2013 IEEE 12th International Symposium on Parallel and Distributed Computing

Detecting Unstable Conjunctive Locality-Aware Predicates in Large-Scale Systems

Min Shen, Ajay D. Kshemkalyani and Ashfaq Khokhar
Dept. of Computer Science, University of Illinois at Chicago,
Chicago, IL 60607-7053, USA
Email: {mshen6, ajay, ashfaq}@uic.edu

***Abstract*—Recently, the concept of locality-aware predicates (LAP) has been proposed. A LAP models a predicate within a local region of the whole network in a large-scale locality driven system, such as WSNs and modular robotics. In such systems, the cost of doing a global predicate detection is high, besides which, a predicate over the state of a local region better captures properties of local interactions. Thus, LAP detection becomes a relevant and interesting problem. In this paper, we explore the problem of detecting unstable conjunctive LAP, and develop a scale-free algorithm by running an interval-based detection algorithm using a vector clock built on-the-fly for processes in the local region. More importantly, we develop the encoded vector clock (EVC) technique. EVC makes detecting unstable conjunctive LAP more practical in large-scale systems by reducing the storage cost.**

***Keywords*-distributed system; predicate detection; locality-aware; encoding; large-scale network**

I. Introduction

In recent years, distributed systems have found applications in new areas such as modular robotics [1], [2], [3] and wireless sensor networks (WSNs) [4]. These emerging areas share some common properties such as being large-scale and locality-driven. These properties lead to the need for robust and scalable algorithms to manage, monitor, and reason about the distributed execution in these applications.

To better observe the distributed properties that occur in the distributed execution of such large-scale systems, we recently proposed the concept of locality-aware predicates (LAP) [4]. Locality-aware predicates model predicates within a local area rather than the entire network in a large-scale system. In a large-scale locality-driven system, such as WSNs and modular robotics, we have the following.

1) The interactions are local and driven by neighborhood proximity.
2) The cost of doing a global predicate detection is high.
3) Properties of local interactions can only be captured by predicates over local regions.

These factors make locality-aware predicate detection a relevant and interesting problem. For example, to detect an explosion event in a WSN deployed field, we need to detect both the temperature and the sound level being above certain threshold, such as "*temperature* $> 150°C$" and "*sound* $> 60dB$". In order to correctly detect the explosion, this predicate needs to be detected on processes that are close to each other rather than far apart. Consider the predicate "number of tokens less than 5". When users are interested in knowing the state of a local region for 5-mutual exclusion, detecting this predicate within the local region provides more insight towards this region than detecting it within the entire network.

In our previous work [4], we reasoned that the key aspects in detecting locality-aware predicates are to specify the local detection region and to detect such predicates in a consistent manner. Therein, we also studied the problem of detecting stable LAPs, which are LAPs that remain true once they are found true by a consistent observation within the region.

The problem of detecting distributed properties that keep fluctuating with time is more challenging. Thus, the detection of unstable LAP, where the predicate may hold only intermittently, needs to be addressed. The problem of detecting unstable predicates within the entire system has been well studied in the literature [5], [6], [7]. Much of the literature focuses on conjunctive predicates, where the predicate can be expressed as the conjunction of predicates that are defined on variables local to a single process. This is because of the exponential complexity that may occur otherwise [7]. In this paper, we study the problem of detecting unstable conjunctive locality-aware predicates.

Contributions

1) We design the regional vector clock using virtual IDs in the local region.
2) We develop a scale-free algorithm, i.e., an algorithm whose complexity is independent of the size of the system, for detecting unstable conjunctive LAP in a large-scale system.
3) More importantly, we develop the encoded vector clock (EVC) technique which optimizes the time and space complexity of vector clocks. We show how to detect unstable conjunctive LAP using EVC. This makes detecting unstable conjunctive LAP more practical in a large-scale system.

Organization

Section II gives the system model and a background on predicate detection. Section III presents the basic detection algorithm which is scale-free. Section IV presents the encoded vector clock (EVC) technique and shows how we use it to optimize the detection algorithm. Conclusions and future work are discussed in Section V.

II. System Model and Background

In this section, we define the system model that will be used in later sections. We also briefly survey the work on predicate detection, especially for unstable predicates.

A. System Model

A distributed system is 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 $l = |L|$, and let d denote the degree of the graph, defined as the max degree of any node in the graph. The N processes asynchronously communicate with each other via logical channels. 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.

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 message reception, message sending, or an internal event. Let $E = \cup_{i \in P} e \in E_i$ denote the set of events executed in a distributed execution. The causal precedence relation between events induces an irreflexive partial order on E. This relation is defined as Lamport's "happens before" relation [8], and denoted as $\prec$. An execution of a distributed system is thus denoted by the tuple $(E, \prec)$.

B. Predicate Detection

There are several predicate types and algorithms studied in the literature (see the survey chapter by Kshemkalyani and Singhal [9]). We briefly summarize some of the more important classes of predicates.

When categorizing predicates based on the function on the variables/states involved in the predicate, there are these classes [10]:

1) A *relational predicate* is a predicate that is expressed as an arbitrary relation on the variables in the system. Let x_i and y_j be local variables at process P_i and P_j, respectively. $\psi =$ "$sum(x_i, y_j) = 45$" is a relational predicate.
2) A *conjunctive predicate* is a predicate that can be expressed as the conjunction of local variables. $\chi =$ "$x_i > 35 \land y_j < 32$" is a conjunctive predicate. In general, $\chi = \land_i \chi_i$.

When categorizing predicates based on their detectability, there are these classes:

1) A *stable predicate* is a predicate that remains true once it becomes true [11].
2) An *unstable predicate* is a predicate that is not stable and hence may hold only intermittently [10].

A stable/unstable predicate can be either conjunctive or relational. There is literature studying detection of unstable conjunctive predicates [5], [6]. However, due to the exponential complexity, few have studied the unstable relational predicate detection problem.

Unstable Predicates

Detecting unstable predicates over a distributed execution is important for various purposes such as monitoring, synchronization, coordination, and debugging. Due to the asynchrony in message transmissions and in local executions, different executions of the same distributed program can generate different sequences of global states. Therefore, whether an unstable predicate is detected within all consistent observations of an execution or within some consistent observation of an execution, can be different. Thus, two modalities have been defined under which a predicate Φ can hold [7], [10].

1) *Possibly(*Φ*)*: There exists a consistent observation of the execution such that Φ holds in a global state of the observation.
2) *Definitely(*Φ*)*: For every consistent observation of the execution, there exists a global state of it in which Φ holds.

Cooper and Marzullo gave an algorithm to detect *Possibly(*Φ*)* and *Definitely(*Φ*)* for a relational unstable predicate [7]. However, this algorithm has exponential complexity. This is expected because unstable predicate detection essentially is an NP-complete problem. However, for conjunctive unstable predicates, there exist detection algorithms based on intervals that incur only a polynomial complexity [5], [6]. For such predicates, an interval at a process P_i is defined as the time duration in which the local predicate is true. Due to the lack of synchronized physical clocks at each process, the start and end events of an interval x, denoted as $\min(x)$ and $\max(x)$, respectively, are identified by vector clocks [12], [13]. The detection of either *Possibly(*Φ*)* or *Definitely(*Φ*)* is to identify a set of intervals, containing one interval per process in which the local predicate is true, such that a certain condition is satisfied within the set under either modality. The conditions for *Possibly(*Φ*)* or *Definitely(*Φ*)* to hold within a set X of intervals have been shown as follows [5], [6], [14]:

$$\textit{Possibly}(\Phi) : \forall x_i, x_j \in X, \max(x_i) \not\prec \min(x_j) \quad (1)$$

$$\textit{Definitely}(\Phi) : \forall x_i, x_j \in X, \min(x_i) \prec \max(x_j) \quad (2)$$

Garg and Waldecker proposed a detection algorithm for *Possibly(*Φ*)* [5]. However, that algorithm is not based on intervals. They later proposed another detection algorithm for *Definitely(*Φ*)* [6], based on the concept of intervals. An interval-based algorithm that works for both *Possibly(*Φ*)* and *Definitely(*Φ*)* was also presented [9]. For large-scale systems, an efficient hierarchical algorithm to detect regional/local predicates as well as global predicates in the *Definitely*(Φ) modality was recently proposed [15].

III. Detecting Unstable Conjunctive LAP

A. Modeling the Local Region

A key aspect in using locality-aware predicates is to specify the region within which the predicate is to be detected. We model this region as an area centered at process P_r with which the user interacts, and that region contains all processes within a distance of h hops from P_r. We call h the radius of the region, which is a parameter specified by the user. To represent this model in the network, we construct a topology that forms a local breadth-first search tree (BFST) at process P_r using links in L. To represent a h-radius region, the height of the BFST is h. To construct such a topology, we can adapt the distributed BFST algorithm [16], which constructs a BFST within the whole network, into one that imposes a height constraint. This topology represents the region within which the predicate is to be detected.

B. Establishing the Regional Vector Clock

Although the topology representing the detection region is constructed, we cannot directly apply the detection algorithms [6], [9] for detecting unstable conjunctive LAP. These interval-based algorithms require the establishment of vector clocks, which are used to identify the start and end events of an interval. In a large-scale system, it is impractical to assume a vector clock is initially maintained for the entire system. Churn in the system makes such a system-wide vector clock further impractical. Even if we are to establish such vector clocks for the system, it will incur an $O(N)$ (N is the number of processes in the entire system) storage cost at each process. This causes the solution to be non-scale-free. Being scale-free is important for LAP detection algorithms, because we do not want to incur time and space complexities relevant to the size of the entire network to observe only a part of the system. Thus, we need to dynamically establish a vector clock only for the processes in the detection region.

To establish the regional vector clock, each process in the detection region needs to be assigned a unique virtual ID that is within the range $[1, n]$, where n is the number of processes in the detection region. By constructing the local BFST in Section III-A, we assume that P_r has already collected the real IDs of all processes in the local BFST and each process knows its parent and children in the BFST. P_r then assigns the virtual ID 1 to itself and a unique virtual ID in the range $[2, n]$ for every other process in the local BFST. In this way, each process gets mapped to a unique position in the size-n vector. Process P_r then broadcasts this mapping between the real IDs to the virtual IDs within the local BFST. Each process that receives this map initializes a size-n vector V. To capture all causal relations consistently during the initial phase of establishing the regional vector clock, another convergecast and broadcast [9] need to be performed within the local BFST. This ensures that no message piggybacked with a vector clock timestamp is received until the recipient process in the detection region has established the size-n vector locally. This establishment of the regional vector clock is shown in Algorithm 1.

Algorithm 1 Establishing Regional Vector Clock (Code for P_i in the region)

```
integer: $VID$
array of integer: $V$
HashMap of ⟨integer, integer⟩: $map$ // Maps real IDs of
        // processes in BFST to virtual IDs in the range $[1, n]$

$P_r$ initiates the algorithm:
1.  $VID = 1$;
2.  generate $map$, with the constraint that $map(r) = 1$;
3.  broadcast $ASSIGN(map, n)$ message in the local BFST;

$P_i (i \neq r)$ receives $ASSIGN(map, n)$ message from parent:
4.  $VID = map(i)$;
5.  initialize size $n$ vector $V$;
6.  if ($P_i$ is not leaf node) then
7.      send $ASSIGN(map, n)$ message to all children;
8.  else
9.      convergecast $FINISH$ message to $P_r$;

$P_r$ receives $FINISH$ message:
10. broadcast $READY$ message within the local BFST;

$P_i$ receives $READY$ message:
11. start piggybacking messages with vector clock timestamps;
```

Notice that, by broadcasting the mapping between the real IDs to the virtual IDs within the local BFST, every process in the detection region knows the identifications of all processes within the region. In this way, each process in the region can also determine whether it is sending/receiving a message to/from a process outside the region.

The regional vector clocks are updated by the following rules [12], [13].

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

For detecting unstable LAP, the regional vector clock only tracks the causal relations among the processes in the local

Algorithm 2 Detection Algorithm for Unstable Conjunctive Predicates, adapted from [9]

```
queues for all $n$ processes in the detection region:
$Q_1, Q_2, \ldots, Q_n \leftarrow \perp$
sets of integer: updatedQueues, newUpdatedQueues $\leftarrow \{\}$
integer: $VID$

When interval $x$ finishes at $P_i$:
1.  send $(\min(x), \max(x), VID)$ to $P_r$;

On receiving an interval $(\min(I), \max(I), VID)$ at $P_r$:
2.  Enqueue the interval onto queue $Q_{VID}$;
3.  if (number of intervals on $Q_{VID}$ is 1) then
4.    updatedQueues = {VID};
5.    while (updatedQueues is not empty)
6.      newUpdatedQueues = {};
7.      for each $a \in updatedQueues$ do
8.        if ($Q_a$ is not empty) then
9.          $x$ = head of $Q_a$;
10.         for $b = 1 \ldots n, b \neq a$ do
11.           if ($Q_b$ is not empty) then
12.             $y$ = head of $Q_b$;
13.             if ($\min(x) \not\prec \max(y)$) then    // $Definitely$
14.               add $b$ to newUpdatedQueues;
15.             if ($\min(y) \not\prec \max(x)$) then    // $Definitely$
16.               add $a$ to newUpdatedQueues;
17.             if ($\max(x) \prec \min(y)$) then    // $Possibly$
18.               add $a$ to newUpdatedQueues;
19.             if ($\max(y) \prec \min(x)$) then    // $Possibly$
20.               add $b$ to newUpdatedQueues;
21.     Delete heads of all $Q_h$ where $h \in newUpdatedQueues$;
22.     updatedQueues = newUpdatedQueues;
23. if (all queues are non-empty) then
24.   report predicate detected. Heads of queues form the
      solution.
```

BFST. However, whenever there is a message going out of or coming into the region, a potential transitive causal relation is introduced if some process inside the region sends a message to a process outside the region which later sends another message back into the region. In order for the detection algorithm to work correctly, those transitive causal relations also need to be captured. This requires all the processes outside the region to store the vector clock timestamps they receive piggybacked on messages and to piggyback those timestamps with every outgoing message. However, processes outside the detection region do not advance the vector clock timestamps stored locally, since they do not contribute to the causal relations between processes inside the detection region.

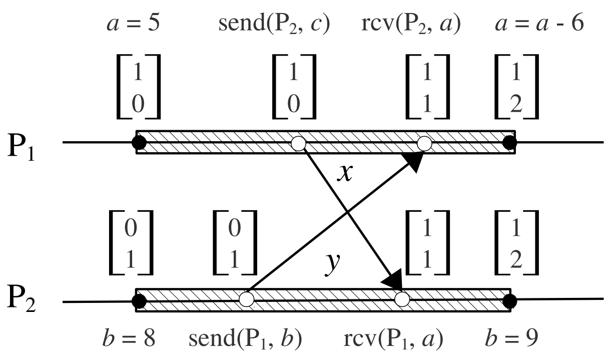
C. Detecting Unstable Conjunctive LAP

With the regional vector clock established, we can run the detection algorithm given as Algorithm 2, based on [9], within the detection region. Process P_r locally maintains n queues, $Q_1, Q_2, \ldots, Q_n$ numbered using virtual IDs. Whenever a new interval x finishes at some process in the detection region, this process sends the vector clock timestamps of $\min(x)$ and $\max(x)$ and its virtual ID to P_r. P_r then enqueues the interval x onto queue Q_i. By tracking the intervals from all n processes, P_r repeatedly checks the heads of all n queues using the conditions in (1) or (2) in Section II to check whether *Possibly(Φ)* or *Definitely(Φ)* is detected within the region. This check is done with virtual IDs. If any interval is found to violate those conditions, P_r deletes this interval from the corresponding queue.

D. Ticking at Relevant Communication Events

In Algorithm 2, observe that the 4 causality tests in the innermost loop essentially check if $e_i \prec f_j$, where e_i and f_j are either the start or end events of some intervals, which are *relevant events* to the detection of LAP. This test using vector timestamps is $O(1)$ time, namely if $V(e_i)[i] \leq V(f_j)[i]$. This test is equally valid even if the regional vector clock does not tick at message send and message receive events, as long as such events are not relevant to the local predicate, i.e., do not alter the truth value of the local predicate. Most predicates, such as χ introduced in Section II-B, may not depend on the message send or receive events which only serve to transmit and establish causal relationships among the relevant events. Thus, it is not necessary to advance the local clock component at send and receive events, unless the events modify some variable and change the truth value of the predicate. Thus, we have the rule:

Relevancy test: *Tick the local component of the regional vector clock only at a relevant event, which is defined to be an event that alters the truth value of the predicate.*



$a = 5$
send(P_2, c)
rcv(P_2, a)
$a = a - 6$
$\begin{bmatrix}1\\0\end{bmatrix}$ $\begin{bmatrix}1\\0\end{bmatrix}$ $\begin{bmatrix}1\\1\end{bmatrix}$ $\begin{bmatrix}1\\2\end{bmatrix}$
P_1
x
$\begin{bmatrix}0\\1\end{bmatrix}$ $\begin{bmatrix}0\\1\end{bmatrix}$ $\begin{bmatrix}1\\1\end{bmatrix}$ $\begin{bmatrix}1\\2\end{bmatrix}$
y
P_2
$b = 8$
send(P_1, b)
rcv(P_1, a)
$b = 9$

Figure 1. Illustration of the "relevant event" test. Each process has local variables a, b, and c. The conjunctive predicate $\psi = a_1 > 3 \bigwedge b_2 = 8$ is to be detected. The operations performed at each event are shown. By not ticking the vector clock for message send and receive events that are not relevant, the causal relations between the two intervals x and y at processes P_1 and P_2 are still correctly captured.

Figure 1 illustrates a computation in which the truth values of the local predicates are not changed by the send or receive events. The syntax of a send is $send(dest, var)$, which is to send the value of local variable var to $dest$. The syntax of a receive is $rcv(source, var)$, which is to store the value received from $source$ into local variable var. The predicate is $\psi = a_1 > 3 \bigwedge b_2 = 8$. The send event at P_1 is not relevant to the local predicate. The send event at P_2 does not alter the truth value of the local predicate. The receive event at P_1 modifies the variable a_1 but does not alter the truth value of the local predicate. The receive event at P_2 does not modify any variable on which the local predicate depends.

By applying this relevancy test, multiple events may have the same vector timestamp, such as the two receive events in Figure 1. However, the lattice of relevant events, which now excludes send events and receive events that are not relevant to the predicate (i.e., do not alter the truth value of the predicate), still remains isomorphic to the lattice of timestamps assigned to them. This property is sufficient for the correctness of our algorithm. It is particularly useful in conjunction with the encoded vector clocks optimization that we develop in Section IV.

IV. Encoded Vector Clock (EVC) Optimization

Although the solution proposed in Section III detects the unstable conjunctive LAP in a scale-free manner without incurring a complexity relevant to N, due to message diffusion and the need to capture transitive causal relations, it will eventually incur an $O(n)$ storage cost in every process in the entire system. This is a cost that we cannot afford, especially in a large-scale system.

To solve this problem, we develop the encoded vector clock (EVC) technique. Charron-Bost has shown that to capture the partial order on E, the size of the vector clock can be as large as the dimension of the partial order [17], which is the size of the system N. Instead of using a vector of size $O(N)$, it was suggested that the vector can be encoded into a single number using N distinct prime numbers [17]. In the case of detecting unstable LAP, the dimension of the partial order that is relevant can be captured by the size of the detection region, as shown in Section III. Thus, a regional 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. To build on that work, we develop EVC technique to show how to implement the basic operations of a vector clock.

A. Encoded Vector Clock Operations

Local Tick: Whenever the logical time advances locally, the local component of the vector clock needs to tick. This happens as increasing 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 vector clock timestamps 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:

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

It would be better to merge $Enc(V_1)$ and $Enc(V_2)$ into $Enc(U)$ without knowing the n prime numbers. This can be

Table I
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	$V_1 \prec V_2$: $\forall j \in [1, n]$, $V_1[j] \leq V_2[j]$, and $\exists j$, $V_1[j] < V_2[j]$	$Enc(V_1) \prec Enc(V_2)$: $Enc(V_1) < Enc(V_2)$, and $Enc(V_2) \mod Enc(V_1) = 0$

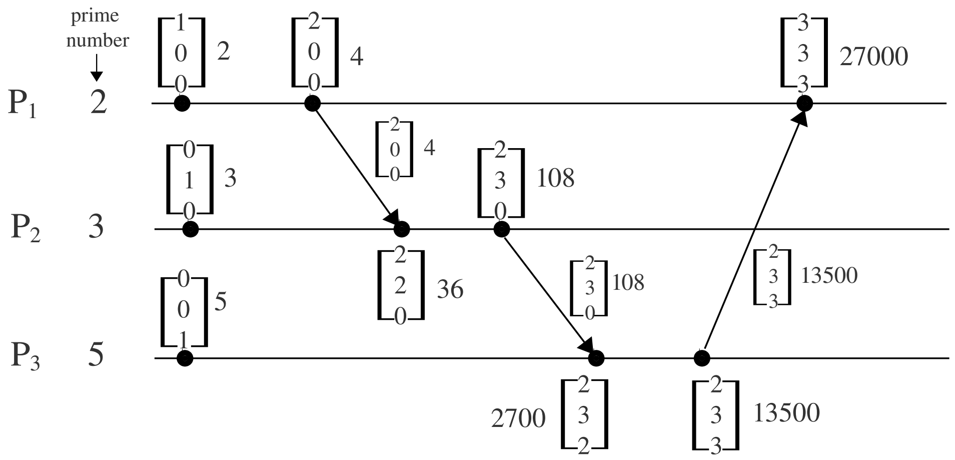


Figure 2. Illustration of using EVC for capturing causal relations. The local prime number for each process is shown beside its ID. The vectors shown in the diagram are only explaining the EVC timestamps. In real scenarios, only the number shown beside each vector is stored and transmitted.

achieved by observing that $Enc(U)$ is the LCM of $Enc(V_1)$ and $Enc(V_2)$. So, by computing the LCM of two EVC timestamps, these two timestamps can be merged without knowing the n prime numbers.

Comparison: Furthermore, the vector clock needs a mechanism to compare two timestamps. To compare two vector clock timestamps, a component-wise comparison between the corresponding elements of two vectors is needed. The comparison has two results:

$$i)\ V_1 \prec V_2 \text{ if } \forall j \in [1,n], V_1[j] \le V_2[j] \text{ and } \exists j, V_1[j] < V_2[j]$$

$$ii)\ V_1 \| V_2 \text{ if } V_1 \not\prec V_2 \text{ and } V_2 \not\prec V_1$$

To compare two EVC timestamps, it is only necessary to test if $Enc(V_j) \bmod Enc(V_i) = 0$. Thus,

$$i)\ Enc(V_1) \prec Enc(V_2) \text{ if } Enc(V_1) < Enc(V_2) \text{ and } Enc(V_2) \bmod Enc(V_1) = 0$$

$$ii)\ Enc(V_1) \| Enc(V_2) \text{ if } Enc(V_1) \not\prec Enc(V_2) \text{ and } Enc(V_2) \not\prec Enc(V_1)$$

The correspondence between the three basic operations of the vector clock and EVC is shown in Table I. These operations using EVC are illustrated in Figure 2. If send events and receive events are not relevant to the local predicates, the local clocks do not need to tick at such events, as explained in Section III-D. In that case, the EVC timestamp 27000 in Figure 2 now is only 60.

With EVC, we can reduce the computing and storage cost for processes within the detection region. Instead of each maintaining a vector of size $O(n)$, processes within the detection region now only need to maintain a single integer.

More importantly, for processes outside the detection region, we can also 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 vector clock locally since there is no corresponding component in the vector clock for P_j, it may still receive multiple messages from within the detection region and needs to be able to merge the vector clock timestamps it receives. When this happens, P_j simply executes the merge operation by calculating the LCM of two numbers.

Figure 3 illustrates how the encoded vector clock works when the detection region is established and the vector clock is maintained only for processes with the region. If send events and receive events are not relevant to the local predicates, the local clocks do not need to tick at such events, as explained in Section III-D. In that case, the EVC timestamp 72 in Figure 3 now is only 6.

After using EVC, when P_r compares two vector clock timestamps at line 13, 15, 17, and 19 in Algorithm 2, P_r will compare two EVC timestamps using the operation described in Table I. Whenever a new interval x finishes at some process in the detection region, the EVC timestamps of events $\min(x)$ and $\max(x)$ are sent to P_r, where LAP detection is performed.

B. Complexity

Comparing with vector clocks, EVC has advantages in time, space, and message size complexity. Each process only needs to store a single number. If we assume that the local space for storing this number is bounded, then the storage cost is only $O(1)$. When reporting intervals using EVC timestamps, the message size complexity also becomes $O(1)$. For time complexity, all operations except computing LCM take $O(1)$ time. For computing $LCM(a,b)$, we have:

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

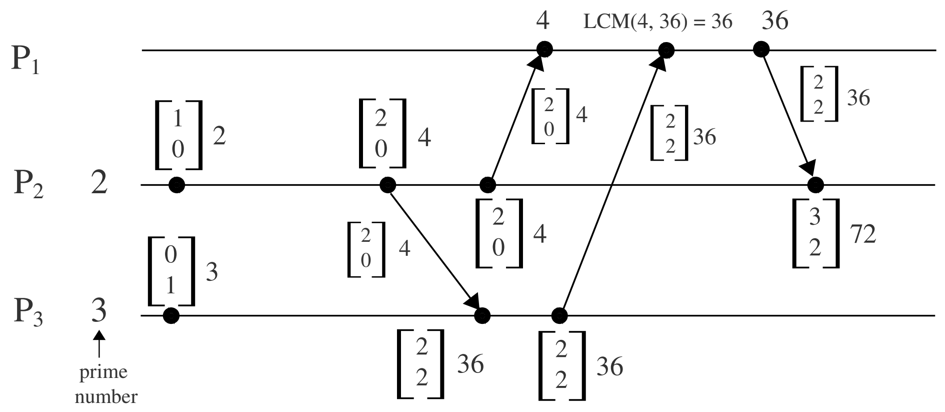


Figure 3. Illustration of using EVC for capturing causal relations within a local region of the network. Here, EVC is built for only P_2 and P_3. The local prime numbers for those two process are shown beside its ID. Again, the vectors in the diagram are only to help understand the EVC timestamps.

By applying the Euclidean algorithm, we can compute $GCD(a, b)$ without factoring the two numbers. It is also well known that the time complexity of Euclidean algorithm is $O(h^2)$, where h is number of digits of the smaller number in base 10. If we assume the numbers are bounded, $O(h^2)$ becomes $O(1)$ and we can compute LCM in $O(1)$ time.

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 [18] which enables us to reuse the smaller numbers. The clock resetting algorithm will incur an $O(n)$ message count complexity and an $O(d)$ storage cost at each process in the region, where d is the maximum degree of processes in the region. The details of adapting the resetting technique are discussed in the next subsection. In Table II, we compare the time complexity and the storage cost of the three basic operations, which are local tick, merge, and compare, for vector clock and EVC.

For a system with N processes and assuming the local predicate becomes true at most m times at each process, Algorithm 2 has a time and space complexity of $O(N^2m)$ at the sink. It also has an $O(mN)$ message count complexity and an $O(mN^2)$ message size complexity. When detecting unstable conjunctive LAP, the factor N is reduced to n, thus resulting in a time and space complexity of $O(n^2m)$ at the sink plus an $O(mn)$ message count complexity and an $O(mn^2)$ message size complexity. After using EVC, the time and space complexity at the sink is further reduced to $O(nm)$, since the intervals are now identified by two integers rather than two vectors. The message size complexity also gets reduced to $O(mn)$.

For non-sink nodes in Algorithm 2, the only complexity comes from maintaining the vector clock. In a system with N processes, that will be an $O(N)$ space complexity. When detecting unstable conjunctive LAP, the space complexity at those non-sink nodes within the detection region becomes $O(n)$, and also for processes outside the detection region. By using EVC, the space complexity for all processes gets further reduced to $O(1)$. Even with the clock resetting, the space complexity for non-sink processes within the detection region is $O(d)$, still less than $O(n)$. This makes Algorithm 2 a better scale-free solution for detecting unstable conjunctive LAP.

C. Resetting EVC

For n processes in the detection region and f_i relevant 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. Fortunately, we can adapt the clock resetting technique [18] to solve this problem.

The clock resetting technique divides the execution of a distributed system into multiple phases. Each time the clock overflows at one process inside the detection region, the resetting algorithm terminates the current phase by sending control messages within the region to make sure there is no

Table II
COMPARISON OF THE TIME AND SPACE COMPLEXITY OF THE THREE BASIC OPERATIONS

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

computation message sending from the previous phase to the next phase. It also introduces artificial causal relations to ensure every event happening in the next phase happens after the events in the previous phase.

For EVC, this clock resetting technique can be used to reset the EVCs of the processes in the detection region. For processes outside the detection region, we can utilize the phase ID to reset their locally stored EVC timestamp. Each process P_i in the detection region maintains the ID of the current phase it is in. Each time the resetting algorithm is executed, the phase ID is increased by 1. Whenever a process in the detection region sends a message to processes outside the region, the phase ID is also piggybacked. Processes outside the detection region also need to store the phase ID, in addition to the EVC timestamp. Whenever a process outside the detection region receives a message with a phase ID larger than the locally stored value, it deduces that a reset has taken place in the detection region and it can safely replace the locally stored EVC timestamp with the one piggybacked on the message.

Furthermore, it is possible that the overflow could happen at a process outside the detection region. If this happens, the outside process starts piggybacking its outgoing messages with a resetting flag. Processes inside the detection region do not reset their EVCs until a message chain is established between the overflowing process and a process inside the detection region. Thus, the resetting technique can be adapted and used as described above to ensure the correctness of the operation even when clock overflow is unavoidable.

V. Conclusions and Future Work

In this paper, we further explored the problem of LAP detection. Focusing on detecting unstable conjunctive LAP, we developed a scale-free solution in which a regional vector clock in the detection region is built on-the-fly and the predicate is detected by an interval-based algorithm [9]. More importantly, we developed the encoded vector clock (EVC) technique that optimizes the detection algorithm by reducing the storage cost from $O(n)$ at every process in the whole network to $O(1)$ for processes outside the detection region and $O(d)$ (with clock resetting) for processes within the region. EVC makes detecting unstable conjunctive LAP more practical in large-scale systems.

There are several potential extensions of locality-aware predicates, such as their adaptation to predicates of different detectability. We also plan to explore locality-aware predicate detection in a weighted graph, and to specify more types of detection region. Additionally, we plan to explore locality-aware predicate in the context of the immediate detection problem [19]. All these will broaden the application of locality-aware predicates.

References

[1] M. De-Rosa, S. Goldstein, P. Lee, P. Pillai, and J. Campbell, “Programming modular robots with locally distributed predicates,” *Proceedings of the IEEE ICRA*, 2008.

[2] ——, “A tale of two planners: Modular robotic planning with ldp,” *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[3] M. De-Rosa, S. Goldstein, J. C. P. Lee, and P. Pillai, “Detecting locally distributed predicates,” *ACM Transactions on Autonomous and Adaptive Systems*, June 2011.

[4] M. Shen, A. Kshemkalyani, and A. Khokhar, “Detecting tree distributed predicates,” *2012 41st International Conference on Parallel Processing Workshops (ICPPW)*, pp. 598–599, 2012.

[5] V. K. Garg and B. Waldecker, “Detection of weak unstable predicates in distributed programs,” *IEEE Transactions on Parallel & Distributed Systems 5, 3*, pp. 299–307, 1994.

[6] ——, “Detection of strong unstable predicates in distributed programs,” *IEEE Transactions on Parallel & Distributed Systems 7, 12*, pp. 1323–1333, 1996.

[7] R. Cooper and K. Marzullo, “Consistent detection of global predicates,” *ACM SIGPLAN Notices. Vol. 26.*, pp. 167–174, 1991.

[8] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM 21, 7*, pp. 558–565, 1978.

[9] A. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[10] R. Cooper and K. Marzullo, “Consistent detection of global predicates,” *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 163–173, 1991.

[11] K. M. Chandy and L. Lamport, “Distributed snapshots: Determining global states in distributed systems,” *ACM Transaction on Computer Systems 3, 1*, pp. 63–75, February 1985.

[12] F. Mattern, “Virtual time and global states of distributed systems,” *Proceedings of the Parallel and Distributed Algorithms Conference*, pp. 215–226, 1988.

[13] C. Fidge, “Logical time in distributed computing systems,” *IEEE Computer*, pp. 28–33, Aug, 1991.

[14] A. Kshemkalyani, “Temporal interactions of intervals in distributed systems,” *Journal of Computer and System Sciences, 52, 2*, pp. 287–298, 1996.

[15] M. Shen and A. Kshemkalyani, “A fault-tolerant strong conjunctive predicate detection algorithm for large-scale networks,” *2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops (IPDPSW 2013)*, 2013.

[16] K. M. Chandy and J. Misra, “Distributed computations on graphs: shortest path algorithms,” *Communications of the ACM*, pp. 833–838, 1982.

[17] B. Charron-Bost, “Concerning the size of logical clocks in distributed systems,” *Inf. Process. Lett. 39, 1*, pp. 11–16, 1991.

[18] L.-H. Yen and T.-L. Huang, “Resetting vector clocks in distributed systems,” *Journal of Parallel and Distributed Computing, 43, 1*, pp. 15–20, 1997.

[19] A. Kshemkalyani, “Immediate detection of predicates in pervasive environments,” *Journal of Parallel and Distributed Computing, 72, 2*, pp. 219–230, 2012.