Hierarchical Detection of Strong Unstable Conjunctive Predicates in Large-Scale Systems

Min Shen, *Student Member, IEEE* and Ajay D. Kshemkalyani, *Senior Member, IEEE*

Abstract—In large-scale systems where an on-going monitoring program is needed, the traditional predicate detection algorithms become undesirable due to their high overhead and inability to do repeated detection and to resume the detection after a node failure. This paper presents an on-line decentralized algorithm that detects strong conjunctive predicates in a large-scale system. Our algorithm assumes a preconstructed spanning tree in the system, and detects all satisfactions of the predicate in a hierarchical manner. When a node fails or moves and the structure of the spanning tree is changed, our algorithm is able to recover from this situation and continue the detection of further occurrences of the predicate satisfactions. The hierarchical structure of our algorithm also provides a finer-grained monitoring in those large-scale systems where grouping is established and the monitoring happens at the group level. Comparing with other detection algorithms, our algorithm incurs a low space and time cost, which is distributed across all the nodes in the system, and a low message complexity. Our algorithm is particularly beneficial to resource-constrained systems.

Index Terms—Distributed system, predicate detection, large-scale systems, fault-tolerant

1 Introduction

Detecting predicates over a distributed execution is important for various purposes such as data stream processing, industrial process monitoring, synchronization, coordination, and debugging. In recent years, predicate detection has found applications in large-scale systems such as WSNs [1] and modular robotics [2], [3], where individual nodes have only limited computation and storage resources. With these properties, new solutions that conserve the limited resources and take into consideration the potential failures of the nodes are needed.

There are several predicate classes [4]. Two main classes are:

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. $\Phi = ``x_i + y_j > 40''$ is a relational predicate.
2. A *conjunctive predicate* is a predicate that can be expressed as the conjunction of local predicates. $\Psi = ``x_i < 35 \wedge y_j > 25''$ is a conjunctive predicate.

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 a predicate gets satisfied within all consistent observations of an execution or within some consistent observation of an execution, can be different. Thus, two modalities under which a predicate Φ can hold have been defined [5].

1. $Possibly(\Phi)$: There exists a consistent observation of the execution such that Φ holds in a global state of the observation.
2. $Definitely(\Phi)$: For every consistent observation of the execution, there exists a global state of it in which Φ holds. This type of predicates is also called strong predicates in [6].

Algorithms to detect $Possibly(\Phi)$ and $Definitely(\Phi)$ for a conjunctive or relational predicate are given in [5]. However, it has been shown that detecting a relational predicate is an NP-complete problem. Due to the exponential complexity of detecting relational predicates, most work on predicate detection is focused on conjunctive ones.

In [6], [7], Garg and Waldecker gave centralized algorithms to detect $Definitely(\Phi)$ and $Possibly(\Phi)$, respectively. In [6], they presented an interval-based approach to detect $Definitely(\Phi)$. In [8], an interval-based algorithm that adopts a unified approach to detect both $Possibly(\Phi)$ and $Definitely(\Phi)$ was given. For a system of n processes and an execution in which the local predicate becomes true at most p times at a process, the detection algorithm in [8] has a space and time complexity of $O(pn^2)$. It also generates $O(pn)$ messages, each of size $O(n)$.

Several distributed algorithms were also proposed. Garg and Chase [9] and Hurfin *et al.* [10] presented distributed algorithms to detect $Possibly(\Phi)$. Both algorithms have space, time and message size complexities of $O(mn^2)$, where m is the maximum number of messages sent by any process. Chandra and Kshemkalyani [11] gave a distributed algorithm for detecting $Definitely(\Phi)$. Its space and time complexities are $O(\min(pn^2, mn^2))$ and its message size complexity is also $O(\min(pn^2, mn^2))$. Although [11] is also a distributed

• *The authors are with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, USA. E-mail: {mshen6, ajay}@uic.edu.*

Manuscript received 4 Mar. 2013; revised 25 Nov. 2013; accepted 30 Nov. 2013. Date of publication 12 Dec. 2013; date of current version 15 Oct. 2014. Recommended for acceptance by R. Baldoni.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.306

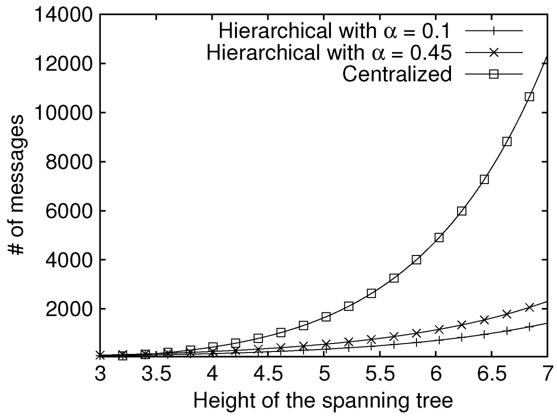

Fig. 1. Message count complexity comparison between hierarchical and centralized detection, with $d = 2$, $p = 20$.

algorithm for detecting $Definitely(\Phi)$, compared with our algorithm, it lacks the capability to do repeated detection and hierarchical detection.

When detecting predicates in continuous monitoring programs, usually the application requires the monitoring program to raise an alarm *each time* the predicate occurs. One example is data stream processing [12], where persistent tracking of the specified pattern, which can be expressed as predicates, is required. Another example is the industrial process monitoring program such as in the chemical manufacturing industry, where the system is monitored for events of both temperature and pressure exceeding certain thresholds. In such cases, the corresponding predicates could become true multiple times, and the monitoring program needs to detect all occurrences. Thus, none of these algorithms [6], [7], [9], [10], [11] are applicable. As shown in [13], these algorithms can detect predicates only once and will hang after the initial detection. They cannot detect multiple occurrences because detecting subsequent occurrences is not simply rerunning those one-time detection algorithms, but requires elaborate processing to ensure safety and liveness. In [13], a centralized repeated detection algorithm which can detect all occurrences of $Definitely(\Phi)$ in $O(pn^3)$ time is given. However, all the time and space costs incurred by this algorithm are at the sink.

This paper presents a decentralized algorithm that detects $Definitely(\Phi)$ in a large-scale system. This algorithm assumes a pre-constructed spanning tree [8], [14] in the system and detects the predicate in a hierarchical manner. By establishing a hierarchy in the system, our algorithm divides the task of detecting a predicate among different levels in the hierarchy. Each node detects the predicate within the subtree rooted at itself. The hierarchical structure of our algorithm also provides a finer-grained monitoring in those large-scale systems where grouping is established and the monitoring is needed at the group level. In addition, our algorithm detects the predicates in a repeated manner [13]. In long-running applications where continuous monitoring is required, repeated detection is essential because manual intervention after one detection of predicate satisfaction to reset the detection algorithm is not practical or even possible.

We assume a degree-bounded spanning tree in the system, where the maximum degree of all nodes is bounded by the value d. For a spanning tree of height h and maximum degree d, our hierarchical algorithm has a global time complexity of $O(d^2pn^2)$ and a global space complexity of $O(pn^2)$, distributed across all nodes in the system. Comparing with the only algorithm capable of doing repeated detection [13], which is centralized and incurs an $O(pn^3)$ time complexity and an $O(pn^2)$ space complexity, our algorithm is superior in performance since d^2 is less than n for any spanning tree with $h > 2$ (essentially any non-centralized configuration). Also, the message count complexity of hierarchical detection is significantly lower. A comparison between our hierarchical algorithm and the centralized repeated algorithm is given in Table 1. Specifically, in Fig. 1, we show more insights towards the differences between the message count complexities of the two algorithms. Here, α is the probability that intervals from d children overlap at one higher level. Notice that, for message size complexity, an additional factor of n will be applied since vector clock is used (see Table 1). This is the case for other predicate detection algorithms, such as [6], [7], [8].

TABLE 1
Complexity Comparison between Hierarchical Detection and the Centralized Repeated Detection Algorithm

	Our Hierarchical Algorithm	Centralized Repeated Detection Algorithm [13]
Space Complexity	$O(pn^2)$ (across all processes)	$O(pn^2)$ (at the sink node)
Time Complexity	$O(d^2pn^2)$ (across all processes)	$O(pn^3)$ (at the sink node)
Message Count	pn	$p\frac{(d^h-2d)(dh-d-h)-d}{(d-1)^2}$
Message Size	pn^2	$pn\frac{(d^h-2d)(dh-d-h)-d}{(d-1)^2}$

We build on our preliminary result [15] by showing how our algorithm handles node failures and mobility in the system and by giving an in-depth complexity analysis.

Contributions

1. We present the first decentralized hierarchical algorithm to detect $Definitely(\Phi)$ in a large-scale distributed system.
2. Hierarchical detection, which is also strongly desirable for large-scale systems, necessarily requires detection of all occurrences of the predicate satisfaction, which we do in our algorithm. None of the existing detection algorithms for $Definitely(\Phi)$ (except the recent centralized algorithm in [13]) can do such repeated detection of all occurrences of Φ. They all hang if a node fails.
3. The hierarchical detection in our algorithm makes it capable of handling node failures or mobility. In our algorithm, each process detects the predicate in the subtree rooted at itself. When a node fails or moves, the detection of the predicate in the system can be easily resumed because our algorithm has the ability to detect a partial predicate of the global predicate and deal with a reconfigured tree. The same cannot be achieved by the existing centralized or distributed detection algorithms.
4. We give a performance analysis of our hierarchical detection algorithm for message, space and time

complexity. The result (summarized in Table 1) shows that our algorithm is superior to the only known algorithm for repeated detection [13], which is centralized.

Section 2 gives the system model and background on interval-based predicate detection. Section 3 presents the hierarchical detection algorithm and its theoretical foundation. Section 4 shows how our algorithm deals with node failures and node mobility. Conclusions are given in Section 5. The detailed complexity analysis is included in the supplementary file which is available in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.306

2 System Model and Background

2.1 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. Let $n = |P|$. The n processes communicate asynchronously with each other via the channels in L. We do not assume FIFO channels, thus the messages may be delivered out of order. The execution of process P_i produces a sequence of events $E_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$, where e_i^k is the kth event at process P_i. An event at a process can be message receiving, message sending, or an internal event. Let $E = \cup_{i \in P} E_i$ 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 [16], and denoted as $\prec$. An execution of a distributed system is thus denoted by the tuple $(E, \prec)$.

If the network is a wireless network, each process can communicate only with other processes within communication range. The topology of such a network is not a complete graph, and messages transmitted within such a network usually traverse multiple hops.

We also assume vector clocks [17], [18], [19], [20] are available. Each process maintains a vector clock V of n integers, which is updated by the following rules.

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

The $\prec$ relation between two events can be checked by comparing their corresponding vector clock timestamps, i.e., $e_i \prec e_j \Leftrightarrow V_{e_i} < V_{e_j}$, where $V_{e_i} < V_{e_j}$ means $\forall a \in [1, n]$, $V_{e_i}[a] \leq V_{e_j}[a]$ and $\exists b \in [1, n]$ such that $V_{e_i}[b] < V_{e_j}[b]$. Henceforth, we use the notation $\prec$ between two events and $<$ between their corresponding vector timestamps interchangeably.

2.2 Background on Interval-Based Detection

An interval at a process P_i is 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 [17], [18], [19], [20]. The detection of either $Possibly(\Phi)$ or $Definitely(\Phi)$ 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 this set. In [7], [6], [21], it was shown that the conditions to be satisfied for $Possibly(\Phi)$ or $Definitely(\Phi)$ to hold within a set X of intervals are as follows:

$$Possibly(\Phi) : \forall x_i, x_j \in X, \max(x_i) \not\prec \min(x_j) \qquad (1)$$

$$Definitely(\Phi) : \forall x_i, x_j \in X, \min(x_i) \prec \max(x_j). \qquad (2)$$

Sink process P_1 locally maintains n queues, $Q_1, Q_2, \ldots, Q_n$. Whenever a new interval x completes at some process P_i, P_i sends the vector clock timestamps corresponding to $\min(x)$ and $\max(x)$ to P_1 (This latency in detection is in all interval-based predicate detection algorithms, except [22]. Note that, some non-interval-based detection algorithms [7] or some algorithms for weaker classes of predicates [2], [3] do not incur such a delay). P_1 then enqueues the interval x onto queue Q_i. By tracking the intervals from all n processes, P_1 checks the heads of all n queues using the condition in Eq. (1) or (2) to see whether $Possibly(\Phi)$ or $Definitely(\Phi)$ is detected. If any interval is found to violate those conditions, P_1 deletes this interval from its corresponding queue.

3 Hierarchical Detection

3.1 Basic Idea and Challenges

Our hierarchical detection algorithm works as follows. We assume a spanning tree is already constructed in the system [8], [14]. The algorithm utilizes this spanning tree to establish a hierarchy for detecting $Definitely(\Phi)$. Each non-leaf node in the tree only maintains queues to track intervals that are sent by its children or that occur locally. Whenever a new interval occurs at a leaf node, it is transmitted to the leaf node's parent which tries to detect the predicate within the subtree rooted at itself. If the predicate is detected in the subtree, the root of the subtree aggregates the set of intervals within which the predicate is detected, and transmits this aggregated interval to its parent. The aggregated interval is treated as a normal interval at the higher levels in the hierarchy, and is used for detecting the predicate within an even larger subtree. Once an aggregated interval is sent to the parent (higher level process), the parent detects occurrences of the predicate within the larger subtree rooted at itself using aggregated intervals received from its children, and generates the aggregated intervals for its level once a satisfaction of the predicate is detected. Whenever the predicate is detected at some subtree, the root of that subtree will perform the operations necessary for doing repeated detection within that subtree. The same procedure repeats at each level of the hierarchy. At the root of the spanning tree, the predicate is detected for the entire system.

Thus, the difficulties in realizing this algorithm are: (1) how to aggregate a set of intervals, and (2) how to do repeated detection of $Definitely(\Phi)$ using aggregated intervals from a lower level in the hierarchy.

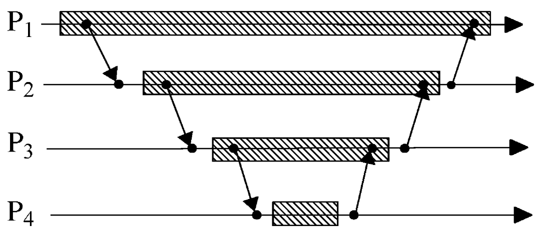

Fig. 2. Approach in [6] works only if the intervals are nested.

In [6], the authors outlined an approach for hierarchical detection of $Definitely(\Phi)$ by trying to address (i) above. But their approach lacks in the following aspects.

1. In [6], the authors assumed a specific partial order in a set of intervals where $Definitely(\Phi)$ is detected. This partial order requires that the intervals in the set can be ordered into $x_1, x_2, \ldots, x_k$ such that $\forall i, j \in [1, k]$, if $i < j$ then $\min(x_i) \prec \min(x_j) \wedge \max(x_j) \prec \max(x_i)$. So their approach requires the set of intervals within which $Definitely(\Phi)$ is detected to be nested as shown in Fig. 2. However, such a relation need not always hold when $Definitely(\Phi)$ is satisfied.
2. The approach in [6] does not do repeated detection. Being able to detect all occurrences of the predicate at each level is essential to hierarchical detection.

We assume the hierarchy is formed as shown in Fig. 3a. From Fig. 3b, observe that the first set of intervals detected at P_2 satisfying $Definitely(\Phi)$ consists of x_1 and x_2, and its aggregation will be sent to the process in the higher level, i.e., P_3. In addition to receiving this solution set, after P_3 receives interval x_5 from P_4 and interval x_4 occurs at P_3, P_3 will start the detection at the higher level. However, $Definitely(\Phi)$ cannot be detected in the set $\{x_1, x_2, x_4, x_5\}$. If only a one-time detection algorithm runs at P_2, which is the case in the approach in [6], then the only set of intervals P_2 ever reports to P_3 is $\{x_1, x_2\}$ and the later occurrence of the predicate for P_1 and P_2 in the set $\{x_1, x_3\}$ will remain undetected. Therefore, the set $\{x_1, x_3, x_4, x_5\}$ within which the predicate could be detected for all 4 processes will never be detected by P_3. Notice that, in this example, no single process is detecting the predicate for the entire system. Only hierarchical detection is performed. This example shows that being able to find *all occurrences* of the predicate at *each level* is necessary to the hierarchical detection algorithm.

Without a proper way to aggregate intervals and without a way to repeatedly detect predicates, the approach given in [6] will fail to detect the predicates at the intermediate nodes as well as at the top of the hierarchy.

3.2 Example Scenario of Our Algorithm

In this subsection, still using Fig. 3, we show how our algorithm handles an example scenario where the user wants to monitor the system for a certain event "$\Phi = \wedge_i Temp_i > 50deg$", where $Temp_i$ is the temperature reading at process P_i.

When $Definitely(\Phi)$ is first detected in $\{x_1, x_2\}$ at P_2 for processes P_1 and P_2, our algorithm will identify one interval from this set such that it will never form part of a future solution set detected by P_2. After identifying such an interval, in this case x_2, P_2 will remove it from its corresponding queue

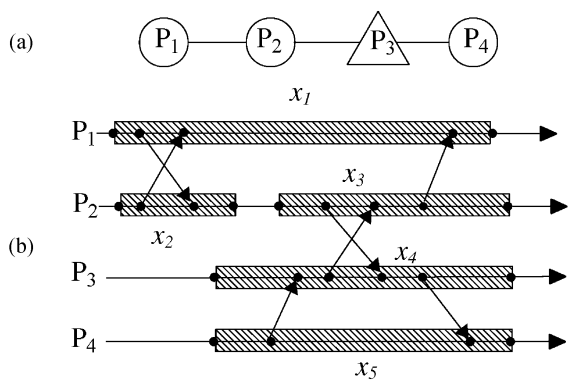

Fig. 3. (a) Spanning tree consists of 4 processes. (b) Timing diagram showing the relation between intervals.

after sending the aggregated interval of set $\{x_1, x_2\}$ to P_3. P_2 then continues the detection for later occurrences of the predicate. When interval x_3 finishes, P_2 will detect a second occurrence of the predicate in the set $\{x_1, x_3\}$ and send another aggregated interval to P_3. At process P_3, after local interval x_4 finishes and P_3 receives intervals from both its children, P_2 and P_4, P_3 will attempt to detect the predicate within the tree rooted at itself. The first attempt will fail, since the set $\{x_1, x_2, x_4, x_5\}$ does not satisfy $Definitely(\Phi)$. As part of this failed attempt, P_3 will remove the aggregation of $\{x_1, x_2\}$ from its queue. When P_3 receives the aggregation of $\{x_1, x_3\}$ from P_2, a second attempt to detect the predicate begins. This time, the predicate is detected in the set $\{x_1, x_3, x_4, x_5\}$. Thus the predicate is detected for all 4 processes. After the first detection at P_3, the algorithm will not hang. Another interval from the solution set will be identified for removal, and the detection will continue running.

From this example, we can observe that the key aspects of our hierarchical detection algorithm lie in

1. the way to aggregate a solution set, and
2. the way to identify at least one interval from a solution set for removal to ensure progress for repeated detection

at *each level* in the hierarchy. In the rest of this section, we will show how we solve these problems.

3.3 Aggregation of Intervals to Detect $Definitely(\Phi)$

In [6], [21], it was shown that for $Definitely(\Phi)$ to hold in a set X of intervals, the following needs to be true

$$\forall x_i, x_j \in X, \min(x_i) < \max(x_j).$$

This property was named as $overlap(X)$. Our objective is to decentralize the detection of $Definitely(\Phi)$. We first consider the scenario where $Definitely(\Phi)$ has been detected in each of the two sets of intervals X and Y and we want to detect $Definitely(\Phi)$ in $X \cup Y$.

Assume now, we have 4 processes in the system with their timing diagram shown in Fig. 4a. The intervals occurring at each process are shaded. The vector clock timestamps identifying the lower and higher bound of each interval are also illustrated in the figure. Intervals x_1 from process P_1 and x_2 from process P_3 form set X, while intervals y_1 and y_2 from process P_2 and P_4, respectively, form set Y. It can be checked that both $overlap(X)$ and $overlap(Y)$ are true.

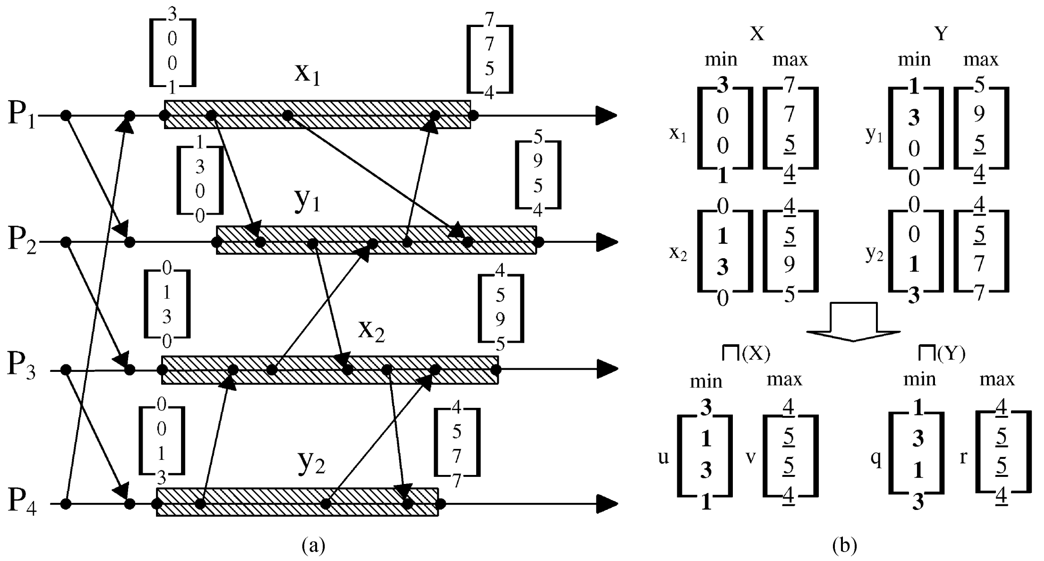

Fig. 4. Example showing the aggregation of intervals for detecting $Definitely(\Phi)$. (a) The timing diagram of the system is given. An interval from each process is marked in shade along with the vector clock timestamps identifying the lower and higher bounds. (b) The two sets of intervals X and Y consisting of intervals from (a) are shown. The way to aggregate each set is also illustrated. Component-wise maximums among all lower bounds in the same set are marked in bold while the component-wise minimums among all higher bounds in the same set are marked in underline.

To show that $Definitely(\Phi)$ is also detected in all 4 processes, equivalently $overlap(X \cup Y)$, we need to show

$$\forall i, j \in \{1,2\}, \min(x_i) < \max(y_j) \wedge \min(y_j) < \max(x_i). \quad (3)$$

From Fig. 4b, we can observe that, if we take the component-wise maximum of $\min(x_1)$ and $\min(x_2)$ (illustrated in bold) to form a new vector u, then the first conjunct in Eq. (3) is equivalent to

$$\forall j \in \{1,2\}, u < \max(y_j). \quad (4)$$

Likewise, taking the component-wise minimum of $\max(y_1)$ and $\max(y_2)$ (illustrated in underline) to form another new vector r, Eq. (4) is equivalent to $u < r$. The same operations can also be applied to show the second conjunct of Eq. (3) using aggregated vectors q and v.

This gives the inspiration to aggregate a set of intervals into a single one to detect $Definitely(\Phi)$ in a larger set of intervals. For sets X and Y in Fig. 4, their aggregated intervals are denoted as $\sqcap(X)$ and $\sqcap(Y)$, respectively. The way to aggregate those two sets using component-wise minimum or maximum is shown in Fig. 4b.

Formally, for an arbitrary set X of intervals, satisfying $overlap(X)$, we define an aggregation function $\sqcap(X)$ of intervals in X, in terms of vector timestamps, as:

$$\forall i \in [1,n], \min(\sqcap(X))[i] = \max_{x\in X}(\min(x)[i]) \quad (5)$$

$$\forall i \in [1,n], \max(\sqcap(X))[i] = \min_{x\in X}(\max(x)[i]). \quad (6)$$

With this formal definition of the aggregation function $\sqcap$, we show the following theorem.

Theorem 1. *Let X, Y and Z be sets of intervals, such that $Z = X \cup Y$. Then $overlap(Z)$ iff $overlap(X) \wedge overlap(Y) \wedge overlap(\sqcap(X), \sqcap(Y))$.*

Proof. ($\Rightarrow$) $overlap(X)$ and $overlap(Y)$ are clearly true since $X, Y \subseteq Z$. Now consider an interval $y \in Y$. Since $overlap(Z)$, $\forall x \in X, \min(x) < \max(y)$. Thus $\min(\sqcap(X)) < \max(y)$. Since this is true for all $y \in Y$, $\min(\sqcap(X)) < \max(\sqcap(Y))$. The same deduction applies to $\min(\sqcap(Y)) < \max(\sqcap(X))$. So, we have $overlap(\sqcap(X), \sqcap(Y))$.

($\Leftarrow$) From $overlap(\sqcap(X), \sqcap(Y))$ we have $\min(\sqcap(X)) < \max(\sqcap(Y)) \wedge \min(\sqcap(Y)) < \max(\sqcap(X))$. For any interval $x \in X$, we have $\min(x) < \min(\sqcap(X))$. For any interval $y \in Y$, we have $\max(\sqcap(Y)) < \max(y)$. Since $\min(\sqcap(X)) < \max(\sqcap(Y))$, we have for any $x \in X$ and any $y \in Y$, $\min(x) < \max(y)$. Similarly, we can deduce that for any $x \in X$ and any $y \in Y$, $\min(y) < \max(x)$. Since we already have $overlap(X)$ and $overlap(Y)$, now we have $overlap(Z)$. □

Theorem 1 shows that we can aggregate a set of intervals X into a single interval $\sqcap(X)$ which can represent the entire set in detecting $Definitely(\Phi)$ within an even larger set of intervals. $\sqcap(X)$ is uniquely identified by $\min(\sqcap(X))$ and $\max(\sqcap(X))$. These are not events but cuts in execution $(E, \prec)$, identified by their vector timestamps.

Theorem 1 only covers the union of two sets of intervals. In the spanning tree, some processes may have more than 2 children. Below, we extend Theorem 1 to scenarios involving more than two sets of intervals.

Lemma 1. *Let $X_1, X_2, \ldots, X_d$ be d sets of intervals, and Z be the union of all d sets. Thus $Z = \cup_{i=1}^{d} X_i$. Then $overlap(Z)$ iff $\wedge_{i=1}^{d} overlap(X_i) \wedge overlap(\sqcap(X_1), \sqcap(X_2), \ldots, \sqcap(X_d))$.*

Proof. ($\Rightarrow$) $\wedge_{i=1}^{d} overlap(X_i)$is clearly true since $X_i \subset Z$. Since $overlap(Z)$, we have $\forall i, j \in [1,d], overlap(X_i \cup X_j)$. Thus, according to Theorem 1, we have $\forall i, j \in [1,d]$, $overlap(\sqcap(X_i), \sqcap(X_j))$. This means, $\forall i, j \in [1,d], \min(\sqcap(X_i)) < \max(\sqcap(X_j))$. Thus, we have $overlap(\sqcap(X_1), \sqcap(X_2), \ldots, \sqcap(X_d))$.

($\Leftarrow$) Since $\wedge_{i=1}^{d} overlap(X_i) \wedge overlap(\sqcap(X_1), \sqcap(X_2), \ldots, \sqcap(X_d))$, we have $\forall i, j \in [1,d], overlap(X_i \cup X_j)$. This means, by picking any two intervals y_1, y_2 from Z, it is always true that $\min(y_1) < \max(y_2)$. This is because there will always be a pair of $i, j \in [1, d]$, such that $y_1 \in X_i$ and $y_2 \in X_j$. So, we have *overlap*(Z). □

For our hierarchical detection algorithm, each process P_i in the spanning tree detects $Definitely(\Phi)$ within the subtree rooted at itself. Once the predicate is detected, P_i aggregates the set of intervals within which the predicate is detected using $\sqcap$ and sends the aggregated interval to its parent. At higher levels in the spanning tree, the predicate within the subtree will be detected based on aggregated intervals received from child processes. Lemma 1 ensures that, by testing the *overlap* property on the aggregated intervals, the predicate can be detected within a larger set of intervals. At higher levels, the aggregation function will also be applied to the aggregated intervals. However, we notice that, for two sets of intervals X and Y,

$$\sqcap(\sqcap(X), \sqcap(Y)) = \sqcap(X \cup Y). \tag{7}$$

So, applying the aggregation function on aggregated intervals is equivalent to applying it on the union of all sets.

3.4 Repeated Detection

In [13], the author showed how repeated detection can be done in the centralized $Definitely(\Phi)$ detection algorithm. Basically, repeated detection requires identifying a certain interval from a solution set such that this single interval cannot be part of a future solution set.

Doing the same in the hierarchical detection algorithm is more complex. In the hierarchical algorithm, the detection takes place at each level. At higher levels, the solution set consists of both aggregated intervals and non-aggregated intervals. Each aggregated interval represents a solution set at the lower level. Identifying a certain interval for removal now is to identify a solution set that cannot be part of a future solution at a higher level, and removing an aggregated interval x in the solution set will remove all the intervals aggregated by x. This is very different from the situation in the centralized algorithm in which the sink only needs to consider non-aggregated intervals. Below, we show how repeated detection can be done in the hierarchical detection algorithm.

First, for aggregated intervals generated at the same process, we have

Theorem 2. *For an aggregated interval* $\sqcap(X)$ *generated at process* P_a *and a later aggregated interval* $\sqcap(X')$ *generated at the same process,* $\min(\sqcap(X)) < \max(\sqcap(X)) < \min(\sqcap(X')) < \max(\sqcap(X'))$.

Proof. Since $\sqcap(X)$ is an aggregated interval, the set of intervals X it aggregates satisfy the condition *overlap*(X). Thus $\forall x_i, x_j \in X, \min(x_i) < \max(x_j)$. Also, according to the definition in (Eqs. (5), (6)), we know that the elements in $\min(\sqcap(X))$ and $\max(\sqcap(X))$ are equal to the component-wise maximum or minimum among all $\min(x_i)$ or $\max(x_i)$, respectively. Since $\forall x_i, x_j \in X, \min(x_i) < \max(x_j)$, we have $\forall x_i, x_j \in X, \forall l \in [1, n], \min(x_i)[l] \le \max(x_j)[l]$. Thus, $\forall l \in [1, n], \min(\sqcap(X))[l] \le \max(\sqcap(X))[l]$. So, $\min(\sqcap(X)) < \max(\sqcap(X))$. The same can also be shown for $\sqcap(X')$.

Since $\sqcap(X')$ is generated after $\sqcap(X)$, it means X' is a solution set within the subtree rooted at P_a that occurs after the solution set X. Thus, there exists at least one interval x'_b in X', such that x'_b occurs after the corresponding interval x_b in X which comes from the same process. So, we have $\max(x_b) < \min(x'_b)$. Also, according to the definition in (Eqs. (5), (6)), we know that $\forall x_i \in X$, $\forall x'_i \in X', \max(\sqcap(X)) < \max(x_i) \wedge \min(x'_i) < \min(\sqcap(X'))$. Thus, $\max(\sqcap(X)) < \min(\sqcap(X'))$. □

For any two intervals x and x' that occur (local intervals) or are generated (aggregated intervals) at the same process, if $\max(x) < \min(x')$, we call x' a successor of x and denote it as $succ(x)$. Theorems 1 and 2 prove that the aggregated intervals are treated just as the non-aggregated intervals at the higher levels in the hierarchy. Now we show how we can identify an interval, aggregated or not, from a solution set such that it can be safely removed.

In order for an interval x_i in a solution set to be part of a future solution set, there needs to be at least one interval x_j from the same solution set, such that *overlap*$(x_i, succ(x_j))$ is true. From [13], we know that this is equivalent to

$$\min\big(succ(x_j)\big) < \max(x_i). \tag{8}$$

Then, if for all intervals $x_j (j \neq i)$ from the solution set X, Eq. (8) is false, we have that *overlap*$(x_i, succ(x_j))$ is false for all $x_j (j \neq i)$ from the solution set. Thus x_i can be safely removed from the head of the queue. So, we have:

$$\text{remove } x_i \text{ iff}$$
$$\forall x_j \in X(j \neq i), \min\big(succ(x_j)\big) \not< \max(x_i). \tag{9}$$

Since $\max(x_j) < \min(succ(x_j))$, from [13], we know the test condition in Eq. (9) can be approximated to

$$\text{remove } x_i \text{ iff } \forall x_j \in X(j \neq i), \max(x_j) \not< \max(x_i). \tag{10}$$

Since we do not know the values in $\min(succ(x_j))$ until that interval gets reported from the lower level, in order to identify the interval for removal as soon as possible, the approximated condition in Eq. (10) is what we use to prune the queues. Although it is only an approximation, we now show that it is actually correct and capable of always identifying at least one interval for removal.

Theorem 3 (Safety). *Once a solution set* X *is detected at any process in the hierarchy, only intervals* $x_i \in X$ *(*x_i *may be aggregated or not) that cannot be part of another solution are removed from their queues.*

Proof. Since Eq. (10) ⇒ Eq. (9), any interval removed using the condition in Eq. (10) will also satisfy the condition in Eq. (9). Thus, those intervals cannot be part of any future solution set. Therefore, even if Eq. (10) is only an approximation, it still guarantees safety. □

Theorem 4 (Liveness). *For any solution set* X *detected at any process in the hierarchy, at least one interval (aggregated or not) gets removed from its queue.*

Proof. Assume that the condition in Eq. (10) cannot be satisfied by some solution set X. Then, it means that for any intervals $x_i \in X$, aggregated or not, there exists another interval $x_j \in X$, such that $\max(x_j) < \max(x_i)$. This condition will eventually cause one interval x_k to satisfy $\max(x_k) < \max(x_k)$, which is impossible. So the assumption is false, and thus the condition in Eq. (10) holds for any solution set. Thus, Eq. (10) guarantees liveness.

Algorithm 1: Hierarchical decentralized detection of conjunctive definitely predicates, adapted from [13] (Code for P_i)

```
number of children: $l$
queue for $P_i$: $Q_0 \leftarrow \perp$
queues for children: $Q_1, Q_2, \ldots, Q_l \leftarrow \perp$
set of int: updatedQueues, newUpdated $\leftarrow \{\}$
int: count
On receiving an interval from child $P_j$ at $P_i$:
1  Enqueue the interval onto queue $Q_j$;
2  if number of intervals on $Q_j$ is 1 then
3      updatedQueues = {j};
4      while updatedQueues is not empty do
5          newUpdated = {};
6          for each $a \in updatedQueues$ do
7              if $Q_a$ is not empty then
8                  $x =$ head of $Q_a$;
9                  for $b = 0 \ldots l (b \neq a)$ do
10                     if $Q_b$ is not empty then
11                         $y =$ head of $Q_b$;
12                         if $\min(x) \nless \max(y)$ then
13                             add $b$ to newUpdated;
14                         if $\min(y) \nless \max(x)$ then
15                             add $a$ to newUpdated;
16         Delete heads of all $Q_c$ where $c \in newUpdated$;
17         updatedQueues = newUpdated;
18         if all queues are non-empty $\wedge$ $updatedQueues = \emptyset$ then
19             if $P_i$ has parent in the spanning tree then
20                 report $\sqcap$(heads of all queues) to parent;
21             else
22                 report predicate detected;
23             for $a = 0 \ldots l$ do
24                 count = 0;
25                 for $b = 0 \ldots l (b \neq a)$ do
26                     for $c = 1 \ldots n$ do
27                         if $\max(head(Q_a))[c] < \max(head(Q_b))[c]$ then
28                             $count++$;
29                             break;
30                 if $count = l$ then
31                     add $a$ to newUpdated;
32             Delete heads of all $Q_a$ where $a \in newUpdated$;
33             updatedQueues = newUpdated;
```

With the safety and liveness of the condition in Eq. (10) proved, we can safely use it to prune the queues so that future occurrences of the predicate at each level in the hierarchy can be repeatedly detected. □

3.5 Hierarchical Detection Algorithm

With Theorems 1, 3 and 4, we have the theoretical foundation for the hierarchical detection algorithm outlined in Section 3.1. The algorithm is listed in Algorithm 1. Each process in the spanning tree tracks the intervals occurring locally and those sent from its children. The intervals sent from a child process can be non-aggregated intervals or aggregated ones, depending on whether the child is a leaf node. By checking the intervals received in the queues (Lines (1)-(17)), each process attempts to detect the predicate within the subtree rooted at itself. Once a solution set is found (Line (18)), the root of the subtree aggregates the set and sends it to its parent (Lines (19)-(20)). At the higher level in the hierarchy, the parent determines if the predicate can be detected in an even larger subtree rooted at itself by repeating the same detection procedure (Lines (1)-(17)). When the root of the spanning detects a solution set, a satisfaction of the predicate is detected within the whole system (Lines (21)-(22)).

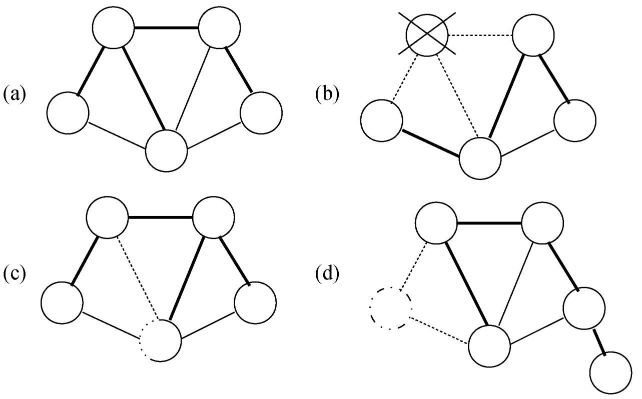

Fig. 5. (a) Initial topology of the spanning tree. Bold lines indicate tree edges, while dashed lines indicate disconnected tree edges due to either (b) node crash, (c) decreasing power, or (d) node mobility. In order to reconstruct the spanning tree, new tree edges are added.

Each time the predicate is detected at some process, Lines (23)-(33) prune the heads of the queues so that future occurrences of the predicate at the same level can be repeatedly detected. For each interval x_i in the solution set X, this procedure checks x_i against all other intervals x_j in X to see if $\forall x_j \in X (j \neq i), \max(x_j) \nless \max(x_i)$. Each time an interval x_i is to be checked, a counter is initialized to 0. For each interval $x_j (j \neq i)$, if $\max(x_j) \nless \max(x_i)$ then the counter is increased by 1. After x_i is checked against all other intervals x_j, if the counter equals l, which is the total number of intervals in the solution set X minus 1, then interval x_i satisfies the condition in Eq. (10). Thus, we can safely remove x_i from the corresponding queue. In Algorithm 1, the intervals to be processed can be aggregated intervals. Thus when comparing the vector timestamps of two intervals (Lines (12), (14), (26)-(27)), we cannot compare them in $O(1)$ time as we can do with the normal intervals. This will affect the time complexity of this algorithm. For details, please refer to the supplementary file which is available online.

Although Algorithm 1 has the same basic structure as the centralized algorithm given in [13], it is essentially different. Algorithm 1 detects $Definitely(\Phi)$ in a hierarchical manner and performs tests on aggregated intervals. Instead of one central server process maintaining n queues, each process in Algorithm 1 maintains queues only for itself and its children in the spanning tree. When the predicate is detected at non-root processes, the solution set is aggregated for processes in the higher level to detect the predicate in a larger area.

To summarize, Theorems 1, 3, and 4 together guarantee that Algorithm 1 is correct, meaning that all the predicate occurrences in the system are detected and there are no false alarms.

4 Fault-Tolerance

4.1 Potential Failures In the System

In a large-scale wireless network, the following situations could result in changes to the spanning tree. 1) A node crashes and

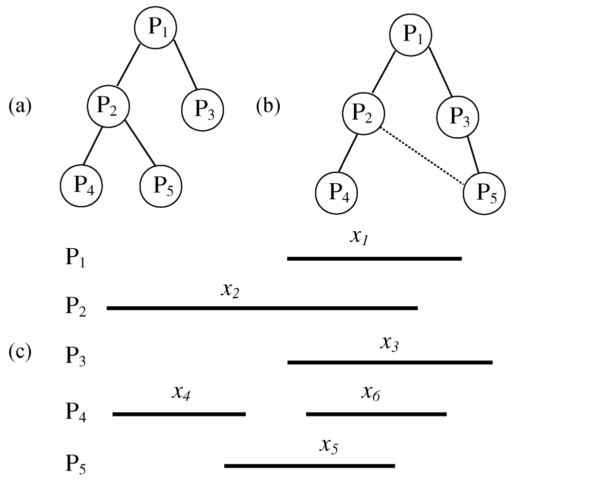

Fig. 6. (a) Spanning tree consists of 5 processes. (b) Due to either node mobility or decreasing power of process P_5, the structure of the spanning tree is changed. (c) Intervals on each process are represented by line segments.

Algorithm 2: Hierarchical Detection with Fault-Tolerance (Code for P_i)

```
Interval: $previous \leftarrow \perp$
set of boolean: $flag$
$P_i$ loses a child $P_j$:
1  $flag_j =$ true;
$P_i$ gains a new child $P_k$:
2  if $Q_k$ exists then
3      $flag_k =$ false;
4  else
5      create a local queue $Q_k$ initialized to $\perp$;
On receiving the first interval $x_k$ from newly added child $P_k$:
6  if $overlap(x_k, previous)$ then
7      send $\sqcap(x_k, previous)$ to $parent$;
Each time $P_i$ reports an interval $x$ to $parent$:
8  $previous = x$;
Each time $P_i$ removes an interval from local queue $Q_j$:
9  if $flag_j$ and $Q_j$ is empty then
10     remove $Q_j$;
```

loses all the communication link with its previous neighbors. 2) A node moves out of the communication range of some or all of its previous neighbors. 3) A node reduces communication range due to loss of power. These situations are illustrated in Fig. 5. In a large-scale wireless network, these situations can be common, especially if individual node only has limited power and is prone to crash failures. To handle these situations, a mechanism to reconstruct the spanning tree after the node failures is required. In this paper, we do not consider the problem of reconstructing the degree-bounded spanning tree after node failures. We expect some spanning tree reestablishment mechanism at a lower layer to handle the reconstruction of the degree-bounded spanning tree. This problem is universal, ranging from electric power transmission grids to LAN configurations to higher-layer tree overlays. Readers are referred to the reconstruction approaches by Yang and Fei [23] and Jeon *et al.* [24]. We instead focus on how the hierarchical detection algorithm should behave to seamlessly transition across the changes in the underlying spanning tree caused by the reconstruction mechanism. We assume that failures do not occur during message transmissions. This assumption is reasonable because, compared with the duration of intervals, the time to transmit a message is usually much less.

No matter which of the above listed three situations happens, their effects to individual nodes in the spanning tree after the reconstruction mechanism takes place can be categorized into one or more of the following three changes:

1. The node loses one child.
2. The node gains an additional child.
3. The node's parent gets switched.

Thus in order for our algorithm to correctly resume the detection, meaning no missed predicate occurrences, it needs to be able to handle these listed changes at all the nodes that are affected by the reconstruction mechanism.

4.2 Dealing with Changes in the Spanning Tree

The main challenge is to prevent missed predicate occurrences. It is possible to miss a predicate occurrence in 2 ways: when the changes to a node happen before the node's local interval finishes and when the changes happen after the local interval finishes.

Take the example in Fig. 6 as an illustration. The messages are not drawn in the timing-diagram for clarity. If two intervals satisfy the *overlap* relation, they appear as overlapped in the horizontal direction in Fig. 6c. The spanning tree is initially constructed as shown in Fig. 6a. The failure happens, say for instance, when process P_5 moves away to become P_3's child. This triggers the spanning tree reconstruction mechanism and the reconstructed spanning tree is shown in Fig. 6b.

If this change happens after interval x_5 finishes, the reconstruction affects processes P_2, P_3, and P_5. For P_2, it loses one child, and the interval x_5 received by P_2 now has no corresponding child process. For P_3, it gains a new child. However, there is no corresponding interval queue in P_3 to accept intervals from P_5. In addition, the intervals P_3 later reports to P_1 is the aggregation of intervals from both P_3 and P_5. Compared with the intervals P_1 receives from P_3 before the spanning tree changes, the new intervals from P_3 now represent a different set of processes: $\{P_3, P_5\}$. For P_5, it now has a different parent process. Thus, it needs to report its intervals to the new parent. These effects on processes P_2, P_3 and P_5 resulting from the spanning tree reconstruction could all potentially cause missed predicate occurrences.

On the other hand, P_5 and P_2 could be disconnected before interval x_5 finishes. In this scenario, the aggregated intervals reported by P_2 do not contain interval x_5. If P_3 reports interval x_3 to P_1 before P_5 becomes P_3's child, then it is possible that P_1 will not receive interval x_5 as part of any aggregated intervals and thus miss the detection of the predicate in the set $\{x_1, x_2, x_3, x_5, x_6\}$.

With the above two possible ways in which the changes in the spanning tree can affect the detection algorithm, we observe that our algorithm's hierarchical detection manner can help to seamlessly transition across the changes and guarantee the correctness.

Our proposed solution works as follows:

1. Whenever a process P_i loses a child P_j, P_i flags the queue corresponding to P_j as lost. However, P_i does

not remove this queue until all the remaining intervals in the queue are processed and the queue becomes empty. If P_j once again becomes P_i's child before the queue turns empty, P_i removes the flag.

2. When P_i gains a new child P_k, P_i needs to create a new queue to process intervals from P_k. Also, when P_i receives the first interval from P_k, it needs to check whether this interval overlaps with the most recent interval reported by P_i to its parent. If so, P_i needs to send a new aggregated interval to its parent.
3. When P_i switches parent, P_i reports subsequent intervals to the new parent.
4. When P_i's child P_j sends intervals that represent a different set of processes, P_i appends the new intervals into the same queue corresponding to P_j.

4.3 Algorithm Augmentation for Fault-Tolerance

The addition of fault-tolerance to the hierarchical detection algorithm is listed in Algorithm 2. After the spanning tree is changed, each process that has its parent or children changed in the reconstructed spanning tree needs to update its local queues accordingly (Lines 1-7). Each process also needs to remember the most recent reported interval in order to guarantee all predicate occurrences are detected (Line 8).

Although the structure of the spanning tree changes, due to the fact that the hierarchical manner of the algorithm stays the same, it will not affect the detection of the future occurrences of the predicate within the system. Furthermore, Theorems 1, 3, and 4 still ensure the correctness of further detections of the predicate.

Even if multiple occurrences of the failures in Section 4.1 happen concurrently, our algorithm is capable of resuming the detection after the spanning tree is locally reconstructed for each such failure. This is because, in Algorithm 2, each process P_i only needs to check the changes in its children and parent after the spanning tree reconstruction. Thus, each process P_i only needs to react to local changes.

Below, we show that Algorithm 2 guarantees all predicate occurrences are detected with the presence of potential failures happening during the detection. Notice that, we only show that this statement is guaranteed for the entire system, not for every subset of processes grouped by a subtree in the system. This is because, with fault-tolerance considerations, the spanning tree may keep changing. Thus, any subtree in the system may exist only temporarily. Requiring the detection of all occurrences of the predicate within any subtree while the subtree itself could change at any time is thus impractical.

Theorem 5 (Completeness). *Any occurrence of the predicate for the entire system will be detected by some solution set X at the root process.*

Proof. With fault-tolerance handling, Theorem 1 is not impacted because the logic to aggregate intervals stays the same. Theorem 2 will also not be affected because we do not remove intervals even if the corresponding process disconnects. Thus, even with potential nodes leaving and joining, it is always true that for an aggregated interval $\sqcap(X)$ and a later aggregated interval $\sqcap(X')$ generated at the same process, we can find a interval x'_b in X' and the corresponding entry x_b in X such that $\max(x_b) < \min(x'_b)$. With Theorem 2 not affected, Theorems 3 and 4 will also hold.

Furthermore, if process P_i switches parent before its local interval finishes, we also make sure P_i's new parent will process this local interval if it can be part of a global solution set. Thus, if interval x_i from P_i is part of a global solution set, x_i will become part of an aggregated interval and eventually reach the root process where the occurrence of the predicate will be detected. □

With Theorems 1, 3, 4, and 5, it is guaranteed that our proposed fault-tolerance handling will ensure the correctness of the detection algorithm when transitioning across changes of the underlying spanning tree.

5 Discussion

In this paper, we proposed the first decentralized hierarchical algorithm that repeatedly detects all occurrences of $Definitely(\Phi)$ for a conjunctive predicate Φ. Such an algorithm is essential for large-scale systems, particularly when the system is subject to node crashes. Our algorithm detects the predicate at each level in the hierarchy, and thus is able to detect a partial predicate of the global predicate. This enables our algorithm to easily resume the detection after a node crashes or moves. Compared with the only other algorithm capable of doing repeated detection [13], our algorithm distributes a lower time cost, and the same space cost, across all processes in the network, and reduces the number of control messages significantly. A detailed complexity analysis is given in the supplementary file which is available online and [15].

References

[1] M. Shen, A.D. Kshemkalyani, and A. Khokhar, "Detecting Stable Locality-Aware Predicates," *J. Parallel Distrib. Comput.*, vol. 74, no. 1, pp. 1971-1983, Jan. 2014.

[2] M. De-Rosa, S. Goldstein, P. Lee, P. Pillai, and J. Campbell, "Programming Modular Robots with Locally Distributed Predicates," in *Proc. IEEE ICRA*, 2008, pp. 3156-3162.

[3] M. De-Rosa, S. Goldstein, P. Lee, J. Campbell, and P. Pillai, "Detecting Locally Distributed Predicates," *ACM Trans. Autonom. Adapt. Syst.*, vol. 6, no. 2, p. 13, June 2011.

[4] C.M. Chase and V.K. Garg, "Detection of Global Predicates: Techniques and Their Limitations," *Distrib. Comput.*, vol. 11, no. 4, pp. 191-201, Oct. 1998.

[5] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," in *Proc. ACM/ONR Workshop Parallel Distrib. Debugging*, 1991, pp. 167-174.

[6] V.K. Garg and B. Waldecker, "Detection of Strong Unstable Predicates in Distributed Programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 12, pp. 1323-1333, Dec. 1996.

[7] V.K. Garg and B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 3, pp. 299-307, Mar. 1994.

[8] A. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, Systems*. Cambridge, U.K.: Cambridge Univ. Press, 2008.

[9] V. Garg and C. Chase, "Distributed Algorithms for Detecting Conjunctive Predicates," in *Proc. 15th IEEE ICDCS*, 1995, pp. 423-430.

[10] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal, "Efficient Distributed Detection of Conjunctions of Local Predicates," *IEEE Trans. Softw. Eng.*, vol. 24, no. 8, pp. 664-677, Aug. 1998.

[11] P. Chandra and A. Kshemkalyani, "Distributed Algorithm to Detect Strong Conjunctive Predicates," *Inf. Process. Lett.*, vol. 87, no. 5, pp. 243-249, Sept. 2003.

[12] P. Chandra and A. Kshemkalyani, ''Data Stream Based Global Event Monitoring Using Pairwise Interactions,'' *J. Parallel Distrib. Comput.*, vol. 68, no. 6, pp. 729-751, June 2008.

[13] A.D. Kshemkalyani, ''Repeated Detection of Conjunctive Predicates in Distributed Executions,'' *Inf. Process. Lett.*, vol. 111, no. 9, pp. 447-452, Apr. 2011.

[14] F. Gärtner, *A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms, Swiss Federal Inst. Technol.(EPFL), Lausanne, Switzerland*, 2003.

[15] M. Shen and A. Kshemkalyani, ''A Fault-Tolerant Strong Conjunctive Predicate Detection Algorithm for Large-Scale Networks,'' in *Proc. 27th IEEE IPDPS Workshops*, 2013, pp. 1460-1469.

[16] L. Lamport, ''Time, Clocks, the Ordering of Events in a Distributed System,'' *Commun. ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

[17] M. Raynal and M. Singhal, ''Logical Time: Capturing Causality in Distributed Systems,'' *Computer*, vol. 29, no. 2, pp. 49-56, Feb. 1996.

[18] F. Mattern, ''Virtual Time and Global States of Distributed Systems,'' in *Proc. Parallel Distrib. Algorithms Conf.*, 1988, pp. 215-226.

[19] C. Fidge, ''Logical Time in Distributed Computing Systems,'' *Computer*, vol. 24, no. 8, pp. 28-33, Aug. 1991.

[20] R. Baldoni and M. Raynal, ''Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems,'' *IEEE Distrib. Syst. Online*, vol. 3, no. 2, pp. 1-17, Feb. 2002.

[21] A. Kshemkalyani, ''Temporal Interactions of Intervals in Distributed Systems,'' *J. Comput. Syst. Sci.*, vol. 52, no. 2, pp. 287-298, Apr. 1996.

[22] A. Kshemkalyani, ''Immediate Detection of Predicates in Pervasive Environments,'' *J. Parallel Distrib. Comput.*, vol. 72, no. 2, pp. 219-230, Feb. 2012.

[23] M. Yang and Z. Fei, ''A Proactive Approach to Reconstructing Overlay Multicast Trees,'' in *Proc. 23rd INFOCOM*, 2004, pp. 2743-2753.

[24] J. Jeon, S. Son, and J. Nam, ''Overlay Multicast Tree Recovery Scheme Using a Proactive Approach,'' *Comput. Commun.*, vol. 31, no. 14, pp. 3163-3168, Sept. 2008.

Min Shen holds a BS in computer science from Nanjing University. He is currently a PhD student at the Department of Computer Science at the University of Illinois at Chicago, USA. His research interests include distributed algorithms, predicate detection and wireless sensor networks. He is a Student Member of the IEEE.

Ajay D. Kshemkalyani received the BTech degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987, and the MS and PhD degrees in computer and information science from The Ohio State University, USA in 1988 and 1991, respectively. He spent six years at IBM Research Triangle Park working on various aspects of computer networks before joining academia. He is currently a professor at the Department of Computer Science at the University of Illinois at Chicago, USA. His research interests are in distributed computing, distributed algorithms, computer networks, and concurrent systems. In 1999, he received the US National Science Foundation Career Award. He previously served on the editorial board of the Elsevier journal *Computer Networks*, and is currently an editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has coauthored a book entitled *Distributed Computing: Principles, Algorithms, and Systems* (Cambridge University Press, 2008). He is a distinguished scientist of the ACM and a Senior Member of the IEEE.