Dotted Version Vectors: Efficient Causality Tracking for Distributed Key-Value Stores

Nuno Preguiça
CITI/DI
FCT, Universidade Nova de Lisboa
Monte da Caparica, Portugal
nmp@di.fct.unl.pt

Carlos Baquero, Paulo Sérgio Almeida,
Victor Fonte, Ricardo Gonçalves
CCTC/DI
Universidade do Minho
Braga, Portugal
{cbm,psa,vff,tome}@di.uminho.pt

***Abstract*—In cloud computing environments, data storage systems often rely on optimistic replication to provide good performance to geographically disperse users and to allow operation 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. In this paper, first we review, and expose problems with current approaches to causality tracking in optimistic replication: these either lose information about causality or do not scale, as they require replicas to maintain information that grows linearly with the number of clients or updates. Then, we propose a novel, scalable solution that fully captures causality: it maintains very concise information that grows linearly only with the number of servers that register updates for a given data element, bounded by the degree of replication. Moreover, causality can be checked in $O(1)$ time instead of $O(n)$ time for version vectors. We have integrated our solution in Riak, and results with realistic benchmarks show that it can use as little as 10% of the space consumed by current version vector implementation, which includes an unsafe pruning mechanism.**

I. Introduction

The design of Amazon's Dynamo system [1] was an important influence to a new generation of databases, such as Cassandra [2] and Riak[1], 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 conflicting goals of *consistency*, *availability* and *partition-tolerance* only two of those can be achievable in the same system [3], [4]. Facing wide area 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 physical timestamps to arbitrate which concurrent updates should prevail. As a consequence some updates will be lost since a *last writer wins (LWW)* policy is enforced over concurrent updates. An approach avoiding lost updates must be able to maintain divergency until it can be reconciled: concurrent updates must be not only accurately detected, but also fully preserved.

Accurate tracking of concurrent data updates can be achieved by a careful use of well established causality tracking mechanisms [5], [6], [7], [8]. In particular, for data storage systems, version vectors [6] 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, current cloud storage systems, e.g. Dynamo and Riak, make several compromises regarding causality tracking, leading to lost updates and/or introduction of false concurrency. The reasoning behind these compromises is to achieve higher scalability, limiting the number of entries in the version vectors.

In this article we propose to overcome these limitations and present a new, and simple, causality tracking solution that allows both scalable and fully accurate tracking. The key idea of our approach is to maintain the identifier of the event separate from the causal past. Besides allowing the size of information to be bounded by the degree of replication, instead of the number of clients, this approach allows to verify causality in constant time (instead of $O(n)$ time with version vectors).

The remainder of this paper is organized as follows. Section II presents the system model. Section III surveys the current solutions used for causality tracking and discusses their limitations. Section IV presents the new mechanism: *Dotted Version Vectors*. Section V evaluates the performance of the proposed mechanism. Section VI discusses related work, and Section VII concludes the paper with some final remarks.

II. System model

Storage systems for cloud computing environments can be seen as composed of a set of interconnected storage server nodes that provide a data read/write service to a much larger set of clients. In a data center, a typical configuration includes both storage nodes and application server nodes. In each application server node, multiple threads execute independently, acting as an independent clients. Thus, even if the data center includes a similar number of storage nodes and application servers nodes, the number of clients if much larger.

Without loss of generality, we consider a standard key-value store interface that exposes two operations: reading, GET(KEY):VALUES,CONTEXT; and writing, PUT(KEY,VALUE, CONTEXT). (A delete operation can be implemented, for example, by executing a PUT() with a special value.) A given key is replicated in a subset of the server nodes, which we call

[1] http://www.basho.com/Riak.html

the replica nodes for that key. The GET() operation returns one or more concurrent values that are collected in the node handling the request, and an immutable CONTEXT that encodes the causal knowledge in the value(s). The PUT() operation submits a single value that supersedes all values associated to the supplied immutable CONTEXT. This CONTEXT is obtained in the client by a previous GET() operation.

Typically, clients engage in a sequence of GET() followed by PUT() using the last received CONTEXT, but any combination is allowed. Provided only a *nil* or a previously received CONTEXT is used, they can be re-used multiple times and in any order. A replica node allows multiple clients to concurrently update values for the same key. i.e. any interleaving of GET() and PUT() is permissible among clients. We assume no affinity, so clients are free to read values from a replica node and write them to different ones. The encoded causality information encoded in CONTEXT allows linking a GET() to a subsequent PUT().

The GET() and PUT() operations can be associated to given *read* and *write* quorums, that control the number or replica nodes contacted. However, this aspect will be largely orthogonal to the causality tracking task.

A salient aspect in these systems, is that all data changes to any given key are mediated by the server nodes that act as replica nodes for that key. Thus, since clients cannot engage in direct communication with each other, one can aim for scalable solutions for causality tracking that do no depend on the number of clients. Otherwise, known scalability lower bounds would apply [9].

An important aspect to consider when reasoning about the scalability of these approaches, is the existence of three different orders of magnitude at play: a small number of replica nodes for each key; a large number of server nodes; a huge number of clients, keys and issued operations. Thus, a scalable solution should avoid mechanisms that are linear with the highest magnitude and, if possible, it should strive to match the lowest scale, i.e. the number of replica nodes.

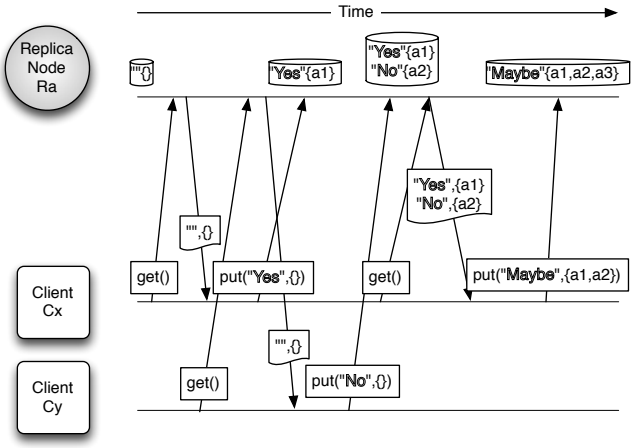

Fig. 1. Two clients concurrently modifying the same key on a replica node. Causal histories.

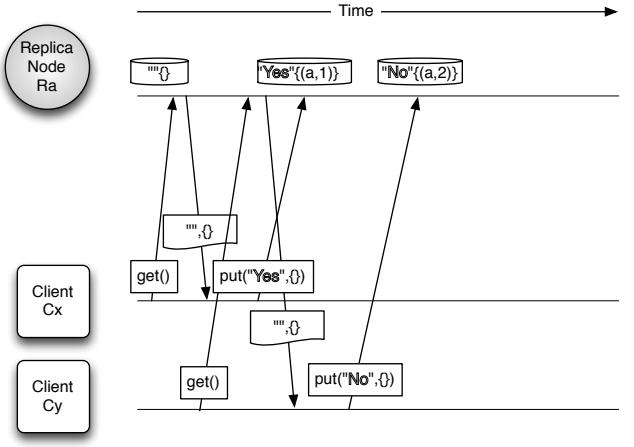

Fig. 2. Two clients concurrently modifying the same key on a replica node. Version vector with one id-per-server.

III. CURRENT APPROACHES

Under the defined system model one can consider the possible executions and the causality patterns that they produce. In these executions causality can be precisely characterized by *causal histories* [7]. Causal histories are sets of unique update event identifiers. Here we will consider the composition of unique server ids and a monotonic integer counter. The partial order of causality can be precisely tracked by comparing these sets by set inclusion. An history A causally precedes C iff $A \subset C$. Two histories are concurrent if neither include the other: $A \parallel B$ iff $A \not\subseteq B$ and $B \not\subseteq A$.

In Figure 1 we consider an execution where two clients, C_x and C_y, concurrently update the same key in the same replica node, R_a. Since updates are concurrent, the replica node will store the two conflicting values and only later a new value is produced that takes those values into account and produces a new value that resolves the conflict. Values whose causal history is strictly included in the causal history of another value are replaced by the later value: e.g. in this figure, MAYBE causally succeeds both YES and NO ($\{a_1\} \subset \{a_1, a_2, a_3\}, \{a_2\} \subset \{a_1, a_2, a_3\}$), and replaces them.

In systems that enforce a *last writer wins* policy, such as in Cassandra, concurrent updates are not represented in the stored state and only the last update prevails. Under LWW, for the run in Figure 1 where we have two concurrent updates, since YES happened to be registered before NO, it will be lost and the final outcome is NO. This leads to loss of information and the semantic conflict resolution that previously lead to MAYBE is no longer possible. Although some specific application semantics are compatible with a LWW policy, this simplistic approach is not adequate for many other application semantics. A correct tracking of concurrent updates is essential to allow all updates to be considered upon conflict resolution.

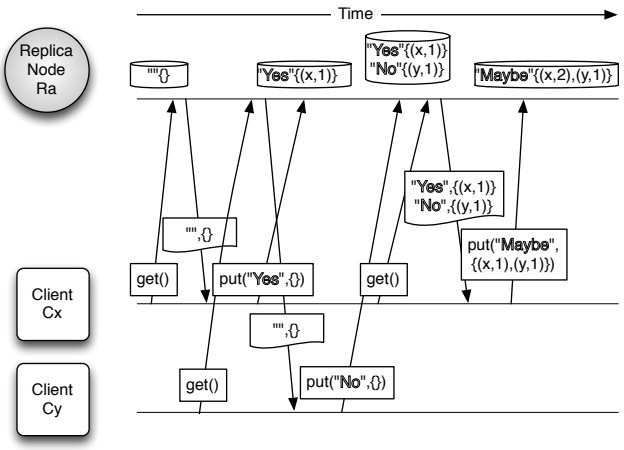

Fig. 3. Two clients concurrently modifying the same key on a replica node. Version vector with one id-per-client.

A. Version vectors with id-per-server

If causality is tracked by version vectors with one entry per server node, 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 identify the concurrent values and typically the last write will prevail. Figure 2 illustrates this situation. In the example NO overwrites YES. Notice that if the updates where handled by different replica nodes, say a and b, then they would be properly marked and could later be determined to be concurrent – unlike a pure LWW policy where one would have a higher "real time" clock and always overwrite the other (as occurs in Cassandra).

In all examples we represent version vectors as a set of pairs that depict an id and its associated integer counter, e.g. $\{(x,2),(y,4),(z,1)\}$. The classic representation for this version vector, $[2,4,1]$, assumes a total order on node ids, $x < y < z$, and a fixed number of nodes.

Version vectors with one id-per-server cannot express the degree of concurrency permitted by the system model and the consequence is the serialization of concurrent updates to the same server and in practice a LWW policy on those updates. The Dynamo system uses one entry per server node and thus is sensible to this issue.

One can argue that the server node could instead verify that the new update is concurrent with its current version by checking that the version vector included in the operation does not dominate the version vector of the current version (since $\{\} \not\geq \{(a,1)\}$ it was based on a past value). In this case, the replica node could reject the update, implementing a conditional write semantics. This approach is used, e.g., in the CVS version control system (although not necessarily relying on version vectors). In Coda [10], the update is accepted but no further access is allowed to the file until the conflict is solved by a special mechanism. These solutions go against the usual policy of write availability [1], the norm in modern key-value stores.

A simple solution to the lack of expressive power in the id-per-server approach is to use an id-per-client and thus match the number of concurrency allowed in the model. This was the approach followed in the Riak data store.

B. Version vectors with id-per-client

If each active client is allowed one entry in the version vector, one has enough entries to express the concurrency that the system produces. Once the server node correctly identifies each client (e.g. by including client ids in the PUT() API) it can adequately register updates by associating them to the client id. Figure 3 illustrates this. Notice that the two concurrent updates are properly registered and that the semantic reconciliation is performed and overwrites the previous values ($\{(x,1)\} < \{(x,2),(y,1)\}, \{(y,1)\} < \{(x,2),(y,1)\}$).

In the Riak implementation, since clients are stateless and do not keep their last counter locally, it is important that the last GET() CONTEXT includes the client's last write. The default quorum settings in Riak, ensures a *read your writes* semantics [11], so that the most recent update by a given client is present in the CONTEXT.

Although this approach does capture all system concurrency, it does so at the expense of a large state growth. In each key all replica nodes will end up storing the ids of all the clients that ever issued writes on that key. The Riak system tries to compensate this by using version vector pruning, a concept also used in the Dynamo system. However pruning has important consequences.

C. Version vector pruning

Consider a value v_p in the data store under some key. The context meta-data c_p (e.g. version vector) associated to the value v_p identifies the causal history of all the updates reflected on that value. By pruning c_p, some of these update events are lost from the context, and another competing value v_q for the same key and with context c_q, can now be compared differently. For instance, consider c'_p as the pruned version of c_p, if we have $c_q < c_p$, it can become that $c_q \parallel c'_p$. Old and obsolete data can creep back from the past.

In the example of Figure 3, in the last server version, pruning could be performed by forgetting the entry for client y. If by any chance, a different replica with the pruned vector received the original PUT from client y, value "NO" would revive and the system would assume it to be concurrent with "MAYBE".

Pruning typically leads to the introduction of false concurrency and further need for reconciliation. The higher the degree of pruning, the higher is the degree of false concurrency in the system. Depending on the properties of the reconciliation algorithms this can possibly even lead to lost updates.

Both Dynamo and Riak use pruning to bound the growth of version vectors. Dynamo does not state what is the maximum number of entries per clock. On the other hand, Riak does not use a simple value for this. Instead, it uses an interval of values in which it may prune or not the clock, depending on

data freshness. By default, clocks with size between 20 and 50 are subject to pruning (uses timestamps to determine data freshness), while after 50 they are always pruned.

In the next section we present a solution to the problems identified in the current mechanisms and systems. The new mechanism, *dotted version vectors*: (1) accurately tracks all the causality allowed in the system model (that reflects the operation of modern key-value stores); (2) scales with the (small) number of replica nodes for a key; (3) and does not require pruning.

IV. Dotted Version Vectors

We now present a concise and accurate representation for the clocks to be used as a substitute for the classic version vectors in key-value stores. The mechanism allows a lossless representation of causality (contrary to, e.g., Plausible Clocks) while only using server-based ids, and only a component per replica node, thus avoiding the space consumption explosion that occurs in id-per-client approaches.

One could be led to think that the conciseness obtained using server-based ids would contradict Charron-Bost minimality result [9]. Such is not the case because, not only the problems addressed by version vectors and vector clocks are different [12], but essentially because the present scenario does not involve direct client-to-client interaction: all interactions are intermediated by a server node.

Using server-based ids, a precise representation of causality could be obtained by using causal histories (sets of uniquely identified events in the causal past); but these are unreasonable in practice due to space consumption. On the other hand, a version vector compresses causal histories by representing, for each component, all events up to a given sequence number, i.e., a downward closed set of events (or down-set for short). This is not enough to identify the concurrent versions generated when several clients perform a get of some key from one server and then all perform a put, using the same causal past.

The basic idea of dotted version vectors (DVV) is to add the possibility of representing an individual event (a "dot") with an isolated sequence number outside the contiguous range. The dot is kept separate from the causal past and it identifies the globally unique event being described, the new version in this case. This allows describing events as concurrent (due to having different dots) even if they were generated from the same causal past (and have identical version vectors in the causal past).

Figure 4 clarifies the differences in expressive power of causal histories, version vectors and dotted version vectors. There we assume three nodes $\{a, b, c\}$ and events tagged with sequence numbers in each node. A causal history is able to represent an arbitrary set of events such as $\{a_3, b_1, b_2, c_2, c_4\}$ (left); a version vector is able to represent a down-set: all events up to some given number for each node such as $\{a_1, a_2, a_3, b_1, b_2, c_1, c_2\}$ (center); a dotted version vector is able to represent a down-set plus an extra isolated event, such as $\{a_1, a_2, a_3, b_1, b_2, c_1, c_2, c_4\}$ (right), where event c_4 falls outside the range from 1 to 2.

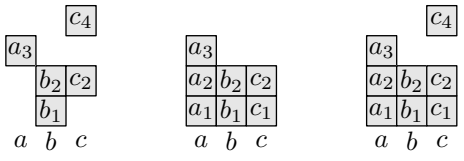

Fig. 4. Sets of events representable by causal histories (left), version vectors (center) and dotted version vectors (right).

Keeping the event described separate from the causal past allows the order between events to be computed in $O(1)$: to see if $X \leq Y$ it is enough to check if the sequence number for the dot of X is less or equal to the corresponding sequence number in the Y version vector.

A. Definition

A dotted version vector (DVV) is a logical clock which consists of a pair d, v, where v is a traditional version vector (a mapping from identifiers to integers) and the dot d is a pair (i, n), with i a node identifier and n an integer. The dot represents the globally unique event being described, while the VV represents the causal past. The events represented by a DVV can be characterized by a semantic function from DVV (or sets of clocks) to causal histories:

$$\begin{aligned} \mathcal{C}[\![((i,n),v)]\!] &= \{i_n\} \cup \mathcal{C}[\![v]\!] \\ \mathcal{C}[\![v]\!] &= \bigcup_{(i,n)\in v} \{i_m \mid 1 \leq m \leq n\}, \end{aligned}$$

where i_n denotes the event with sequence number n generated by node i, and $\mathcal{C}[\![v]\!]$ is the semantic function for traditional version vectors. With this definition, the causal history:

$$\{a_1, a_2, a_3, b_1, b_2, c_1, c_2, c_4\}$$

that cannot be represented by a version vector, will be represented by the dotted version vector $((c, 4), \{(a, 3), (b, 2), (c, 2)\})$. An alternative, more compact, encoding can share the node id in the version vector and use a triplet to represent the dot. In this example, it would lead to the compact representation $\{(a, 3), (b, 2), (c, 2, 4)\}$.

B. Partial order

The partial order on DVV can be defined, as usual, in terms of inclusion of causal histories; i.e.:

$$X \leq Y \iff \mathcal{C}[\![X]\!] \subseteq \mathcal{C}[\![Y]\!],$$

which amounts to:

$$((i,n),u) \leq ((j,m),v) \iff \{i_n\} \cup \mathcal{C}[\![u]\!] \subseteq \{j_m\} \cup \mathcal{C}[\![v]\!].$$

Given that each dot is generated as a globally unique incomparable event, using the notational convenience of considering a mapping to return 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: $u \leq v \iff \forall_{(i,n)\in u}.\, n \leq v(i)$.

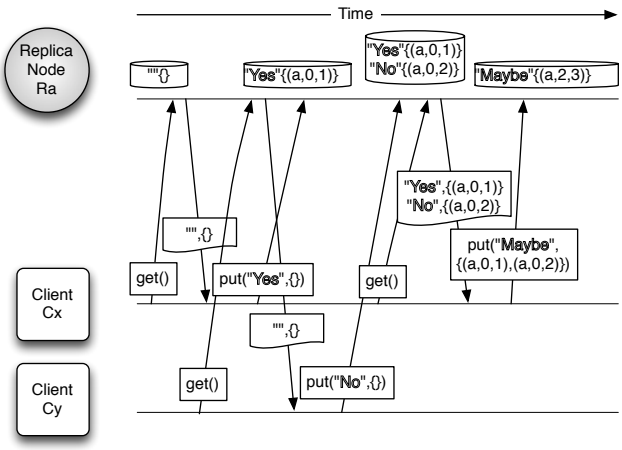

Fig. 5. Two clients concurrently modifying the same key on a replica node. Dotted version vectors.

1) O(1) computation of the partial order: A significant difference between DVV and traditional clock mechanisms such as VV is that while these conflate present event and causal past, DVV keep the present event (in the dot) separated from the causal past. An important consequence of this is that if the dot in X is contained in the causal past of Y, it means that Y was generated causally following X, which implies that Y also contains the causal past of X. This means that for any DVV $((i,n),u)$ and $((j,m),v)$, if $n \leq v(i)$ then it follows that we must also have $u \leq v$. This means that there is no need for the comparison of the VV components and the order can be computed as an $O(1)$ operation, simply as:

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

This partial order allows concurrent clocks even using only a single replica node identifier. As an example:

$$((r,4),\{(r,3)\}) \parallel ((r,5),\{(r,3)\}),$$

as they represent the causal histories:

$$\{r_1,r_2,r_3,r_4\} \parallel \{r_1,r_2,r_3,r_5\},$$

This situation will arise when $((r,4),\{(r,3)\})$ is stored in a replica node and a client, which has previously read some value and got a context which represents the same causal past $\{(r,3)\}$, now performs a put using this context. This situation is very common but cannot be handled with current mechanisms using server-based identifiers.

In Figure 5, we present our usual run using dotted version vectors in the alternative compact notation. It can be seen that causality is accurately tracked, even tough per-server identifiers are used. Notice that the replica node, when handling a *put* operation on behalf of a client node, looks at the current versions in the data store to determine the next available integer and uses it to register the update and delete obsolete versions, if any.

C. Kernel operations for eventual consistency

We have seen that, as soon as clients can perform concurrent updates managed by a single replica node, several concurrent versions may have to be kept in that node. These version sets are returned by a get operation, and their clocks are supplied as the context in a put operation.

In this section we argue that, even though a key-value store wants to make available to clients the *get* and *put* operations (essentially), the mechanics of a distributed key-value store, in terms of causality tracking, should be based on two more core functions on the sets of logical clocks of replicas:

- **sync**(S_1, S_2): takes two clock sets and returns a clock set. It returns a set of the more up-to-date concurrent clocks for the union of S_1 and S_2, while discarding obsolete clocks. When accessing versions stored in different replicas, this operation identifies the versions that are not overwritten by some other version;
- **update**(S, S_r, r): takes a set of clocks S (the context supplied by the client), the set of clocks S_r in the replica node (for the given key), and the replica node id r, and returns a clock for the new version written by a client in a put operation. This new DVV contains a globally unique event and also dominates all clocks sent in the context. When writing a new version, this operation can be used to generate a new clock that overwrites clocks of previously read versions.

The idea is that both a get and a put are complex operations which have several steps (e.g., for a get, receiving request, asking replica nodes for values, merging the partial results, and finally replying). To reason about clock mechanisms it helps to consider these more core sync and update functions and to define a get using sync, and a put using both sync and update. Different clock mechanisms, such as DVV, can then be defined for the use in the sync and update operations.

We now describe how a key-value store can implement the operations it intends to make available to clients (get and put) by using the kernel operations sync and update. The proposed mechanics have some deviation from the current practice. We will discuss practical impact in Section V.

Operation ***get***(k)*:* When a client asks some server node N to perform a get of some key k (step 1 in Figure 6):

- N computes the set of replica nodes R for k;
- N asks a subset of nodes in R for the value for that key. Depending on the expected semantics, this subset may contain, for example, a single node or a quorum of nodes (step 2);
- N waits for the replies (step 3);
- N performs a reduce of the replies using the sync operation, which discards obsolete values and returns only the more up-to-date concurrent values, and replies to the client (step 4).

Operation ***put***(k, v, S)*:* When a client asks some server node N to perform a put for some key (step 1 in Figure 7):

- N computes the set of replica nodes R for k;

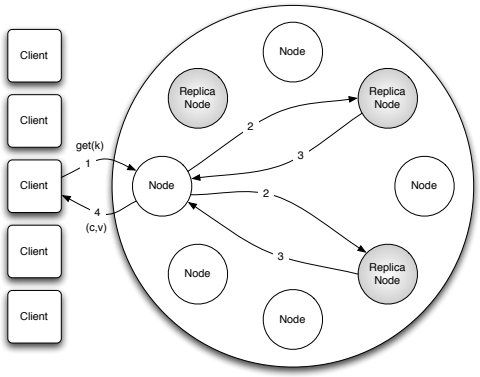

Fig. 6. A get operation.

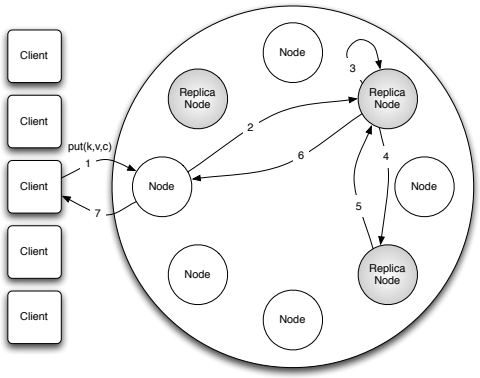

Fig. 7. A put operation.

- if N is a replica node for k, then N will coordinate the request; otherwise N will forward the request to some replica node for k, that will act as coordinator (step 2);
- the coordinator C performs an update operation, resulting in a clock value $u = \text{update}(S, S_C, C)$, performs a sync between u and the local set of concurrent instances, which serves to discard instances now made obsolete, and stores the result of the sync $S'_C = \text{sync}(S_C, \{u\})$ (step 3); this state transition from S_C to S'_C, comprised of the composition of update and sync, must be implemented as an atomic operation (for any given key; different keys can be handled concurrently);
- C sends S'_C to a subset of other nodes in R. Depending on the expected semantics, this subset may, for example, be empty or contain a quorum of nodes (step 4);
- each of those nodes performs a sync between S'_C and the local set of instances, stores the result of the sync $S'_i = \text{sync}(S_i, S'_C)$, and acknowledges to C;
- C waits for the replies (if the subset is not empty) (step 5);
- C acknowledges to node N (step 6), which in turn acknowledges to the client (or C acknowledges directly if that is possible) (step 7).

Replication-aware client library: We have described the steps taken when a generic load balancer or client library is used. While any node can coordinate a get operation, for a put the coordinator must be a replica node. Using a replication-aware client library or load balancer will help in reducing the response time of a put by removing the forwarding hop.

D. Operations on DVV

1) The sync function: The function sync produces a set of concurrent clocks that describe the collective causal past of the two sets of clocks in the parameters. It simply returns elements from the sets supplied, and it can have a general definition only in terms of the partial order on clocks, regardless of their actual representation:

$$\begin{aligned}\text{sync}(S_1, S_2) &= \{x \in S_1 \mid \nexists y \in S_2 . x < y\} \cup \\ &\quad \{x \in S_2 \mid \nexists y \in S_1 . x < y\}\end{aligned}$$

2) The update function: An update registered on a replica node r containing the set of versions S_r (for the given key), and with client supplied context S can have a reference definition in terms of causal histories using replica node ids plus sequence numbers to denote globally unique events, as:

$$\begin{aligned}\text{update}(S, S_r, r) &= \{r_{n+1}\} \cup \bigcup S \qquad \text{with} \\ &\quad n = \max(\{0\} \cup \{x \mid r_x \in \bigcup S_r\}).\end{aligned}$$

This definition states that the result of an update contains the causal history supplied by the client plus a new unique event tagged by the replica node using the next available value in the sequence for that node (for the given key).

To define the update function over dotted version vectors, we make use of some auxiliary functions. The ids function gives the set of identifiers in a pair, VV, DVV or set of DVV:

$$\begin{aligned}\text{ids}((i, \_)) &= i, \\ \text{ids}((i, n), v) &= \{i\} \cup \text{ids}(v), \\ \text{ids}(X) &= \{\text{ids}(x) \mid x \in X\}.\end{aligned}$$

The $\lceil \_ \rceil_\_$ function takes a DVV or set of DVV and a replica node identifier and returns the maximum sequence number of the events from that replica node represented:

$$\begin{aligned}\lceil((i, n), v)\rceil_r &= \max(\{n \mid i = r\} \cup \{v(r)\}), \\ \lceil S \rceil_r &= \max(\{0\} \cup \{\lceil C \rceil_r \mid C \in S\}).\end{aligned}$$

The update function can now be defined:

$$\text{update}(S, S_r, r) = ((r, \lceil S_r \rceil_r + 1), \{(i, \lceil S \rceil_i) \mid i \in \text{ids}(S)\}).$$

It is easy to see that under this definition, the update function produces a DVV that represents the same causal history as the one produced by the reference definition in terms of causal histories, while keeping the new globally unique event separated in the dot.

E. Correctness

The operations on a key-value store invoked by clients (get and put) resort to the kernel operations sync and update. These operate on (and return) sets of clocks. DVV are expressive enough to represent causality accurately in the key-value store setting because they exploit an important point: that single clocks are not a first class entity that can be operated upon by clients. A client may perform a get, which may return a set of concurrent replicas and an opaque context for the corresponding set of clocks. The client may then use the context on a subsequent put operation, but cannot operate upon individual clocks from that context. E.g., a client, having receiving 3 concurrent replicas, cannot invoke a put that depends only on two of them, because a single opaque context is returned by a get and this same context must be supplied in the subsequent put.

First we define the following predicate that formalizes the notion of down-set for sets of clocks:

Definition 1 (Downset): A set of clocks S is called a down-set, written downset(S), if and only if $\forall i \in \text{ids}(S). \forall\, 1 \leq n \leq \lceil S \rceil_i . i_n \in \mathcal{C}[\![S]\!]$.

This is true for sets of clocks for which the union of the corresponding causal histories are downward closed sets, under the order over events $r_i \leq s_j \iff r = s \wedge i \leq j$. In other words, the predicate is true if, for each node r, the set contains all events generated at r up to some sequence number.

The reason that makes it possible to have an accurate representation of causality using DVV is that, as we will show, all sets of clocks, kept at replica nodes or returned to clients, are down-sets. One of the reasons for this is that the set of down-sets is closed under the sync operation:

Lemma 1: If X and Y are down-sets, then sync(X, Y) is also a down-set.

Proof: Trivial, from the definition of sync: the set of events corresponding to the result of sync is the union of the corresponding events of the arguments, i.e., $\mathcal{C}[\![\text{sync}(X, Y)]\!] = \mathcal{C}[\![X]\!] \cup \mathcal{C}[\![Y]\!]$, which means it is a downward closed set. ∎

Proposition 1: For each key, in a given system containing a set of replica nodes R, each $r \in R$ storing a replica set S_r, then $\forall r \in R.\, \text{downset}(S_r)$.

Proof: By induction on a trace of core operations (sync and update) that produces the current state. (Even though there is concurrent execution of get and put by different clients, the core sync and update operations are atomic; they can be serializable into some sequential trace.) Any new set of clocks S'_r to be stored in a replica node r can only be the result of one of two cases: 1) in a coordinator node, from the atomic composition of sync and update: $S'_r = \text{sync}(S_r, \{u\})$, where $u = \text{update}(S, S_r, r)$. By the induction hypothesis the sets of clocks stored, namely S_r, are down-sets; S sent by the client as context is also a down-set, from the previous lemma, because it is the result of one or more sync operations of down-sets stored in replica nodes. As S and S_r are down-sets, S'_r will also be a down-set because: although $\{u\}$ itself may not be a down-set, for any identifier i other than r, the computed mapping $(i, \lceil S \rceil_i)$ represents a contiguous range of events starting from 1 for identifier i in the corresponding causal history of S; the sync between $\{u\}$ and S_r will therefore represent a contiguous range of events in what concerns these identifiers; for identifier r, as S_r represents all events from r up to $\lceil S_r \rceil_r$, and u contains only one more event with number $\lceil S_r \rceil_r + 1$, then S'_r represents a contiguous range starting from 1 for id r; 2) in a replica node r that receives S'_C computed by the coordinator C in a put, $S'_r = \text{sync}(S_r, S'_C)$ will also be a down-set because S'_C sent by the coordinator is a down-set, S_r is a down-set by the induction hypothesis and because the sync of down-sets produces down-sets. ∎

Having shown that all sets of DVV stored or returned to clients are down-sets, we now show that each DVV generated encodes causality information correctly and that the exposed operations perform as they should: get returns sets of concurrent DVV and put discards obsolete replicas and stores concurrent updates.

From the definition of sync, the set of DVV in the result represents the union of the corresponding causal histories of its arguments. This means that the result of a get describes accurately the causal past of the replicas replied to the coordinator of the get and reduced through the repeated use of sync. It can also be seen from the definition of sync that it discards obsolete replicas, returning only the more up-to-date ones.

From the above lemma and proposition, the clock set S sent from a client is a down-set. Given that a DVV can represent a down-set plus an isolated event, this means that any new DVV u, which is computed by the update $u = \text{update}(S, S_C, C)$ represents accurately the causality information for the new replica: the union of the causal histories corresponding to clocks in S (the context sent by the client) plus a new globally unique event (using the next available sequence number for the coordinator id, for that key). Each put obtains a globally unique event because each makes use of the atomic composition of the operations update and sync: if two puts for the same key are being served concurrently, one will use the S'_C generated by the other in its invocation of update.

In a put, the sync of the singleton $\{u\}$ containing the newly generated DVV with the existing replicas discards obsolete values while maintaining the set of DVV that represent the concurrent replicas. The resulting set of DVV is then sent to other replica nodes, where a sync similarly discards obsolete replicas and keeps the more up-to-date concurrent ones.

V. Evaluation

We implemented and evaluated the usage of Version Vectors (VV) and Dotted Version Vectors (DVV) in Riak. Implementing DVV in Riak was rather trivial. The PUT operation pipeline was modified to follow the description of Figure 7. In addition to the DVV source code, only a hundred of lines of code were modified to accommodate these changes.

Regarding the evaluation, the setup was a Riak cluster running on 6 machines, and another machine simulating the clients. The simulated client's request rate was chosen to ensure resource exhausting did not occur during the evaluation.

Workload	Type	Get		Put		Update		Clock	Values
		Mean (ms)	95th (ms)	Mean (ms)	95th (ms)	Mean (ms)	95th (ms)	Size (bytes)	per Key (average)
60% GET	VV	7.65	15.9	5.71	10.1	14.4	24.0	790	1.34
10% PUT	DVV	3.16	5.25	4.31	6.27	7.76	10.9	127	1.31
30% UPD	$\frac{DVV}{VV}$	**0.41**	**0.33**	**0.76**	**0.62**	**0.54**	**0.46**	**0.16**	**0.98**
30% GET	VV	10.4	21.6	7.48	13.8	18.8	31.9	859	1.20
10% PUT	DVV	3.45	5.83	4.56	6.59	8.39	11.8	123	1.16
60% UPD	$\frac{DVV}{VV}$	**0.33**	**0.27**	**0.61**	**0.48**	**0.45**	**0.37**	**0.14**	**0.97**

TABLE I
DVV AND VV BENCHMARKS WITH A GENERIC APPROACH.

Workload	Type	Get		Update		Clock	Values
		Mean (ms)	95th (ms)	Mean (ms)	95th (ms)	Size (bytes)	per Key (average)
Browsing Mix	VV	2.15	3.63	5.00	7.70	159	1.00081
	DVV	2.01	3.49	5.70	8.80	89	1.00051
	$\frac{DVV}{VV}$	**0.94**	**0.96**	1.13	1.15	**0.56**	**0.99970**
Shopping Mix	VV	2.84	5.00	6.80	11.0	117	1.00066
	DVV	2.77	4.94	7.70	12.8	82.0	1.00039
	$\frac{DVV}{VV}$	**0.98**	**0.99**	1.13	1.16	**0.70**	**0.99973**
Ordering Mix	VV	7.70	16.2	14.4	24.0	682	1.00549
	DVV	2.95	4.76	7.40	10.0	113	1.00425
	$\frac{DVV}{VV}$	**0.38**	**0.29**	**0.51**	**0.42**	**0.17**	**0.99877**

TABLE II
DVV AND VV BENCHMARKS WITH TPC-W APPROACH.

We used N=3, R=W=2, with N being the total number of replicas, while R and W depict the size of the read and write quorum. There were 50k keys, accessed using a Pareto distribution (20% of the keys were requested 80% of the time), and value payload was set to a fixed size of 1 KB or 5KB. Runs took 20 minutes each, sufficient time to achieve enough operations (about 2 million) and a stable state for measuring. million operations.

We considered three core actions on clients: (1) a simple GET, returning value(s); (2) a blind PUT where a value is written in a given key, with no causal context supplied (this operation will always increase concurrency); (3) an update, that is expressed by a GET returning value(s) and a context, a 50 ms delay, and a PUT that re-supplies the context and writes a value that supersedes those acquired in the GET (reducing the possible concurrency in the GET).

From these three core actions we evaluated two benchmarks that considered different workload mixes. The first benchmark was to do a simple generic distribution load, with the proportion of blind puts kept at 10% and interchanged proportions of 30% versus 60% for gets and updates. The size per value was fixed at 1KB. The second benchmark was to simulate TPC-W [13] workloads, using the "Shopping Mix" (80% reads, 20% writes) with a fixed value size of 5KB, the "Ordering Mix" (50% reads, 50% writes) and the "Browsing Mix" (95% reads, 5% writes), both with 1KB per value. Reads were done with the normal GET operation, while writes were done in updates, a GET followed by a PUT.

A. Comparison of overall latency

The first, generic, benchmark results are in table I, while the TPC-W approach benchmark results are in table II. Both tables show the DVV/VV ratio that helps compare the two mechanisms, values smaller than 1.0 show an improvement and are depicted in **bold**.

In all tests we find that clock size is always (much) smaller in DVV, even with the (default) pruning that occurs with Riak VV. One can also confirm that pruning is occurring, because all the tests reveal that there were more concurrent values in the VV case. The difference in the number of values per key between the two logical clocks, results from false conflicts created by pruning. We recall that since Riak VV resort to pruning they do not reliably represent concurrency, and introduce false conflicts that need to be resolved. Having no pruning, our DVV implementation accurately tracks concurrency, while still allowing an expressive reduction of metadata size. It is easy to see that even if the default pruning activation threshold was lowered in Riak VV case, although it would reduce clock sizes, this would also lead to an increase of false concurrency and higher numbers of values per key.

Regarding performance, the generic benchmark results show that using a value payload of 1KB, the write and read operations were much better then using VV. Having less conflicts, and factoring the smaller clock size, on average operations transfer smaller data (1.8KB versus 1.2KB).

In the TPC-W case, the first thing we can see is that concurrency (rate of conflicts, measured by values per object) is very low, as it would be expected in a more realistic setting (concurrency rates in Dynamo's paper [1] are very similar to these). Read operations were always better, or pretty even between both mechanisms. This is to be expected since the read pipeline was not modified by our implementation, but DVV is usually smaller, thus requiring less data to be transferred.

Write operations were pretty good in the ordering mix, since (like the generic approach) each value was 1KB and the difference in clock size was significant. In contrast, the browsing mix also had 1KB per value, but the difference in clock sizes was not very large (too few writes in 20 minutes for the VV clock to grew significantly, but with time, it would probably grow much larger). Then, on average, values with VV and DVV had 1.16KB and 1.09KB in size, respectively. The same can be said of the shopping mix, in this case 5.12KB and 5.08KB for the VV and DVV, respectively. Therefore, in the shopping mix and browsing mix, the difference in clock size was not sufficient to make up for the changes we had to made in the write pipeline. Simply put, using DVV in Riak, when writing some value, the coordinator has to send every conflicting value to replicas. Moreover, if the coordinator is not a replica for that key, then it has to forward the write request to a new coordinator that is also a replica. In the standard Riak implementation, the VV case, the write pipeline is simpler, only the new client value is passed to replicas and every node can be a coordinator for any write request.

B. Comparison of clock sizes

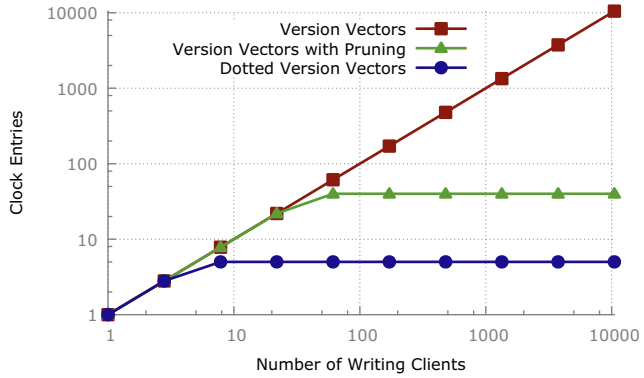

Fig. 8. Theoretical growth in clock size, as factor of writing clients.

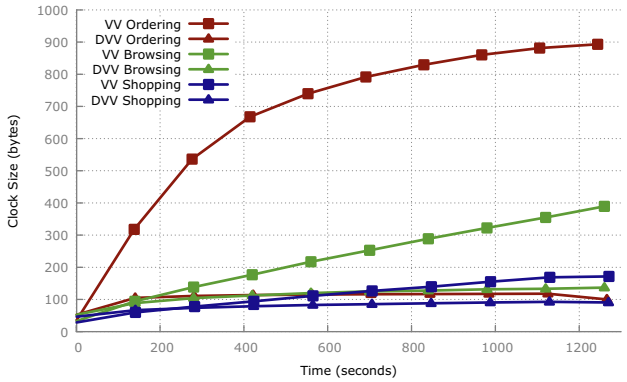

Fig. 9. Real growth in clock size using TPC-W workload mixes.

Figure 8 illustrates the theoretical effect of the number of writing clients in the number of clock entries, and thus the overall clock size. DVV stabilizes in size when the number of entries reaches the replication factor (usually 3), while VV with id-per-client grow indefinitely. Thus, in practice, systems usually resort to pruning to control its growth. In Riak's case, the imposed limit to the number of kept ids, and the trigger to pruning, is in the 20 to 50 range.

In Figure 9 we depict the evolution of the average clock size per key during the execution of the TPC-W based benchmarks in the Riak cluster. Here we can see that the DVV size becomes constant after a short time, whereas the Riak VV size is constantly increasing until stabilizing somewhere close to 1KB in the Ordering Mix case. Notice that it only stabilizes because of pruning in Riak VV, if not that it would grow linearly. The other workloads using Riak VV did not have enough time to stabilize, but would eventually be similar to the ordering mix. DVV has more or less 3 entries per clock (N=3) and size of 100 bytes, thus 1000 bytes in average for each VV means that it has roughly 30 entries, in line with the pruning range of Riak $[20 - 50]$.

VI. Related Work

Version tracking solutions as used in cloud storage systems are rooted on Lamport' seminal work on the definition and role of causality in distributed systems [5]. This work was the foundation for subsequent advances in causality's basic mechanisms and theory, including the introduction of version vectors [6] for tracking causality among replicas in a distributed storage system and vector clocks [14], [15] for tracking causality of events in a distributed systems.

Most of this initial work dealt with a fixed, mostly small, number of participants. Later, several systems introduced mechanisms for the dynamic creation and retirement of vector entries to be used when a server enters and leaves the system. While some techniques required the communication with several other servers [16], others required communication with a single server [17]. Interval Tree Clocks [18] are able to track causality in a dynamic, decentralized scenario where entities can be autonomously created and retired. Other systems, such as Dynamo [1], use unsafe techniques to remove entries, that are expected not to be necessary, based on time.

Even with these mechanisms, tracking causality through version vectors or vector clocks requires a space linear with the number of entities in the system, posing scalability problems for system with a large number of elements [9]. This problem is experienced in practice, for example, in cloud computing storage systems, as discussed in Section III.

The Roam system [19] runs a consensus protocol to decrease, in all servers, the value of all entries of the version vector by a constant value. The system only keeps the entries that are larger than zero. The *dependency sequences* [20] mechanism assumes a scenario where dynamic, weakly-connected sets of entities (mobile hosts) communicate through designated proxy entities chosen from a stable, well-connected (mobile service stations). The mechanism maintains information about the causal predecessors of each event. It needs to take periodic global snapshots to prune discardable causality-tracking metadata.

In Depot [21], the version vector 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 version vectors 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 storage systems explore the fact that they manage a large number of objects to maintain less information for each object. In Microsoft's WinFS [22], a base version vector for all objects is maintained for the file system, and each object maintains only the difference for the base in a concise version vector. In Cimbiosys [23], the authors suggest the use of the same technique in a peer-to-peer system. These systems, as they maintains only one entry per server, cannot generate two concurrent version vectors for tagging concurrent updates submitted to the same server from different clients, as discussed in Section III. In a separate WinFS work [24] the authors describe a mechanism that allows encoding of

non sequential causal histories by registering exceptions to the sequence; e.g. $\{a_1, a_2, b_1, c_1, c_2, c_3, c_7\}$ could be represented by $\{(a, 2), (b, 1), (c, 7)\}$ plus exceptions $\{c_4, c_5, c_6\}$. One important feature in dotted version vectors is that following the update rules for the target environment, at most a single update event that is outside the initial sequence is needed, and thus a single *dot* is enough. Additionally, by isolating the dot that identifies the version, causality can be checked in $O(1)$ time instead of $O(n)$ time.

Another direction is to use unsafe space-folding approaches that can reduce the storage and communication overhead at the expense of less accuracy of the causality relation captured by these mechanisms. Although devised as an alternative not to version vectors but to vector clocks, *plausible clocks* [25] propose techniques for condensing event counting from multiple replicas over the same vector entry. The resulting order does not contradict the causal precedence relation but because counters are effectively shared between processes, some concurrent events will be perceived as causally related. In fact, the previously mentioned Lamport clocks [5], are a notable example of plausible clocks.

VII. Conclusion

We have introduced *dotted version vectors*, a novel solution for tracking causal dependencies among update events. The base idea of our solution is to add the capability to represent an extra isolated event over the downward closed causal history described by version vectors.

Dotted version vectors allow an accurate tracking of causality among updates executed by multiple clients, while using server-based identifiers. Their size is only linear with the number of servers that register the updates, being bounded by the degree of replication. When compared with previous accurate proposals that require client-based identifiers, linear with the number of clients, our solution is much more efficient, as the number of clients tends to be several orders of magnitude larger than the number of servers that register updates for a given data element. Additionally, causality can be checked in $O(1)$ time instead of $O(n)$ of previous proposals.

Our solution is simple and practical: we have modified the Riak key-value store to use it. Evaluation showed a significant reduction in the size of metadata, and a good reduction in the latency when serving requests. Other relevant benefits are the elimination of false conflicts, and the simplification of the key-value store API, avoiding the need to generate and transmit globally unique client identifiers.

References

[1] 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 *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, April 2010.

[3] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '00. New York, NY, USA: ACM, 2000, pp. 7–.

[4] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent available partition-tolerant web services," in *In ACM SIGACT News*, 2002, p. 2002.

[5] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[6] D. S. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *Transactions on Software Engineering*, vol. 9, no. 3, pp. 240–246, 1983.

[7] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distributed Computing*, vol. 3, no. 7, pp. 149–174, 1994.

[8] M. Raynal and M. Singhal, "Logical time: Capturing causality in distributed systems," *IEEE Computer*, vol. 30, pp. 49–56, Feb. 1996.

[9] B. Charron-Bost, "Concerning the size of logical clocks in distributed systems," *Information Processing Letters*, vol. 39, pp. 11–16, 1991.

[10] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," in *Thirteenth ACM Symposium on Operating Systems Principles*, vol. 25, Asilomar Conference Center, Pacific Grove, US, 1991, pp. 213–225.

[11] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch, "Session guarantees for weakly consistent replicated data," in *International Conference on Parallel and Distributed Inormation Systems*, Austin, TX, US, Sep. 1994.

[12] J. B. Almeida, P. S. Almeida, and C. Baquero, "Bounded version vectors," in *DISC*, ser. Lecture Notes in Computer Science, R. Guerraoui, Ed., vol. 3274. Springer, 2004, pp. 102–116.

[13] T. P. P. C. (TPC)., "Tpc benchmark w(web commerce) specification version 1.8," 2002.

[14] C. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *11th Australian Computer Science Conference*, 1989, pp. 55–66.

[15] F. Mattern, "Virtual time and global clocks in distributed systems," in *Workshop on Parallel and Distributed Algorithms*, 1989, pp. 215–226.

[16] R. A. Golding, "A weak-consistency architecture for distributed information services," *Computing Systems*, vol. 5, pp. 5–4, 1992.

[17] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in *Sixteen ACM Symposium on Operating Systems Principles*, Saint Malo, France, Oct. 1997.

[18] P. S. Almeida, C. Baquero, and V. Fonte, "Interval tree clocks," in *Proceedings of the 12th International Conference on Principles of Distributed Systems*, ser. OPODIS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 259–274.

[19] D. H. Ratner, "Roam: A scalable replication system for mobile and distributed computing," Ph.D. dissertation, 1998, uCLA-CSD-970044.

[20] R. Prakash and M. Singhal, "Dependency sequences and hierarchical clocks: Efficient alternatives to vector clocks for mobile computing systems," *Wireless Networks*, pp. 349–360, 1997, also presented in Mobicom96.

[21] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," in *OSDI 2010*, Oct. 2010.

[22] D. Malkhi and D. B. Terry, "Concise version vectors in winfs," in *DISC*, ser. Lecture Notes in Computer Science, P. Fraigniaud, Ed., vol. 3724. Springer, 2005, pp. 339–353.

[23] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat, "Cimbiosys: a platform for content-based partial replication," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2009, pp. 261–276.

[24] L. Novik, I. Hudis, D. B. Terry, S. Anand, V. Jhaveri, A. Shah, and Y. Wu, "Peer-to-peer replication in winFS," Microsoft Research (MSR), Tech. Rep. MSR-TR-2006-78, Jun. 2006.

[25] F. J. Torres-Rojas and M. Ahamad, "Plausible clocks: constant size logical clocks for distributed systems," *Distributed Computing*, vol. 12, no. 4, pp. 179–196, 1999.