Unilateral version vector pruning using loosely synchronized clocks

Yasushi Saito
Imaging Systems Laboratory
HP Laboratories Palo Alto

HPL-2002-51
March 5th , 2002*

E-mail: ysaito@hpl.hp.com

replication, version vector, vector clock, distributed system

This paper presents a new lightweight algorithm for pruning inactive entries in version vectors (VVs). This algorithm lets each node remove inactive VV entries without any coordination with other nodes. It achieves this feature by devising a new way to compare two version vectors based on loosely synchronized clocks and placing a timing restriction on the behavior of the application. VVs computed by our algorithm can accurately and completely capture the "happened-before" relation between events just like ordinary VVs. This paper proves the correctness of our algorithm as well.

Unilateral version vector pruning using loosely synchronized clocks

Yasushi Saito
HP Labs, Storage Systems Department
ysaito@hpl.hp.com

March 5, 2002

Abstract

This paper presents a new lightweight algorithm for pruning inactive entries in version vectors (VVs). This algorithm lets each node remove inactive VV entries without any coordination with other nodes. It achieves this feature by devising a new way to compare two version vectors based on loosely synchronized clocks and placing a timing restriction on the behavior of the application. VVs computed by our algorithm can accurately and completely capture the "happened-before" relation between events just like ordinary VVs. This paper proves the correctness of our algorithm as well.

1 Introduction

A version vector (VV) is a data structure, kept for each copy (replica) of a modifiable object, that summarizes the set of updates applied to the replica. It is a generic mechanism for detecting Lamport's happen-before relationship [9] between events in a distributed system, and its uses include detecting concurrent updates in a replicated database system [11, 8, 14], distributed system monitoring and debugging [6], and efficient update propagation in mobile database systems [14, 12, 7].

This paper presents a new lightweight algorithm for pruning inactive entries in VVs.[1] Our algorithm addresses a key shortcoming of VVs: they encode the set of nodes (computers) that participate in the VV protocol, and their size and computational overheads grow with the number of nodes. This problem becomes serious especially when a system experiences frequent and dynamic changes to the participant set [12, 14]. In such an environment, the size of VVs grows unboundedly even when the number of active nodes is small, because VVs must keep the entries for all nodes that once participated in the protocol in the past. Such a system must prune inactive VV entries systematically. A VV pruning algorithm should satisfy the following goals:

Localization: VV pruning shall require no synchronous communication among nodes. Localization not only reduces the cost of the algorithm, but also gives managerial latitude to the system. For example, it allows VV pruning to be performed when a node boots, every night when the node is idle, or whenever a node wishes.

Livelock freedom: A slow or unavailable node shall not hamper pruning process running on other nodes. This property is crucial since VV-pruning is most often needed when some nodes remain inactive for a long period.

In the past, VV pruning has usually been implemented using a distributed consensus protocol, which is not only expensive but also prone to livelocking. In contrast, our algorithm achieves the above goals by letting nodes unilaterally remove inactive VV entries. The key difficulty we face is inherent asynchrony in distributed systems. For example, a node with a fast clock may remove a VV entry that is considered still "live" by another node with a slower clock.

The algorithm we propose utilizes loosely synchronized clocks and develops a new way of comparing two VVs to coordinate nodes implicitly. VVs computed by this algorithm are *accurate* and *complete*; for any event that "happened before" another, the former event's VV will dominate the latter's, and vice versa. The downside of this algorithm is its reliance on synchronous networking. In particular, it demands that each event be delivered to all live nodes and processed by them within a fixed period. This seemingly strong limitation, however, can usually be circumvented by choosing a long VV-entry expiration period (e.g., a month), because VV-pruning is a non-critical, background operation that can be delayed if needed.

[1] The algorithm is for pruning entries in a VV itself. It is not for pruning data structures associated with version management, such as an update log in a replicated database system. The reader should consult [15] for algorithms for dealing with the latter issue.

Section 1.1 presents the basic "textbook" VV algorithm and explains the difficulty of VV-entry pruning. Section 1.3 overviews VV-pruning algorithms proposed in the past. Section 2 presents our algorithm, and Section 3 proves its correctness.

1.1 Description of the basic VV algorithm

Figure 1 shows the basic VV algorithm, introduced first in the LOCUS distributed operating system [11, 16]. A VV is a table that partially maps a *node ID* to a *timestamp*. It is kept for each copy (replica) of an object replicated in a distributed system. A node ID is any bit-string that uniquely identifies a node that stores a replica (e.g., an IP address). A timestamp is any monotonically increasing value, but most systems use a counter that increments when an update locally happens at a node. We follow this convention in our presentation. The VV attached to an event or a data item captures a distributed cut of the system, i.e., the perception of the state of all nodes in the system by the event issuer [3]. This algorithm supports node (replica) addition simply by letting a new node copy the VV (and the database contents) from another existing node [7, 12, 14]. Other nodes treat non-existent entries in a VV as zero. Thus, the VV-entry comparison operator $\prec$ is defined as in Table 1.

This paper focuses on the use of VV for concurrency detection. Specifically, we assume that an incoming event is compared against the VV representing the state of the replica (variable *vv* in Figure 1) and that the incoming event is processed (i.e., applied to the replica) immediately after its arrival.

This VV algorithm can provably capture the "happened-before" relationship [9] between two events E_1 (with VV vv_1) and E_2 (with VV vv_2) as follows [5, 10]:

- If Dominates(vv_1, vv_2), then E_2 "happened-before" E_1.
- If Dominates(vv_1, vv_2), then E_1 "happened-before" E_2.
- If neither VV dominates the other, then neither of the events "happened-before" the other. These events are said *conflicting*, or *concurrent*.

Figures 2 and 3 show examples of the use of VVs. Figure 2 shows how VVs can discover non-concurrent (consistent) updates. Figure 3 shows how VVs can detect concurrent updates.

1.2 Difficulty of VV-entry pruning

A VV entry should be removed when a node retires or stops issuing updates so as to reduce both the spatial and computational overheads of VV management. Removing a VV en-

```
// Per-node persistent variables.
type VV = NodeID ↦ Timestamp
     Update = ⟨vv: VV, data: Whatever⟩

var ts: Timestamp
    vv: VV

// Called to issue a new event.
proc IssueUpdate(data)
  ts ← ts + 1
  vv[myself] ← ts
  u ← Update⟨vv ← vv, data ← data⟩
  Send u to other nodes.

// Called when an update arrives at a node.
proc ReceiveUpdate(u)
  if Equal(vv, u.vv) or Dominates(vv, u.vv) then
    Duplicate event reception.
  if Dominates(u.vv, vv) then
    Apply the update.
  else
    Conflict! resolve the update

  // Compute the pairwise maxima of the two VVs.
  for $i \in \mathrm{dom}(u.vv) \cup \mathrm{dom}(vv)$
    if $vv[i] \prec u.vv[i]$
      $vv[i] \leftarrow u.vv[i]$

proc Equal($vv_1$,$vv_2$): bool
  return $vv_1[i] \approx vv_2[i], \forall i \in \mathrm{dom}(vv_1) \cup \mathrm{dom}(vv_2)$

proc Dominates($vv_1$,$vv_2$): bool
  if Equal($vv_1$, $vv_2$) then return false
  return $vv_1[i] \prec vv_2[i]$ or $vv_1[i] \approx vv_2[i]$,
       $\forall i \in \mathrm{dom}(vv_1) \cup \mathrm{dom}(vv_2)$
```

Figure 1: *The basic Version-vector algorithm. Function "*dom*(vv)" returns the domain (keys) of version vector vv. Operators $\prec$ and $\approx$ are defined in Table 1.*

	$n \notin \mathrm{dom}(vv_b)$	$n \in \mathrm{dom}(vv_b)$
$n \notin \mathrm{dom}(vv_a)$	$vv_a[n] \approx vv_b[n]$	$vv_a[n] \prec vv_b[n]$
$n \in \mathrm{dom}(vv_a)$	$vv_a[n] \succ vv_b[n]$	$*$

Table 1: *Decision matrix for operators "$\prec$", "$\approx$", "$\succ$", to support node addition. For example, if, for some node n, $n \notin \mathrm{dom}(vv_a)$ and $n \in \mathrm{dom}(vv_b)$ then the algorithm decides that $vv_a[n] \prec vv_b[n]$. "$*$" means that the result is computed by comparing $vv_a[n]$ and $vv_b[n]$ arithmetically.*

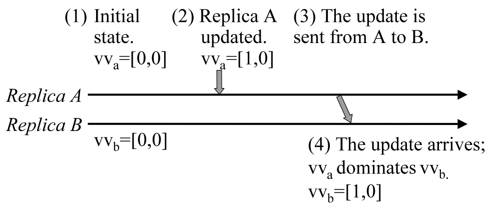
(1) Initial state. vv_a=[0,0]
(2) Replica A updated. vv_a=[1,0]
(3) The update is sent from A to B.
Replica A
Replica B
vv_b=[0,0]
(4) The update arrives; vv_a dominates vv_b. vv_b=[1,0]

Figure 2: *In this example, two replicas, a and b, start from the identical state in step (1), with version vectors* $[0,0]$ *(i.e., entries for a and b both being 0). Replica a is updated in step (2), and the update is sent to b in step (3). Since a is strictly newer than b, replica b detects no conflict in step (4).*

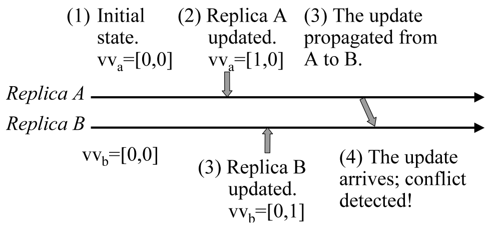
(1) Initial state. vv_a=[0,0]
(2) Replica A updated. vv_a=[1,0]
(3) The update propagated from A to B.
Replica A
Replica B
vv_b=[0,0]
(3) Replica B updated. vv_b=[0,1]
(4) The update arrives; conflict detected!

Figure 3: *In this example, replicas a and b start from the identical state in step (1). Two updates are issued concurrently at replicas a and b concurrently in steps (2) and (3). When a's update is sent to b, replica b detects a conflict, because neither vv_a nor vv_b dominates the other.*

try is not trivial, however, because simply erasing the entry unilaterally causes both false-positive and false-negative concurrency detection. The first problem is shown in Figure 4. Here, if two concurrent events, received by nodes *a* and *b*, fail to reach each other, then these events could falsely be considered non-concurrent. The second case is shown in Figure 5; when one event is sent from node *a* to *b* just after *a* deleted a VV entry (but *b* hasn't), then two non-concurrent events could be diagnosed as concurrent. These examples demonstrate that nodes cannot simply remove VV entries at their will; they must agree on the timing and the names of the entries to be removed beforehand.

1.3 Related work

Several systems run distributed consensus protocols to let nodes agree on VV entries to be pruned. TSAE [7] runs a two-phase protocol for removing an entry for a retiring node from VVs of other nodes. In the first phase, the retiring node circulates the "retirement" message to other nodes. After confirming the reception of retirement by all the nodes, the retiring node circulates another message to let the nodes delete the entry from their VVs. Roam [13, 14] supports deletion of entries for retiring as well as inactive nodes. Any node in Roam

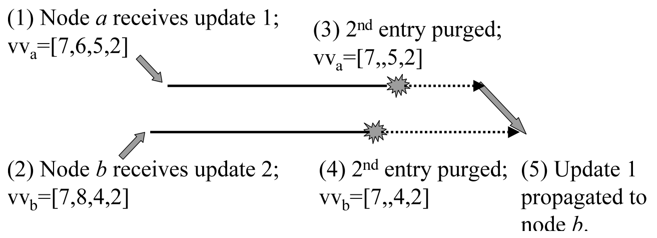
(1) Node a receives update 1; vv_a=[7,6,5,2]
(3) 2nd entry purged; vv_a=[7,,5,2]
(2) Node b receives update 2; vv_b=[7,8,4,2]
(4) 2nd entry purged; vv_b=[7,,4,2]
(5) Update 1 propagated to node b.

Figure 4: *The first problem of naive unilateral VV-entry removal. Initially, nodes a and b receive two concurrent updates 1 and 2 with VVs VV_a and VV_b in steps (1) and (2). The system remains quiescent, and the two updates fail to reach each other until the second entry in their VVs are pruned in steps (3) and (4). In step (5), update 1 is sent from node a to node b, but b fails to detect a conflict.*

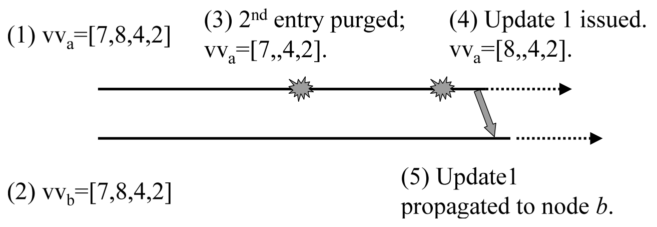
(1) vv_a=[7,8,4,2]
(3) 2nd entry purged; vv_a=[7,,4,2].
(4) Update 1 issued. vv_a=[8,,4,2].
(2) vv_b=[7,8,4,2]
(5) Update1 propagated to node b.

Figure 5: *The second problematic case. Initially, in steps (1) and (2), the state of node a and b is identical. In step (3), because node a's clock is faster than b's, the second entry in VV_a is pruned before VV_b's. Later, in step (4), node a issues a new update 1 that "happens-after" the update currently stored in both a and b. When update 1 arrives at node b, b detects a false conflict.*

can start the pruning process autonomously by proposing the value it wants to subtract from a VV entry. The proposed value propagates among nodes epidemically. After the proposal finishes circulating, each node subtracts the agreed-on value from the entry; the entry is removed from the VV if its value becomes zero. These consensus-based algorithms are expensive and prone to livelocking.

Adya [1] proposes a VV-pruning algorithm that deletes entries unilaterally without clock-skew compensation. Thus, it is inaccurate (it may detect conflicts falsely), but for their purpose of the summarization of causal dependency, inaccuracy only results in a delay of program execution.

Bayou [12] supports removing VV entries for retiring nodes through an ingenious node-naming scheme. A newly created node (say, a) communicates with a node already in the system (say, b) and receive b's timestamp (say T_b). Node a then assumes the ID of $\langle T_b, b\rangle$. This ID-selection scheme allows each node to determine whether a missing entry in its VV is due to a new node creation or a node removal. While this algorithm achieves our goals of localization and livelock freedom, it still has two shortcomings. First, the size of node IDs grows gradually as replicas are created and deleted. Second, it does not support removing VV entries for live but inactive nodes.

Arora [2] proposes a VV algorithm in which the range of values in each entry is bounded, and thus the total size of the VV is bounded, assuming that the set of replicas is fixed. It achieves this property by forcing each node to communicate to others at a consistent pace and not allowing the entry "wrap-round" too quickly. It does not, however, support VV-entry pruning.

2 VV pruning using loosely synchronized clocks

Our entry-removing algorithm is based on a simple intuition: just remove entries unilaterally when they are found to be inactive for a long period, e.g., a month. The two problems with the naive implementation, demonstrated in Section 1.2, are addressed as follows. The first problem is fundamentally inevitable in an asynchronous, livelock-free removal algorithm. In fact, its inevitability is a direct consequence of Charron-Bost's theorem [4], which proved that detecting causality accurately in an N-node asynchronous distributed system requires a use of version vectors of size at least N. Thus, we punt and simply prohibit such situation from happening: our algorithm demands synchronous networking, i.e., it demands that any event be propagated to all live nodes within a fixed period.[2] The second problem is solved by creating a "grace period" long enough to absorb both network delay and clock skew, just before a node removes entries. During the grace period, timestamps about to be pruned are considered equal even when even when they are not.

Figure 6 shows our new algorithm. We extend the VV and embed, in each entry, the physical-clock time the entry was last updated by the node corresponding to the entry. (Systems that use a physical clock as a timestamp, e.g., TSAE [7], need not this change). The definition of the new VV-entry comparison operator, $\prec^n, \approx^n, \succ^n$, are shown in Table 3. This algorithm makes several timing assumptions about the system's behavior, as summarized in Figure 7.

This algorithm prunes VV entries in two stages: *deactivation*, and *deletion*. An entry is deactivated when it remains unchanged for D_{retire} seconds. A deactivated entry is logically treated as a deleted entry, but is kept in the VV so as to absorb clock skew. The entry is actually removed from the VV when it remains unchanged for D_{delete} seconds. D_{retire} and D_{delete} are any values that satisfy the relations defined in Table 2. In practice, we anticipate that both deactivation and deletion are performed periodically in a batch, especially when a node manages replicas of many objects.

$$\begin{aligned} D_{retire} &> D_{prop} + D_{net} + D_{skew} \\ D_{delete} &> D_{retire} + D_{net} + D_{skew} \end{aligned}$$

Table 2: *Definition of* D_{retire} *and* D_{delete}.

3 Correctness proof

This section proves that our new VV algorithm accurately and completely captures the "happened-before" relationship between events. For the sake of the proof, we simulate the basic VV algorithm (Figure 1) in parallel with our new algorithm and show that a VV computed by our algorithm (*vv* hereafter) dominates another if and only if the "imaginary" VV computed by the basic algorithm (*iv* hereafter) dominates its counterpart. Since the basic VV algorithm is proven correct [5, 10], this argument suffices to show the correctness of our algorithm.

[2]Alternatively, a node that fails to receive an update within the period must commit suicide. Our algorithm does not provide a mean of deciding whether a node should commit suicide or not.

	$n \notin \mathrm{dom}(vv_b)$	$n \in \mathrm{dom}(vv_b), \neg\mathrm{Active}(vv_b[n])$	$n \in \mathrm{dom}(vv_b), \mathrm{Active}(vv_b[n])$
$n \notin \mathrm{dom}(vv_a)$	$vv_a[n] \approx^n vv_b[n]$ $_{\langle 1 \rangle}$	$vv_a[n] \approx^n vv_b[n]$ $_{\langle 4 \rangle}$	$vv_a[n] \prec^n vv_b[n]$ $_{\langle 7 \rangle}$
$n \in \mathrm{dom}(vv_a), \neg\mathrm{Active}(vv_a[n])$	$vv_a[n] \approx^n vv_b[n]$ $_{\langle 2 \rangle}$	$vv_a[n] \approx^n vv_b[n]$ $_{\langle 5 \rangle}$	$*$ $_{\langle 8 \rangle}$
$n \in \mathrm{dom}(vv_a), \mathrm{Active}(vv_a[n])$	$vv_a[n] \succ^n vv_b[n]$ $_{\langle 3 \rangle}$	$*$ $_{\langle 6 \rangle}$	$*$ $_{\langle 9 \rangle}$

Table 3: *New decision matrix for operator "$\prec^n$". "$*$" means that the result is computed by arithmetically comparing $vv_a[n]$ and $vv_b[n]$. Tags such as $_{\langle 1 \rangle}$ and $_{\langle 2 \rangle}$ are used to refer to the rules in the correctness proof.*

3.1 Notations

- $a.vv$ is the value of variable vv (Figure 6) at node a.
- $u.vv$ is the VV associated with update u.
- $a.iv$ and $u.iv$ are the value of the VV computed by the basic VV algorithm, corresponding to $a.vv$ and $u.vv$, respectively.
- Time mentioned in the proof is hypothetical physical-clock time observed by one arbitrary node in the system. Notice that, with this definition, the following relationship always holds between hypothetical time T and the physical-clock time T' observed at any node:

$$T - D_{skew} < T' < T + D_{skew}. \qquad (1)$$

3.2 Completeness

We first prove the completeness of the algorithm; i.e., that whenever the basic algorithm detects a concurrency between two events, our algorithm also detects the concurrency. The opposite property, i.e., that a concurrency detected by our algorithm is always genuine, is proved in Section 3.3.

Claim 1 *Suppose that, for two VVs vv_1 and vv_2, there exists node x such that $x \in (\mathrm{dom}(vv_1) \cap \mathrm{dom}(vv_2))$. Then,*

$$vv_1[x] \prec^n vv_2[x] \quad \textit{iff.} \quad iv_1[x] \prec iv_2[x],$$

and

$$vv_1[x] \approx^n vv_2[x] \quad \textit{iff.} \quad iv_1[x] \approx iv_2[x].$$

Proof. The entry for x in a VV can be incremented only by node x, which never decrements its own entry. Thus, as far as x is in both VVs, the entries have the same meaning as those of the imaginary VVs. ■

Now, we prove the completeness by proving that comparison of individual entries in vv will yield the same result as that in iv.

Theorem 1 *Suppose that node a receives update u from node b at time T. If $a.iv[x] \prec u.iv[x]$ for some node x, then $a.vv[x] \prec^n u.vv[x]$.*

Proof. Let u' be the update that first set $x.iv[x]$ to $u.iv[x]$, and $T_{u'}$ be the time u' was issued (by node x). Notice that, from the way $u'.iv[x].wc$ is calculated and (1), the following inequality holds.

$$u'.iv[x].wc - D_{skew} < T_{u'} < u'.iv[x].wc + D_{skew}. \qquad (2)$$

We first prove that $x \in \mathrm{dom}(u.vv)$ and $\mathrm{Active}(u.vv[x])$ at time T. Lets calculate the last moment that node a can receive u. From Assumption 1, a must receive u' by $T_{u'} + D_{prop}$ (otherwise, $a.iv[x]$ becomes $u.iv[x]$). On the other hand, T_d, the earliest moment at which $u.vv[x]$ can be deleted (by either node a or the sender of u) is:

$$\begin{aligned} T_d &= u.iv[x].wc + D_{delete} \\ &> u.iv[x].wc + D_{prop} + D_{net} + D_{skew} \\ &> T_{u'} + D_{prop}. \end{aligned}$$

Thus, u must arrive at a before $u.vv[x]$ is deleted.

We now enumerate the state of $a.vv$ at time T and show that in all the cases, node a will conclude that $a.vv[x] \prec^n u.vv[x]$.

Case 1: $x \in \mathrm{dom}(a.vv)$. Rule $\langle 9 \rangle$ in Table 3 apply. From Claim 1, the theorem holds.

Case 2: $x \notin \mathrm{dom}(a.vv)$. Rule $\langle 3 \rangle$ in Table 3 will decide that $a.vv[x] \prec^n u.vv[x]$. ■

Theorem 2 *Suppose that node a receives update u from node b at time T. If $a.iv[x] \succ u.iv[x]$ for some node x, then $a.vv[x] \succ^n u.vv[x]$.*

Proof. Let u' be the update that first set $x.iv[x]$ to $u.iv[x]$, and $T_{u'}$ be the time u' was issued (by node x). Notice that the inequality (2) holds here as well.

We show that $x \in \mathrm{dom}(a.vv)$ and $\mathrm{Active}(a.vv[x])$. Node a must receive u' time $T_{u'} + D_{prop}$ by the latest. In contrast,

```
// Per-node persistent variables.
type Entry = ⟨ts: Timestamp , wc: PhysicalClock ⟩
    VV = NodeID ↦ Entry

var ts: Timestamp
    vv: VV

// Same as in Figure 1.
proc IssueUpdate(data)
  ts ← ts + 1
  vv[myself] ← ⟨ts, Now⟩
  u ← Update⟨vv ← vv, data ← data⟩
  Send v to other nodes.

proc ReceiveUpdate(u)
  if Equal(vv, u.vv) or Dominates(vv, u.vv) then
    Duplicate update reception.
  if Dominates(u.vv, vv) then
    Apply the update.
  else
    Conflict! resolve the update

  for i ∈ dom(u.vv)∪dom(vv)
    if vv[i] $\prec^n$ u.vv[i]
      vv[i] ← u.vv[i]

proc Equal($vv_1$,$vv_2$): bool
  PruneVV($vv_1$)
  PruneVV($vv_2$)
  return $vv_1$[i] $\approx^n$ $vv_2$[i], ∀i ∈ dom($vv_1$)∪dom($vv_2$)

proc Dominates($vv_1$,$vv_2$): bool
  if Equal($vv_1$, $vv_2$) then return false
  PruneVV($vv_1$)
  PruneVV($vv_2$)
  return $vv_1$[i] $\prec^n$ $vv_2$[i] or $vv_1$[i] $\approx^n$ $vv_2$[i],
        ∀i ∈ dom($vv_1$)∪dom($vv_2$)

// PruneVV can be called at any time to prune VV entries.
proc PruneVV(vv)
  for n ∈ dom(vv)
    if  vv[n].wc < Now - $D_{delete}$ then
      Delete the entry for n from vv

// Used to compute $\prec^n$. See also Table 3.
proc Active(ent)
  return ent.wc < Now - $D_{retire}$
```

Figure 6: *Version vector algorithm with lightweight pruning support. "Now" returns the value of the local physical clock.*

1. Any event is propagated to any other node within D_{prop} seconds after it was issued, unless the replica has been continuously dead during that period. In other words, the algorithm demands that any change to a VV entry be propagated to all nodes within D_{prop} seconds.

2. Maximum physical-clock skew among nodes is D_{skew} seconds.

3. Any network message sent from one node arrives and is processed by the destination within D_{net} seconds.

Figure 7: *Timing requirements of the VV-pruning algorithm*

T_d, the earliest moment at which $a.vv[x]$ can be deleted, is computed as follows:

$$\begin{aligned} T_d &= a.vv[x].wc + D_{delete} \\ &> a.vv[x].wc + D_{prop} + D_{net} + D_{skew} \\ &> T_{u'} + D_{prop}. \end{aligned}$$

Thus, the algorithm uses only the last row in Table 3 in this case. By combining the last row and Claim 1, the algorithm will determine that $a.vv[x] \succ^n u.vv[x]$. ■

Theorem 3 *Suppose that node a receives update u from replica b at time T. If $a.iv[x] \approx u.iv[x]$ for some node x, then $a.vv[x] \approx^n vv_b[x]$.*

Proof. Let u' be the update that first set $x.iv[x]$ to $u.iv[x]$, and $T_{u'}$ be the time u' was issued by node x. Notice that the inequality (2) holds here as well.

Case 1: $x \in \mathrm{dom}(a.vv)$.

If $x \in \mathrm{dom}(u.vv)$, then the theorem holds from Rules ⟨5⟩, ⟨6⟩, ⟨8⟩, and⟨9⟩. Suppose otherwise, i.e., b removed $b.vv[x]$ before sending u to a. The earliest moment that b can remove $b.vv[x]$, T_d, is computed as follows:

$$\begin{aligned} T_d &\geq b.iv[x].wc + D_{delete} - D_{skew} \\ &= a.iv[x].wc + D_{delete} - D_{skew} \\ &> a.iv[x].wc + D_{retire}. \end{aligned}$$

Thus, ¬Active($a.vv[x]$) holds. Combine that with Claim 1, and the theorem holds.

Case 2: $x \notin \mathrm{dom}(a.vv)$.

In this case, $x \notin \mathrm{dom}(u.vv)$, because the "wc" field in the two VVs are identical and the algorithm will remove $u.vv[x]$ before the comparison. Thus, the algorithm will determine from Rule 1 that the two VV entries are equal. ■

3.3 Accuracy

This section proves that a concurrency detected by our algorithm is a genuine concurrency.

Theorem 4 *Suppose that node a receives update u from node b at time T. If $a.vv[x] \prec^n u.vv[x]$ for some node x, then $a.iv[x] \prec u.iv[x]$.*

Proof. Theorems 1, 2, 3 establish the following relationships.

$$\begin{aligned}
a.iv[x] \prec u.iv[x] &\Rightarrow a.vv[x] \prec^n vv_b[x], && \text{(Theorem 1)}\\
a.iv[x] \approx u.iv[x] &\Rightarrow a.vv[x] \approx^n vv_b[x], && \text{(Theorem 3)}\\
a.iv[x] \succ u.iv[x] &\Rightarrow a.vv[x] \succ^n vv_b[x]. && \text{(Theorem 2)}
\end{aligned}$$

Now, by exploiting the fact that the combinations of "$\prec, \approx, \succ$" and "$\prec^n, \approx^n, \succ^n$" both define total orderings, we can prove the reverse property easily.

For example, Theorem 5 is proved by combining Theorems 2 and 3.

$$\begin{aligned}
a.vv[x] \not\succ^n u.vv[x] \quad &\wedge \quad a.vv[x] \not\approx^n u.vv[x]\\
&\Rightarrow \quad a.iv[x] \not\succ u.iv[x] \wedge a.iv[x] \not\succ u.iv[x]\\
&\text{i.e.,}\\
a.vv[x] \prec^n u.vv[x] \quad &\Rightarrow \quad a.iv[x] \prec^n u.iv[x]. \blacksquare
\end{aligned}$$

Theorem 5 *Suppose that node a receives update u from node b at time T. If $a.vv[x] \succ^n u.vv[x]$ for some node x, then $a.iv[x] \succ u.iv[x]$.*

Proof. The combination of Theorems 1 and 3 proves this theorem. ■

Theorem 6 *Suppose that node a receives update u from node b at time T. If $a.vv[x] \approx^n u.vv[x]$ for some node x, then $a.iv[x] \approx u.iv[x]$.*

Proof. The combination of Theorems 1 and 2 proves this theorem. ■

4 Conclusions

This paper presented an algorithm for pruning inactive entries in version vectors. This algorithm offers two properties crucial in practical distributed services. First, it requires no synchronous coordination among nodes, thereby allowing nodes to prune entries when they desire. Second, it can prune entries belonging to nodes that remain incommunicable. It still cannot escape from the fundamental trade-off between asynchrony and liveness in distributed systems in that it demands that clock skew among nodes be bounded and each event be delivered to all other nodes within a fixed period. In many applications, however, these limitations are practically a non-issue, because VV-entry pruning is a background operation that can be delayed, and one can avert the problem simply by choosing a long expiration period during which all events are certainly delivered to all nodes.

Acknowledgements

I thank Christos Karamanolis and John Wilkes for helping me out with ideas and presentations.

References

[1] Atul Adya and Barbara Liskov. Lazy consistency using loosely synchronized clocks. In *16th Symp. on Princ. of Distr. Comp. (PODC)*, pages 73–82, Santa Barbara, CA, USA, August 1997.

[2] Anish Arora, Sandeep Kulkarni, and Murat Demirbas. Resettable vector clocks. In *ACM Symp. on Princ. of Distr. Comp. (PODC)*, 2000.

[3] Ozalp Babaoglu. *Distributed Systems*, chapter 4, pages 55–96. Addison-Wesley, 1993.

[4] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, July 1991.

[5] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia, 1988.

[6] Vijay K. Garg and Craig M. Chase. Distributed algorithms for detecting conjunctive predicates. In *IEEE International Conference on Distributed Computing Systems*, June 1995.

[7] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California Santa Cruz, December 1992. Tech. Report no. UCSC-CRL-92-52, ftp://ftp.cse.ucsc.edu/pub-/tr/ucsc-crl-92-52.ps.Z.

[8] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX Winter Tech. Conf.*, pages 95–106, New Orleans, LA, USA, January 1995. http://www.cs.cmu.edu/afs/cs/project/coda/Web-/docdir/usenix95.pdf.

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

[10] Friedmann Mattern. Virtual time and global states of distributed systems. In *Int. W. on Parallel and Dist. Algorithms*, pages 216–226. Elsevier Science Publishers B.V. (North-Holland), 1989. http://www.informatik.tu-darmstadt.de/VS/Publikationen/.

[11] D. Scott 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. *IEEE Transactions on Soft. Eng.*, SE-9(3):240–247, 1983.

[12] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *16th Symp. on Op. Sys. Principles (SOSP)*, pages 288–301, St. Malo, France, October 1997.

[13] David Ratner, Peter Reiher, and Gerald Popek. Dynamic version vector mainenance. Technical Report CSD-970022, UCLA, June 1997. Available through http:/-/www.csindex.com.

[14] David H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, UC Los Angeles, 1998. Tech. Report. no. UCLA-CSD-970044, http://ficus-www.cs.ucla.edu-/ficus-members/ratner/papers/diss.ps.gz.

[15] Yasushi Saito. Optimistic replication algorithms, May 2000. General examination report. Available at http:/-/www% -.cs.washington.edu/homes/yasushi/replica.ps.

[16] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The Locus distributed operating system. In *9th Symp. on Op. Sys. Principles (SOSP)*, pages 49–70, Bretton Woods, NH, USA, October 1983.