On the Growth of the Prime Numbers Based Encoded Vector Clock

Ajay D. Kshemkalyani   Bhargav Voleti

University of Illinois at Chicago

ajay@uic.edu

Overview

Introduction

- Scalar clocks: $e \rightarrow f \Rightarrow C(e) < C(f)$
- Vector clocks: $e \rightarrow f \iff V(e) < V(f)$
  - Fundamental tool to characterize causality
  - To capture the partial order $(E, \rightarrow)$, size of vector clock is the dimension of the partial order, bounded by the size of the system, n
  - Not scalable!
- encoding of vector clocks (EVC) using prime numbers to use a single number to represent vector time
  - big integer EVC grows fast, eventually exceeds size of vector clock

Contribution

Evaluate and analyze the growth rate of EVC using simulations

Vector Clock Operation at a Process P_i

1. Initialize V to the 0-vector.
2. Before an internal event happens at process P_i, $V[i] = V[i] + 1$ (local tick).
3. 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.
4. 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.

Encoded Vector Clock (EVC) and Operations

- A vector clock $V = \langle v_1, v_2, \cdots, v_n \rangle$ can be encoded by n distinct prime numbers, $p_1, p_2, \cdots, p_n$ as:

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

- EVC operations: Tick, Merge, Compare
- **Tick** at P_i: $Enc(V) = Enc(V) * p_i$

EVC Operations (contd.)

- **Merge:** For $V_1 = \langle v_1, v_2, \cdots, v_n \rangle$ and $V_2 = \langle v'_1, v'_2, \cdots, v'_n \rangle$, merging 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:

$$\begin{aligned} Enc(V_1) &= p_1^{v_1} * p_2^{v_2} * \cdots * p_n^{v_n} \\ Enc(V_2) &= p_1^{v'_1} * p_2^{v'_2} * \cdots * p_n^{v'_n} \\ Enc(U) &= \prod_{i=1}^{n} p_i^{\max(v_i, v'_i)} \end{aligned}$$

However,

$$Enc(U) = LCM(Enc(V_1), Enc(V_2)) = \frac{Enc(V_1) * Enc(V_2)}{GCD(Enc(V_1), Enc(V_2))}$$

EVC Operations (contd.)

- **Compare:**

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

Thus, to manipulate the EVC,

- Each process needs to know only its own prime
- Merging EVCs requires computing LCM
  - Use Euclid's algorithm for GCD, which does not require factorization

Correspondence of Operations

Table: 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 $Enc(V_2) \mod Enc(V_1) = 0$

Operation of the Encoded Vector Clock

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: Operation of EVC t_i at process P_i.

Illustration of Using EVC

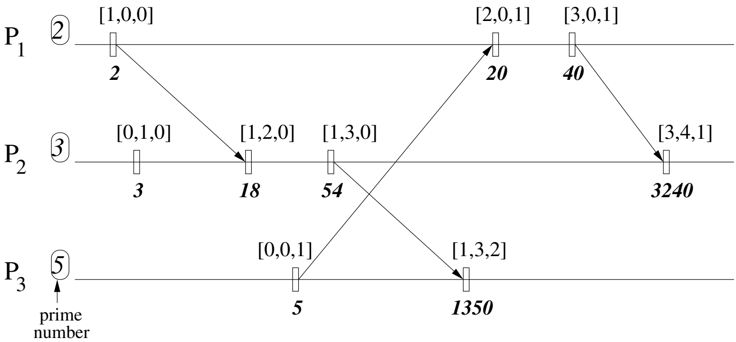

Figure: The vector timestamps and EVC timestamps are shown above and below each timeline, respectively. In real scenarios, only the EVC is stored and transmitted.

Simulation Assumptions

- Simulated distributed executions with a random communication pattern
- pr_s: probability of a send (versus internal) event
- Used first n primes for the n processes
- *Overflow process*: that process which is earliest to have its EVC size exceed $32n$ bits

Number of Events until EVC Size exceeds 32n bits

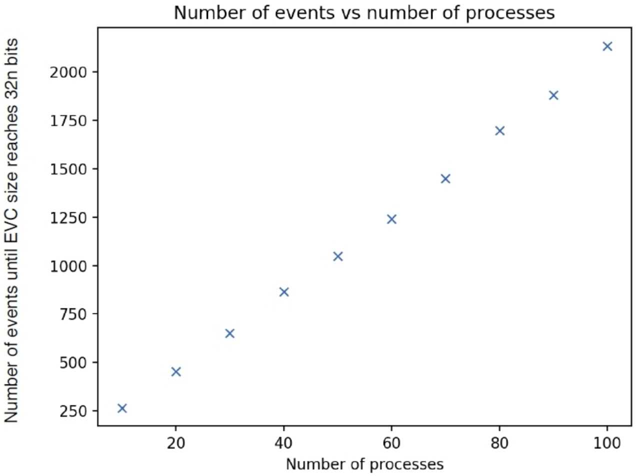

Figure: Average of 10 runs. $pr_s = 0.6$.

- Typically, 21-25 events/process before EVC size exceeded 32n

Strawman Analysis

- $pr_s = 0.6$ implies every third event is a receive event.
- Consider $n = 60$. 60 lowest prime numbers needs 8 bit representation.
- At each event, EVC size increases by 8 bits (local tick)
- At a receive event, (every 3rd event), size of EVC doubles due to LCM
- Worst-case progression of size of EVC in bits approx. as:

$$8, 16, 32 \text{ and } 40 \text{ (event } e_i^3), 48, 56, 112 \text{ and } 120 \text{ (event } e_i^6),$$
$$128, 136, 272 \text{ and } 280 \text{ (event } e_i^9), 288, 296, 592 \text{ and } 600 \text{ (event } e_i^{12}),$$
$$608, 616, 1232 \text{ and } 1240 \text{ (event } e_i^{15}), 1248, 1256, 2512 \text{ and } 2520 \text{ (event } e_i^{18})$$

- At the 18th event at P_i, the EVC size exceeds $60 \times 32 = 1920$ bits
- As per simulation, overflow happens at the 1250/60th event, or the 21st event, at the overflow process

Size of EVC as a Function of Number of Events

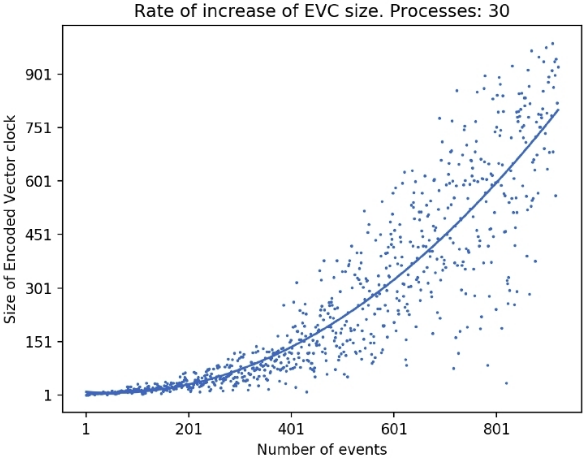

Figure: pr_s= 0.5.

- About 900 events until EVC size reached 960 (= 30 × 32) bits at overflow process

Size of EVC as a Function of Number of Events

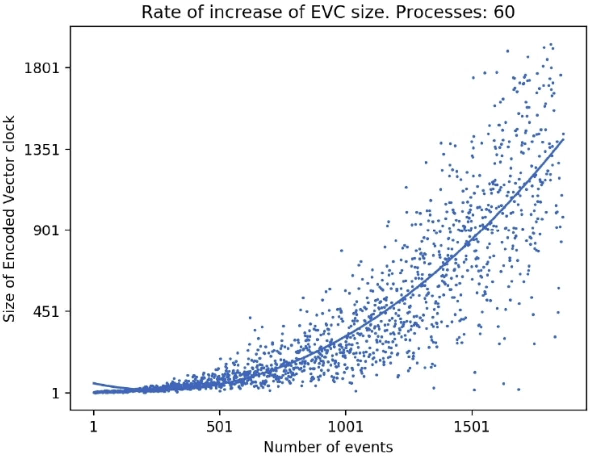

Figure: $pr_s = 0.5$.

- About 1800 events until EVC size reached 1920 ($= 60 \times 32$) bits at overflow process

Size of EVC as a Function of Number of Events

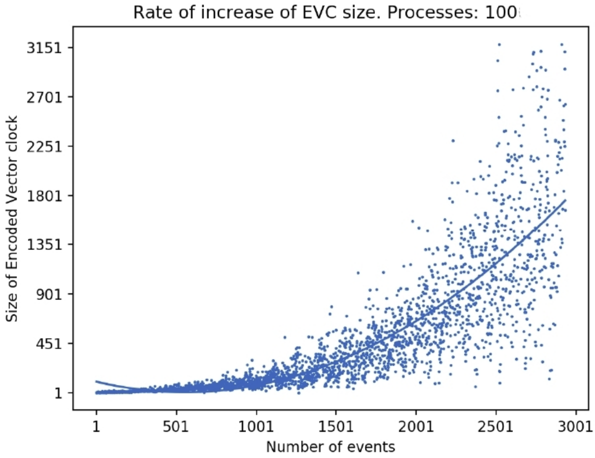

Figure: $pr_s = 0.5$.

- About 3000 events until EVC size reached 3200 ($= 100 \times 32$) bits at overflow process

Number of Events until Overflow 32n bits (function of Event Types)

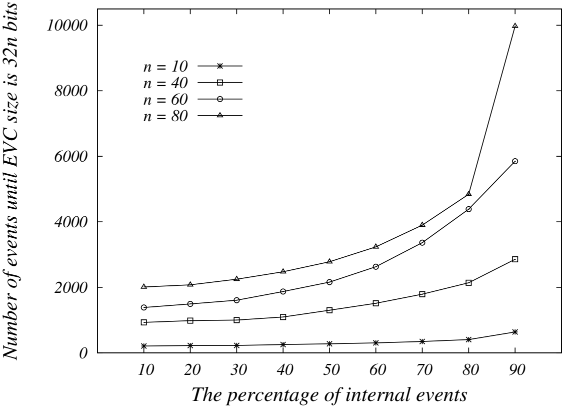

Figure: Varied percentage of internal events (out of internal, send, and receive events)

- Receive events cause EVC to grow very fast due to LCM

Strawman Analysis

- Consider $n = 60$, and $prob(int.\ event) = 0.9$, $prob(receive\ event) = 0.05$
- For $n = 60$, 60 lowest prime numbers needs 8 bit representation.
- At each event, EVC size increases by 8 bits (local tick)
- At a receive event, (every 20th event), size of EVC doubles due to LCM
- Worst-case progression of size of EVC in bits approx. as:

$$8, \cdots 152, 304 \text{ and } 312 \text{ (event } e_i^{20}\text{)},$$
$$320, \cdots 464, 928 \text{ and } 936 \text{ (event } e_i^{40}\text{)},$$
$$944, \cdots 1088, 2176 \text{ and } 2184 \text{ (event } e_i^{60}\text{)}$$

- At the 60th event at P_i, or 3600th event in execution, the EVC size exceeds $60 \times 32 = 1920$ bits
- As per simulation, overflow happens at the 6000th event. Apply correction:
    - In the initial window before steady state, more than 20 non-receive events per receive event

Scalability of EVCs

EVC timestamps grow very fast. To alleviate this problem:

1. Tick only at relevant events, e.g., when the variables alter the truth value of a predicate
   - On social platforms, e.g., Twitter and Facebook, max length of any chain of messages is usually small
2. Application requiring a vector clock is confined to a subset of processes
3. Reset the EVC at a strongly consistent (i.e., transitless) global state
4. Use logarithms to store and transmit EVCs
   - Local tick: single addition
   - Merge and Compare: Take anti-logs and then logs,
     - complexity is subsumed by that of GCD computation
     - extra space is only scratch space

Case Study

Detecting memory consistency errors in MPI one-sided applications using EVC in the MC-CChecker tool

- Relevant events were the synchronization events; only these were timestamped
- Each concurrent region in the code was a unit of computation; boundary between two concurrent regions corresponded to a global transitless state
  - MC-CChecker safely reset EVCs at the start of each concurrent region
- Execution time and memory usage using EVC in MC-CChecker were much lower than using traditional vector clocks

Conclusions

- Studied the encoding of vector clocks using prime numbers, to use a single number to represent vector time
- Simulations show that the single integer EVCs grow fast
  - Analyzed the growth rate
  - Receive events cause EVCs to grow much faster due to LCM
  - Proposed several solutions to deal with this problem
    - Tick at relevant events; detection regions; reset EVC at transitless global state; use logs of EVCs
  - Case study: Detecting memory consistency errors in MPI one-sided applications
    - Using EVCs far more memory- and time-efficient than using traditional vector clocks

Thank You!