Improving on Version Stamps

Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte

Departamento de Informática, Universidade do Minho
Largo do Paço, 4709 Braga Codex

Abstract. Optimistic distributed systems often rely on version vectors or their variants in order to track updates on replicated objects. Some of these mechanisms rely on some form of global configuration or distributed naming protocol in order to assign unique identifiers to each replica. These approaches are incompatible with replica creation under arbitrary partitions, a typical operation mode in mobile or poorly connected environments. Other mechanisms assign unique identifiers relying on statistical correctness. In previous work we have introduced an update tracking mechanism that overcomes these limitations. This paper presents results from recent experimentation, that brought to surface a particular pattern of operation that results in an unforeseen, unlimited growth in space consumption. We also describe informally a new update tracking mechanism that does not exhibit this pathological growth while providing guaranteed unique identifiers for a dynamic number of replicas under arbitrary partitions and the same functionality of version vectors.

1 Introduction

Tracking update dependencies on optimistic replication systems often resorts to the use of version vectors [5] or some of its variants [7]. These mechanisms have been devised and have been successfully supporting traditional scenarios with a fixed number of replicas. Extensions to version vectors have been proposed [6] in order to accommodate a variable number of replicas. However, when trying to cope with the problem of replica identification there is an implicit assumption of global configuration or of a well connected environment in which a distributed naming protocol can be run and replica retirement detected. These assumptions are incompatible with replica creation (and retirement) on distributed systems subject to arbitrary partitions, a typical mode of operation of mobile or poorly connected environments. Other approaches tackle this identification problem relying on statistical correctness [4]. These approaches, not only may lead to occasional errors (which, even very rare, may be unacceptable), but also lead to large identifiers.

Previous work of the present authors [2] introduced the Version Stamp mechanism, a form of decentralized version vector overcoming these limitations. It enables autonomous identity management, update tracking and comparison, solely relying in local or pair-wise knowledge. This confinement is possible because: its structure allows a local management of the identity namespace (both local generation of identifiers when forking replicas and merging of identifiers when joining replicas); and the information about updates to replicas is based on the identity namespace in such a way as to allow global comparisons. This structure was devised in such a way that it should naturally grow and collapse as replicas are created and merged in the system.

Version Stamps have been employed in the Panasync [1] decentralized file replication system. Recent experimentation, however, brought to surface a particular pattern of operation that, when repeatedly applied, leads to an unnecessary unbounded growth of its structure.

This paper identifies this particular pattern of operation and illustrates how the version stamp structure degenerates under its occurrence. It also proposes a new version tracking mechanism – inspired by recent insights on autonomous identity management – that does not exhibit this unnecessary structural growth.

2 Version Stamps

Version stamps were devised in order to track update dependencies across a set of replicas in a mobile or poorly connected environment. In this setting, autonomous creation of replicas and pair-wise reasoning over update dependencies are crucial requirements. Operations on version stamps cannot depend on a global view of the system and thus they rely exclusively on local knowledge of each replica.

The structure of a version stamp is made of an *identity* and an *update* component. The identity component distinguishes each replica from all coexisting ones, in any possible configuration. It is also used as an available namespace from which new identities can be generated. This identity generated is achieved by namespace division as described below. The update component records "when" (in which state) changes were applied to a replica. It consists of a single identity-like value collected from the identity of its ancestor when the update was performed.

Three operations are provided: a **fork** operation supports the creation of new replicas whose state is cloned from the original; a **join** operation supports the merging of two replicas keeping one and retiring the other; and an **update** operation accounts for changes on the state of a replica.

An update operation simply copies the *identity* to the *update* component. This means that after an update, subsequent ones do not affect a version stamp. This is an example of the goal, in the design of version stamps, to discard information that is irrelevant to the comparison of coexisting elements in a configuration.

At a fork operation the *identity* of the resulting version stamps is recursively constructed by appending either '0' or '1' to the right of each component of the ancestor *identity*. A fork does not modify the *update* component as it does not introduce any update event (the ones tracked by the mechanism): it simply copies the *update* component to the new version stamps. Regarding identity management, the transformation applied to the identity component can be perceived as the subdivision of the namespace available to a particular replica. Although a local operation, the resulting namespaces are guaranteed to be globally unique and thus distinguish the two new replicas from all the others in the current configuration.

When a join between two elements occur the resulting *identity* is built by merging the two ancestors *identity* components. The *update* component is built likewise, merging the two ancestor *update* components; this reflects the combined knowledge of past updates. Upon the join operation, the resulting identity namespace can be perceived as the union of the two ancestors namespaces. This resulting namespace can then be recursively collapsed each time sibling identity namespaces are present (namespaces that have been previously split upon a fork operation). Since the *identity* component always *dominates* the *update* component (which records information regarding past state changes), this simplification propagates to the *update* component. Ultimately, joining every coexisting replica leads to a completely collapsed version stamp, bringing its structure to its initial value, that is, two empty sets. This division and collapsing feature is an intended design goal of the version stamp mechanism.

Figure 1 shows an example of the version stamp mechanism in action on a replicated system. In this example a version stamp is represented by an [*update* | *identity*] pair, the ϵ denotes an empty set and the δ denotes a local event of state change. Though not shown in this example, as stated above, joining the two remaining replicas would completely collapse the resulting version stamp. A detailed and formal description of Version Stamps including its proof of correctness can be found in [2].

$$\begin{tikzcd}
 & & & & & {[0 \mid 00]} \arrow[r, "\delta"] & {[00 \mid 00]} \arrow[ddr] & & \\
 & & & {[\epsilon \mid 0]} \arrow[r, "\delta"] & {[0 \mid 0]} \arrow[ur] \arrow[dr] & & & & \\
{[\epsilon \mid \epsilon]} \arrow[r, "\delta"] & {[\epsilon \mid \epsilon]} \arrow[r, "\delta"] & {[\epsilon \mid \epsilon]} \arrow[ur] \arrow[dr] & & & {[0 \mid 01]} & & {[00+1 \mid 00+1]} \arrow[r, "\delta"] & {[00+1 \mid 00+1]} \\
 & & & {[\epsilon \mid 1]} \arrow[r, "\delta"] & {[1 \mid 1]} \arrow[r, "\delta"] & {[1 \mid 1]} \arrow[urr] & & &
\end{tikzcd}$$

Fig. 1. A set of partially ordered events with version stamps.

2.1 Pollution of the namespace

Exercising the version stamp mechanism, we have observed an undesired growth of version stamps under a simple pattern of operation. This problem is illustrated in Figure 2, which shows the identity component in a scenario where three replicas are created and then we repeat a pattern in which two of them are joined and forked again, while alternating replicas.

Although we end up with only three replicas, the identity components are much more complex than in the configuration after the first two forks (with the same number of replicas). In this scenario, the Version Stamp mechanism leads to a overly refined namespace which cannot be simplified upon these interleaving joins.

This growth gets worse every time this operation pattern occurs and recent experimentation does indicate that this can be a fairly common case in practical usage scenarios. Furthermore, when an update occurs, the identity component is copied to the update component, thus aggravating this problem.

This degeneration of the namespace does not imply that the version stamp mechanism is incorrect but that it may consume an unreasonable amount of space. As a result, this growth pattern of version vectors can severely affect its practical application.

$$\begin{tikzcd}
 & 0 \arrow[drr] & & & 00+100 & & \\
{} \arrow[ur] \arrow[dr] & & 10 \arrow[r] & 0+10 \arrow[ur] \arrow[dr] & & & 010+1010+110 \\
 & 1 \arrow[ur] \arrow[dr] & & & 01+101 \arrow[r] & 01+101+11 \arrow[ur] \arrow[dr] & \\
 & & 11 \arrow[urrr] & & & & 011+1011+111
\end{tikzcd}$$

Fig. 2. Pollution in the identity component of version stamps.

3 Dynamic Map Clocks

Dynamic Map Clocks imports from version stamps the basic features that support autonomous identity management. Noticing that the lack of counters, on version stamps, impose important restrictions on identity management that ultimately contributed to the identified growth problem, dynamic map clocks will combine the use of counters with a more flexible identity management scheme.

The important property that rules identity management for update tracking is the allocation to each replica of at least one identity that is exclusive to that replica in a given moment. When an update needs registering in a given replica, one identity must be chosen among its exclusive identities and the associated counter must be incremented. This identity does not need to be the same for all updates in that replica and replicas can handover identities to other replicas.

As a consequence of these insights it is easy to conceive a scheme where replicas can fork by either specializing a binary identity (forking 010 would derive 0100 and 0101) or by partitioning controlled identities that were obtained upon joins (forking $010 + 10 + 111$ could derive $010 + 111$ and 10). This is the basic mechanism that supports dynamic map clocks and Figure 3 shows a run that illustrates this. More complex rules are enforced when handling joins and setting counters upon joins.

3.1 Non-pollution of the namespace

Considering the namespace pollution problem that was present on version stamps, it is easy to verify that dynamic map clocks are much more flexible on the han-

$$\begin{tikzcd}
 & & & & & {[0_3 \mid 00]} \arrow[r, "\delta"] & {[00_4 \mid 00]} \arrow[dr] & & \\
 & & & {[\epsilon_2 \mid 0]} \arrow[r, "\delta"] & {[0_3 \mid 0]} \arrow[ur] \arrow[dr] & & & & \\
{[\epsilon_0 \mid \epsilon]} \arrow[r, "\delta"] & {[\epsilon_1 \mid \epsilon]} \arrow[r, "\delta"] & {[\epsilon_2 \mid \epsilon]} \arrow[ur] \arrow[dr] & & & {[0_3 \mid 01]} & & {[00_4 + 1_4 \mid 00 + 1]} \arrow[r, "\delta"] & {[00_4 + 1_5 \mid 00 + 1]} \\
 & & & {[\epsilon_2 \mid 1]} \arrow[r, "\delta"] & {[1_3 \mid 1]} \arrow[r, "\delta"] & {[1_4 \mid 1]} \arrow[urr] & & &
\end{tikzcd}$$

Fig. 3. A set of partially ordered events with dynamic map clocks.

dling of names. Figure 4 shows how the run that depicted a name pollution pattern in version stamps is easily handled by this mechanism.

$$\begin{tikzcd}
 & 0 \arrow[drr] & & & 0 & & \\
\epsilon \arrow[ur] \arrow[dr] & & 10 \arrow[r] & 0 + 10 \arrow[ur] \arrow[dr] & & & 10 \\
 & 1 \arrow[ur] \arrow[dr] & & & 10 \arrow[r] & 10 + 11 \arrow[d] \arrow[ur] \arrow[dr] & \\
 & & 11 \arrow[urrr] & & & 1 & 11
\end{tikzcd}$$

Fig. 4. Non-pollution of the identity component in the dynamic map clock mechanism.

In some way, dynamic map clocks try to ally the innovative management of identities that stems from version stamps with the benefits of classical counters and their synthetic encoding of updates.

4 Discussion

Handling replication in highly decentralized systems and large scale settings – in number of nodes, geographical distance or communication latency between nodes– often implies the use of optimistic techniques in order to improve availability. In these scenarios, replicas are allowed to diverge from a consistent global state but reconciliation procedures and update propagation strategies are put in place so that consistency can eventually be restored. All these operations must rely on some dependency tracking mechanism in order to infer the causal relations between replica states. As mentioned before, the standard version vector mechanism assumes a consistent management of replicas names.

In decentralized distributed systems that face partitions, large membership changes under churn and disconnected operation, one can no longer rely on the assumption that globally unique names are available. A way of approaching these settings is to avoid determinism altogether and rely on probabilistic approaches such as generating random replica names, and assume some risk of name collisions [3], or using sets of hashes of replica state to detect updates [4] once again assuming some risk of collision. If reliability cannot be compromised, as is often the case, only a deterministic approach is appropriate. Determinism can only be obtained by recursive generation of names, the approach developed and formalized in our previous work on version stamps. However, as we have shown in the present paper, recursive generation of names can easily introduce important growth problems in the space consumed by the version stamp mechanism.

With dynamic map clocks we achieve a better handling of the data space by avoiding unnecessary partitions of identifiers and concentrating on the important property that each replica must at a given moment have exclusive access to at least one globally unique identifier. Unlike version stamps, that do not use counters, dynamic map clocks are, in a sense, a hybrid mechanism that also relies on counters for registering updates. This usage of counters leads to important savings in size. In addition, although the examples here have only shown runs with join operations, dynamic map clocks allow the use of messages when sending metadata and support unidirectional updating of dependency information.

While the theory of dynamic map clocks is still under development and a proper formalization and formal proof is ongoing work, we already have a running implementation of the mechanism. This implementation has been tested on long random runs under various numbers of replicas and always evaluated as correct. This evaluation is done by contrasting the causal pre-order that the mechanism derives with the equivalent pre-order derived by causal histories. Causal histories are implemented by assuming global knowledge and adding unique update events to a set of events in each replica and relating this sets by set inclusion (see [2, 8] for more details on causal histories). We can comment, from our experience, that incorrect mechanisms typically fail these checks after a small number of steps and do not stay correct in long random runs.

Another important property of dynamic map clocks, not present in version stamps, is their potential use as substitutes for vector clocks in autonomous decentralized settings. Vector clocks, that are at the core of causal message delivery protocols and distributed debugging, also rely on globally unique names thus facing the same problems of version vectors.

References

1. Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Panasync: Dependency tracking among file copies. In Paulo Guedes, editor, *Ninth ACM SIGOPS European Workshop*, pages 7–12. DIKU - University of Copenhagen, 2000.
2. Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Version stamps – decentralized version vectors. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 544–551. IEEE Computer Society, 2002.
3. Gerald A. Heuer. Estimation in a certain probability problem. *The American Mathematical Monthly*, 66(8):704–706, 1959.
4. Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *Proceedings of the 23nd International Conference on Distributed Computing Systems (ICDCS)*, pages 670–677. IEEE Computer Society, 2003.
5. D. Stott Parker, Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, David Edwards, Stephen Kiser, and Charles Kline.

Detection of mutual inconsistency in distributed systems. *Transactions on Software Engineering*, 9(3):240–246, 1983.

6. David Ratner, Peter Reiher, and Gerald Popek. Dynamic version vector maintenance. Technical Report CSD-970022, Department of Computer Science, University of California, Los Angeles, 1997.
7. Yasushi Saito and Marc Shapiro. Optimistic replication. Technical Report MSR-TR-2003-60, Microsoft Research, 2003.
8. R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 3(7):149–174, 1994.