks in constant time; as for ch and $\text{vv}_{\text{client}}$, they have to compare the context to every version's clock; $\text{vv}_{\text{server}}$ and dvvs always compare the context to a single clock, and in addition, dvvs has to traverse lists of versions;
- *sync* resembles *discard*, but instead of comparing a set of versions to a single context, it compares two sets of versions. Thus, ch, $\text{vv}_{\text{client}}$ and dvv complexities are similar to *discard*, but quadratic with V instead of linear. Since $\text{vv}_{\text{server}}$ and dvvs have only one clock, the complexity of *sync* is linear on V.

7.1 Evaluation

We implemented both dvv and dvvs in Erlang, and integrated it with our fork of the NoSQL Riak datastore[2]. To evaluate the causality tracking accuracy of dvvs, and its ability to overcome the sibling explosion problem, we setup two equivalent 5 node Riak clusters, one using dvvs and the other $\text{vv}_{\text{server}}$.

We then ran a script[3] equivalent to the following: Peter (P) and Mary (M) write and read 50 times each to the same key, with read-write cycles interleaved (P writes then reads, next M writes then reads, in alternation). Figure 3 shows the growth in the number of siblings with every new write. The cluster with $\text{vv}_{\text{server}}$ had an explosion of false concurrency: 100 concurrent versions after 100 writes. Every time a client wrote with the its latest context, the clock in the server was already modified, thus generating and adding a sibling. However, with dvvs, although each write still conflicted with the latest write from the other client, it detected and removed siblings that were causally older (all the siblings present at the last read by that client). Thus, the cluster with dvvs had only two siblings after the same 100 writes: the last write from each client.

Finally, dvvs has already seen early adoption in the industry, namely in Riak, where it is the default logical clock mechanism in the latest release. As expected, it overcame the sibling explosion problem that was affecting real world Riak deployments, when multiple clients wrote on the same key.

[1] For simplicity of notation, we use the big O variant: $\tilde{O}$, that ignores logarithmic factors in the size of integer counters and unique ids.

[2] `https://github.com/ricardobcl/Dotted-Version-Vectors`

[3] `https://gist.github.com/ricardobcl/4992839`

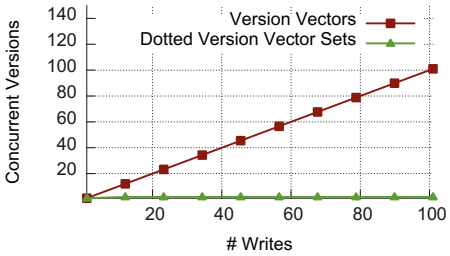

Fig. 3. Results of running two interleaved clients with 50 writes each

8 Related Work

The role of causality in distributed systems was introduced by Lamport [11], establishing the foundation for the subsequent mechanisms and theory [11,14,20,19,2,4]. In Section 3 we discussed the problems of solutions commonly used in eventually consistent stores. In this section, we discuss other related work.

Variability in the Number of Entities. The basic vector based mechanisms can be generalized to deal with a variable number of nodes or replicas. The common strategy is to map identifiers to counters and handle dynamism in the set of identifiers. Additions depend on the generation of unique identifiers. Removals can require communication with several other servers [7], or to a single server [15,1]. While dvv and dvvs avoid identifier assignment to clients, these techniques could support changes in the set of servers.

Exceptions on Conflicts. Some systems just detect the concurrent PUT operations from different clients and reject the update (e.g. version control systems such as CVS and subversion) or keep the updates but do not allow further accesses until the conflict is solved (e.g. original version of Coda [8]); in these cases, using version vectors (vv) with one entry per server is sufficient. However, these solutions sacrifice write availability which is a key "feature" of modern geo-replicated databases.

Compacting the Representation. In general, using a format that is more compact than the set of independent entities that can register concurrency, leads to lossy representation of causality [4]. Plausible clocks [21] condense event counting from multiple replicas over the same vector entry, resulting in false concurrency. Several approaches for removing entries that are not necessary have been proposed, some being safe but requiring running consensus (e.g. Roam [18]), and others fast but unsafe (e.g. Dynamo [5]) potentially leading to causality errors.

Extensions and Added Expressiveness. In Depot [12], the vv associated with each update only includes the entries that have changed since the previous update in the same node. However, each node still needs to maintain vv that include entries for all clients and servers; in a similar scenario, the same approach could be used as a complement to our solution. Other systems explore the fact that they manage a large number of objects to maintain less information for each object. WinFS [13] maintains a base vv for all

objects is the file system, and for each object it maintains only the difference for the base in a concise vv. Cimbiosys [17] uses the same technique in a peer-to-peer system. These systems, as they maintain only one entry per server, cannot generate two vv for tagging concurrent updates submitted to the same server from different clients, as discussed in Section 3 with $\mathsf{vv}_{\mathsf{server}}$. WinFS includes a mechanism to deal with disrupted synchronizations that allow to encode non sequential causal histories by registering exceptions to the events registered in vv; e.g. $\{a_1, a_2, b_1, c_1, c_2, c_4, c_7\}$ could be represented by $\{(a,2),(b,1),(c,7)\}$ plus exceptions $\{c_3, c_5, c_6\}$. However, using dvv with its system workflow, at most a single update event that is outside the vv is needed, and thus a single *dot* per version is enough. dvvs goes further, by condensing all causal information in a vv, while being able to keep multiple implicit *dots*. This ensures just enough expressiveness to allow any number of concurrent clients and still avoids the size complexity of encoding a generic non sequential ch. Wang et. al. [22] have proposed a variant of vv with $O(1)$ comparison time (like dvv), but the vv entries must be kept ordered which prevents constant time for other operations. Furthermore, it also incurs in the problems associate with $\mathsf{vv}_{\mathsf{server}}$, which we solved with dvvs.

9 Closing Remarks

We have presented in detail Dotted Version Vectors, a novel solution for tracking causality among update events. The base idea is to add an extra isolated event over a causal history. This is sufficiently expressive to capture all causality established among concurrent versions (siblings), while keeping its size linear with the number of replicas.

We then proposed a more compact representation — Dotted Version Vector Sets — which allows for a single data structure to accurately represent causal information for a set of siblings. Its space and time complexity is only linear with the number of replicas plus siblings, better than all current mechanisms that accurately track causality.

Finally, we introduced a general workflow for requests to distributed data stores. It abstracts and factors the essential operations that are necessary for causality tracking mechanisms. We then implemented both our mechanisms using those kernel operations.

Acknowledgements. This research was partially supported by FCT/MCT projects PEst-OE/EEI/UI0527/2014 and PTDC/EEI-SCR/1837/2012; by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n^{o} 609551, SyncFree project; by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-037281.

References

1. Almeida, P.S., Baquero, C., Fonte, V.: Interval tree clocks. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 259–274. Springer, Heidelberg (2008)
2. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Trans. Comput. Syst. 5(1), 47–76 (1987)

3. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 2000, p. 7. ACM, New York (2000)
4. Charron-Bost, B.: Concerning the size of logical clocks in distributed systems. Information Processing Letters 39, 11–16 (1991)
5. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Proceedings of Twenty-First ACM SIGOPS SOSP, pp. 205–220. ACM (2007)
6. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. ACM SIGACT News, 2002 (2002)
7. Golding, R.A.: A weak-consistency architecture for distributed information services. Computing Systems 5(4), 379–405 (1992)
8. Kistler, J.J., Satyanarayanan, M.: Disconnected operation in the Coda file system. In: Thirteenth ACM Symposium on Operating Systems Principles, vol. 25, pp. 213–225. Asilomar Conference Center, Pacific Grove (1991)
9. Klophaus, R.: Riak core: building distributed applications without shared state. In: ACM SIGPLAN Commercial Users of Functional Programming, CUFP 2010, p. 14:1. ACM, New York (2010), http://doi.acm.org/10.1145/1900160.1900176
10. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 35–40 (2010)
11. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
12. Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., Walfish, M.: Depot: Cloud storage with minimal trust. In: OSDI 2010 (October 2010)
13. Malkhi, D., Terry, D.: Concise version vectors in winFS. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 339–353. Springer, Heidelberg (2005)
14. Parker, D.S., Popek, G., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., Kline, C.: Detection of mutual inconsistency in distributed systems. Transactions on Software Engineering 9(3), 240–246 (1983)
15. Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Sixteen ACM Symposium on Operating Systems Principles, Saint Malo, France (October 1997)
16. Preguiça, N., Baquero, C., Almeida, P.S., Fonte, V., Gonçalves, R.: Brief announcement: Efficient causality tracking in distributed storage systems with dotted version vectors. In: Proceedings of the 2012 ACM Symposium on PODC, pp. 335–336. ACM (2012)
17. Ramasubramanian, V., Rodeheffer, T.L., Terry, D.B., Walraed-Sullivan, M., Wobber, T., Marshall, C.C., Vahdat, A.: Cimbiosys: a platform for content-based partial replication. In: Proceedings of the 6th USENIX Symposium on NSDI, Berkeley, CA, USA, pp. 261–276 (2009)
18. Ratner, D., Reiher, P.L., Popek, G.J.: Roam: A scalable replication system for mobility. MONET 9(5), 537–544 (2004)
19. Raynal, M., Singhal, M.: Logical time: Capturing causality in distributed systems. IEEE Computer 30, 49–56 (1996)
20. Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: In search of the holy grail. Distributed Computing 3(7), 149–174 (1994)
21. Torres-Rojas, F.J., Ahamad, M.: Plausible clocks: constant size logical clocks for distributed systems. Distributed Computing 12(4), 179–196 (1999)
22. Wang, W., Amza, C.: On optimal concurrency control for optimistic replication. In: Proc. ICDCS, pp. 317–326 (2009)