Version Stamps — Decentralized Version Vectors

Paulo Sérgio Almeida Carlos Baquero Victor Fonte

Departamento de Informática, Universidade do Minho
Largo do Paço, 4709 Braga Codex, Portugal
E-mail: {psa,cbm,vff}@di.uminho.pt

Abstract

Version vectors and their variants play a central role in update tracking in optimistic distributed systems. Existing mechanisms for a variable number of participants use a mapping from identities to integers, and rely on some form of global configuration or distributed naming protocol to assign unique identifiers to each participant. These approaches are incompatible with replica creation under arbitrary partitions, a typical mode of operation in mobile or poorly connected environments. We present an update tracking mechanism that overcomes this limitation; it departs from the traditional mapping and avoids the use of integer counters, while providing all the functionality of version vectors in what concerns version tracking.

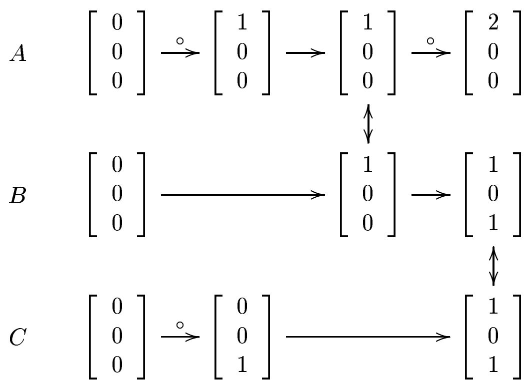

Figure 1. Use of version vectors to track updates among three replicas.

1. Introduction

Mobile computing has evolved in the previous decade into what is now a common mode of operation for a significant share of distributed systems. This mobile context helped to promote optimistic strategies and, with them, the need for version vectors in update tracking. Nevertheless, the same mobile context also brings to surface some of the limitations of version vectors, in particular concerning the identification of participating entities in the computation in such potentially dynamic environments.

The concept of version vector [13] is connected to the twin concept of vector clock [6, 11], and both are rooted on causality in distributed systems [10]. These concepts share an equivalent structure that consists in a mapping from process/replica identifiers to integer counters, $I \hookrightarrow \mathbb{N}$. In practice, version vectors and vector clocks are more often represented as a fixed sequence of integer counters, $\{1, 2, \ldots, k\} \to \mathbb{N}$, which is a reasonable choice as long as the number of entities is known in advance. Figure 1 shows an execution in a replicated system where fixed size version vectors are used to track updates to each of the three replicas in the system. The direction of evolution is represented by the arrows, with dot annotated arrows, $\overset{\circ}{\to}$, depicting updates on a given element of the system.

Although structurally similar, vector clocks and version vectors play different roles on distributed systems. Vector clocks are known to provide a view over a distributed computation, different events being identified by distinct vector clock values[1]. The role of version vectors is to detect mutual inconsistency among replicas and to determine the most recent version among two causally related replicas. All replicas that have seen the same updates, typically after a synchronization procedure, share the same version vector value – see again Figure 1.

A well known problem of version vectors and vector clocks is that they are unbounded in size [15, 18]. In fact, they are twice unbounded. Each integer counter can grow indefinitely and the number of identified entities can also grow unbounded.

A less known problem, which we address in this paper, resides in the identification requirement of both version vec-

[1] In Fidge Logical Time, two events share the same clock value when representing a synchronization event between two instances. Usually, asynchronous message passing is assumed and this does not occur.

tors and vector clocks [3, 14]. Each participating entity must be assigned a unique identifier in order to obtain a proper mapping to integer counters. In a well connected environment, it would be simple to request a unique identifier from a server or to run a distributed protocol for the generation of a unique identifier. Such protocols are not possible in the current mobile setting when subject to partitioned operation. Moreover, significant technology and research trends are pointing towards wireless ad hoc networking setups, where entities are autonomous and operate in local clusters on a proximity basis [12, 4, 7]. In such environments, partitioned operation is the common mode of operation and an answer to the identification problem must be sought.

In circumstances in which we can afford probabilistically unique identifiers, algorithms may resort to some form of random based ids in order to cope with replica creation under partitioned environments. Contrary to these approaches, our work does not rely on probabilistic uniqueness and assumes that guaranteed unique identifiers must be provided.

1.1. Fixed vs. Variable number of Replicas

Classic replication systems operate over a well defined number of replicas. Such is the case of the system depicted in Figure 1. The more general case of a dynamic replication system, introduces the need to accommodate replica creation and retirement. One approach would be to represent replica creation by introducing new horizontal lines and new replica identifiers in the system representation, and likewise to discontinue those lines towards the future, upon replica retirement.

The approach we follow, instead, represents all the functionality of replica creation, synchronization and retirement by two simple constructs: replica forking and joining of replicas. Synchronization can then be represented by joining two replicas and forking the resulting one. An example is presented in Figure 2.

This dynamic replication system is more general than the fixed one and can be used to encode the latter. In Figure 3 we give the intuition to this encoding by representing under fork-and-join dynamics a traditional version vector setting for three replicas, using the same names for elements in equivalent positions and omitting the name of extra elements. From this example, it is also easy to see that an equivalent mapping can be found for runs with a variable number of replicas.

1.2. Frontier Elements vs. All Elements

In certain circumstances, one may want to relate any two elements occurring in the distributed evolution, that is, all

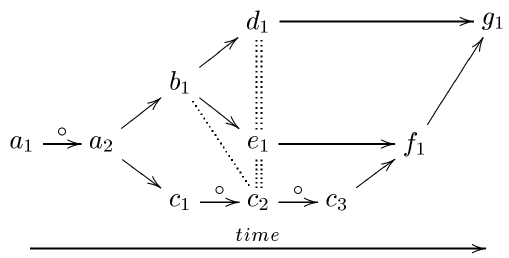

Figure 2. Some possible evolutions of data elements showing two frontiers of coexisting elements (denoted by single and double-dotted lines).

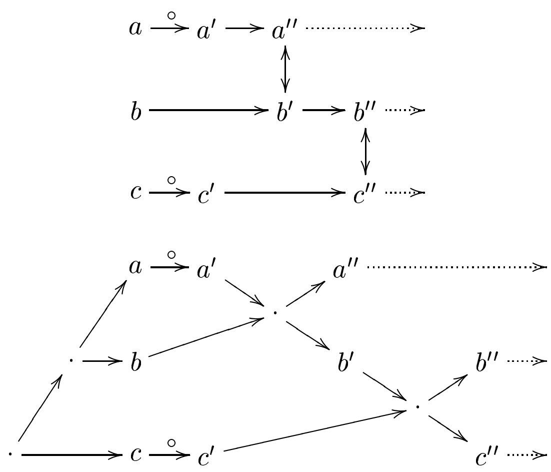

Figure 3. Encoding a fixed number of replicas (top) under fork-and-join dynamics (bottom).

elements in the distributed computation are subject to ordering. For instance, in the computation depicted in Figure 2, one may want to inquire how c_2 and a_1 relate and determine that a_1 is in the past of c_2. Such querying could be necessary when debugging a recorded execution of the replicated system.

In other circumstances, namely in update tracking, one may only need to relate coexisting elements, that is, only elements in the same reachable configuration. If this is the case, it wouldn't make sense to query how c_2 and a_1 relate since these elements never coexist in any arbitrary system evolution. In this sense, a reachable configuration is perceived as forming a *frontier*. Any two elements that are connected by a direct arrowed path never coexist, and consequently never belong to the same frontier of contemporaneous elements.

If we concentrate on element c_2 we can observe that, for the depicted evolution, there are two possible frontiers to which c_2 can belong. The first, represented by a single dotted line, might occur if c_1 gave place to c_2 before the bifurcation of b_1. The second frontier, double dotted, occurs if b_1's bifurcation is prior to c_2's transformation into c_3. In fact, it is possible that both frontiers occur in a particular system run.

In any case, an ordering system that targets frontier elements should have enough information to relate any two events that can occur in any possible system frontier. It is intuitive to accept that ordering of frontier elements is sufficient for version management, since only coexisting elements are subject to queries on their relation properties. We believe that this observation can have an important impact on the design of future version management techniques.

Under the distinction that we have just presented it is now clear that traditional version vectors are overly expressive: they are capable of overall ordering albeit in their application context a frontier ordering would be sufficient. One could conjecture that a compressed substitute of version vectors would be conceivable for traditional settings with fixed numbers of entities, and such substitute would not contradict Charron-Bost minimality results [5] (stated in the context of vector clocks but easily inferable for version vectors). This is not, however, the purpose of this article.

It is easy to conclude that classical (fixed size) version vectors are associated to frontiers of constant size, the vector dimension, while dynamic forms of version vectors, c.f. [15], act on variable frontiers.

Our goal is to develop a decentralized, autonomous form of version vectors – named *version stamps* – that allows frontier ordering with autonomous creation of identifiers from any available replica. By considering frontier ordering we seek a compact solution to the identification problem that can act as an alternative to version vectors in dynamic settings.

1.3. Structure of the Paper

The rest of the paper is structured as follows. The next section introduces a model of causal histories of events, using a global view on events. Sections 3 and 4 develop the concept of version stamps and introduce a set of invariants over their structure. Section 5 establishes a functional equivalence between version stamps and causal histories, and Section 6 refines the version stamp model while keeping the equivalence. Section 7 concludes the article.

2. Causal Histories in Dynamic Settings

Detection of version dependencies among data elements can be constructed over a notion of causal history of update events [16]. In the construction of such history we assume a global view over the system in order to obtain a description that is intuitively correct. Afterwards, a version stamping system that does not rely on a global view will be constructed and proved to represent the same dependency order between elements that can be derived from the causal history.

To model causal histories we keep a mapping from element identities to sets of update events. Since we are only interested in comparing frontier elements, we only keep in the mapping the set of elements that define each frontier (thus elements that may have existed in its past are not included). This map can be seen as representing a "current configuration".

Operations (update, fork and join) are described by transformations between configurations.

We use the traditional notation for functions: $\{a \mapsto \{x\}, b \mapsto \{y,z\}, c \mapsto \{x,z,w\}\}$ represents a function that maps elements a, b and c to sets of events; some events (like x and z) can be in the causal history of several elements.

Notation. We use $\{F; a \mapsto x, b \mapsto y\}$ to represent a function that maps a to x, b to y and that maps other elements in the domain according to function F. This notation expresses also that both a and b do not belong to the domain of F. This is useful to perform "pattern matching" over functions (Note that using $F \cup \{a \mapsto x, b \mapsto y\}$ does not imply that $x, y \notin \mathrm{dom}(F)$.). A similar notation can be used for 'pattern matching' over sets: $\{A; a, b\}$ denotes a set $A \cup \{a, b\}$ such that $a, b \notin A$.

Definition 2.1. *An initial configuration can be captured by* $\{a \mapsto \{\}\}$ *and represents a system with one data element. From any reachable configuration, the following transformations can occur:*

- $\{C; a \mapsto A\} \stackrel{\mathbf{update}(a)}{\longrightarrow} \{C; a' \mapsto A \cup \{e\}\}$ *with* $e \notin \mathcal{E}(\{C; a \mapsto A\})$,
- $\{C; a \mapsto A\} \stackrel{\mathbf{fork}(a)}{\longrightarrow} \{C; b \mapsto A, c \mapsto A\}$,
- $\{C; a \mapsto A, b \mapsto B\} \stackrel{\mathbf{join}(a,b)}{\longrightarrow} \{C; c \mapsto A \cup B\}$,

with $\mathcal{E}(\{C\}) \doteq \bigcup\{C(i) \mid i \in \mathrm{dom}(C)\}$.

Although mapping only "current" elements, the corresponding event sets store all update events that have occurred in the causal history of each element: events are not discarded. A global view is present because each update event has a global unique identity that cannot be computed by only looking at the element being updated.

When querying the relationship between elements, according to known updates, the goal is to distinguish three possible situations: Equivalence – the same set of events; Obsolescence – all the update events and at least one more in the dominating element; Mutual inconsistency – at least

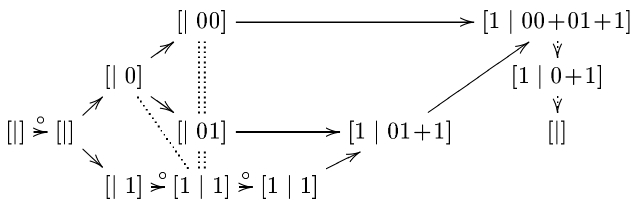

Figure 4. Version Stamps.

one different update event in each element. Given a configuration $\{C; a \mapsto X, b \mapsto Y\}$:

- a **equivalent to** b iff $X = Y$,
- a **obsolete relative to** b iff $X \subset Y$,
- a **inconsistent with** b iff $X \not\subseteq Y$ and $Y \not\subseteq X$.

Comparison of elements in a frontier can be deduced from the causal histories as defined above. In fact, all these situations are represented by a pre-order on the elements of a given frontier. Given a configuration C, for any two elements a, b in the domain of C, we have:

$$a \sqsubseteq_C b \iff C(a) \subseteq C(b).$$

The simplicity of this model is only possible in the presence of a global view over the set of events in the system.

3. Version Stamps

Our goal is to devise a stamping mechanism that can be used to infer the order between frontier elements that is induced by comparing sets of causal histories (as described above). The mechanism must not depend on any form of global view; it must work autonomously and rely only on the local information that is kept within the data elements being operated upon. An efficient use of space is also highly desirable in order to support a practical use.

We now present an informal description of version stamps. Figure 4 presents the example from Figure 2 where the version stamp corresponding to each element is shown. Each version stamp is made up of two components, which we represent as $[update \mid id]$. The *id* component acts as the element identity: it distinguishes the element from all other coexisting elements (in a frontier). The *update* component stores information about which updates are known to a given element. It avoids the use of counters and consists of a single id-like value which collects *id*'s as they were (in ancestor elements) when updates were performed. Each component is presented as a sum of binary strings.

The first two version stamps in the left show that when the frontier is only one element updates do not need to have expression on the stamps. In fact, the update operation simply copies *id* into *update*; 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 frontier.

At a fork operation the *id* in the resulting stamps is recursively constructed by appending either 0 or 1 to the right of the ancestor *id*. A fork does not modify the *update* component as it does not introduce any update event (the ones tracked by the mechanism).

When a join between two elements occurs the resulting *id* is built by merging the two ancestor *id*'s. The *update* component is built likewise, merging the two ancestor *update* components; this reflects the combined knowledge of past updates.

An important property of the mechanism is the possible simplification of stamps after joins. The intuition is that a join decreases the number of elements in a frontier, leading to smaller identities being needed to distinguish them. A fork followed by a join of the resulting elements should result in an element with the original *id*. The intermediate elements *id*'s only differ in the appended 0 and 1; after being merged they are collapsed into the original *id*. (A simplification of *id* induces also a simplification of *update*.) Some analogies can be made: the simplification of minterms in boolean algebra, the collapsing of neighbour blocks in the buddy memory allocation system [9] or collecting weights in Huang's termination detection algorithm [8]. Likewise, *id*'s denote non-intersecting parts of 'the whole'; their complexity adjusts dynamically, reflecting the granularity of the frontier of coexisting elements.

3.1. Synopsis of formal presentation

The locality goal of the mechanism can be seen to be met by looking at the definition of the operations (below). To prove that version stamps can be used to infer the same order as induced by causal histories, we split the presentation of version stamps and proof of correctness in several steps.

We start by presenting a non-reducing version of the mechanism, in which no simplification at joins occur, and prove several auxiliary invariants that characterize some properties of version stamps. Afterwards, we show that both causal histories and the non-reducing version of the mechanism induce the same pre-order between elements at any given frontier. To do this we must first prove a stronger result that implies the required equivalence. Finally, we present a rewriting rule on version stamps that represents the simplification after a join. We show that it preserves all previously defined invariants as well as the proved result relating causal histories to version stamps.

4. Version Stamps: Non-Reducing

A version stamp is a pair (u, i), respectively the *update* and the *id*. Both components share the same structure, and are members of a set N (names). We now characterize N.

Let Σ^* be the partially ordered set of all finite binary strings (sequences of $\{0, 1\}$) ordered by:

$$r \sqsubseteq s \iff r \textbf{ is a prefix of } s.$$

We have, for example, $01 \sqsubseteq 011$ and $01 \parallel 00$ (we use $\parallel$ to denote non-comparability). The null string is denoted by ϵ; it constitutes the bottom of Σ^*: $\epsilon \sqsubseteq s$ for all strings s.

Definition 4.1. *N is the set of all finite antichains in Σ^*, ordered by:*

$$n_1 \sqsubseteq n_2 \iff \forall r \in n_1 . \exists s \in n_2 . r \sqsubseteq s.$$

For example, $\{0, 01\}$ is not a valid element of N because $0 \sqsubseteq 01$, and we have $\{00, 011\} \sqsubseteq \{000, 011, 1\}$ and $\{00, 10\} \not\sqsubseteq \{000, 011, 1\}$ as well.

As the order defined on N is the classic order in lower powerdomains [17], at first sight looks like we are in the presence of a pre-order. However, N was defined in a way so that it is a partial order and not merely a pre-order. More specifically:

Proposition 4.2. *N is a partial order; moreover it is a join semillatice with join given by:*

$$n_1 \sqcup n_2 \doteq \{s \in n_1 \cup n_2 \mid (s \sqsubseteq r \in n_1 \cup n_2) \Rightarrow s = r\}.$$

(That is the join of two names is the set of all maximal elements in their union.)

Proof. N is isomorphic to $\mathcal{O}(\Sigma^*)$ (the down-sets of strings) ordered by inclusion, which is a complete lattice. $\square$

Informally, the antichains in N can be seen to represent the maximal elements of down-sets, the order defined corresponds to inclusion of down-sets and the join corresponds to union of down-sets. For example, $\{00, 011\} \sqcup \{000, 01, 1\} = \{000, 011, 1\}$.

We now proceed with the definition of the first model of version stamps, in which we do not include simplification after joins. For presentation purposes, we describe the operations on version stamps using configurations that map elements to version stamps. This facilitates relating causal histories to version stamps. It is important to emphasize that this does not, however, imply that operations require a global view: the operations manipulate the version stamps of the operated upon elements, which themselves require no global view (contrary to the what happens in causal histories, where an update operation makes use of globally unique update events). The order derived from stamps only makes use of local stamp information as well.

Definition 4.3. *An initial configuration can be captured by $\{a \mapsto (\{\epsilon\}, \{\epsilon\})\}$ and represents a system with one data element. From any reachable configuration, the following transformations can occur:*

- $\{V; a \mapsto (u, i)\} \overset{\mathbf{update}(a)}{\longrightarrow} \{V; a' \mapsto (i, i)\}$,
- $\{V; a \mapsto (u, i)\} \overset{\mathbf{fork}(a)}{\longrightarrow} \{V; a' \mapsto (u, i0), a'' \mapsto (u, i1)\}$ *with* $nx \doteq \{sx \mid s \in n\}, x \in \{0, 1\}$ *being the concatenation of a digit lifted to sets of strings,*
- $\{V; a \mapsto (u_a, i_a), b \mapsto (u_b, i_b)\} \overset{\mathbf{join}(a,b)}{\longrightarrow} \{V; c \mapsto (u_a \sqcup u_b, i_a \sqcup i_b)\}$.

The update component simply copies the *id* into *update*; fork maintains the *update* component and appends either a 0 or a 1 to each string in the *id* component; the join operation performs joins of names for each component. It is easy to see that under the above definitions, the components in the resulting stamps are well-formed names (antichains of strings).

We now define the pre-order on the elements of a configuration V obtained from the version stamps in V, that will be used to make the correspondence with causal histories. Given a configuration V, for any two elements a, b in the domain of V, we have:

$$a \mathrel{\underset{\sim_V}{\sqsubset}} b \iff \mathrm{fst}(V(a)) \sqsubseteq \mathrm{fst}(V(b)).$$

Towards proving a proposition that relates causal histories with version stamps we establish now some auxiliary properties of configurations of version stamps.

Invariant 4.4. (I_1) *In any reachable configuration V: $\forall (a, (u, i)) \in V. u \sqsubseteq i$.*

Proof. See [2]. $\square$

This invariant states that in a version stamp the *update* is always dominated by *id*. This property will ensure, on reducible version stamps models, that there is no obsolete information on *update* when replicas converge and *id* simplifications are possible.

Invariant 4.5. (I_2) *In any reachable configuration V: $\forall \{x \mapsto (u_x, i_x), y \mapsto (u_y, i_y)\} \subseteq V.\ \forall r \in i_x, s \in i_y.\ r \parallel s$.*

Proof. See [2]. $\square$

This second invariant brings attention to some structural properties of the *id*'s that are present in a configuration. In a given frontier of elements each string that is present in a given *id* will be non-comparable to all other strings in the same or another *id*. Consequently, all *id*'s in a frontier are non-comparable.

Invariant 4.6. (I_3) *In any reachable configuration V:* $\forall\{x \mapsto (u_x, i_x), y \mapsto (u_y, i_y)\} \subseteq V.\ \forall r \in u_x.\ \{r\} \sqsubseteq i_y \Rightarrow \{r\} \sqsubseteq u_y$.

Proof. See [2]. □

This invariant implies a weaker one: $\forall\{x \mapsto (u_x, i_x), y \mapsto (u_y, i_y)\} \subseteq V.\ u_x \sqsubseteq i_y \Rightarrow u_x \sqsubseteq u_y$. The pertinence of this last invariant can be illustrated by an example. Suppose two non-comparable elements $a \parallel b$ with version stamps (u_a, i_a), (u_b, i_b). If an update occurs on one of them, for instance $update(a)$, we must be sure that a (a' after update) remains non-comparable to b, and $b \sqsubseteq a'$ does not happen (recall that causal histories ensure this by using *fresh* event names on updates). Since $update(a)$ produces version stamp (i_a, i_a) then our property $u_b \sqsubseteq i_a \Rightarrow u_b \sqsubseteq u_a$ means that in order for $b \sqsubseteq a'$ to occur, then $b \sqsubseteq a$ must also occur in the first place.

5. Correspondence between causal histories and version stamps

We now show that version stamps as defined above can be used to derive the pre-order between elements according to inclusion of causal histories. As we described above, comparing elements in a configuration C of causal histories can be done according to:

$$a \precsim_C b \iff C(a) \subseteq C(b).$$

If we have a configuration V of version stamps that corresponds to C (whose version stamps are derived from the same system execution as C), being the order between elements obtained from V:

$$a \precsim_V b \iff \text{fst}(V(a)) \sqsubseteq \text{fst}(V(b)),$$

we want to prove that both C and V induce the same pre-order, i.e. $\precsim_C = \precsim_V$. This means we want to show that:

$$C(a) \subseteq C(b) \Leftrightarrow \text{fst}(V(a)) \sqsubseteq \text{fst}(V(b)).$$

It can be seen that a direct proof by induction of this equivalence fails. This failure is in itself an interesting result and can be briefly explained by the following insight: knowing how elements compare according to causal history inclusion in a given configuration is not enough to know how they will compare in the configuration obtained after performing a given operation. In other words, even though we are not interested in knowing the exact update events in causal histories, we need to know something more than just how they compare even if comparison is all we are interested in.

Technically, we need to prove a stronger equivalence, which will be used as a stronger induction hypothesis in the proof. We show then, the following stronger proposition. (We use fst and snd for the projections on the first and second components of a pair. We also use the notation $f[A]$ for the direct image of A under f, that is $f[A] = \{f(x) \mid x \in A\}$.)

Proposition 5.1. *Given any distributed execution with causal histories $C_0 \longrightarrow C_1 \longrightarrow \ldots \longrightarrow C_k$ and with version stamps $V_0 \longrightarrow V_1 \longrightarrow \ldots \longrightarrow V_k$, it is true that* $\text{dom}(C_k) = \text{dom}(V_k)$ *and* $C_k(x) \subseteq \bigcup C_k[S] \Leftrightarrow \text{fst}(V_k(x)) \sqsubseteq \bigsqcup \text{fst}[V_k[S]]$, *for all* $x \in \text{dom}(C_k)$ *and* $\emptyset \subset S \subseteq \text{dom}(C_k)$.

Proof. See [2]. □

From the previous proposition, the result we want to show follows, as stated by:

Corollary 5.2. *Given any distributed execution with causal histories $C_0 \longrightarrow C_1 \longrightarrow \ldots \longrightarrow C_k$ and with version stamps $V_0 \longrightarrow V_1 \longrightarrow \ldots \longrightarrow V_k$, it is true that* $\text{dom}(C_k) = \text{dom}(V_k)$ *and* $C_k(x) \subseteq C_k(y) \Leftrightarrow \text{fst}(V_k(x)) \sqsubseteq \text{fst}(V_k(y))$, *for all x, y in* $\text{dom}(C_k)$.

Proof. Substitute S by $\{y\}$ in the previous proposition. □

6. Simplifying version stamps upon joins

We now describe a rewriting rule that can be applied to a version stamp and perform the simplifications that have been informally introduced in Figure 4. Such simplifications reflect, as already discussed, the dynamic adaptation of *id*'s to the 'shape' of the frontier. This simplification is essential towards obtaining a realistic implementation, by minimizing the space requirements of version stamps.

The simplification of a version stamp that results from a join is attempted by repeatedly applying the following rewriting rule until it is no longer possible to apply it:

$$(u, \{i; s0, s1\}) \xrightarrow{\alpha} (u', \{i; s\}),$$

with

$$u' = \begin{cases} u \setminus \{s0, s1\} \cup \{s\} & \text{if } s0 \in u \text{ or } s1 \in u, \\ u & \text{otherwise.} \end{cases}$$

One property of a rewriting $(u, i) \xrightarrow{\alpha} (u', i')$ that follows trivially from the order on names is that $u' \sqsubseteq u$ and $i' \sqsubseteq i$. As the order on names is well-founded (there are no infinite descending chains of names), only a finite number of rewritings can be applied to a stamp. It is also easy to see that the rewriting is confluent. Therefore, a stamp can be rewritten into a unique normal form.

We omit the proof of confluence as it is intuitive and concentrate on the correctness of the transformation. For that we need to show that applying a rewriting $(u, \{i; s0, s1\}) \xrightarrow{\alpha} (u', \{i; s\})$ to a version stamp in a configuration V results in a configuration V' where: the rewritten version stamp consists of two wellformed names (antichains), the invariants I_1, I_2, I_3 are maintained, and the relation from dom(V) to $\mathcal{P}$(dom(V)) expressed by

$$R(V) \doteq \{(x, S) \mid \text{fst}(V(x)) \sqsubseteq \bigsqcup \text{fst}[V[S]]\}$$

is the same in V', i.e. $R(V) = R(V')$.

Wellformedness of u' and $\{i; s\}$. Regarding $\{i; s\}$, as $\{i; s0, s1\}$ is an antichain, we have for every $r \in i$ that $r \parallel s0$ and $r \parallel s1$; therefore $r \parallel s$, which means that $\{i; s\}$ is also an antichain. Regarding u', if neither $s0$ nor $s1$ belong to u, then $u' = u$. Otherwise, we have for every $r \in u \setminus \{s0, s1\}$ that: $s0 \not\sqsubseteq r$ and $s1 \not\sqsubseteq r$ (because $u \sqsubseteq \{i; s0, s1\}$), and $r \not\sqsubseteq s$ (because u is an antichain); therefore, $r \parallel s$, which means that $u \setminus \{s0, s1\} \cup \{s\}$ is an antichain.

Invariant I_1. This is a local invariant on each stamp; it suffices to show that $u' \sqsubseteq \{i; s\}$. If neither $s0$ nor $s1$ belong to u, then $u' = u \sqsubseteq \{i; s\}$ (as $u \sqsubseteq \{i; s0, s1\}$). Otherwise, it is also trivial that $u' = u \setminus \{so, s1\} \cup \{s\} \sqsubseteq \{i; s\}$, for the same reason.

Invariant I_2. This invariant involves pairs of stamps; it suffices to consider the cases where the rewritten stamp is involved. For any other stamp (u_x, i_x) in V and string $r \in i_x$, due to Invariant I_2 on V we have: $r \parallel s0$, $r \parallel s1$, therefore $r \parallel s$; and also $r \parallel t$ for all $t \in i$; therefore $r \parallel v$ for all strings $v \in \{i; s\}$.

Invariant I_3. The invariant involves expressions of the form $\{r\} \sqsubseteq i_y \Rightarrow \{r\} \sqsubseteq u_y$, for stamps (u_x, i_x), (u_y, i_y), and $r \in u_x$. As for the previous invariant, it suffices to consider the cases where the rewritten stamp is involved:

$(u', \{i; s\}) = (u_y, i_y)$. Suppose $\{r\} \sqsubseteq \{i; s\}$; then, $\{r\} \sqsubseteq \{i; s0, s1\}$ and by I_3 on configuration V $\{r\} \sqsubseteq u$. If neither $s0$ nor $s1$ belong to u, then $u' = u$ and $\{r\} \sqsubseteq u'$. Otherwise, as $\{r\} \sqsubseteq \{i; s\}$, we have $r \neq s0$ and $r \neq s1$; therefore, $\{r\} \sqsubseteq u \setminus \{so, s1\} \cup \{s\} = u'$.

$(u', \{i; s\}) = (u_x, i_x)$. Suppose $\{r\} \sqsubseteq i_y$ with $r \in u'$. If neither $s0$ nor $s1$ belong to u, then $u' = u$, $r \in u$; therefore, by I_3 on V, we have $\{r\} \sqsubseteq u_y$. Otherwise, in which case $u' = u \setminus \{so, s1\} \cup \{s\}$, we have $r \neq s0$ and $r \neq s1$; also $r \neq s$, otherwise we would have $\{s\} \sqsubseteq i_y$, impossible by I_2 on V; therefore $r \in u$ and by I_3 on V we have $\{r\} \sqsubseteq u_y$.

Preservation of R. We prove that applying a rewriting $(u, \{i; s0, s1\}) \xrightarrow{\alpha} (u', \{i; s\})$ to a version stamp in a configuration V results in a configuration V' so that $R(V) = R(V')$. First we show that $x\ R(V)\ S \Rightarrow x\ R(V')\ S$. Suppose $x\ R(V)\ S$, i.e. fst$(V(x)) \sqsubseteq \bigsqcup$fst$[V[S]]$. We must consider two cases:

rewriting of $V(x)$. If $x \in S$ then $x\ R(V')\ S$ holds trivially; if $x \notin S$, as $u' \sqsubseteq u$, we have $u' \sqsubseteq u \sqsubseteq \bigsqcup$fst$[V[S]] = \bigsqcup$fst$[V'[S]]$.

rewriting of $V(y)$ with $y \in S$. The case $x = y$ is trivial (and already covered above). Otherwise $x \neq y$; let $Z = S \setminus \{y\}$; we have $V|Z = V'|Z$ and also $V(x) = V'(x)$; therefore fst$(V'(x)) \sqsubseteq \bigsqcup$fst$[V'[Z]] \sqcup u$. If $s0 \notin u$ and $s1 \notin u$ we have $u' = u$ and $x\ R(V')\ S$ holds trivially. Otherwise, in which case $u' = u \setminus \{s0, s1\} \cup \{s\}$, due to I_1 and I_2 (and $x \neq y$) we have that $s0$ and $s1$ do not belong to fst$(V(x))$; therefore the inequality above still holds replacing u by u', and thus we have $x\ R(V')\ S$.

Now we show that $x\ R(V')\ S \Rightarrow x\ R(V)\ S$. Suppose $x\ R(V')\ S$, i.e. fst$(V'(x)) \sqsubseteq \bigsqcup$fst$[V'[S]]$. We must consider two cases:

rewriting of $V(y)$ with $y \in S$. The case $x = y$ is trivial; if $x \neq y$, given that $u' \sqsubseteq u$, we have fst$(V(x)) =$ fst$(V'(x)) \sqsubseteq \bigsqcup$fst$[V'[S]] \sqsubseteq \bigsqcup$fst$[V[S]]$.

rewriting of $V(x)$. The case $x \in S$ is trivial; in the case $x \notin S$, we have that no string in fst$[V'[S]]$ is greater than or equal to s, otherwise, due to I_1 there would exist a string r in snd$[V'[S]]$ such that $s \sqsubseteq r$, something impossible because, due to I_2, no string in snd$[V'[S]]$ can be comparable to s Therefore, $s \notin u'$ which means we are in the case where $u' = u$ and so we have also fst$(V(x)) \sqsubseteq \bigsqcup$fst$[V[S]]$.

7. Conclusions

Both version vectors and vector clocks rely on the availability of identifiers that can support their ordering technique. We have argued that operation under partitioned operation and mobility prevents the use of traditional techniques for unique identifier generation, and that these operation modes are already common and call for appropriate solutions. Additionally, data management under these operation modes is mostly based on optimistic techniques and therefore requires robust dependency tracking solutions.

In this article we addressed the identification problem in the context of data dependency tracking. In order to achieve this goal we had to distinguish the ordering of elements in a frontier from the ordering of any two elements in a system run, thus contributing to the clarification of the role of version vectors. This distinction, together with the presence of the identification problem, raises a set of research lines, one of which was developed in the article. The other lines concern the design of decentralized vector clocks, by exploring autonomous identifiers on overall ordering, and the search for a more compact (possibly bound) form of version vectors on settings with fixed identifiers and frontier ordering.

We have developed a model of causal histories that is adapted to dynamic settings exhibiting autonomous interaction. We presented a version stamping mechanism that only relies on information that is locally available, overcoming the need for a global view. Finally, we established and proved a correspondence which states that the relation between any two given elements in a frontier, according to inclusion of causal histories, can be computed by their version stamps.

Version stamps, having solved the autonomous identification problem while addressing frontier ordering, provide an adequate dependency tracking mechanism that operates in scenarios where this functionality was not available.

The presented version stamp mechanism has been implemented in the PANASYNC project[2] [1]. This project is an application of version stamps to file replication, providing a set of tools for dependency tracking on single file copies. The project provides a C++ STL based library implementing version stamps.

Acknowledgements

The authors would like to thank José Bacelar Almeida, Jorge Sousa Pinto, Maria João Frade, Nuno Lopes and the anonymous reviewers for their comments.

References

[1] P. S. Almeida, C. Baquero, and V. Fonte. Panasync: Dependency tracking among file copies. In P. Guedes, editor, *Ninth ACM SIGOPS European Workshop*, pages 7–12. DIKU - University of Copenhagen, 2000.

[2] P. S. Almeida, C. Baquero, and V. Fonte. Version stamps — decentralized version vectors. Technical Report UMDITR2002.01, Departamento de Informática, Universidade do Minho, Mar. 2002.

[3] C. Baquero and F. Moura. Causality in autonomous mobile systems. In *Third European Research Seminar on Advances in Distributed Systems*. Broadcast, EPFL-LSE, Apr. 1999.

[4] M. Butrico, H. Chang, N. Cohen, and D. G. Shea. Data synchronization in mobile network computer – reference specification. In *WMR'98, ECOOP'98 Workshop Reader*. Springer Verlag, 1998.

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

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

[7] J. Haartsen, M. Naghshineh, J. Inouye, O. Joeressen, and W. Allen. Bluetooth: Vision, goals, and architecture. *ACM Mobile Computing and Communications Review*, 2(4):38–45, Oct. 1998.

[8] S.-T. Huang. Detecting termination of distributed computations by external agents. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS)*, pages 79–84, Washington, DC, 1989. IEEE Computer Society.

[9] K. C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, 1965.

[10] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

[12] R. Morris, J. Jannoti, F. Kaashoek, J. Li, and D. Decouto. Carnet: A scalable ad hoc wireless network system. In P. Guedes, editor, *Ninth ACM SIGOPS European Workshop*, pages 61–65. DIKU - University of Copenhagen, 2000.

[13] 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*, 9(3):240–246, 1983.

[14] 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.

[15] D. Ratner, P. Reiher, and G. Popek. Dynamic version vector maintenance. Technical Report CSD-970022, Department of Computer Science, University of California, Los Angeles, 1997.

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

[17] M. B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16:23–36, 1978.

[18] F. Torres-Rojas and M. Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–196, 1999.

[2] `http://sourceforge.net/projects/panasync`.