Encoded Vector Clock: Using Primes to Characterize Causality in Distributed Systems

Ajay D. Kshemkalyani
University of Illinois at Chicago
Chicago, Illinois
ajay@uic.edu

Ashfaq Khokhar
Iowa State University
Ames, Iowa
ashfaq@iastate.edu

Min Shen
University of Illinois at Chicago
Chicago, Illinois
victor.nju@gmail.com

ABSTRACT

The vector clock is a fundamental tool for tracking causality in distributed applications. Unfortunately, it does not scale well to large systems because each process needs to maintain a vector of size n, where n is the total number of processes in the system. To address this problem, we propose the encoding of the vector clock using prime numbers to use a single number to represent vector time. We propose the operations on the encoded vector clock (EVC). We then show how to timestamp global states and how to perform operations on the global states using the EVC. We also discuss scalability issues of the EVC.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; *Concurrent algorithms*;

KEYWORDS

causality, vector clock, encoding, prime numbers, happened before relation

ACM Reference Format:
Ajay D. Kshemkalyani, Ashfaq Khokhar, and Min Shen. 2018. Encoded Vector Clock: Using Primes to Characterize Causality in Distributed Systems. In *ICDCN '18: 19th International Conference on Distributed Computing and Networking, January 4–7, 2018, Varanasi, India.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3154273.3154305

1 INTRODUCTION

A distributed system is modeled as an undirected graph (P, L), where P is the set of processes and L is the set of communication links connecting them. Let $n = |P|$ and let d denote the degree of the graph. Between any two processes, there may be at most one logical channel over which the two processes communicate asynchronously. A logical channel from P_i to P_j is formed by paths over links in L. We do not assume FIFO logical channels; thus the messages may be delivered out of order. Let l denote the number of logical channels in the system.

The execution of process P_i produces a sequence of events $E_i = \langle e_i^0, e_i^1, e_i^2, \cdots \rangle$, where e_i^k is the k^{th} event at process P_i. An event at a process can be an *internal* event, a *message sending* event, or a *message reception* event. Let $E = \bigcup_{i \in P} \{e \mid e \in E_i\}$ denote the set of events in a distributed execution. The causal precedence relation between events induces an irreflexive partial order on E. This relation is defined as Lamport's "happened before" relation [14], and denoted as $\rightarrow$. An execution of a distributed system is thus denoted by the tuple $(E, \rightarrow)$. Lamport designed the scalar clock, which is a function C that assigns integer timestamps to events such that if $e \rightarrow f$, then $C(e) < C(f)$. However, the drawback of scalar clocks is that $C(e) < C(f)$ does not imply that $e \rightarrow f$.

Mattern [15] and Fidge [4] designed the vector clock which assigns a vector V to each event such that $e \rightarrow f \iff V(e) < V(f)$. Each process P_i maintains a vector clock V. Events are timestamped by the current clock value. The vector clocks, initialized to the 0-vector, are updated by the following rules.

(1) Before an internal event happens at process P_i, $V[i] = V[i]+1$ (local tick).
(2) Before process P_i sends a message, it first executes $V[i] = V[i] + 1$ (local tick), then it sends the message piggybacked with V.
(3) When process P_i receives a message piggybacked with timestamp U, it executes
$\forall k \in [1 \ldots n], V[k] = \max(V[k], U[k])$ (merge);
$V[i] = V[i] + 1$ (local tick)
before delivering the message.

The vector clock is a fundamental tool to characterize causality in distributed executions [11, 19]. Each process needs to maintain a vector of size n, where n is the total number of processes in the system, to represent the local vector clock. Unfortunately, this does not scale well to large systems. Several works in the literature attempted to reduce the size of vector clocks [12, 16, 21, 22], but they had to make some compromises in accuracy or alter the system model, and in the worst-case, were as lengthy as vector clocks. To address this problem, we propose the encoding of the vector clock using prime numbers to use a single number to represent vector time.

In Section 2, we give the encoding of the vector clock, and the operations on the encoded vector clock (EVC). In Section 3, we give mechanisms to timestamp global states and operations on the global states using EVC. In Section 4, we discuss scalability techniques for EVC. We give concluding remarks in Section 5.

2 ENCODED VECTOR CLOCK

Charron-Bost has shown that to capture the partial order $(E, \rightarrow)$, the size of the vector clock is the dimension of the partial order [2], which is bounded by the size of the system, n. Instead of using a

ICDCN '18, January 4–7, 2018, Varanasi, India

ACM ISBN 978-1-4503-6372-3/18/01…$15.00
https://doi.org/10.1145/3154273.3154305

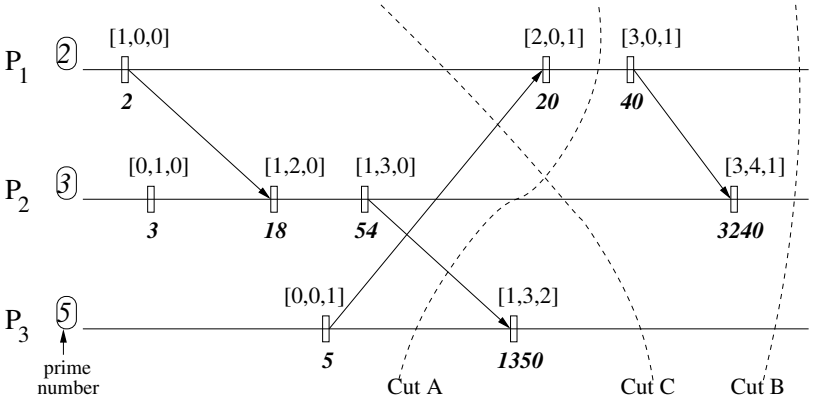

Figure 1: Illustration of using EVC. The vector timestamps and EVC timestamps are shown above and below each timeline, respectively. In real scenarios, only the EVC is stored and transmitted.

vector of size n, we propose that the vector can be encoded into a single number using n distinct prime numbers. The encoding of vector clocks using primes was used for detecting locality-aware conjunctive predicates in large-scale systems [20]. A vector clock containing n elements, $V = \langle v_1, v_2, \cdots, v_n \rangle$, can be encoded by n distinct prime numbers $p_1, p_2, \cdots, p_n$ as:

$$Enc(V) = p_1^{v_1} * p_2^{v_2} * \cdots * p_n^{v_n}$$

However, only being able to encode a vector clock into a single number is insufficient to track causal relations. We develop the EVC technique to show how to implement the basic operations of a vector clock. The EVC at each process P_i is initialized to 1. For a vector clock to work, it needs three basic operations: local tick, merge, and compare. Below, we implement these basic operations using EVC.

2.1 Encoded Vector Clock Operations

Local Tick: Whenever the logical time advances locally at P_i, the local component of the vector clock needs to tick. This increases the local component in the vector by 1:

$$V[i] = V[i] + 1$$

While using EVC, this operation is equivalent to multiplying the EVC timestamp by the local prime number p_i,

$$Enc(V) = Enc(V) * p_i$$

Merge: Whenever one process sends a message to another process, with a vector clock timestamp piggybacked, the recipient of the message needs to merge the piggybacked vector clock with its own local vector clock. For two vector clock timestamps

$$V_1 = \langle v_1, v_2, \cdots, v_n \rangle \text{ and } V_2 = \langle v_1', v_2', \cdots, v_n' \rangle$$

merging them yields:

$$U = \langle u_1, u_2, \cdots, u_n \rangle, \text{ where } u_i = \max(v_i, v_i')$$

The encodings of V_1, V_2, and U are:

$$\begin{aligned} Enc(V_1) &= p_1^{v_1} * p_2^{v_2} * \cdots * p_n^{v_n} \\ Enc(V_2) &= p_1^{v_1'} * p_2^{v_2'} * \cdots * p_n^{v_n'} \\ Enc(U) &= \prod_{i=1}^{n} p_i^{\max(v_i, v_i')} \end{aligned}$$

We do not have access to the vector components v_i and v_i' ($i = 1 \ldots n$) to generate $Enc(U)$. Furthermore, it would be better to merge $Enc(V_1)$ and $Enc(V_2)$ into $Enc(U)$ without knowing the n prime numbers. (One advantage of this is protection against attacks.) This can be achieved by observing that

$$Enc(U) = LCM(Enc(V_1), Enc(V_2))$$

So, by computing the LCM of two EVC timestamps, these two timestamps can be merged without knowing the n prime numbers.

Comparison: A mechanism to compare two vector clock timestamps is needed. Let $\mathcal{V}^{\mathcal{E}}$ denote the set of vector timestamps of events. Then $(\mathcal{V}^{\mathcal{E}}, <)$ is isomorphic to $(E, \rightarrow)$ [4, 15]. To compare two distinct vector clock timestamps, a component-wise comparison between the corresponding elements of two vectors is needed. The comparison has two results (the tests $V_1 < V_2$ and $V_2 < V_1$ are symmetrical):

$$\begin{aligned} &i)\ V_1 < V_2 \text{ if } \forall j \in [1, n], V_1[j] \leq V_2[j] \text{ and} \\ &\qquad\qquad \exists j, V_1[j] < V_2[j] \\ &ii)\ V_1 \| V_2 \text{ if } V_1 \not< V_2 \text{ and } V_2 \not< V_1 \end{aligned}$$

Let $\mathcal{ENCV}^{\mathcal{E}}$ denote the set of encoded vector timestamps of events. To compare two (distinct) EVC timestamps, it is only necessary to test if $Enc(V_j) \bmod Enc(V_i) = 0$. Thus,

$$\begin{aligned} &i)\ Enc(V_1) \prec Enc(V_2) \text{ if } Enc(V_1) < Enc(V_2) \text{ and} \\ &\qquad\qquad Enc(V_2) \bmod Enc(V_1) = 0 \\ &ii)\ Enc(V_1) \| Enc(V_2) \text{ if } Enc(V_1) \not\prec Enc(V_2) \text{ and} \\ &\qquad\qquad Enc(V_2) \not\prec Enc(V_1) \end{aligned}$$

The correspondence between the three basic operations of the vector clock and EVC is shown in Table 1. These operations using

Table 1: Correspondence between vector clocks and EVC.

Operation	Vector Clock	Encoded Vector Clock
Representing clock	$V = \langle v_1, v_2, \cdots, v_n \rangle$	$Enc(V) = p_1^{v_1} * p_2^{v_2} * \cdots * p_n^{v_n}$
Local Tick (at process P_i)	$V[i] = V[i] + 1$	$Enc(V) = Enc(V) * p_i$
Merge	Merge V_1 and V_2 yields V where $V[j] = \max(V_1[j], V_2[j])$	Merge $Enc(V_1)$ and $Enc(V_2)$ yields $Enc(V) = LCM(Enc(V_1), Enc(V_2))$
Compare	$V_1 < V_2$: $\forall j \in [1, n]$, $V_1[j] \leq V_2[j]$, and $\exists j$, $V_1[j] < V_2[j]$	$Enc(V_1) \prec Enc(V_2)$: $Enc(V_1) < Enc(V_2)$, and $Enc(V_2) \mod Enc(V_1) = 0$

(1) Initialize $t_i = 1$.
(2) Before an internal event happens at process P_i, $t_i = t_i * p_i$ (local tick).
(3) Before process P_i sends a message, it first executes $t_i = t_i * p_i$ (local tick), then it sends the message piggybacked with t_i.
(4) When process P_i receives a message piggybacked with timestamp s, it executes
$t_i = LCM(s, t_i)$ (merge);
$t_i = t_i * p_i$ (local tick)
before delivering the message.

Figure 2: Operation of EVC t_i at process P_i.

EVC are illustrated in Figure 1 using an example execution over three processes. As our technique encodes vector clocks of events, the encoded vector clock/timestamps of events, $(\mathcal{ENCV}^{\mathcal{E}}, \prec)$ is isomorphic to $(E, \rightarrow)$ and to $(\mathcal{V}^{\mathcal{E}}, <)$.

Thus, the encoded vector clock t_i (initialized to 1) is operated at process P_i as shown in Figure 2. To manipulate the EVC, each process needs to know only its own prime and not the primes of other processes. Merging two EVCs requires computing the LCM, which does not require factorization (see Section 2.2.2).

2.2 Complexity

Comparing with vector clocks, EVC has advantages in time, space, and message size complexity. Each process only needs to store and transmit a single number. If we assume that the local space for storing and transmitting this number is bounded, then the storage cost and message space overhead is only $O(1)$.

In general, we analyze the complexity of vector clocks and EVC assuming bounded storage using the uniform cost model, which is suitable for analysis when the numbers fit into a single machine word. We analyze the complexity of EVC assuming unbounded storage using the logarithmic cost model, which assigns a cost to every machine operation that is a function of the number of bits involved, and is suitable when the numbers are unbounded. The logarithmic cost model is used to compute the bit complexity. For a EVC value H, we use h to denote the number of bits or digits in H. Thus, $h = \log H$.

2.2.1 Local Tick.

- **Unbounded storage:** If we assume the EVC has unbounded storage, the bit complexity of multiplying two numbers of size H is $O(h^2)$ using naive multiplication. This becomes $O(h(\log h)(\log\log h))$ using the Schonhage-Strassen or other modern algorithms. However, for the local tick, the prime number that the EVC value H is being multiplied with is assumed to have bounded storage (fits in one machine word). Hence, the multiplication has bit time complexity $O(h)$.
- **Bounded storage:** If we assume the EVC has bounded storage, the multiplication of EVC with the prime number has $O(1)$ time complexity.

2.2.2 Merge or Computing LCM. To compute $LCM(a, b)$, we have:

$$LCM(a, b) = \frac{a * b}{GCD(a, b)}$$

By applying the Euclidean algorithm, we can compute $GCD(a, b)$ without factoring the two numbers. The time complexity is the number of steps in Euclid's algorithm, multiplied by the computational cost of each step. Let $\hbar$ be number of digits of the smaller number in base 10. (Note that $\hbar$ and h are of the same order.) It is well known that the number of steps required is never more than five times the number $\hbar$ [5, 6, 13].

- **Bounded numbers:** In the uniform cost model, each step of the algorithm takes constant time. The total running time for GCD is $O(\hbar)$; this can be expressed as $O(1)$ because $\hbar$ is bounded by the word size. So we can compute GCD and LCM in $O(1)$ time.
- **Unbounded numbers:** In the logarithmic cost model, it is well known that the overall bit time complexity of the Euclidean algorithm for GCD is $O(h^2)$ [3, 7] by employing the Euclid algorithm together with a classical mod operation. This can be reduced using recursive reduction techniques, and be brought down to $O(M(h)\log h)$, where $M(h)$ is the bit complexity of multiplication of two h-bit integers [3]. The best-known bound for $M(h)$ is $O(h(\log h)(\log\log h))$, as derived from modern transform and convolution techniques based on the Schonhage-Strassen or other algorithms for fast large integer multiplication [3, 17]. Thus, the complexity of the recursive GCD algorithms is:

$$O(h(\log^2 h)(\log\log h))$$

This results in quasilinear algorithms for the GCD and LCM.

Table 2: Comparison of the time complexity of the three basic operations and the space complexity, for vector clock and EVC.

	Vector Clock (bounded storage) (uniform cost model)	Encoded Vector Clock (unbounded storage) (logarithmic cost model)	Encoded Vector Clock (bounded storage) (uniform cost model)
Local Tick	$O(1)$	$O(h)$	$O(1)$
Merge	$O(n)$	$O(h(\log^2 h)(\log\log h))$	$O(1)$
Compare	$O(n)$	$O(h(\log h)(\log\log h))$	$O(1)$
Storage	$O(n)$	$O(h)$	$O(1) + O(d)$ (with resetting)

2.2.3 Compare.

- **Unbounded numbers:** The complexity of a mod operation modulo H is asymptotically the same as a size-H multiply. The time to check $Env(V_2) \mod Enc(V_1) = 0$ is the same time taken to multiply two large numbers of h bits. The best-known bit complexity can be calculated using the Schonhage-Strassen bound, as $O(h(\log h)(\log\log h))$ [3].
- **Bounded numbers:** In the uniform cost model, the time complexity to check $Env(V_2) \mod Enc(V_1) = 0$ is $O(1)$.

2.2.4 Storage.

- **Unbounded numbers:** The storage complexity is $O(h)$.
- **Bounded numbers:** In the uniform cost model, the storage complexity is $O(1)$. The only drawback for assuming a bounded space for storing the numbers is that eventually it will overflow. When overflow happens, we can adapt the vector clock resetting technique [25] which enables us to reuse the smaller numbers. The clock resetting algorithm will incur an $O(l)$ message count complexity and an $O(d)$ storage cost at each process. The resetting technique is outlined in Section 4.3.

In Table 2, we compare the time complexity of the three basic operations (local tick, merge, compare), and the storage cost, for vector clock and EVC. Note that the logarithmic cost model for computing the complexities for the unbounded EVC storage case is different from the uniform cost model used to compute the complexities for the bounded EVC storage case and for vector clocks.

2.3 Resilience to Churn

Churn refers to the dynamic joining and departure of processes. EVCs (and the operations on them) can operate correctly without any change and without any overhead in the face of churn, because the prime number of each process is independent of the others, and the EVC timestamp of an event also encodes its causal history into a single number. Optimizations, such as reducing the EVC values by a factor corresponding to the component of the departed process, require an engineered solution.

In comparison, vector clocks can also handle churn but may require some adaptation of the basic protocol and/or the operations and incur a corresponding overhead. In the simple approach, when a process joins, the vector size is increased and when a process departs, the vector size is not reduced [4]. Approaches that reduce the vector size when a process departs incur a change to the protocol [18, 23].

3 EVC TIMESTAMPS OF CUTS

3.1 Cuts

A cut is a prefix of the execution $(E, \rightarrow)$ and the state after the events of a cut represents a global state [1]. A downward closed prefix of $(E, \rightarrow)$ represents a consistent global state, and is a meaningful observable state of the execution [1]. The set of consistent cuts $CCuts$ forms a lattice $(CCuts, \subset)$ under the set inclusion relation [15]. Vector timestamps are assigned to cuts in order to reason with cuts [8, 15].

Let $\downarrow e = \{f \mid f \rightarrow e \ \bigwedge f \in E\} \cup \{e\}$ denote the causal history of event e. $\downarrow e$ is a consistent cut. In general, the union of the causal histories of any subset X of events is a consistent cut. Thus, $cut(X) = \bigcup_{e_i \in X} \downarrow e_i$ is consistent even if the events in X form a cut that is not consistent.

The surface of a cut $S(cut)$ is the set that contains the last event of the cut cut at each process. Formally, $S(cut) = \{e_i^k \mid e_i^k \in cut \bigwedge e_i^{k+1} \notin cut\}$. For a cut cut, we define $\widehat{cut} = \bigcup_{e_i \in S(cut)} \downarrow e_i$ to be the smallest consistent cut that is larger than or equal to the cut cut. If cut is consistent, then $cut = \widehat{cut}$, whereas if cut is not consistent, then $cut \subset \widehat{cut}$.

3.2 EVC Timestamp of a Cut

Vector timestamp of a cut cut, $V(cut)$, is defined as

$$\begin{aligned} \forall k \in [1, n], V(cut)[k] &= V(e_k)[k], \text{ for } e_k \in S(\widehat{cut}) \\ &= \max_{e_i \in S(cut)} V(e_i)[k] \end{aligned}$$

Let event $e_i \in S(cut)$ occur at process P_i and let the vector timestamp of e_i, $V(e_i) = \langle v_1^i, v_2^i, \cdots v_n^i \rangle$. Likewise, let event $\hat{e}_i \in S(\widehat{cut})$ occur at process P_i and let the vector timestamp of $\hat{e}_i, V(\hat{e}_i) = \langle \hat{v}_1^i, \hat{v}_2^i, \cdots \hat{v}_n^i \rangle$. We can then observe that

$$\begin{aligned} Enc(V(cut)) &= \prod_{i=1}^{n} p_i^{\hat{v}_i^i} \\ &= \prod_{i=1}^{n} p_i^{\max(v_i^1, v_i^2, \cdots, v_i^n)} \end{aligned}$$

To compute $Enc(V(cut))$, we do not have access to the individual components of vector timestamps of events in $S(cut)$. Moreover, it would be better to combine $Enc(V(e_1))$, $Enc(V(e_2))$, ..., $Enc(V(e_n))$ into $Enc(V(cut))$ without knowing the n prime numbers. This can be achieved by observing that

$$Enc(V(cut)) = LCM(Enc(V(e_1)), Enc(V(e_2)), \cdots, Enc(V(e_n))).$$

Table 3: Correspondence between operations on cuts using vector clocks and EVC.

Operation	Vector Clock	Encoded Vector Clock
Cut	$\forall k \in [1,n], V(cut)[k] = \max_{e_i \in S(cut)} V(e_i)[k]$ (cut may not be consistent) $\forall k \in [1,n], V(cut)[k] = V(e_k)[k]$ for $e_k \in S(cut)$ (cut is consistent)	$Enc(V(cut)) = LCM(Enc(V(e_1)), Enc(V(e_2)), \cdots, Enc(V(e_n)))$, where $e_i \in S(cut)$
Common past	$\forall k \in [1,n], V(CP(cut))[k] = \min_{e_i \in S(cut)} V(e_i)[k]$	$Enc(V(cut)) = GCD(Enc(V(e_1)), Enc(V(e_2)), \cdots, Enc(V(e_n)))$, where $e_i \in S(cut)$
Intersection	If $V(cut1)[j] = v_j$ and $V(cut2)[j] = v'_j$, $V(cut1 \bigcap cut2)[j] = \min(v_j, v'_j)$	Merge $Enc(V(cut1))$ and $Enc(V(cut2))$ yields $Enc(V) = GCD(Enc(V(cut1)), Enc(V(cut2)))$
Union	$V(cut1 \bigcup cut2)[j] = \max(v_j, v'_j)$	$Enc(V) = LCM(Enc(V(cut1)), Enc(V(cut2)))$
Compare	$V(cut1) < V(cut2)$: $\forall j \in [1,n], V(cut1)[j] \leq V(cut2)[j]$, and $\exists j, V(cut1)[j] < V(cut2)[j]$	$Enc(V(cut1)) \prec Enc(V(cut2))$: $Enc(V(cut1)) < Enc(V(cut2))$, and $Enc(V(cut2)) \mod Enc(V(cut1)) = 0$

So, by computing the LCM of n EVC timestamps of events, the encoded timestamp of the consistent cut can be computed without knowing the n prime numbers. The LCM of n numbers can be computed iteratively, and its complexity is $n-1$ times the complexity of a single LCM. By extending the results of Section 2.2, the time complexity of computing the LCM of n EVC timestamps is $O(n \times h(\log^2 h)(\log\log h))$ assuming unbounded storage for EVCs and $O(n)$ assuming bounded storage for EVCs.

Example 1: For Cut A in Figure 1, for events $e_i \in S(CutA)$, $V(e_1) = [2,0,1]$, $V(e_2) = [1,3,0]$, and $V(e_3) = [0,0,1]$. We have $V(CutA) = [2,3,1]$.

- Using prime numbers, $Enc(V(CutA)) = 2^{\max(2,1,0)} \times 3^{\max(0,3,0)} \times 5^{\max(1,0,1)} = 4 \times 27 \times 5 = 540$.
- We have $Enc(V(e_1)) = 20$, $Enc(V(e_2)) = 54$, and $Enc(V(e_3)) = 5$. Without using prime numbers, $Enc(V(CutA)) = LCM(Enc(V(e_1)), Enc(V(e_2)), Enc(V(e_3))) = LCM(20, 54, 5) = 540$.

Thus, $Enc(V(CutA))$ is the same value with and without using the prime numbers.

3.3 EVC Timestamp of Cut Representing Common Past

For a cut cut, we can define its common-past $CP(cut)$ to be the execution prefix such that the prefix is in the causal history of every element in $S(cut)$. $CP(cut) = \bigcap_{e_i \in S(cut)} \downarrow e_i$ [8]. The common-past of a cut is useful for discarding obsolete information in distributed databases, checkpointing, and designing protocols for the replicated log and replicated dictionary problems [10, 24]. We define the vector timestamp of $CP(cut)$, $V(CP(cut))$, as

$$\forall k \in [1,n], V(CP(cut))[k] = \min_{e_i \in S(cut)} V(e_i)[k]$$

As before, let event $e_i \in S(cut)$ occur at process P_i and let the vector timestamp of e_i, $V(e_i) = \langle v_1^i, v_2^i, \cdots v_n^i \rangle$. We can then observe that

$$Enc(V(CP(cut))) = \prod_{i=1}^{n} p_i^{\min(v_i^1, v_i^2, \cdots, v_i^n)}$$

To compute $Enc(V(CP(cut)))$, we do not have access to the individual components of vector timestamps of events in $S(cut)$. Moreover, it would be better to combine $Enc(V(e_1))$, $Enc(V(e_2))$, ... $Enc(V(e_n))$ into $Enc(V(CP(cut)))$ without knowing the n prime numbers. This can be achieved by observing that

$$Enc(V(CP(cut))) = GCD(Enc(V(e_1)), Enc(V(e_2)), \cdots, Enc(V(e_n))).$$

So, by computing the GCD of n EVC timestamps of events, the encoded timestamp of the consistent cut can be computed without knowing the n prime numbers. The GCD of n numbers can be computed iteratively, and its complexity is $n-1$ times the complexity of a single GCD. By extending the results of Section 2.2, the time complexity of computing the GCD of n EVC timestamps is $O(n \times h(\log^2 h)(\log\log h))$ assuming unbounded storage for EVCs and $O(n)$ assuming bounded storage for EVCs.

For unbounded storage, an alternate bound for $GCD(a_1, \ldots, a_n)$, based on the Euclidean algorithm, also computes the GCD iteratively but counts the total number of division operations. The derivation leverages the fact that after each GCD calculation, the GCD reduces by a factor of at least two, (else if it remains the same, only one division is used). There are $\log_2 a_1$ (rather than n) terms in the series $\sum_{k=1}^{\log_2 a_1} \log a_{i_k}$. The total number of division operations is less than $O((\log a_1)(\log a_n))$ or simply $O(h^2)$. As each division costs $O(h(\log h)(\log\log h))$, this bound is better if

$$O(h^3(\log h)(\log\log h)) < O(nh(\log^2 h)(\log\log h))$$

which may not be true for large numbers a_i.

Example 2: For Cut B in Figure 1, for events $e_i \in S(CutB)$, we have $V(e_1) = [3,0,1]$, $V(e_2) = [3,4,1]$, and $V(e_3) = [1,3,2]$. $V(CutB) = [3,4,2]$, whereas we have $V(CP(CutB)) = [1,0,1]$.

- Using prime numbers, $Enc(V(CP(CutB))) = 2^{\min(3,3,1)} \times 3^{\min(0,4,3)} \times 5^{\min(1,1,2)} = 2 \times 1 \times 5 = 10$.
- We have $Enc(V(e_1)) = 40$, $Enc(V(e_2)) = 3240$, and $Enc(V(e_3)) = 1350$. Without using prime numbers, $Enc(V(CP(CutB))) = GCD(Enc(V(e_1)), Enc(V(e_2)), Enc(V(e_3))) = GCD(40, 3240, 1350) = 10$.

Thus, $Enc(V(CP(CutB)))$ is the same value with and without using the prime numbers.

Table 4: Comparison of the time complexity of the operations on cuts using vector clocks and EVC.

	Vector Clock (bounded storage) (uniform cost model)	Encoded Vector Clock (unbounded storage) (logarithmic cost model)	Encoded Vector Clock (bounded storage) (uniform cost model)
Computing timestamp	$O(n^2)$ (*cut* may not be consistent) $O(n)$ (*cut* is consistent)	$O(nh(\log^2 h)(\log\log h))$	$O(n)$
Computing common past	$O(n^2)$	$O(nh(\log^2 h)(\log\log h))$	$O(n)$
Intersection and union	$O(n)$	$O(h(\log^2 h)(\log\log h))$	$O(1)$
Compare	$O(n)$	$O(h(\log h)(\log\log h))$	$O(1)$

Matrix clocks, first defined by Wuu and Bernstein [24], use a $n \times n$ matrix M of clock values, where the $M[j,k]$th entry at P_i denotes P_i's knowledge of P_j's knowledge of the latest local clock value at P_k. Note that $M[j]$, the jth row of the matrix timestamp of an event e, corresponds to the vector timestamp of the event at P_j in the surface of the cut $\downarrow e$, denoted by $V((S(\downarrow e))_j)$, and this can be encoded by EVC as shown above. Thus, the matrix clock can be encoded as a vector of length n of EVCs. The common past of events $(S(\downarrow e))_j$, for all j, identifies the execution prefix that is known to all processes and thus can be discarded from the local log at event e.

Example 3: For the event e with EVC = 3240 in Figure 1, $Enc(M(e))$ = [40, 3240, 5]. We have $V(\downarrow e) = [3,4,1]$, $V((S(\downarrow e))_1) = [3,0,1]$, $V((S(\downarrow e))_2) = [3,4,1]$, $V((S(\downarrow e))_3) = [0,0,1]$, whereas we have $V(CP(\downarrow e)) = [0,0,1]$. By applying a logic similar to example 2, it follows that:

- Using prime numbers,
  $Enc(V(CP(\downarrow e))) = 2^{\min(3,3,0)} \times 3^{\min(0,4,0)} \times 5^{\min(1,1,1)} = 1 \times 1 \times 5 = 5$.
- $Enc(V((S(\downarrow e))_1)) = 40, Enc(V((S(\downarrow e))_2)) = 3240, Enc(V((S(\downarrow e))_3)) = 5$, Without using prime numbers,
  $Enc(V(CP(\downarrow e))) = GCD(40, 3240, 5) = 5$.

The EVC of the execution prefix that can be safely discarded is 5.

3.4 Other Operations on Cuts

Intersection and Union: For two vector clock timestamps of (consistent) cuts $cut1$ and $cut2$, let

$$V(cut1) = \langle v_1, v_2, \cdots, v_n \rangle \text{ and } V(cut2) = \langle v'_1, v'_2, \cdots, v'_n \rangle$$

We have that

$$V(cut1 \bigcap cut2) = \langle u_1, u_2, \cdots, u_n \rangle, \text{ where } u_i = \min(v_i, v'_i)$$

$$V(cut1 \bigcup cut2) = \langle u_1, u_2, \cdots, u_n \rangle, \text{ where } u_i = \max(v_i, v'_i)$$

The encodings of $V(cut1)$, $V(cut2)$, $V(cut1 \bigcap cut2)$, and $V(cut1 \bigcup cut2)$ are:

$$\begin{aligned} Enc(V(cut1)) &= p_1^{v_1} * p_2^{v_2} * \cdots * p_n^{v_n} \\ Enc(V(cut2)) &= p_1^{v'_1} * p_2^{v'_2} * \cdots * p_n^{v'_n} \\ Enc(V(cut1 \bigcap cut2)) &= \prod_{i=1}^{n} p_i^{\min(v_i, v'_i)} \\ Enc(V(cut1 \bigcup cut2)) &= \prod_{i=1}^{n} p_i^{\max(v_i, v'_i)} \end{aligned}$$

To compute the encodings of the vector timestamps of the intersection and union cuts, we do not have access to the individual components of the vector timestamps of $cut1$ and $cut2$. Moreover, it would be better to compute $Enc(V(cut1 \bigcap cut2))$ and $Enc(V(cut1 \bigcup cut2))$ without knowing the n prime numbers. This can be achieved by observing that

$$\begin{aligned} Enc(V(cut1 \bigcap cut2)) &= GCD(Enc(V(cut1)), Enc(V(cut2))) \\ Enc(V(cut1 \bigcup cut2)) &= LCM(Enc(V(cut1)), Enc(V(cut2))) \end{aligned}$$

So, by computing the GCD and the LCM of two EVC timestamps of cuts, the EVC timestamps of the intersection and the union cuts, respectively, can be computed without knowing the n prime numbers. The time complexity is $O(1)$, namely the cost of a single GCD or LCM operation, assuming bounded storage, or $O(h(\log^2 h)(\log\log h))$ assuming unbounded storage to represent the EVCs. This assumes the encoded vector timestamps of $cut1$ and $cut2$ are available. These should be available since we are operating in the EVC domain.

Example 4: Consider the intersection and union of Cut A and Cut C shown in Figure 1. Using vector timestamps of cuts and prime numbers, we have:

- $V(CutA) = [2,3,1]; V(CutC) = [1,3,2]$.
- $V(CutA \bigcap CutC) = [1,3,1]; V(CutA \bigcup CutC) = [2,3,2]$
- $Enc(V(CutA \bigcap CutC)) = 2^1 * 3^3 * 5^1 = 270$;
  $Enc(V(CutA \bigcup CutC)) = 2^2 * 3^3 * 5^2 = 2700$

Using the encodings of the vector timestamps of Cut A and Cut C and without using prime numbers, we have by using the expression from Section 3.2, $Enc(V(CutA)) = LCM(20, 54, 5) = 540$ and $Enc(V(CutC)) = LCM(2, 54, 1350) = 1350$. We then have:

- $Enc(V(CutA \bigcap CutC)) = GCD(Enc(V(CutA)), Enc(V(CutC)))$
  $= GCD(540, 1350) = 270$.
- $Enc(V(CutA \bigcup CutC)) = LCM(Enc(V(CutA)), Enc(V(CutC)))$
  $= LCM(540, 1350) = 2700$.

Comparison: The comparison of two distinct consistent cuts $cut1$ and $cut2$ in $CCuts$ results in one of two outcomes: i) $cut1 \subset cut2$ (or symmetrically, $cut2 \subset cut1$), or ii) $cut1 \not\subset cut2$ and $cut2 \not\subset cut1$, i.e., $cut1 \| cut2$. To compare two EVC timestamps of cuts $cut1$ and $cut2$, it is only necessary to test if $Enc(V(cut2))$ mod $Enc(V(cut1)) = 0$. Thus,

i) $Enc(V(cut1)) \prec Enc(V(cut2))$ if $Enc(V(cut1)) < Enc(V(cut2))$ and $Enc(V(cut2))$ mod $Enc(V(cut1)) = 0$

ii) $Enc(V(cut1)) \| Enc(V(cut2))$ if $Enc(V(cut1)) \not\prec Enc(V(cut2))$ and $Enc(V(cut2)) \not\prec Enc(V(cut1))$

For unbounded storage, the time to check $Env(V(cut2))$ mod - $Enc(V(cut1)) = 0$ is asymptotically the same as the time taken to multiply two large numbers of h bits. The best-known bit complexity is based on the Schonhage-Strassen bound, as $O(h(\log h)(\log\log h))$. For bounded numbers, the time complexity to check $Env(V(cut2))$ mod $Enc(V(cut1)) = 0$ is $O(1)$.

The encoded vector clock timestamps of consistent cuts, denoted $(\mathcal{ENCV}^{CC}, \prec)$, is isomorphic to $(\mathcal{V}^{CC}, <)$, the vector clock timestamps of consistent cuts, and to $(CCuts, \subset)$.

Table 3 gives the correspondence between the operations on cuts using vector clocks and using EVC. In Table 4, we compare the time complexities between the operations on cuts using vector timestamps and using EVCs.

4 SCALABILITY

For n processes in the system and f_i events at each process P_i, the maximum EVC timestamp across all processes is $O(\prod_{i=1}^{n} p_i^{f_i})$. From this observation, we can see that EVC timestamps grow very fast and overflow is unavoidable. We can use several strategies to alleviate this problem.

4.1 Relevant Events

It suffices if the local clock does not tick at every event but only at events that are relevant to the application. Thus, the EVC does not grow so fast. This strategy is explained in the context of predicate detection [20]. The local clock should tick only when the variables in the predicate alter the truth value of the predicate.

We also note that on social platforms such as Twitter and Facebook, the maximum length of any chain of messages is usually small, after which that chain of posts dies out.

4.2 Detection Regions

In large-scale systems, the application requiring a vector clock may be confined to only a subset of m processes, where $m < n$. An example of this is locality-aware predicate detection [20]. The subset of m processes forms a detection region. Processes within the detection region maintain a single number for the EVC. More importantly, for processes outside the detection region, we can cut down the storage cost and make the solution more practical for large-scale systems. For a process P_j outside the region, when it first receives a message piggybacked with an EVC timestamp, it simply stores this single number. Although P_j will not tick the EVC locally since there is no corresponding component in the vector clock for P_j, it may still receive multiple messages. Each time this happens, P_j simply executes the merge operation by calculating the LCM of two numbers. (P_j needs to store the EVC and to do the merge because it may later send messages back into the detection region, directly or transitively.)

4.3 Resetting EVC

We can adapt the clock resetting technique [25] to solve the problem when the clock overflows. This technique divides the execution of a distributed system into multiple phases. Each time the clock overflows at any process, the resetting algorithm terminates the current phase by sending control messages while ensuring there is no computation message sending from the current phase to the next phase, nor from the next phase to the current phase. The reset protocol involves a period of send inhibition of messages, and the local clock gets reset in a strongly consistent (i.e., transitless) global state [9].

(1) Initialize $t_i = 0$.
(2) Before an internal event happens at process P_i, $t_i = t_i + \log(p_i)$ (local tick).
(3) Before process P_i sends a message, it first executes $t_i = t_i + \log(p_i)$ (local tick), then it sends the message piggybacked with t_i.
(4) When process P_i receives a message piggybacked with timestamp s, it executes
$t_i = s + t_i - \log(GCD(\log^{-1}(s), \log^{-1}(t_i)))$ (merge);
$t_i = t_i + \log(p_i)$ (local tick)
before delivering the message.

Figure 3: Operation of the logarithm of the EVC, t_i, at process P_i.

4.4 Using logarithms of EVC

As the EVC technique uses exponentiation, we propose the use of logarithms to store and transmit the EVCs. This can result in a reduction in the size of EVCs. Before proceeding with this approach, we note that since logarithms involve finite-precision arithmetic, their use is subject to the introduction of errors due to the limited precision. The use of logarithms requires a careful analysis of the errors introduced. Yet, we outline this approach due to its promise.

The logarithms can be taken to any base greater than 1. Henceforth, we omit mention of the base. The three operations of the EVC, using the representation of logarithms, are as follows.

Local Tick: At a process P_i, the local tick updates $\log(Enc(V))$ as follows.

$$\log(Enc(V)) = \log(Enc(V)) + \log(p_i)$$

Merge: Let vector U denote the merge of vectors V_1 and V_2, and let x and y denote the logarithms of $Enc(V_1)$ and $Enc(V_2)$, respectively. Then,

$$\begin{aligned}\log(Enc(U)) &= \log(LCM(Enc(V_1), Enc(V_2)))\\ &= \log(Enc(V_1)) + \log(Enc(V_2))\\ &\quad - \log(GCD(Enc(V_1), Enc(V_2)))\\ &= x + y - \log(GCD(\log^{-1}(x), \log^{-1}(y)))\end{aligned}$$

Compare: Let x and y denote the logarithms of $Enc(V_1)$ and $Enc(V_2)$, respectively, and let N denote the set of natural numbers.

$$i)\ x \prec y \text{ if } x < y \text{ and } \log^{-1}(\log(\frac{Enc(V_2)}{Enc(V_1)})) \in N$$
$$\text{if } x < y \text{ and } \log^{-1}(y - x) \in N$$
$$ii)\ x \| y \text{ if } x \not\prec y \text{ and } y \not\prec x$$

The local tick is implemented by a single addition. Unfortunately, the merge and the comparison operations require taking the antilogarithms, which may require a large amount of scratch space. The bit time complexity of taking the log and anti-log using the arithmetic-geometric mean iteration method is $O(M(r) \log r)$, where r refers to the number of digits of precision and $M(r)$ is the complexity of the multiplication module. For the merge operation, this cost can be assumed to be less than or of the same order as that of computing the GCD. The test for the compare operation is subject to errors due to the finite precision used in representing logarithms. The amount of errors can be reduced if for each tick operation, a conversion from the log domain to the integer domain, followed by multiplication in the integer domain and then a conversion back to the log domain is performed.

In summary, the logarithm of the encoded vector clock, t_i (initialized to 0) at process P_i, is operated as shown in Figure 3. The advantage is that the storage of the EVC and transmission of the piggybacked timestamp can be done using a single (smaller) number, and the extra space required is only scratch space.

5 CONCLUSIONS

We proposed the encoding of the vector clock using prime numbers, to use a single number to represent vector time. We gave the operations on the EVC. We also showed how to timestamp global states using EVC, and various operations on these global states using EVC. To manipulate the EVC, every process only needs to know its own prime, and not the primes of other processes. Further, to compute the equivalent of the maximum of two vector clocks, a process needs to find the largest common multiple of their EVCs, which does not require factorization. In addition to the obvious advantage of saving space, the time complexity of most operations using EVC (with bounded storage) is lower than that using traditional vector clocks. The one exception is timestamping a consistent global state; however, cut operations are not performed frequently.

A drawback of EVCs is that they grow very fast and overflow soon occurs. Therefore, we examined scalability approaches for the EVC to deal with this. These included ticking the clock only at application-relevant events and only at processes where such events occur, and resetting the EVC throughout the system when it overflows at some process. We also proposed the use of logarithms of EVCs to store and transmit EVCs to cut down the overhead.

The approach of using logarithms of EVCs appears promising although an analysis of errors introduced due to finite-precision logarithms is required before reaching any conclusions. Another direction for further work is to examine how quickly the EVC grows in practice for real applications or via simulations. Further, by examining the length of the causal chain of relevant events/messages in social platforms (e.g., Twitter and Facebook) where the number of users is potentially large, one could discover firm evidence as to whether or not EVCs are advantageous on social platforms.

ACKNOWLEDGMENTS

The authors would like to thank Bhaskar Dasgupta for suggesting the alternate bound on computing the GCD of n numbers in the unbounded storage case.

REFERENCES

[1] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75. https://doi.org/10.1145/214451.214456

[2] Bernadette Charron-Bost. 1991. Concerning the Size of Logical Clocks in Distributed Systems. *Inf. Process. Lett.* 39, 1 (1991), 11–16. https://doi.org/10.1016/0020-0190(91)90055-M

[3] Richard Crandall and Carl Pomerance. 2001. *Prime Numbers: A Computational Perspective (1st Ed.).* Springer, New York, NY, USA.

[4] Colin J. Fidge. 1991. Logical Time in Distributed Computing Systems. *IEEE Computer* 24, 8 (1991), 28–33. https://doi.org/10.1109/2.84874

[5] H. Grossman. 1924. On the Number of Divisions in Finding a G.C.D. *The American Mathematical Monthly* 31, 9 (1924), 443.

[6] Ross Honsberger. 1976. *Mathematical Gems II.* The Mathematical Association of America.

[7] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[8] Ajay D. Kshemkalyani. 1998. Causality and Atomicity in Distributed Computations. *Distributed Computing* 11, 4 (1998), 169–189. https://doi.org/10.1007/s004460050048

[9] Ajay D. Kshemkalyani. 1998. A Framework for Viewing Atomic Events in Distributed Computations. *Theor. Comput. Sci.* 196, 1-2 (1998), 45–70. https://doi.org/10.1016/S0304-3975(97)00195-3

[10] Ajay D. Kshemkalyani. 2004. The Power of Logical Clock Abstractions. *Distributed Computing* 17, 2 (2004), 131–150. https://doi.org/10.1007/s00446-003-0105-9

[11] Ajay D. Kshemkalyani and Mukesh Singhal. 2011. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press.

[12] Sandeep S. Kulkarni and Nitin H. Vaidya. 2017. Effectiveness of Delaying Timestamp Computation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017.* 263–272. https://doi.org/10.1145/3087801.3087818

[13] Gabriel Lame. 1844. Note sur la limite du nombre des divisions dans la recherche du plus grand commun diviseur entre deux nombres entiers. *Comptes Rendus Acad. Sci.* 19 (1844), 867–870.

[14] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM 21, 7* (1978), 558–565. https://doi.org/10.1145/359545.359563

[15] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. *Proceedings of the Parallel and Distributed Algorithms Conference* (1988), 215–226.

[16] Sigurd Meldal, Sriram Sankar, and James Vera. 1991. Exploiting Locality in Maintaining Potential Causality. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing (PODC '91).* ACM, New York, NY, USA, 231–239. https://doi.org/10.1145/112600.112620

[17] Niels Moller. 2008. On Schonhage's algorithm and subquadratic integer gcd computation. *Math. Comp.* 77, 261 (2008), 589–607.

[18] Golden G. Richard III. 1998. Efficient Vector Time with Dynamic Process Creation and Termination. *J. Parallel Distrib. Comput.* 55, 1 (1998), 109–120. https://doi.org/10.1006/jpdc.1998.1493

[19] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing* 7, 3 (1994), 149–174. https://doi.org/10.1007/BF02277859

[20] Min Shen, Ajay D. Kshemkalyani, and Ashfaq A. Khokhar. 2013. Detecting Unstable Conjunctive Locality-Aware Predicates in Large-Scale Systems. In *IEEE 12th International Symposium on Parallel and Distributed Computing, ISPDC 2013, Bucharest, Romania, June 27-30, 2013.* 127–134. https://doi.org/10.1109/ISPDC.2013.25

[21] Mukesh Singhal and Ajay D. Kshemkalyani. 1992. An Efficient Implementation of Vector Clocks. *Inf. Process. Lett.* 43, 1 (1992), 47–52. https://doi.org/10.1016/0020-0190(92)90028-T

[22] Francisco J. Torres-Rojas and Mustaque Ahamad. 1999. Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. *Distributed Computing* 12, 4 (1999), 179–195. https://doi.org/10.1007/s004460050065

[23] Xinli Wang, Jean Mayo, Wei Gao, and James Slusser. 2006. An Efficient Implementation of Vector Clocks in Dynamic Systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2.* 593–599.

[24] Gene T.J. Wuu and Arthur J. Bernstein. 1984. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing (PODC '84).* ACM, New York, NY, USA, 233–242. https://doi.org/10.1145/800222.806750

[25] Li-Hsing Yen and Ting-Lu Huang. 1997. Resetting Vector Clocks in Distributed Systems. *J. Parallel Distrib. Comput.* 43, 1 (1997), 15–20. https://doi.org/10.1006/jpdc.1997.1330