# Brief Announcement: Decoupling Version Identification from Causality Tracking Information in Distributed Storage Systems

Nuno Preguiça, CITI, DI/FCT/Univ. Nova de Lisboa
Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, Ricardo Gonçalves,HASLab, Univ. Minho

## 1 Introduction

Tracking causality is one of the fundamental problems in distributed systems. In distributed storage systems, version vectors [3] are used pervasively to track dependencies between replica versions. In these systems, version vectors tend to have a dual functionality: identify a version and encode causal dependencies.

In this paper, we argue that by decoupling version identification and information on causal dependencies it is possible to provide a causality tracking solution that is both scalable and fully accurate. The key idea of our approach is to maintain the identifier of the version separate from the causal past. This simple idea is quite powerful, allowing to verify causality in constant time (instead of $O(n)$ for version vectors) and allowing to precisely track causality with information with size bounded by the degree of replication (instead of information size bounded by the number of clients as in some cloud storage systems, such as Riak).

## 2 Version vectors basics

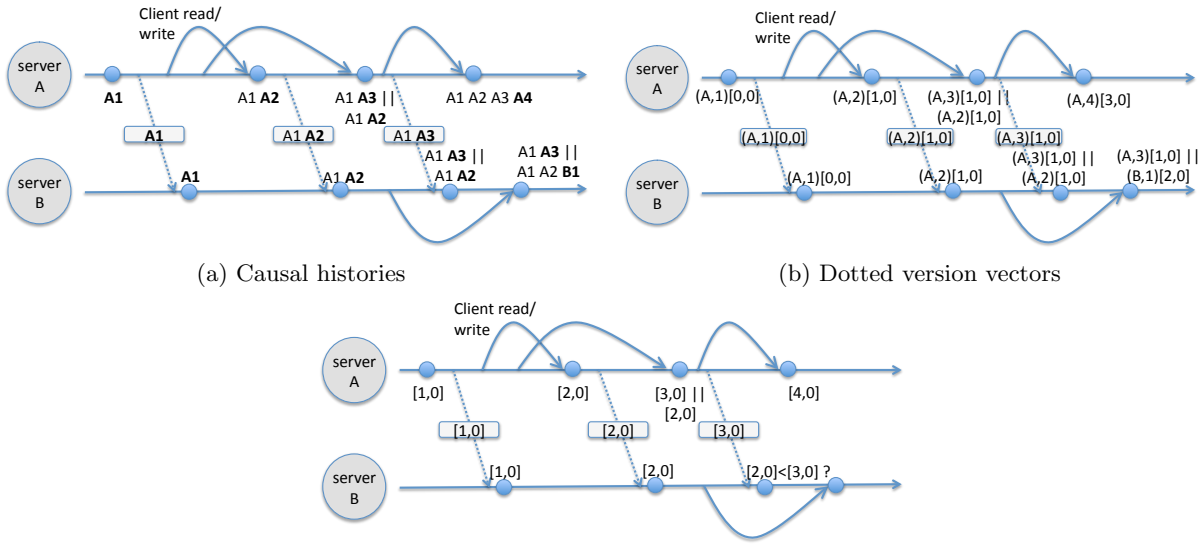
In a distributed systems, causality can be precisely characterized by *causal histories* [5]. Causal histories are sets of unique event identifiers. Each event, $a$, is assigned a new unique identifier, $id_a$, and its causal history, $H_a$, will include this identifier and the set, $P_a$, of identifiers for all events that causally precede $a$ ($H_a = \{id_a\} \cup P_a$). The partial order of causality can be precisely tracked by comparing these sets by set inclusion. An history $H_a$ causally precedes $H_b$ iff $H_a \subset H_b$. Two histories are concurrent if neither include the other: $H_a \parallel H_b$ iff $H_a \not\subseteq H_b \wedge H_b \not\subseteq H_a$. It is possible to compare if two events are equal by comparing their identifier: $a = b$, iff $id_a = id_b$.

Version vectors (VV) are an efficient mechanism to encode causal histories in distributed storage systems. When considering VVs, unique identifiers are the composition of unique site ids and a monotonic integer counter. A version vector, $V$, maintains for each site, $s_i$, an integer $V[s_i] = n_i$ encoding that event identifiers $(s_i, 1), \ldots, (s_i, n_i)$ are included in the set represented by $V$ (assuming that the first assigned identifier in $s_i$ is $(s_i, 1)$). VVs are used to verify the causality among replica versions: $V_a \leq V_b$, iff $\forall s, V_a[s] \leq V_b[s]$, which is no more that the application of set-inclusion defined for causal histories.

By the definition of causal history, it is clear that it is possible to verify if an event $a$ causally precedes an event $b$ by simply verifying if its identifier $id_a$ is contained in the set $P_b$ of events that precede event $b$ : $a < b$, iff $id_a \in P_b$ (or $id_a \in H_b \wedge id_a \neq id_b$). Two events are concurrent if neither causally precedes the other: $a \parallel b$ iff $id_a \notin P_b \wedge id_b \notin P_a$. VVs do not allow the use of the set-contains operation when verifying the causality dependencies of two events, as the version identifier is not known as it is diluted in the VV. In the next section, we present dotted version vectors, a causality tacking mechanism that decouples version identifiers and causality tracking information, precisely encoding the causal history in a distributed storage system.

## 3 Dotted version vectors

A dotted version vector (DVV) [4] is a logical clock which consists of a pair $(d, v)$, where $v$ is a traditional version vector and the dot $d$ is a pair $(i, n)$, with $i$ a node identifier and $n$ an integer.




(a) Causal histories

(b) Dotted version vectors

(c) Version vectors

Figure 1: System with two servers and a single object. Client interactions are presented as curves. Server synchronizations are presented as dotted lines, showing the propagated information. The causality information maintained after each relevant event is shown close to each small circle. with || meaning that concurrent versions are maintained.

The dot is the version identifier and it represents the globally unique event being described, while the VV represents the causal past. The events represented by a DVV can be characterized by the following semantic function from DVVs to causal histories:

$$\mathcal{C}[\![((i,n),v)]\!] \quad = \quad \{i_n\} \cup \bigcup_{\forall j} \{j_m \mid 1 \leq m \leq v[j]\},$$

where $i_n$ denotes the event with unique identifier $(i, n)$.

From the definition of causal histories, it follows immediately that an event $a$ with DVV $((i_a, n_a), v_a)$ causally precedes an event $b$ with DVV $((i_b, n_b), v_b)$: $a < b$, iff $n_a \leq v_b[i_a]$ (i.e., the event identifier of $a$ is in the causal past of $b$). Two events are concurrent if neither causally precedes the other: $a \parallel b$ iff $n_a > v_b[i_a] \wedge n_b > v_a[i_b]$.

As an example, we present the evolution of the versions of an object maintained in two storage servers using both causal histories (Figure 1a) and DVVs (Figure 1c). As it is possible to observe, DVVs are the immediate representation of causal histories, with the version identifier decoupled from the causal past. Next, we discuss the advantages and disadvantages of DVVs.

$O(1)$ **causality verification** As discussed before, verifying if one event $a$ precedes some other event $b$ can be done in constant time, by simply verifying if the event identifier (dot) of $a$ is reflected in the version vector that summarizes the causal past of $b$. This follows immediately from the definition of causal histories.

Although in many cases this can be just a theoretical curiosity, with the growing number of sites involved in distributed systems, this is becoming increasingly important.

**Efficient causality tracking in replicated storage systems** Distributed file systems (DFS - e.g. Locus, Coda, Ficus), usually use VVs with one entry per server. This is sufficient for detecting concurrency between versions stored in servers. For detecting concurrency between the version in a server and the version a client wants to write, the client can record the VV of the version it has read. When writing back his changes, if the VV is different, a concurrent update is detected. In this case, systems as Coda require the conflict to be solved before the file could be accessed

again. With DVV, conflicts could be detected by comparing only the dot, instead of the full VV – a different dot, present in the server replica, would mean a conflict.

When a storage system maintains multiple versions, the problems gets more complex. As exemplified in the replica A of Figure 1c, the same strategy can be used to detect concurrent writes from two clients. The problem that arises is what VV to use to identify the second version. When using an entry per server, any VV generated will dominate the VV of the previous version – in the example, $[2,0] < [3,0]$. This can cause problems if it is necessary to compare the two versions, as it would happen in server B, after receiving the version tagged with VV $[3,0]$. This shows that VVs with one entry per server are insufficient to track causality among versions generated concurrently by multiple clients. The system could store the information that both VVs are in fact concurrent, but this information would need to be propagated with the VVs.

An alternative used in cloud storage systems, such as Riak, is to keep one entry in the VV per client. This is inefficient as VVs can grow very large. To address this problem these systems prune VVs optimistically, which is unsafe, possibly leading to lost updates and/or to the introduction of false concurrency. Safe mechanisms for pruning VVs requires global knowledge – e.g. Golding [1] has proposed an asynchronous solution for the problem.

DVVs can precisely track causality among versions concurrently created by multiple clients using one entry per replica server, as DVVs decouple the version identifier and the causal past. When a client submits a version that is concurrent with the version in the server, a new DVV is generated that correctly tracks causality. In the example of figure 1b, we have $(3,0)[1,0] \parallel (2,0)[1,0]$. The evaluation with a modified version of Riak that includes DVV has shown a significant reduction in the size of metadata, and a good reduction in the latency when serving requests [4].

**Comparison with related works** Vector clocks (VCs) are used to track causal dependencies among events in a distributed system. The same approach proposed in DVVs could be used with VCs, as VCs use essentially the same mechanism as VVs, with the difference that VVs only record events that generate new data versions and VCs record all events in a distributed system.

Version vectors with exceptions (VVE) [2] extend VVs by recording non-continuous sequences of events for each site, thus allowing to represent any causal history. Thus, VVEs do not incur in the same problem as VVs for precisely tracking causality among updates concurrently executed by multiple clients. For allowing to verify causality in $O(1)$, VVEs could be extended with the approach proposed in this paper for decoupling version identification and causal past.

Wang et. al. [6] have proposed a variant of VVs with $O(1)$ comparison time. The solution keeps VV entries ordered, which leads to $O(n)$ time for other operations. Furthermore, as a simple VV it also incurs in the problems of VVs for tracking causality among concurrent client updates.

# References


[1] Richard Golding. *Weak-consistency Group Communication and Membership*. PhD thesis, University of California at Santa Cruz, 1992.

[2] Dahlia Malkhi and Douglas B. Terry. Concise version vectors in winfs. *Dist. Computing*, 20(3):209–219, 2007.

[3] D. Stott Parker and et. al. Detection of mutual inconsistency in distributed systems. *Transactions on Software Engineering*, 9(3):240–246, 1983.

[4] Nuno M. Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Dotted version vectors: Logical clocks for optimistic replication. *CoRR*, abs/1011.5808, 2010.

[5] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 3(7):149–174, 1994.

[6] Weihan Wang and Cristiana Amza. On optimal concurrency control for optimistic replication. In *Proc. ICDCS*, pages 317–326, 2009.