

Predicate Detection in Large-Scale Locality-Driven Networks

BY

Min Shen

B.E. (Nanjing University, China) 2009

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2014

Chicago, Illinois

Defense Committee:

Ajay Kshemkalyani, Chair and Advisor

Ashfaq Khokhar

Ugo Buy

Venkat Venkatakrishnan

Kunpeng Zhang, Information and Decision Science

Copyright by

Min Shen

2014

To my wife,

Lulu Zhang,

for accompanying me along this journey called life.

ACKNOWLEDGMENTS

I want to thank my advisor Prof. Ajay Kshemkalyani for his tremendous support and assistance. He has gone a long way to provide me with valuable help especially when I needed it most. His guidance has lighted the path for me to accomplish my research goals. I would like to express my deep gratitude towards all his contributions of time and ideas.

I would like to also thank my other committee, Prof. Ashfaq Khokhar, Prof. Ugo Buy, Prof. Venkat Venkatakrishnan, and Prof. Kunpeng Zhang for their support and insights towards my research.

Last but not least, I also want to thank my family for showing great patience and providing comfort during the most difficult time. Their support is what makes this thesis possible.

MS

TABLE OF CONTENTS

<u>CHAPTER</u>		<u>PAGE</u>
1	INTRODUCTION	1
1.1	Motivation	1
1.2	Related Work	8
1.3	Goals and Contributions	13
1.3.1	Goals	13
1.3.2	Contributions	17
2	SYSTEM MODEL AND BACKGROUND	22
2.1	System Model	22
2.2	Background	25
2.2.1	Stable/Unstable Predicates	26
2.2.2	<i>Possibly/Definitely</i> Time Modalities	27
2.2.3	Conjunctive/Relational Predicates	31
2.2.4	Existing Detection Algorithms	34
2.2.4.1	Detecting Stable Predicates Using Snapshots	34
2.2.4.2	Unstable Predicate Detection Algorithms	35
3	DETECTING STABLE LOCALITY-AWARE PREDICATES . .	38
3.1	Locality-Aware Predicates	39
3.1.1	Motivation	39
3.1.2	Detecting Locality-Aware Predicates	40
3.2	Modeling Area of Interest	44
3.3	Consistent Sub-cut Construction	52
3.4	Detecting Locality-Aware Predicates	59
3.4.1	Formal Definition	59
3.4.2	Stable LAP Detection Algorithm	60
3.5	Complexity Analysis	64
3.5.1	Algorithm 1 (Local BFST)	64
3.5.2	Algorithm 2 (Consistent Sub-cut)	66
3.5.3	Algorithm 3 (Stable LAP)	67
3.5.4	Comparison with Other Algorithms	68
3.6	Special Cases	69
4	DETECTING UNSTABLE LOCALITY-AWARE PREDICATES	71
4.1	Detecting Unstable Conjunctive LAP	71
4.1.1	Establishing the Regional Vector Clock	71
4.1.2	Detecting Unstable Conjunctive LAP	74

TABLE OF CONTENTS (Continued)

<u>CHAPTER</u>		<u>PAGE</u>
4.1.3	Ticking at Relevant Communication Events	76
4.2	Encoded Vector Clock (EVC) Optimization	78
4.2.1	Encoded Vector Clock Operations	79
4.2.2	Complexity	83
4.2.3	Resetting EVC	85
5	HIERARCHICAL REPEATED DETECTION	87
5.1	Hierarchical Detection	88
5.1.1	Basic Idea and Challenges	88
5.1.2	Example Scenario of Our Algorithm	91
5.1.3	Aggregation of Intervals to Detect <i>Definitely</i> (Φ)	92
5.1.4	Repeated Detection	97
5.1.5	Hierarchical Detection Algorithm	100
5.2	Fault-Tolerance	103
5.2.1	Potential Failures In the System	103
5.2.2	Dealing with Changes In the Spanning Tree	105
5.2.3	Algorithm Augmentation for Fault-Tolerance	107
5.3	Complexity	110
5.3.1	Message Complexity	110
5.3.2	Space Complexity	117
5.3.3	Time Complexity	118
5.3.4	Cost of Maintaining the Spanning Tree	119
6	INSTANTANEOUS DETECTION	121
6.1	Instantaneously Modality	121
6.2	Detecting Predicates in Physical Time	123
6.2.1	Basic Ideas	123
6.2.2	Hierarchical Detection	124
6.2.3	Repeated Detection for <i>Instantaneously</i> (δ, ϕ)	129
6.2.4	Detection Pruning Technique	132
6.2.5	Integrated Detection Algorithm	135
6.2.6	Complexity Analysis of the Detection Algorithm	136
6.3	Evaluation	139
6.3.1	Simulator Design	139
6.3.2	Simulation Parameters	142
6.3.3	Simulation Result	143
7	CONCLUSION AND FUTURE WORK	147
	APPENDICES	150
	Appendix A	151
	Appendix B	153

TABLE OF CONTENTS (Continued)

<u>CHAPTER</u>	<u>PAGE</u>
CITED LITERATURE	155
VITA	160

LIST OF TABLES

<u>TABLE</u>		<u>PAGE</u>
I	COMPARISON OF FEATURES OF ALGORITHMS FOR DETECTING STABLE PREDICATES IN LARGE-SCALE DISTRIBUTED SYSTEMS.	18
II	COMPARISON OF FEATURES OF ALGORITHMS FOR DETECTING UNSTABLE PREDICATES.	20
III	COMPLEXITY EVALUATION OF STABLE LAP ALGORITHMS IN A DEGREE- D BOUNDED NETWORK.	65
IV	COMPLEXITY COMPARISON BETWEEN LAP ALGORITHMS AND SOME EXISTING ALGORITHMS TO DETECT STABLE PREDICATES.	68
V	CORRESPONDENCE BETWEEN VECTOR CLOCKS AND EVC	81
VI	COMPARISON OF THE TIME AND SPACE COMPLEXITY OF THE THREE BASIC OPERATIONS	84
VII	COMPLEXITY COMPARISON BETWEEN HIERARCHICAL DETECTION AND THE CENTRALIZED REPEATED DETECTION ALGORITHM	111

LIST OF FIGURES

<u>FIGURE</u>		<u>PAGE</u>
1	Example to illustrate $Possibly(\Phi)$ and $Definitely(\Phi)$. (a) The timing diagram of the execution. (b) The state lattice for the execution.	29
2	Example to illustrate the conditions for $Definitely(\Phi)$ and $\neg Possibly(\Phi)$ for two processes.	33
3	Four types of messages.	43
4	Illustration of covering an area centered at P_0 with radius 2. Bold edges are tree edges. (a) A spanning tree might exclude some processes such as x and y because it does not find shortest paths (b) A local BFST includes all the processes.	45
5	Illustration of dynamic changes to <i>child_list</i> . (a) Initially P_j is P_i 's child, with P_i 's <i>dist</i> being 5 and P_j 's <i>dist</i> being 6. (b) P_j discovers a shorter path with <i>dist</i> being 5 and sends a <i>length</i> message to P_i . (c) P_j discovers a shorter path with <i>dist</i> being 4 and sends a <i>length</i> message to P_i . (d) P_j discovers a shorter path with <i>dist</i> being 3 and sends a <i>length</i> message to P_i . In (b) and (c), P_i can discover P_j is no longer its child and can safely remove P_j from its <i>child_list</i> in line (18). In (d), P_i becomes P_j 's child and resets its <i>child_list</i> in line (9).	51
6	Illustration of the “relevant event” test. Each process has local variables a , b , and c . The conjunctive predicate $\psi = a_1 > 3 \wedge b_2 = 8$ 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.	77
7	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. .	82

LIST OF FIGURES (Continued)

<u>FIGURE</u>		<u>PAGE</u>
8	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.	83
9	The approach in (17) works only if the intervals are nested.	89
10	(a) The spanning tree consists of 4 processes. (b) Timing diagram showing the relation between intervals.	90
11	Example showing the aggregation of intervals for detecting <i>Definitely</i> (Φ). (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.	93
12	(a) The 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.	104
13	(a) The 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.	106
14	Message complexity comparison between hierarchical and centralized detection, with $d = 2, p = 20$	113
15	Message complexity comparison between hierarchical and centralized detection, with $d = 4, p = 20$	114
16	Message complexity comparison of hierarchical detection within networks of the same size but with different spanning tree topologies. Note that $p = 20$	115

LIST OF FIGURES (Continued)

<u>FIGURE</u>		<u>PAGE</u>
17	Message complexity comparison between hierarchical detection and centralized detection. (a) $d = 2, h = 12$ (b) $d = 8, h = 4$. Note that $p = 20$	116
18	(a) The timing diagram of the execution. (b) The state lattice for the execution. Bold edges indicate current run δ	122
19	Examples showing the aggregation of intervals for detecting <i>Possibly</i> (ϕ). In (a), the timing diagram of the system is given. Intervals from each process is marked in shade along with the vector clock timestamps identifying the lower and higher bounds. In (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 in (b). Notice that, component-wise maximum among all lower bounds in the same set are marked in bold while the element corresponding to <i>ORIG</i> (x_i) or <i>ORIG</i> (y_i) for each interval x_i and y_i is marked in bold and italic.	125
20	Example illustrating the repeated detection technique in the context of aggregated intervals. (a) Timing diagram showing 4 processes with their corresponding intervals illustrated in shade. (b) Spanning tree consisting of these 4 processes establishes a hierarchy in the network. Note that dotted line indicates an interval start marker, thin straight line indicates an unaggregated interval, and the bold straight line indicates an aggregated interval. Only relevant ones are drawn.	130
21	By sending markers right after local interval starts, our detection algorithm is able to prune false positives. (a) P_2 and P_3 send markers to P_1 right after their local intervals start. (b) P_2 and P_3 send intervals to P_1 for detection. (c) P_1 's <i>Arriv</i> queue maintaining the orders in which intervals and interval start markers are received at P_1 . Circled entries represent the interval start markers of the corresponding intervals. Since interval x_1 's entry appears before interval x_2 and x_3 's interval start markers, P_1 deduces that x_1 does not overlap with x_2 or x_3	134
22	Architecture of the simulator.	140
23	Showing accuracy of the detection method with (a) increasing message transmission time(here $n = 100, t = 1200s, miet = 10ms$), (b) increasing mean inter-event time(here $n = 100, t = 1200s, mtt = 20ms, lid = 20s, hid = 30s$).	145

LIST OF ABBREVIATIONS

LAP	Locality-Aware Predicates.
BFST	Breadth-First Search Tree.
EVC	Encoded Vector Clock.

SUMMARY

Global predicate detection is an important problem in many distributed systems. It aims at detecting whether a certain distributed property has been satisfied during the system execution. The traditional algorithms to solve this problem are based on logical clocks. They explore the causal relationship between events occurring during the execution of the distributed program. However, this also makes these algorithms not suitable to detect global properties in a physical time and dynamic fashion. Furthermore, in arising new types of distributed systems, such as large-scale wireless sensor networks and modular robotics, where events are locality driven and the scale of the system is large with individual process having limited computation resources, the traditional algorithms simply cannot handle the complexities.

To address these issues, we first propose the concept of locality-aware predicate (LAP) which aims at detecting predicates within a local region. Although a LAP detects predicates only within a certain local region, observing the local area consistently requires considering the entire system in a consistent manner. This raises the challenge of making the complexities of the corresponding predicate detection algorithms scale-free. We design algorithms for detecting both stable and unstable LAPs and show that our algorithms are scale-free.

We also propose the methodology of hierarchical repeated detection. This approach of detecting predicates relies only on neighborhood communications to collaboratively detect global predicates. Compared with the traditional predicate detection algorithms, hierarchical detection incurs a much smaller cost which is distributed across all nodes in the system, and is

SUMMARY (Continued)

capable of continuing the detection even when individual node crashes during the detection. The performance evaluation of our proposed algorithm shows its great potential for detecting predicates in networks where individual process only has limited computation resources and is prone to crash failure.

Furthermore, we propose the Instantaneously detection algorithm which is built on top of the hierarchical detection methodology. We combine several approximation techniques with the hierarchical detection method to make the algorithm capable of detecting predicates in physical time even when synchronized physical clock is not available in the system. Since the detection is achieved via neighborhood communication only, this algorithm can detect predicates in physical time with a high accuracy while still relies solely on logical clocks.

CHAPTER 1

INTRODUCTION

1.1 Motivation

A distributed system is a collection of independent entities interconnected via a communication network that cooperate with each other to solve problems. Being characterized as a collection of autonomous processes communicating over a communication network, a distributed system has the following unique features (1):

- **No common physical clock:** Although each process might have its own physical clock, there is no way to perfectly synchronize the physical clocks of all processes in the distributed system. Thus, asynchrony amongst processes is unavoidable in a distributed system.
- **No shared memory:** Being connected via a communication network, the only way to share information with other processes in a distributed system is through message passing. This feature also implies the absence of common physical clock.

In the past few decades, distributed systems have gained wide applications. The motivation for using distributed systems involves some or all of the followings (1):

- **Inherently distributed computations:** For some applications, such as a server/client application or money transferring applications, the underlying computation is inherently distributed.

- **Resource sharing:** In some applications, such as a large-scale distributed database, it can be very expansive or even impractical to replicate the data on all hosts. Furthermore, storing all the data on a single host can also incur heavy costs. For such applications, dividing the data into smaller chunks and distributing them onto multiple hosts is necessary.
- **Accessing geographically remote data:** In order for a process to access data that is not local, message passing between the two hosts to deliver the data is necessary. Furthermore, with the advances in the design of mobile devices and in the wireless communication technique, systems such as Wireless Sensor Networks (WSNs) emerge which can share data that is geographically distributed within the network.
- **Enhanced reliability:** Because of the geographical distribution of processes as well as the potential replication of data and executions, the failure of a single host has very limited impact on the entire network. The distributed system inherently provides increased reliability.
- **Increased performance/cost ratio:** The cooperation between processes in the distributed system partitions a task across various processes in the network. This feature as well as the sharing of data and resource increase the performance/cost ratio for a distributed system.

A major problem in reasoning with the execution of a distributed system is the detection of distributed properties. Detecting such properties over a distributed execution is important for various purposes such as monitoring, synchronization, coordination, and debugging of a

distributed system. However, the difficulty of observing the states of a distributed system is very high. This is because of the inherent features of a distributed system, i.e., the absence of common physical clocks and shared memories. Arisen from these features, the asynchrony of the message transmission and local executions in a distributed system present challenges to observing its state. Furthermore, due to the concurrent execution at each process and the varying communication delays, different executions of the same distributed program can generate different sequences of global states. This non-determinism further adds to the complexity of detecting distributed properties.

To solve this problem, many distributed predicate detection algorithms have been proposed. Predicate detection is to specify a statement on the variables local to different processes in the distributed system and aims at detecting whether this statement becomes true during the execution of the distributed system. For example, deadlock detection is a predicate detection problem. A deadlock represents a system state in which a subset of the processes are blocked on one another, each waiting for a resource to be released by another. The wait-for relation can be represented by a graph and a deadlock indicates that there is a cycle in the graph. This condition can be specified using local variables indicating the wait-for relation. Another example of predicate detection is the mutual exclusion problem. The mutual exclusion problem requires that at any time the number of processes in the critical section is no more than 1. This condition can also be specified using local variables indicating whether a process is in the critical section. Thus, it can be monitored by a predicate detection algorithm. A predicate can be stable or unstable. A stable predicate stays true once it becomes true; the unstable

predicate does not have such a property. The deadlock detection problem is a stable predicate while the mutual exclusion problem is an unstable predicate.

In recent years, new types of distributed systems have emerged in areas such as wireless sensor networks (WSNs) and modular robotics. These emerging areas share some common properties such as large scale and being locality driven. Also, individual processes in such systems usually only have limited computation resources. To manage, monitor, and reason about the distributed execution in these systems, predicate detection has also found applications in these new areas (2; 3; 4). For example, in a WSN deployed environment, the user might be interested in seeing whether the average temperature in the monitored field is above a certain value. In this case, each sensor node has its own temperature reading $temp_i$, which is a local variable, and the situation that the user wants to monitor can be expressed as a predicate on local variables: “ $\text{avg}_{i \in P}(temp_i) \geq 20^\circ C$ ”. The detection of a satisfaction of this predicate indicates that the situation which the user is interested in has occurred in the monitored field. However, due to the inherent differences between these new systems and the traditional distributed systems, new solutions that take into consideration the uniqueness of these new systems are needed.

We observe that in traditional distributed systems, the predicate detection algorithms are run for the entire system. This is due to the fact that geographical information has no impact on the detection of distributed properties in a traditional distributed system. For example, when a deadlock is detected in the system, the processes involved in the cyclic waiting can be either geographically far apart from each other or just sitting next to each other; when two processes

enter the critical section simultaneously and the mutual exclusion condition is broken, the two processes could be either across the entire network or just neighbors to each other.

However, when the number of processes becomes large and the events are locality driven, the situation becomes different. For example, to detect an explosion event in a WSN deployed field, we need to detect both the temperature and the sound level as “*temperature* > 150°C” and “*sound* > 60dB”. Assuming each sensor can only sense one particular parameter, this statement is a predicate specified on the local variables of multiple sensors. However, to make the detection of such a predicate meaningful, the processes whose local variables satisfy this predicate should be close to each other. If the above predicate is detected in two sensors which are far apart from each other, then the satisfaction of this predicate may not necessarily mean that an explosion has happened in the field. Running the predicate detection algorithm within the entire network thus have two drawbacks when detecting predicates in such situations. (i) The cost of running the algorithm for the whole network is high. (ii) The information gained does not represent the predicate well enough. Thus, from the above example, we can see that when detecting a distributed property in such networks, users are usually more interested in the state of a local region rather than the entire network.

Another observation is that the existing unstable predicate detection algorithms can only detect the first occurrence of the predicate satisfaction. This is due to the fact that in traditional distributed systems, predicate detection is more commonly used for debugging purposes. In such situations, the users are more interested in knowing whether a certain condition has been met during the execution of the distributed program. For example, the user can define an unstable

predicate as “ $x_i > 30 \wedge y_j < 25$ ” where x_i and y_j are local variables on processes P_i and P_j . When such a condition is satisfied in the system, the user wants to hang the execution of the distributed program. In such a situation, it is only necessary to know the first time this unstable predicate becomes true in the system and there is no need to know the subsequent occurrences of the predicate satisfactions.

On the other hand, when continuous monitoring is required in the system, knowing all occurrences of predicate satisfactions becomes necessary. This is especially true for unstable predicates. One example is data stream processing, where persistent tracking of the specified pattern, which can be expressed as predicates, is required. In such cases, the corresponding predicates could become true multiple times, and the monitoring program needs to detect all occurrences. As shown in (5), the traditional detection 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. Furthermore, in large-scale locality-driven networks, node failures can be common. Thus, a solution that can detect all occurrences of the predicate satisfactions and can also recover from the potential node crash failures is also necessary in a large-scale locality-driven network.

Also, we observe that, due to the lack of synchronized physical clocks, the traditional detection algorithms rely solely on logical clocks and the causal relationships between events. Thus, they can only make statements about whether the specified predicate has been satisfied based on all consistent observations of the distributed program’s execution. While this is good

at guaranteeing the occurrence or the opposite of a certain property during the distributed program's execution, this is insufficient to detect predicates in physical time. This is because physical time detection is concerned with only the current run, while traditional detection algorithms are concerned with all consistent runs of the distributed program. In long-running applications requiring a continuous monitoring program, the program will not be executed multiple times, thus there is no need to consider predicate satisfactions not within the current run of the program. In such cases, detecting the predicate satisfactions in physical time, i.e. within only the current run of the distributed program, becomes the requirement.

To address the above situations, we propose the concept of locality-aware predicates (LAP). Locality-aware predicates are similar to classical predicates. They can also be categorized in the same way as the classical ones. The difference lies in that LAP detects a predicate in a user-defined area within the system and thus takes into consideration the geographical information of the processes whose local variables satisfy the predicate. We also propose the hierarchical repeated detection algorithm for unstable predicates. It establishes a hierarchy in the network and repeatedly detects all occurrences of the predicate satisfactions at each level in the hierarchy. Due to its ability to detect a partial predicate of the global predicate, this algorithm can easily resume the detection after a node crashes. Furthermore, we propose the Instantaneously detection algorithm, which combines the hierarchical detection methodology with several approximation techniques to enable the detection of predicates in physical time. This algorithm achieves a high detection accuracy without incurring the cost of maintaining physical synchronized clocks.

1.2 Related Work

Detecting whether or not a given distributed property is satisfied is essential to monitoring, synchronization, coordination, and debugging of a distributed system. Predicates can be either stable or unstable. A stable predicate remains true once it becomes true. Detecting whether a certain stable predicate has become true in an ongoing distributed computation is a fundamental problem for many applications in distributed systems. Examples of stable predicates include termination (the system is in a terminated state with processes in an idle state and no messages in the channels), deadlock (a subset of processes are involved in a circular wait), and garbage collection (an object is a garbage if it has no pointer to it).

Snapshots have been used to detect stable predicates. A snapshot of a distributed system consists of a consistent collection of local states of processes and a consistent view of the corresponding channel states. The first paper formalizing and solving the global snapshot problem by Chandy and Lamport (6) assumes FIFO channels. Since the Chandy-Lamport algorithm, recording the snapshot in systems with non-FIFO channels has also been studied (7; 8; 9; 10; 11). The details can be referred from a survey (12). The Lai-Yang algorithm (7) and the Mattern algorithm (11) are *non-freezing* or *non-inhibitory* (8; 9), while the algorithm by Helary (10) is inhibitory in contrast. In recent years, the snapshot problem in large-scale distributed systems, such as P2P networks and supercomputer clusters, has also been studied (13; 14; 15). The corresponding algorithms could be used to detect stable predicates in large-scale systems.

Another recent work (4) introduced the concept of “locally distributed predicates”, which is similar to the concept of LAP that we propose. However, that paper formalized and gave an algorithm to detect only a stable predicate within a linear chain or ring topology. It does not consider the detection of unstable predicates, which are more common in systems such as WSNs or modular robotics, and it associates the predicate with only a linear topology, which is insufficient to represent a local region. Also, (4) assumes FIFO channels, which is not a very practical assumption in networks such as WSNs where communication channels may not be reliable or FIFO. Putting aside the shortcomings in (4), it also shows that detecting predicates in a local region rather than the entire network is starting to gain researchers’ attention.

On the other hand, the unstable predicate does not satisfy the same property of a stable predicate and may hold only intermittently. This makes it more difficult to detect. This is because an instantaneous observation of the whole distributed system is impossible to obtain due to the absence of a common physical clock. Furthermore, the non-determinism arisen from the asynchrony in message transmissions and in local executions adds to the complexity of detecting an unstable predicate. Algorithms to detect general unstable predicates were given in (16). However, it incurs an exponential complexity. In fact, detecting an unstable predicate has been show to be an NP-complete problem. Due to the exponential computation complexity associated with detecting a general unstable predicate, most of the works have been focused on specific subclass, i.e. conjunctive unstable predicates (16), which can be more efficient detected. A conjunctive predicate is one which can be specified in the form of $\wedge_i \Phi_i$, where Φ_i is a local predicate defined on variables local to process P_i . Due to the asynchrony in message

transmissions and in local executions, two modalities under which an unstable predicate Φ holds were also defined (16). The *Possibly*(Φ) modality checks if there exists a consistent observation of the execution such that Φ holds in a global state of the observation, and the *Definitely*(Φ) modality checks if for every consistent observation of the execution, there exists a global state of it in which Φ holds.

In (17; 18), Garg and Waldecker gave centralized algorithms to detect *Definitely* (Φ) and *Possibly*(Φ), respectively. In (17), they presented an interval-based approach to detect predicates under the *Definitely*(Φ) modality. In (1), an interval-based algorithm that adopts a unified approach to detect both *Possibly*(Φ) and *Definitely*(Φ) was given. Several distributed algorithms were also proposed. Garg and Chase (19) and Hurfin et al. (20) presented distributed algorithms to detect *Possibly*(Φ). Chandra and Kshemkalyani (21) gave a distributed algorithm for detecting *Definitely*(Φ).

More recently, predicate detection has been applied to systems such as WSNs. It is argued in (5) that when detecting predicates in continuous monitoring programs such as a WSN, usually the application requires the monitoring program to raise an alarm *each time* the predicate is satisfied. One example is data stream processing (22), 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. None of the above traditional detection algorithms are

capable of conducting such a continuous monitoring of unstable predicates within the system. As shown in (5), 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 (5), a centralized repeated detection algorithm which can detect all occurrences of $Definitely(\Phi)$ is given.

A similar technique called computation slicing was introduced by Garg and Mittal (23). Computation slicing aims at capturing all consistent cuts in the original computation which satisfy a certain global predicate. Those consistent cuts are given in the form of a slice, i.e. a condensed computation, where each event in the slice is a *metaevent* formed by grouping multiple events in the original computation together. Each *metaevent* in the slice corresponds to a join-irreducible element in the sublattice of the lattice of consistent cuts of the original computation induced by the global predicate. The slice has the property that each consistent cut of the slice corresponds to a consistent cut in the original computation that satisfies the global predicate and all such consistent cuts in the original computation correspond to some consistent cut in the slice. In the paper that initially introduced computation slicing (23), a centralized algorithm was given to compute the slice for a *regular* predicate under the *Possibly* modality. Besides that, centralized online (24) and distributed online (25) algorithms are also presented for compute the slice for *regular* predicates under the *Possibly* modality.

Compared with computation slicing, repeated detection has three major differences. First, repeated detection is based on interval solution sets while the computation slicing technique

is based on consistent cuts. On each individual process, the number of intervals in which the local predicate is true is much less than the number of local events. Thus, repeated detection has a much less exploration space than that of the computation slicing. Second, the output of the computation slicing algorithms is a condensed computation. A traversal of the sublattice induced by the slice is still necessary if the user is interested in knowing the states of the system when the predicate becomes true. In repeated detection technique, each predicate satisfaction is directly output. The last major difference lies in the fact that computation slicing technique is currently not able to handle predicates under *Definitely* modality, while repeated detection can detect predicates under this modality.

However, when it comes to detecting unstable predicates in physical time, none of the above logical time based algorithms is sufficient(26). This is because both *Possibly*(ϕ) and *Definitely*(ϕ) predicate detection algorithms are concerned with all consistent runs of the distributed program, while physical time detection is concerned with only the current run. In long-running applications requiring a continuous monitoring program, the program will not be executed multiple times, thus there is no need to consider predicate satisfactions not within the current run of the program. In such cases, detecting the predicate satisfactions in physical time, i.e. within only the current run of the distributed program, becomes the requirement.

We term the detection of predicates/events in physical time as the *Instantaneously* modality. The problem of detecting predicates under the *Instantaneously* modality without access to synchronized physical clocks has been attempted in previous works such as (27; 28). In (27), the author maintains a middle-ware logical clock which adds artificial causal relationships between

events by sending a broadcast message each time a local interval starts. In this way, two events that are previously not comparable using logical time may now become sequentialized. In (28), the authors use the *Definitely*(ϕ) detection algorithm to approximately detect predicates under the *Instantaneously* modality. While this approach ensures every detected predicate satisfaction also occurs in physical time, due to the limitations of the *Definitely*(ϕ) modality, this approach cannot detect the occurrences of predicate satisfactions in physical time that do not satisfy the *Definitely*(ϕ) detection condition. As a result, both methods suffer from a low detection accuracy.

1.3 Goals and Contributions

1.3.1 Goals

Locality-aware predicates aim at specifying and detecting predicates for a specific user-defined local region in large-scale locality driven networks such as modular robotics or WSNs. We name the local region within which the predicate is detected an “area of interest”. In such systems, users are usually more interested in the state of a local region, i.e. an area of interest, rather than the entire system. This is because:

1. The number of processes in the system is large, thus knowing the state of the entire system can be quite expensive.
2. The information gained from the state of a certain local region in the system can better represent a predicate than the state of the entire system.

Our goals in this work are:

1. Formally define locality-aware predicates by designing a model representing the area of interest
2. Design algorithms to detect stable locality-aware predicates.

In solving these problems, we face the following challenges:

1. Although a LAP detects predicates only within a certain area of interest, observing the area consistently still requires considering the entire system in a consistent manner. Since all existing algorithms for getting a consistent view of the system requires either a global snapshot of the entire system or vector clocks of the size of the system, new solutions are thus needed.
2. Since the predicate is only detected within an area of interest, the complexity of detection algorithms should be affected by the size of the area of interest not the size of the entire network. Thus, the detection algorithms should be scale-free.

In addition, we further explore the problem of LAP detection. We extend our LAP detection algorithms to enable the detection of unstable conjunctive LAPs. This makes our LAP detection algorithms able to deal with even more types of predicates. Our goals in this work are:

1. Design algorithms to detect unstable conjunctive LAPs
2. Make our detection algorithms scale-free.

We face the same challenges as we do in solving the stable LAP detection problem. In addition, some challenges unique to the unstable conjunctive LAP detection problem arises:

1. Unstable conjunctive predicate detection algorithms require the establishment of vector clocks. However, in a large-scale network, it is impractical to assume such logical clocks are established within the entire network.

Furthermore, hierarchical repeated detection algorithm aims at distributing the detection of all occurrences of predicate satisfactions within the whole system and providing fault-tolerance. It does so by establishing a hierarchy in the network and detecting all predicate satisfactions at each level in the hierarchy.

Our goals in this work are:

1. Design an algorithm to detect strong unstable conjunctive predicates in a hierarchical and repeated manner.
2. Design the algorithm in such a way that it can recover from node crash failures.

In solving this problem, we face the following challenges:

1. Due to the nature of unstable predicates, their satisfactions within the system keep fluctuating with time. 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. Thus, it is necessary to do repeated detection when detecting unstable conjunctive predicates in the network.

2. Centralized algorithms are not desirable due to the limited computation resources available to individual processes. This is especially true when detecting unstable conjunctive predicates since repeated detection will be performed.
3. Due to the scale of the system is large, individual node crash failures can be common.

Finally, the Instantaneously detection algorithm aims at detecting predicate satisfactions in physical time. It further extends the hierarchical detection methodology by enabling the detection of *Possibly*(Φ) conjunctive predicates. Furthermore, it combines the hierarchical detection algorithm with two novel approximation techniques so that the integrated algorithm can detect predicates in physical time with a high accuracy while still relies solely on logical clocks.

Our goals in this work are:

1. Extend the hierarchical detection methodology to enable the detection of weak unstable conjunctive predicates.
2. Design approximation techniques that work with the hierarchical detection algorithm to detect predicate satisfactions in physical time.

In solving this problem, we face the following challenges:

1. It has been shown in (5) that performing repeated detection under the *Possibly*(Φ) modality incurs an exponential complexity. We need to design approximation algorithms that can bring down the cost.

2. Detecting predicate satisfactions in physical time requires approximation techniques that minimizes the impact of the asynchrony in message transmissions.

1.3.2 Contributions

The contributions in this work are as follows.

By solving the stable LAP problem, we make the following contributions:

1. Motivate and propose the concept of locality-aware predicates in large-scale networks.
2. Propose the first algorithm to detect stable LAP for such networks and assume non-FIFO channels. The algorithm can detect both stable conjunctive LAP and stable relational LAP. The algorithm is highly efficient and the message count, message size, storage cost, and bandwidth complexities are scale-free, i.e., they are independent of the size of the entire network.
3. To design the above algorithm, we also make the following incidental contributions.
 - (a) Present the first distributed algorithm to create a breadth-first search tree (BFST) for a specified region within a network.
 - (b) Present the first distributed algorithm to record a consistent snapshot within a specified region of a network.

The message count, message space, storage cost, and bandwidth complexities of both these algorithms are also scale-free.

Compared with other similar algorithms, our stable LAP detection algorithm has several advantages. We compare the features of these algorithms with our stable LAP detection al-

gorithm in Table I. Specially, we compare with Chandy-Lamport snapshot algorithm (6) and Mattern’s non-FIFO snapshot algorithm (11). We also compare with the two algorithms, *LDP-Basic* and *LDP-Snapshot*, introduced in (4). They both detect a locally distributed predicate within a linear topology only.

TABLE I

COMPARISON OF FEATURES OF ALGORITHMS FOR DETECTING STABLE PREDICATES IN LARGE-SCALE DISTRIBUTED SYSTEMS.

Feature	Non-freezing	Non-FIFO Channel	Locality awareness	Scale-free
Chandy-Lamport (6)	✓	×	No	×
Modified C-L (Section 3.1.2)	✓	×	Spanning tree	×
Mattern’s Non-FIFO Snapshot (11)	✓	✓	No	×
LDP-Basic (4)	✓	×	Linear chain	✓
LDP-Snapshot (4)	×	×	Linear chain	✓
LAP Algorithms	✓	✓	BFST or any tree	✓

By solving the unstable conjunctive LAP problem, we make the following 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 conjunc-

tive LAP using EVC. This makes detecting unstable conjunctive LAP more practical in a large-scale system.

By solving the hierarchical repeated detection problem for *Definitely*(Φ), we also make the following contributions:

1. We present the first decentralized hierarchical algorithm to detect *Definitely*(Φ) 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*(Φ) (except the recent centralized algorithm in (5)) 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 shows that our algorithm is superior to the only known algorithm for repeated detection (5), which is centralized.

TABLE II
COMPARISON OF FEATURES OF ALGORITHMS FOR DETECTING UNSTABLE
PREDICATES.

Feature	Distributed	Repeated Detection	Detection within local regions	Detecting <i>Definitely</i> (Φ)	Detecting <i>Possibly</i> (Φ)
Centralized (6)	×	×	×	✓	✓
Garg and Chase (19)	✓	×	×	×	✓
Hurfin et al. (20)	✓	×	×	×	✓
Chandra and Kshemkalyani (21)	✓	×	×	✓	×
Centralized Repeated Detection (5)	×	✓	×	✓	×
Hierarchical Repeated Detection	✓	✓	✓	✓	×

Compared with other similar algorithms, our hierarchical repeated unstable conjunctive predicate detection algorithm also has several advantages. We compare with the centralized interval-based unstable predicate detection algorithm (1), and several distributed unstable predicate detection algorithms (19; 20; 21) . We also compare with the centralized repeated detection algorithm proposed in (5). The result is shown in Table II.

By solving the Instantaneously detection problem, we make the following contributions:

1. We present the innovative hierarchical detection algorithm for *Possibly*(ϕ) modality. This algorithm relies only on neighborhood communications to collaboratively detect global predicates.
2. We present two approximation techniques that work with the hierarchical detection algorithm to enable the detection of event/predicate in physical time with a high accuracy.

3. We evaluate our detection algorithm by measuring its performance with a parameterized synthetic benchmark and analyzing its accuracy under various conditions.

Compared with the other two algorithms that detect predicates under the *Instantaneously* modality based on logical clocks (27; 28), our algorithm is superior in the fact that it reaches a high detection accuracy under various network conditions.

Chapter 2 gives the system model and background on predicate detection. Chapter 3 discusses the problem of detecting a stable locality-aware predicate and presents the algorithms to do so. Chapter 4 discusses the problem of detection unstable conjunctive LAPs and give the corresponding algorithms. Chapter 5 discusses the problem of detecting a conjunctive unstable predicate for *Definitely*(Φ) in a hierarchical and repeated manner. Chapter 6 discusses the problem of detecting predicates in physical time without relying on the synchronized physical clock. The conclusion and future directions of this work are given in Chapter 7.

Portions of Chapter 3 have been previously published in Elsevier Journal of Parallel and Distributed Computing (29). Portions of Chapter 4 have been previously published in the proceedings of IEEE 12th International Symposium on Parallel and Distributed Computing (30). Portions of Chapter 5 have been previously published in IEEE Transactions on Parallel and Distributed Systems (31).

CHAPTER 2

SYSTEM MODEL AND BACKGROUND

In this chapter, we introduce the concepts that are related to our work and define the terminology that will be used in later chapters.

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 asynchronously communicate with each other via the channels in L . A channel L_{ij} is the communication link from processes P_i to P_j . Since the graph representing the network is undirected, if $L_{ij} \in L$, $L_{ji} \in L$. We do not assume FIFO channels, thus the messages may be delivered out of order. The execution of a process P_i produces a sequence of events $E_i = \langle e_i^0, e_i^1, e_i^2, \dots \rangle$, where e_i^k is the k^{th} 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 (32), and denoted as \prec . An execution of a distributed system is thus denoted by the tuple (E, \prec) .

Two processes P_i and P_j are neighbors if $L_{ij} \in L$. If the network is a wireless network, each process can communicate only with other processes within its communication range. Thus, not all pairs of processes $P_i, P_j \in P$ can be neighbors, and the topology of the network cannot

be considered as a complete graph. Therefore, messages transmitted within such a network usually traverse multiple hops. A message sent along channel L_{ij} is denoted as m_{ij} and has two associated events $send(m_{ij})$ and $recv(m_{ij})$ happening at processes P_i and P_j , respectively.

The state of a process P_i is defined by the contents of variables local to P_i . We denote the state of process P_i after event e_i^j executed and before event e_i^{j+1} as S_i^j . S_i^j is a result of the sequence of all the events executed by P_i up to e_i^j . For an event e_i^k and a process state S_i^j ,

$$e_i^k \prec S_i^j \text{ iff } 1 \leq k \leq j.$$

On the other hand, the state of a channel L_{ij} is defined as the set of messages in transit in L_{ij} , and it depends on the local states of the processes on which the channel is incident. We denote the state of a channel L_{ij} as $SC_{ij}^{x,y}$ and formally define it as follows:

Definition 1. (*Channel state*) The state of a channel $SC_{ij}^{x,y}$ is defined as $\{m_{ij} | send(m_{ij}) \prec S_i^x \wedge recv(m_{ij}) \not\prec S_j^y\}$.

Thus, channel state $SC_{ij}^{x,y}$ denotes the set of messages sent by process P_i up to e_i^x that process P_j have not received until e_j^y .

A global state GS of the execution of a distributed system is a collection of the states of all the processes and channels. For a global state GS to be *consistent*, all messages that are

recorded as received in the global state should also be recorded as sent. Basically, a *consistent global state* ensures that

$$\forall m_{ij}, \text{ if } \text{recv}(m_{ij}) \prec S_j^{y_j} \text{ then } \text{send}(m_{ij}) \prec S_i^{x_i}.$$

A consistent global state denotes a meaningful observation of the distributed system.

A *cut* C is a subset of E such that if $e_i \in C$ then $\forall e'_i \prec e_i, e'_i \in C$. A *consistent cut* (6) C is a subset of E such that if $e \in C$ then $\forall e' \prec e, e' \in C$. A cut divides the set of events in a distributed execution into a PAST and a FUTURE set. The events in C form the PAST set $PAST(C)$, and the events not in C form the FUTURE set $FUTURE(C)$. Thus, a consistent cut ensures that all messages received in the PAST set of the cut was also sent the PAST set. A cut corresponds to a global state of the execution of the distributed system, and a consistent cut corresponds to a consistent global state. Therefore, we denote the state of a process P_i or a channel L_{ij} in a global state, with respect to the corresponding cut C , as S_i^C or SC_{ij}^C , respectively.

We also assume vector clocks (33; 34) are available. Each process P_i maintains a vector clock V_i of n integers. Such a vector clock can provide logical time in the system. Due to the lack of perfectly synchronized physical clocks in a distributed system, logical clocks such as vector clocks can be a substitute for physical clocks in causality-based applications. The vector clock is updated according to 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

$$\forall k \in [1 \dots n], V_j[k] = \max(V_i[k], U[k]);$$

$$V_j[j] = V_j[j] + 1;$$

before delivering the message.

Vector clocks inherently capture the partial order (E, \prec) and the causality relation between events in the execution of a distributed system. The “happens before” relation \prec 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 $V_{e_i}[b] < V_{e_j}[b]$.

2.2 Background

In this section, we survey the existing works on predicate detection in the literature.

The problem of predicate detection is to specify a statement on the variable local to different processes in the network and to detect whether this statement becomes true during the execution of the distributed system. Detecting whether a predicate has become true in an ongoing distributed computation is a fundamental problem for many applications in distributed systems. It is important for various purposes such as monitoring, synchronization, coordination, and debugging of a distributed system. As an example, the system might be monitoring the occurrence of a deadlock in the execution, which is a predicate over local variables of different processes in the network. Predicate detection has also found application in wireless sensor net-

works (WSNs) (2) and modular robotics (3; 4). For example, in a WSN deployed field, to detect whether an explosion has happened, we can define a predicate over the temperature and the loudness of sound as $\Phi = \text{“temperature} > 150^\circ C \text{ and “ sound} > 60dB\text{”}$. The two parameters will be sensed by two different sensors. Thus, this problem is a predicate detection problem. By solving the problem of predicate detection, we gain the ability to monitor the system.

There are many predicate types and detection algorithms studied in the literature (1). In the rest of this section, we summarize the major classes of predicates and present the existing detection algorithms.

2.2.1 Stable/Unstable Predicates

A predicate can be either stable or unstable.

Definition 2. (*Stable predicate*) *A stable predicate is a predicate that remains true once it is found true within a consistent global state (6).*

Definition 3. (*Unstable predicate*) *An unstable predicate is a predicate that is not stable and hence may hold only intermittently (16).*

Examples of stable predicates include termination (the system is in a terminated state with processes in idle state and no in-transit messages in the channels), deadlock (a subset of processes are involved in a circular wait), and garbage collection (an object is a garbage if it has no pointer to it). The stable predicate detection problem has been well-studied and many solutions have been proposed for solving the general problem (e.g., (6; 35; 36; 37)) as well as the special cases (e.g., (38; 39; 40; 41)).

Unstable predicates, on the other hand, is a more general type of predicates. It does not require the predicate to remain true once it becomes true for the first time. For example, in a field deployed with sensors measuring temperature, a predicate $\Psi = \text{“average temperature is above } 50^\circ F\text{”}$ is an unstable predicate. Thus, we can observe that unstable predicates monitor the system for properties that keep fluctuating with time. Due to the nature of unstable predicates and the unavailability of perfectly synchronized physical clock, we cannot detect unstable predicates using the same way as detecting stable predicates by capture a consistent global state. This is because:

- Even the unstable predicate is found true in a consistent global state, it may not have actually held in the execution;
- Even the unstable predicate is true for a transient period in the execution, it may not be detected by periodically capturing a consistent global state.

Thus, the difficulty of detecting unstable predicates makes it necessary to find a new way of detecting predicates.

2.2.2 Possibly/Definitely Time Modalities

To address the challenges presented by unstable predicates, we observe that:

- To detect an unstable predicate, it is necessary to examine the entire execution of the distributed program rather than individual global states of the execution. Thus, it makes sense to define the unstable predicates on the observation of the execution.

- For the same distributed program, even if it is deterministic, the satisfaction of an unstable predicate can change in different executions. Thus, it also makes sense to define the unstable predicates on all observations of the executions of the distributed program.

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 gets detected within all consistent observations of an execution or within some consistent observation of an execution, can be different. Thus, two time modalities under which an unstable predicate Φ can be detected (42) have been defined:

Definition 4. (*Possibly(Φ)*) *There exists a consistent observation of the execution such that Φ can be detected in a global state of the observation.*

Definition 5. (*Definitely(Φ)*) *For every consistent observation of the execution, there exists a global state of it in which Φ can be detected.*

To illustrate *Possibly(Φ)* and *Definitely(Φ)*, we consider the example in Figure 1. The timing diagram of the execution of this distributed program is shown in Figure 1(a). The execution is run at processes P_1 and P_2 . Event e_i^k denotes the k th event at process P_i . Variable a is local to P_1 and variable b is local to P_2 . The state lattice for the execution is shown in Figure 1(b). Each state is labeled by a tuple (c_1, c_2) , where c_1 and c_2 are the event counts at P_1 and P_2 , respectively.

We observe that any path in the state lattice that goes from $(0, 0)$ to $(6, 5)$ can be an execution of the distributed program and provides an serialization of the events happening at

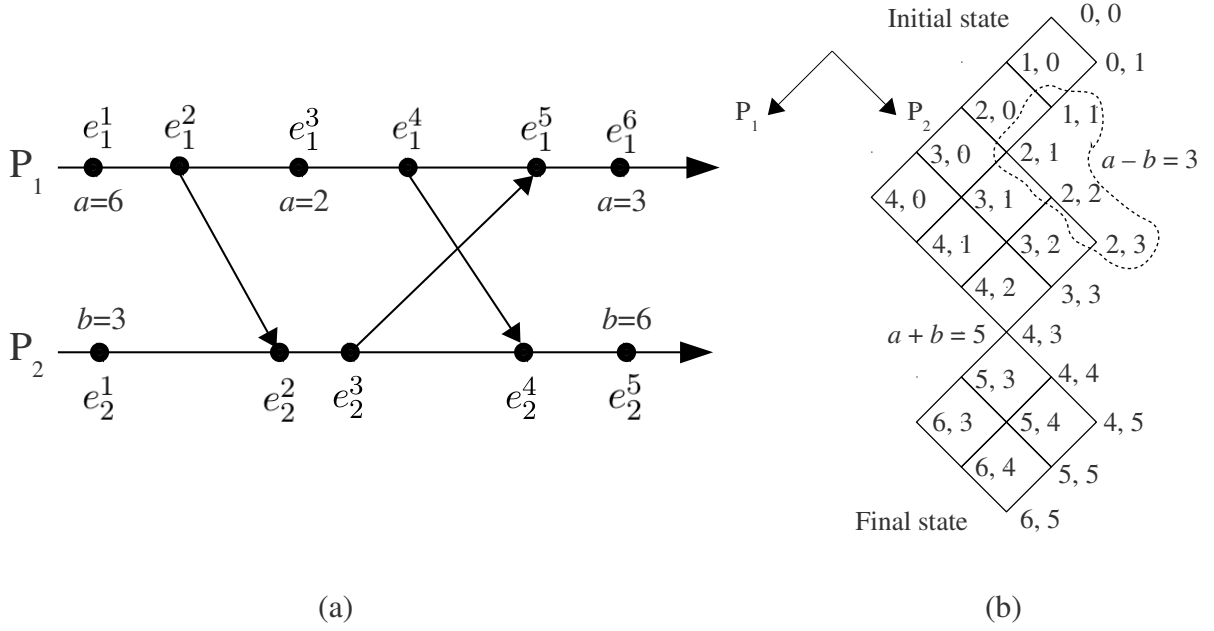


Figure 1. Example to illustrate $Possibly(\Phi)$ and $Definitely(\Phi)$. (a) The timing diagram of the execution. (b) The state lattice for the execution.

two processes which does not break any of the causal relationship shown in the timing diagram.

We also observe that different executions of the distributed program may not pass the same intermediate states as the state transitions from $(0, 0)$ to $(6, 5)$.

We now define two predicates:

$$\Phi = "a + b = 5"$$

$$\Psi = "a - b = 3"$$

From the state lattice in Figure 1, we can observe that

- *Definitely*(Φ) holds. No matter which path in the state lattice the execution of the distributed program traverses, it always goes through state $(4, 3)$. At $(4, 3)$, $a + b = 5$. So, Φ will always be true. Therefore, *Definitely*(Φ) holds.
- *Possibly*(Ψ) holds. Notice that, at state $(2, 1)$ $a - b = 3$. So, there exists at least one state in which Ψ is true. Thus, *Possibly*(Ψ) holds.
- *Definitely*(Ψ) does not hold. The states in which Ψ is true are circled in the dashed line. We observe that, in the state lattice there exist paths from $(0, 0)$ to $(6, 5)$ that do not traverse any state in the circled ones. So, there can be executions of the distributed program in which the predicate Ψ will never be true. Thus, *Definitely*(Ψ) does not hold.

From the above example, we can see that for a predicate Φ , if *Definitely*(Φ) holds then predicate Φ must have become true at some instance during the execution of the distributed program; if only *Possibly*(Φ) holds then we can only say that the predicate might have turned true at some instance but it is not guaranteed. *Possibly*(Φ) is useful when we want to ensure certain property never turns true during the execution. We can guarantee so by detecting that *Possibly*(Φ) does not hold.

Another observation from the above example is that in order to detect an unstable predicate, it seems necessary to explore the entire state lattice. If the event parallelism in the execution of the distributed program is high, the number of states that need to be checked can be up to e^n for n processes and a maximum of e events per process. In fact, it has been shown that detecting unstable predicates using the state lattice is actually an NP-complete problem (42).

2.2.3 Conjunctive/Relational Predicates

In Section 2.2.1, we have shown how to categorize a predicate into either a stable class or an unstable class. Below, we introduce another way of categorizing predicates. Based on the function on the local variables in the predicate statement, we can define the following types of predicates (16):

Definition 6. (*relational predicate*) *A relational predicate is a predicate that is expressed as an arbitrary relation on the variables in the system.*

Definition 7. (*conjunctive predicate*) *A conjunctive predicate is a predicate that can be expressed as the conjunction of local predicates.*

Let x_i and y_j be local variables at process P_i and P_j , respectively. Then $\Phi = \text{"avg}(x_i, y_j) = 35\text{"}$ is a relational predicate, while $\Psi = \text{"}x_i > 20 \wedge y_j < 45\text{"}$ is a conjunctive predicate. This categorization of conjunctive/relational predicates is orthogonal to the categorization of stable/unstable predicates. Thus, a stable predicate can be either conjunctive or relational. So is an unstable predicate.

Although detecting a general unstable predicate may incur an exponential complexity, there exist polynomial solutions for detecting conjunctive unstable predicates. For any cut C of an execution of a distributed program, if the conjunctive unstable predicate Φ is false in this cut C , then there is at least one process P_i such that the state of P_i with respect to C , i.e. S_i^C , will never form part of any cut C' such that Φ is true in C' . This local state of P_i , S_i^C , is hence a *forbidden state* and we can safely advance the local state of P_i to the next event and evaluate

Φ in the resulting cut. In this way, we eliminate the need of doing an exponential search in the state lattice and we have a solution that only need to check $O(en)$ states.

We want to point out here that, being able to detect a conjunctive unstable predicate in polynomial time does not negate the fact that detecting unstable predicates is an NP-complete problem. This is because detecting an arbitrary unstable predicate, i.e. a relational unstable predicate in particular, is inherently an entirely different problem compared to the one of detecting conjunctive unstable predicates only.

In (1), an interval based solution to detect conjunctive unstable predicates has been proposed. An interval at a process P_i is the time duration in which the local predicate is true. Such an interval at process P_i is identified by the pair of events that turn the local predicate true and false, respectively. 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. In the solution proposed in (1), 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 (18; 17; 43), it was shown that the conditions to be satisfied for $Possibly(\Phi)$ or $Definitely(\Phi)$ to be true within a set X of intervals are as follows:

$$Definitely(\Phi) : \forall x_i, x_j \in X, \min(x_i) \prec \max(x_j) \quad (2.1)$$

$$Possibly(\Phi) : \forall x_i, x_j \in X, \max(x_i) \not\prec \min(x_j) \quad (2.2)$$

We illustrate Equation 2.1 and Equation 2.2 using the following examples in Figure 2.

From Figure 2(a), we can observe that for two intervals x_1 and x_2 , if Equation 2.1 is satisfied

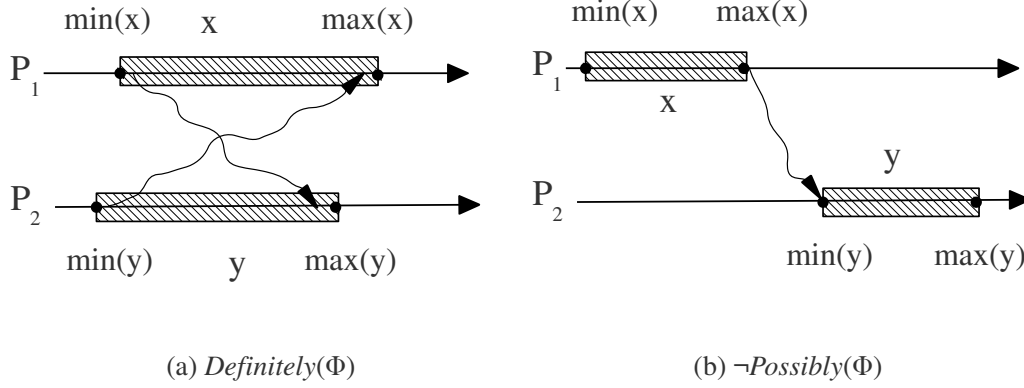


Figure 2. Example to illustrate the conditions for *Definitely*(Φ) and \neg *Possibly*(Φ) for two processes.

on the two intervals, then the two intervals has to overlap at a certain instance in any consistent observation of the execution. Thus, the predicate Φ is guaranteed to be true and *Definitely*(Φ) is detected. From Figure 2(b), we can observe that for two intervals x_1 and x_2 , if the negation of Equation 2.2, i.e. $(\max(x_1) \prec \min(x_2)) \vee (\max(x_2) \prec \min(x_1))$, is satisfied on the two intervals, then the two intervals are guaranteed to not overlap at a certain instance in any consistent observation of the execution. Thus the predicate Φ is guaranteed to be false and *Possibly*(Φ) does not hold. For a system of more than 2 processes, in order for *Possibly*(Φ) or *Definitely*(Φ) to be true in a set of n intervals, Equation 2.1 or Equation 2.2 has to be satisfied for any pair of two intervals in the set, respectively.

2.2.4 Existing Detection Algorithms

In this subsection, we briefly present the existing predicate detection algorithms. In Section 2.2.4.1, we present the algorithms that can detect stable predicates in a non-FIFO network (as the system model assumes non-FIFO channels in Section 2.1). In Section 2.2.4.2, we present the algorithms that detect various types of unstable predicates.

2.2.4.1 Detecting Stable Predicates Using Snapshots

Snapshots have been used to detect stable predicates. A snapshot of a distributed system consists of a consistent collection of local states of processes and a consistent view of the corresponding channel states. The first paper studying the global snapshot problem by Chandy and Lamport (6) assumes FIFO channels. Since then, much literature has also studied this problem with non-FIFO channels (8; 9; 10; 7; 11). The details can be referred from a survey in (12). In recent years, snapshot in large-scale distributed systems, such as P2P networks and supercomputer clusters, has also been studied (13; 14; 15). The corresponding algorithms could be used to detect stable predicates, but they have complexities of $\Omega(N \log N)$, where N is the total number of processes in the entire system.

We will use two techniques from the above papers. One is called *white/red coloring* by Lai and Yang (7), used for recording process states. The other is called *vector counter* by Mattern (11), used for recording channel states in conjunction with the *white/red coloring* coloring technique.

White/red coloring has these basic rules:

1. A process is initially white and immediately becomes red after it takes a local snapshot.

2. A white (or red) process sends white (or red) colored messages.
3. Upon receiving a red message, a white process takes a local snapshot.

Essentially, a process piggybacks a one bit status information on all outgoing communication messages. This indicates whether or not the process has taken its local state. After every process takes its local state, the set of those local states form a consistent cut.

Vector counter technique is for recording channel states. This technique requires each process to separately count the number of white messages that it has sent to any other process in a local vector. Every process also counts the number of white messages received from all other processes. First, a control message circulates through every process, through either a convergecast in a tree topology or circulating around a ring topology, to calculate the total number of white messages sent to each process. After this, the control message is broadcast to all processes, and each process waits until it has received all in-transit white messages according to the corresponding white message counter in the vector. All the white messages received by P_j from P_i after P_j turns red form the channel state SC_{ij} .

2.2.4.2 Unstable Predicate Detection Algorithms

Algorithms to detect both *Possibly*(Φ) and *Definitely*(Φ) for a conjunctive or relational unstable predicate were given in (42). This algorithm utilizes the state lattice and thus incurs an exponential complexity. Due to the exponential complexity of detecting relational predicates, most work on unstable predicate detection is focused on conjunctive ones.

In (17; 18), Garg and Waldecker gave centralized algorithms to detect *Definitely* (Φ) and *Possibly*(Φ), respectively. In (17), the interval-based approach was initially introduced to detect

Definitely(Φ). In (1), an algorithm that adopts a unified approach to detect both *Possibly*(Φ) and *Definitely*(Φ) based on intervals was given. For a network of n processes and an execution in which the local predicate becomes true at most p times at a process, the detection algorithm in (1) has a space and time complexity of $O(pn^2)$. It also generates $O(pn)$ messages, each of size $O(n)$.

The centralized detection algorithm has the drawback that all the computation occurs at a single process. This uneven distribution of time and space complexity makes such algorithms undesirable in systems where individual processes have limited resources. Several distributed algorithms were thus proposed. Garg and Chase (19) and Hurfin et al. (20) presented distributed algorithms to detect *Possibly*(Φ). Both algorithms have space, time and message complexities being $O(mn^2)$, where m is the maximum number of messages sent by any process. Chandra and Kshemkalyani (21) gave a distributed algorithm for detecting *Definitely*(Φ). Its space and time complexities are $O(\min(pn, mn^2))$ and its message complexity is $O(\min(pn^2, mn^2))$.

When detecting unstable predicates in continuous monitoring programs, usually the application requires the monitoring program to raise an alarm *each time* the predicate turns true. In such cases, none of these algorithms (17; 18; 19; 20; 21) are applicable. As shown in (5), these algorithms can detect unstable 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 (5), a centralized repeated detection algorithm which can detect all satisfactions of *Definitely*(Φ) in $O(pn^3)$ time is given. However, all the time/space

costs incurred by this algorithm happen at the sink. It is also shown in (5) that detecting all satisfactions of $Possibly(\Phi)$ takes exponential time.

CHAPTER 3

DETECTING STABLE LOCALITY-AWARE PREDICATES

This chapter is based on our previous publication (29). In this chapter, we discuss the problem of detecting stable locality-aware predicates and present the algorithms to detect such predicates. Unless otherwise specified, all the predicates in the rest of this chapter are stable predicates. In our proposed solution, to detect a stable locality-aware predicate, a 3-stage procedure is required.

1. The interacted process initiates the construction of an overlay network that corresponds to the local region specified by the user (Algorithm 1 in Section 3.2). This incurs a one-time cost for establishing the local region.
2. A distributed snapshot within this region is recorded (Algorithm 2 in Section 3.3) each time the region is to be consistently observed.
3. The recorded snapshot is used for the detection of the predicate (Algorithm 3 in Section 3.4).

The rest of this chapter is organized as follows. Section 2 gives the system model and a background on predicate detection as well as snapshot algorithms. Section 3.1 discusses the challenges in modeling and detecting locality-aware predicates and why a three-stage detection process is necessary. Section 3.2, which focuses on the first stage, introduces how we model the area of interest and proposes the algorithm to construct an overlay network that represents

this area. Section 3.3, which focuses on the second stage, presents how we modify the existing algorithms to solve the new problem - to construct a snapshot within the area of interest in a large-scale non-FIFO network. Section 3.4, which focuses on the third stage, presents the algorithm to detect stable conjunctive LAP and stable relational LAP. Section 3.5 analyzes the complexities of these algorithms in terms of the number of messages, the message sizes, and the storage and bandwidth costs. Section 3.6 briefly discusses some special cases of detecting stable LAP.

3.1 Locality-Aware Predicates

3.1.1 Motivation

Locality-aware predicates aim at specifying predicates for a local region in a large-scale locality driven network such as modular robotics or WSNs. In such a system, the state of a local region, rather than of the entire system, can be of more interest. This is because: (i) the number of processes in the system is large, thus knowing the state of the entire system can be quite expensive; (ii) the processes' interactions are local, driven by neighborhood proximity; and (iii) the properties of these interactions can only be captured by predicates over local regions.

Consider the following examples. In a token-passing system, the detection of a predicate, $\Phi = \text{number of tokens is greater than 5}$, defined for the global system might not contain any useful information, since the system contains many processes and the total number of tokens can easily exceed 5. However, if Φ is defined on a local region, then the detection of this predicate provides insight towards this particular region, and thus better captures the interactions within

the local region. To detect an explosion event in a WSN deployed field, we need to detect both the temperature and the sound level, as “ $temp > 150C \wedge sound > 60dB$ ”. Assuming each sensor can only sense one parameter, this statement is a predicate specified on the local variables of multiple sensors. To make the detection of such a predicate meaningful, the processes whose local variables satisfy this predicate should be close to each other. If the above predicate is detected in sensors which are far apart, then that may not imply that an explosion occurred. In a large-scale locality-driven system, such as WSNs, users are usually interested in such kinds of properties within a certain region. Further examples are the number of patients in a specific area in a WSN monitored hospital environment, and the number of hostile entities in a certain region in a WSN monitored battlefield.

3.1.2 Detecting Locality-Aware Predicates

When using snapshots to detect predicates, we need to build a consistent cut among the processes over which the predicate is to be detected. For locality-aware predicates, the set of processes over which we detect the predicate is not the entire network. One trivial solution is to take a global snapshot and detect the locality-aware predicate based on a subset of this snapshot. However, the complexity of such a solution is affected by the size of the network. To better solve this problem, we need to design algorithms that are *scale-free*, meaning that the size of the entire system does not affect the complexity of the algorithms. For this purpose, we need to take the snapshot only within the area of interest.

One seemingly possible solution to design a scale-free algorithm for recording a snapshot within the area of interest would be to run the Chandy-Lamport snapshot algorithm with hop

count, that is, to count the number of hops the marker message has traversed from the initiator and stop sending markers once the hop count reaches a certain value. We term this algorithm as the modified Chandy-Lamport (modified C-L) algorithm. It has three drawbacks.

1. The overlay network that this algorithm constructs is a spanning tree. If we want to cover all processes within a certain distance from the initiator, a spanning tree is not sufficient since the spanning tree does not find the shortest paths.
2. Using the modified C-L snapshot algorithm will require the communication channels to be FIFO. This is not practical in networks such as WSNs.
3. Most importantly, the modified C-L algorithm can go wrong with the recorded process states and the recorded channel states within the area of interest. This is because it is unable to track messages that transitively traverse outside the area of interest and potentially reenter the area. Fixing this problem requires taking a system-wide global snapshot, which will make the solution non-scale-free.

In essence, the modified C-L algorithm cannot construct a consistent sub-cut over the area of interest in a scale-free manner.

Besides the Chandy-Lamport snapshot algorithm, all the existing non-FIFO snapshot algorithms (13; 14; 15; 7; 10; 11) cannot be directly applied to solve the problem in a non-FIFO network. There are two reasons. First, most of the existing algorithms rely on a spanning tree overlay network (7; 10; 11) or an even more rigid overlay network such as a hypercube (13; 14; 15). Second, when capturing the process states and the channel states, the existing algorithms are designed for the entire network and will cause the complexity to be non-scale-free.

So, a new solution is needed. It needs to be scale-free, capable of constructing an overlay network that represents the area of interest, and effectively and efficiently takes a snapshot within the area of interest. In addition, it detects a predicate within the area of interest. We design such a solution as a three-stage procedure, as outlined in the beginning of this chapter.

A detailed comparison of features between potential solutions to the problem of detecting stable LAP is presented in Table I. Specifically, we compare with the Chandy-Lamport snapshot algorithm (6), the modified C-L algorithm, and Mattern’s non-FIFO snapshot algorithm (11). We also compare with the two algorithms, *LDP-Basic* and *LDP-Snapshot* (4), that can detect a locally distributed predicate (LDP) within a *linear topology* only. Both these algorithms work only if the communication is single-hop to direct physical neighbors, and the underlying channels are FIFO. *LDP-Basic* can only detect a stable predicate that does not depend on channel states, while *LDP-Snapshot* uses freezing to detect a stable predicate that may depend on channel states. Lastly, we compare with our proposed solution, which we term as LAP algorithms. The Lai-Yang algorithm (7) requires unbounded memory to track all past messages, whereas the Helary algorithm (10) is freezing; both these algorithms have highly undesirable features and hence are not considered.

Some classification to design our solutions (LAP algorithms) is introduced next. When taking a snapshot, we need to ensure there is no orphan message (1) present in the snapshot. For the processes within the area of interest, messages can be categorized into 4 types, as illustrated in Figure 3.

1. Messages transmitted entirely within the area of interest,

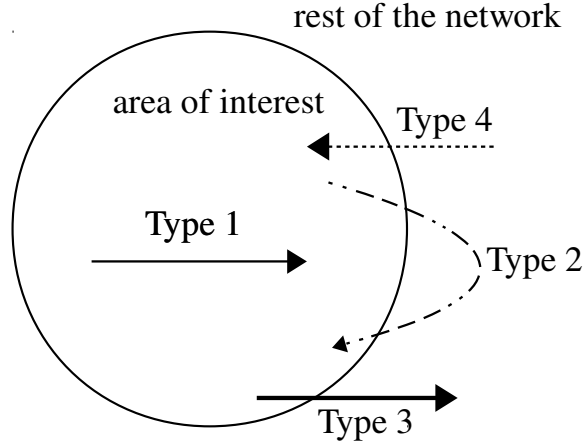


Figure 3. Four types of messages.

2. Messages whose source and destination are within the area of interest, but some of the transmitting intermediate processes are outside the area,
3. Messages sent from within the area of interest to some processes outside it,
4. Messages sent from outside the area of interest to some processes within it.

Since our goal is to take a snapshot for processes within the area of interest, type 3 and 4 messages can be ignored when checking for orphan messages. This is because whether such messages are orphan messages depends on the states of processes outside the local region. So, theoretically we need to check that there are no orphan messages only among type 1 and 2 messages. To classify a message as one of the four types, the algorithm needs to know the source and destination processes, and whether they belong to the area of interest. Although a scale-free solution of space complexity $O(d)$ might seem possible by simply tracking whether each neighbor of a process in the area of interest is also in the area of interest, this will not work because channel states may not be captured correctly. This is because each pair of processes

P_i and P_j communicate over a logical channel C_{ij} . Even though P_i and P_j may be in the area of interest, they may not be neighbors in $(\mathcal{P}, \mathcal{L})$; and therefore it is necessary to know whether each other process is in the area of interest. Furthermore, a message sent along C_{ij} may traverse outside the area of interest over physical links and hence it is necessary to record the states of logical links rather than physical links in the area of interest.

3.2 Modeling Area of Interest

The key aspect of specifying a locality-aware predicate is to specify the area of interest. So the first stage of the solution is to construct a topology that can represent the area of interest. We want to detect the predicate in an area centered at the process P_r the user interacts with and the “radius” of the area is k , meaning that processes in the area are within distance k in terms of the number of edges from P_r . This circular region is a natural model for the area of interest, particularly in WSNs and modular robotics applications because it captures geographical proximity. To achieve this, we need a topology that covers all the processes in such an area; a simple spanning tree will not suffice because it may not include all the processes within k hops from P_r ; see Figure 4 for an illustration of this concept. For this purpose, we use a local breadth-first search tree (BFST) as the topology to model the local region. The local BFST is rooted at process P_r with height k .

A distributed algorithm to construct a BFST was given by Chandy and Misra (44). However the BFST is constructed for the entire network and their algorithm has an $O(N^2d)$ message count complexity. To construct a local BFST, we face the challenge of determining when the algorithm terminates. Trivial solutions such as the asynchronous Bellman-Ford algorithm (1)

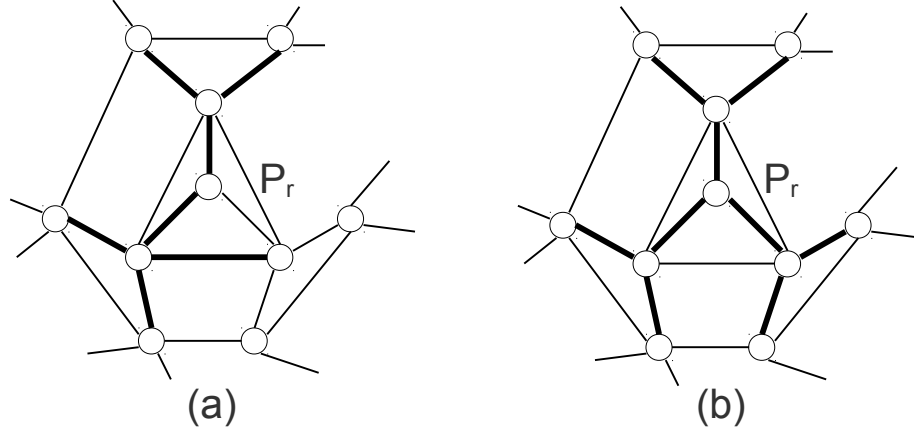


Figure 4. Illustration of covering an area centered at P_0 with radius 2. Bold edges are tree edges. (a) A spanning tree might exclude some processes such as x and y because it does not find shortest paths (b) A local BFST includes all the processes.

do not have any mechanism to determine when to terminate. Although the Chandy-Misra algorithm (44) proposed a way to determine termination by counting the number of acknowledgments, we cannot directly adapt that algorithm by using a hop restriction. This is because a process temporarily k hops away from P_r may later discover a shorter path. Thus, we need to design a solution that can correctly and efficiently determine when the algorithm terminates. Compared with the Chandy-Misra algorithm (44), our algorithm also generates fewer acknowledgment messages. Besides that, our algorithm is capable of determining the list of children for each process in the local BFST when the algorithm terminates. This is important for the later stages in our stable LAP detection solution. We use two types of control messages in the algorithm $Local\ BFST(P_r, k)$, which is listed as Algorithm 1.

Algorithm 1: Local BFST (P_r, k)

```

i.  $P_r$  initiates the construction of local BFST:
1  $dist = 0$ ,  $pred$  undefined,  $child\_list$  empty,  $T = \{i\}$ ;
2 send  $(1, P_r, k)$  to all neighbors;
3  $num = \#$  of neighbors;
ii.  $P_i$  receives  $length(s, P_j, k)$  from  $P_j$ :
4 if  $s < dist$  then
5   if  $dist = \infty$  then
6      $T = T \cup \{i\}$ ;
7   if  $num > 0$  then
8     send  $ack(negative, dist, \emptyset)$  to  $pred$ ;
9    $pred = P_j$ ;  $dist = s$ ;  $child\_list = \langle \rangle$ ;  $num = 0$ ;
10  if  $s + 1 \leq k$  then
11    send a message  $length(s + 1, P_i, k)$  to every neighbor except  $P_j$ ;
12     $num = \#$  of neighbors  $- 1$ ;
13  if  $num == 0$  then
14    send  $ack(positive, dist, T)$  to  $pred$ ;
15     $T = \emptyset$ ;
16 else
17   send  $ack(negative, s, \emptyset)$  to  $P_j$ ;
18   remove  $P_j$  from  $child\_list$ ;
iii.  $P_i$  receives  $ack(*, s, T')$  from  $P_j$ :
19  $T = T \cup T'$ ;
20 if  $s \neq dist + 1$  then
21   discard the  $ack$  message;
22 else
23    $num = num - 1$ ;
24   if  $ack$  is positive then
25     add  $P_j$  to  $child\_list$ ;
26   if  $num == 0$  then
27     if  $pred \neq P_i$  then
28       send  $ack(positive, dist, T)$  to  $pred$ ;
29        $T = \emptyset$ ;
30     else
31       broadcast  $terminate(T)$  on local BFST;
32       terminate the algorithm;
iv.  $P_i$  receives  $terminate(X)$  from  $P_j$ :
33  $X$  identifies the process set in the area of interest;
34 propagate the  $terminate(X)$  on local BFST;
35 terminate the algorithm;

```

1. A message *length* (s, P_j, k) received at P_i indicates that there is a path of length s from P_r to P_i with P_j being the predecessor of P_i . The distance limitation k is also contained in this message.
2. A message *ack* (*positive/negative*, $s, pids$) acknowledges a *length* message sent from P_j to P_i after a certain condition is met. An *ack* can be either positive or negative, and also carries the length parameter s of the corresponding *length* message. The parameter *pids* is a set that contains the process identifiers of some of the nodes in the sub-tree traversed. This parameter is non-empty only on positive *acks*.

Each process P_i also maintains several local variables.

1. *dist*: the length of the shortest path from P_r to P_i known so far. P_r initializes *dist* to 0, other processes initialize *dist* to ∞ .
2. *pred*: the predecessor on the shortest path from P_r to P_i known so far, and is initially undefined. The message *length* (*dist*, *pred*, k) is received from *pred*.
3. *num*: the number of unacknowledged *length* messages, initialized to 0.
4. *child_list*: a list of processes that become a child process of P_i in the tree topology, initialized to the empty list.
5. *T*: a set of some of the process identifiers in the sub-tree, initialized to \emptyset .

Correctness

Observation 1. *Once a process receives a length message, the process will always have some parent pred and will always be a part of the local BFST.*

Observation 2. *For a process P_i in the local BFST, its neighbors that are also in the local BFST are exactly those processes P_j to which P_i sent a length message or from which P_i received a length message.*

Observation 3. *The variable $dist$ is a strictly decreasing function and can change at most k times.*

Theorem 1. *Algorithm 1 identifies all the processes in the local BFST, defined as a BFST rooted at P_r with height k .*

Proof. Define $min_dist(P_i)$ as the length of the shortest path from P_r to P_i . We say that a process gets engaged by a message $length(x, *, *)$ if $x < dist$ at the process. By Observation 1, the process will be part of the local BFST. We prove the theorem by induction on $min_dist(P_i)$, using the hypothesis, “If $min_dist(P_i) = x \leq k$, then P_i is included in the local BFST.”

$min_dist(P_i) = 1$: P_r sends $length(1, P_r, k)$ to all its neighbors. The last engagement of any process P_i having $min_dist(P_i) = 1$ is by the $length(1, P_r, k)$ it receives. By Observation 2, P_i is included in the local BFST.

$min_dist(P_i) = x$ ($x \leq k - 1$): Assume the induction hypothesis is true.

$min_dist(P_i) = x + 1$ ($x \leq k - 1$): By the induction hypothesis, each process P_j such that $min_dist(P_j) = x$ gets last engaged by a message $length(x, *, k)$, and by line (11), sends $length(x + 1, P_j, k)$ to all its neighbors except $pred$, where $min_dist(pred) = x - 1$. Thus, any process P_i such that $min_dist(P_i) = x + 1 \leq k$ will receive at least one $length(x + 1, *, k)$ message, and get last engaged by the first such message. By Observation 2, P_i is included in the local BFST.

Further, if $\text{min_dist}(P_i) = k$, by line (11), P_i will not send any $\text{length}(k+1, P_i, k)$ messages, and no process P_j having $\text{min_dist}(P_j) > k$ will ever be engaged and will not be identified as part of the local BFST. \square

Algorithm 1 is guaranteed to terminate correctly. When process P_i sends the *length* messages to its neighbors, the counter *num* is set to the number of *length* messages sent (line 12). Whenever an *ack* is received, either positive or negative, the counter decreases by 1 (line 23). Furthermore, each time P_i discovers a shorter path, it will reset its counter to 0 (line 9). We also make sure that P_i will only decrease its counter when the distance marked in the *ack* message received matches P_i 's current *dist* (lines 20-21). By performing these operations, P_i can know whether all *length* messages corresponding to its current *dist* have been acknowledged. This is guaranteed to happen because for each *length* message generated, exactly one *ack* message will be sent back. When this happens, all the processes in the temporary sub-tree rooted at P_i (this sub-tree might still change if any process discovers a shorter path via some process outside this sub-tree) also have their *num* being 0. When P_r 's *num* becomes 0, the counter *num* at all processes in the local BFST must have already turned 0. This ensures that all processes in the local BFST have been discovered and every process in the local BFST has discovered the shortest path, because otherwise there will be at least one process whose *num* is not 0.

Upon termination at the root P_r , all processes in the area centered at P_r with radius k form a local BFST, and are also identified at P_r . The identifiers of all the processes get collected in the T parameter at the root. Consider any P_i that has received some *length* message. P_i sends $i \in T$ to the first parent P_j to which it sends a positive *ack*. (P_j has a smaller *dist*

value.) P_j will add i to its T variable (line (19)). Observe from lines (13-14) and (26-28) and Observation 3 that P_j has not yet sent any positive *ack*, or if it has, it will still send one more to a new parent. P_j 's T variable containing i is now included in the first positive *ack* it sends to some parent, say P_k . (P_k has an even smaller *dist* value.) Inductively, within k hops, P_i 's identifier i reaches P_r on some positive *ack*, after which P_r terminates. Thus, every process that has received a *length* message is included in T at P_r .

Each process in this local BFST also knows its parent (*pred*) and children (*child_list*) upon termination. Each process' parent is correctly identified because *pred* is always set to the predecessor on the shortest path known so far. For the list of children, each time a positive *ack* is received at P_i from P_j , P_j has already set its *pred* to P_i . Thus, P_i adds P_j to its *child_list*. However, it is possible that P_j later discovers a shorter path and sets its *pred* to a different process. Thus P_i needs to remove P_j from its *child_list*. This is achieved in our algorithm. Notice that when P_j discovers a shorter path, it will send *length* messages to all its neighbors, except the predecessor on the shorter path; thus, a *length* message gets sent to P_i . When this happens, only one of 3 situations could occur, as illustrated in Figure 5(b,c,d). In each case, P_i can discover that it is no longer P_j 's parent and can safely remove P_j from its *child_list*: this happens in line (18) for cases (b,c) and in line (9) for case (d).

This guarantees that for any process P_i , its *child_list* contains all and only all the processes that becomes P_i 's children.

When the root terminates, it needs to broadcast a *terminate* message in the local BFST, in order for other processes in the local BFST to learn of the termination. This is because they

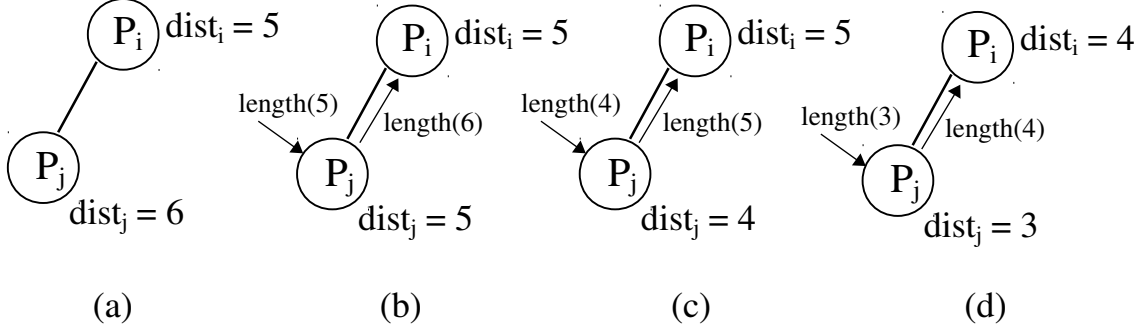


Figure 5. Illustration of dynamic changes to *child_list*. (a) Initially P_j is P_i 's child, with P_i 's *dist* being 5 and P_j 's *dist* being 6. (b) P_j discovers a shorter path with *dist* being 5 and sends a *length* message to P_i . (c) P_j discovers a shorter path with *dist* being 4 and sends a *length* message to P_i . (d) P_j discovers a shorter path with *dist* being 3 and sends a *length* message to P_i . In (b) and (c), P_i can discover P_j is no longer its child and can safely remove P_j from its *child_list* in line (18). In (d), P_i becomes P_j 's child and resets its *child_list* in line (9).

would not otherwise know if a shorter path than that per their current knowledge is still being searched. The non-root processes terminate when they receive the *terminate* broadcast from their parent, *pred*. All the local BFST processes also learn of which other processes are in the local BFST when they receive the *T* parameter on the *terminate* message.

We denote the total number of processes in the entire system as N and the number of processes in the area of interest as n . The complexity of Algorithm 1 is scale-free (to be shown in Section 3.5), meaning that its complexity is affected only by n , k , and d , but not by N . This is a feature shared by all the algorithms introduced in this chapter. Being scale-free is important to locality-aware predicate detection because a locality-aware predicate models a property within a local region. Having an algorithm with complexity relative to N will make it non-scalable and thus not applicable in large-scale distributed systems. We will give the detailed complexity analysis in Section 3.5.

With Algorithm 1, we ensure that in the first stage we can dynamically construct an overlay network covering all the processes in the area of interest with a relatively low cost. Note that Algorithm 1 is not robust to churn because of the intricate interactions among the various *length* and *ack* messages.

3.3 Consistent Sub-cut Construction

Now, we assume that Algorithm 1 has already run and a local BFST rooted at P_r is constructed. In the second stage, a snapshot within the area of interest is taken. We base our algorithm on top of the *white/red coloring* and *vector counter* techniques discussed in Section 2.2.4.1. For capturing process states, the *white/red coloring* technique is sufficient to identify pre-recording and post-recording messages in non-FIFO systems. Also, it does not incur any extra overhead besides associating a one bit data with the messages. However, for capturing channel states, the *vector counter* technique does not solve the problem in a scale-free manner. This technique is designed for capturing channel states while taking a global snapshot. It also ensures that all the in-transit messages while taking the snapshot get delivered to the destination when the algorithm terminates. Directly applying this technique will incur an $O(N)$ storage cost and bandwidth cost, thus causing the solution to be non-scale-free. Instead of maintaining a size N vector to count white messages sent to/received from every process in the system, each process P_i in the area of interest only maintains a size $n = |T|$ vector to count the white messages traversing on logical channels to processes within the area of interest. Recall that T is the set of processes in the local BFST, as computed by Algorithm 1. We assume the n processes in T are $P_{l_1}, P_{l_2}, \dots, P_{l_n}$ and P_r associates this mapping, which maps the ID i of

process $P_i = P_{l_a}$ in the local BFST to a virtual ID a in the range $[1, n]$, with the *terminate* message in Algorithm 1. Thus every process has a unique position in a size- n vector. This is important for the local variables introduced later.

As we need a scale-free algorithm, no process can start coloring the messages white at system initialization time because it does not know the area of interest, and hence would have to track the messages sent to all N processes. As the local BFST is formed on-the-fly, we are faced with the challenge of identifying (i) when to begin coloring the messages as white, (ii) when to begin counting the white messages sent to each other process in the BFST, and (iii) when to start counting the white messages received. These operations are essential to ensure that the recorded channel states are complete. We claim that no coordination is needed among the processes in the local BFST to begin these operations.

Observation 4. *A process in the BFST can begin coloring messages sent to processes in the BFST as white and counting the white messages sent to (and received from) others in the BFST, at any time before recording its local snapshot. The count of incoming white messages should begin no later than receiving the first white message.*

This follows from the fact that each logical channel is independent and the message count is per logical channel. The above observation is implemented in the one-time pre-processing for the algorithm.

So, for our algorithm, each process P_i maintains the following local variables:

1. $color_{l_i}$ records the color of P_i as either white or red; initialized to white.

2. $wmsg_sent_{l_i}[1 \dots n]$ is a vector of size n . $wmsg_sent_{l_i}[j]$ counts the total number of white messages sent to process P_{l_j} in the local BFST.
3. $wmsg_to_recv_{l_i}$ is an integer variable which accumulates the count about the total number of white messages it should receive (from processes that are also in the local BFST) in order to complete the recording of channel states for the consistent sub-cut.
4. $wmsg_recv_{l_i}$ is an integer variable which counts the number of white messages received (from processes that are also in the local BFST).

There are three types of control messages in the algorithm:

1. An *INIT* message gets broadcast within the area of interest via edges of the local BFST constructed in the first stage. It initiates the algorithm and serves as a red communication message in case some process in the local region has not received any communication messages from other processes in the same local region.
2. A *Cvg_Acc_White* message convergecasts the *wmsg_sent* vector.
3. A *Bcast_Acc_White* message broadcasts the *wmsg_sent* vectors accumulated at the root.

In addition to these control messages, each process will also receive white or red communication messages which they also need to handle accordingly.

Our algorithm is listed in Algorithm 2. The one-time pre-processing after the local BFST is constructed involves the following action.

- As part of the initialization, recall that in Algorithm 1, the root process broadcasts the $terminate(T)$ on the local BFST to inform the processes of the IDs of the processes in the

local BFST. As each process gets engaged by the broadcast, it now initializes the size- n vector $wmsg_sent[1 \dots n]$, starts coloring messages white, and counting the number of such messages sent to each other process in the local BFST, and received from processes in the local BFST. (If a white message is received before the $terminate(T)$ broadcast is received, the $wmsg_recvd_{l_j}$ variable is updated right away.)

For each snapshot to be collected after the pre-processing step, the algorithm is executed in five phases:

1. The root of the local BFST initiates a one-to-all broadcast of an *INIT* control message along the BFST edges to inform all processes in the tree about the commencement of taking the snapshot.
2. The number of white messages sent along each logical channel whose both end-points are incident on processes within the local BFST is determined in this phase along with recording the local snapshot. First, upon receiving the *INIT* control message or a red computation message, a white process turns red and records its local state. A process might have already turned red before receiving the *INIT* message; in this case, it simply ignores the *INIT* message. Second, when a leaf node P_{l_i} turns red, it initiates a convergecast to component-wise accumulate the $wmsg_sent_{l_i}[1 \dots n]$ vector in the local BFST.
3. After executing the second phase in which the root accumulates the $wmsg_sent_{l_i}$ vectors from all nodes in the local BFST into $Cvg_Acc_White(W)$, the root initiates a

Algorithm 2: Construction of Consistent Sub-cut(P_r, k, ϕ)

Initialization: P_i initializes the algorithm at the end of Algorithm 1:

1. P_r : On broadcasting *terminate*(T), initialize $wmsg_sent[1 \dots n]$, start coloring messages white and start counting white messages sent to processes in T . Also operate *wmsg_recvd*.
2. $P_i \in$ local BFST: On receiving *terminate*(X), initialize $wmsg_sent[1 \dots n]$, start coloring messages white and start counting white messages sent to processes in T . Also operate *wmsg_recvd*.
 - i. P_r starts collecting the snapshot:
 - 1 send an *INIT*(ϕ) message to all processes connected by local BFST edges and to P_r itself;
 - ii. White process P_i receives an *INIT* or a red communication message from P_j :
 - 2 $color_{l_i} = \text{red}$;
 - 3 **if** $P_{l_i} \neq P_{l_j}$ **then**
 - 4 | send an *INIT* message to all children in local BFST;
 - 5 record local state pertinent to ϕ ;
 - 6 **if** P_{l_i} is a leaf node in local BFST **then**
 - 7 | initiate convergecast *Cvg_Acc_White*($wmsg_sent[1 \dots n]$);
 - iii. P_i receives *Cvg_Acc_White*($W[1 \dots n]$) from P_j :
 - 8 participate in convergecast by accumulating the W vectors from all children in local BFST and its own $wmsg_sent$ vector recorded in the local snapshot;
 - 9 **if** $P_{l_i} \neq P_r$ **then**
 - 10 | send *Cvg_Acc_White*(W) to parent;
 - 11 **else**
 - 12 | initiate broadcast *Bcast_Acc_White*(W);
 - iv. P_i receives an *Bcast_Acc_White*(W) message from P_j :
 - 13 $wmsg_to_recv_{l_i} = W[i]$;
 - 14 **if** P_{l_i} is not a leaf node **then**
 - 15 | propagate *Bcast_Acc_White*(W) to child nodes;
 - v. Red process P_i receives a white or uncolored message msg from P_j :
 - 16 **if** P_{l_j} is in local BFST **then**
 - 17 | record msg as part of channel state $SC_{l_j l_i}$;
 - 18 **if** msg is white **then**
 - 19 | | $wmsg_recvd_{l_i}++$;
 - 20 | **if** $wmsg_to_recv_{l_i} = wmsg_recvd_{l_i}$ **then**
 - 21 | | | terminate the algorithm locally;

$Bcast_Acc_White(W)$ to inform each process in the local BFST of the number of white messages sent to it up to the snapshot recording.

4. As each process gets engaged by the $Bcast_Acc_White$, it saves in $wmsg_to_recv_{l_i} = W[i]$ the number of white messages it should receive in order to complete channel state recording.
5. The channel states are recorded in this phase. When a red process P_{l_i} receives a white or an uncolored message from P_{l_j} (which is also in the local BFST), P_{l_i} adds such a message to the channel state $SC_{l_j l_i}$ and increments $wmsg_recv_{l_i}$ if the message was white. Once P_{l_i} determines that $wmsg_to_recv_{l_i} = wmsg_recv_{l_i}$, it terminates the algorithm locally. It is now ready to participate in the LAP evaluation (Algorithm 3).

Correctness

To prove that Algorithm 2 is correct, we need to show that the process states do not contain an orphan message and that the channel states are complete.

- Since a white process records its local state upon receiving the first red communication message or the *INIT* message, there will be only white communication messages in its recorded local state. Also, since a white process turns red after recording its local state, all the *send* events of white messages would have been recorded. Thus, the process states do not contain orphan messages.

- Since the algorithm does not terminate locally at a red process until it has received the correct number of white messages on each incoming channel from any other process also in the local BFST (line 20), this guarantees that the channel states are complete.

In our algorithm, as the messages are colored white and counted only after the local BFST is formed, no additional overhead is required before the detection of a stable LAP. However, the first snapshot recorded by our algorithm may record some incomplete channel states. This is because some (uncolored) message generated before the pre-processing step may get delivered to the destination after the snapshot and channel state recording is completed at the destination. Such a potentially incomplete channel state recording can be avoided by introducing a small delay between the initialization step and step (i). This is because (even with non-FIFO channels,) messages will eventually be delivered. In wireless transmission protocols such as IEEE 802.11 (45), although the value of the ACK timeout is not defined in the specification, the general setting is $\text{SIFS} + \text{ACK} + \text{Round Trip Propagation Delay}$ (46), which is usually tens of microseconds. Repeated invocations of Algorithm 2 are often needed to test for a stable LAP, and are sequential and spaced apart. Hence, for the second and subsequent invocations, the possibility of an incompletely recorded channel state becomes zero very quickly.

Observe that the definition of a consistent sub-cut (Section 2.1) is not concerned with transitive inconsistencies caused by Type-2 messages of Figure 3. However, observe that we can easily modify Algorithm 2 to prevent inconsistencies caused by Type-2 messages as follows: instead of coloring with two colors, sub-cut snapshot sequence numbers are needed. The message

coloring rule is modified to use sequence numbers. All the processes in the system follow this rule but only the processes in the local BFST execute Algorithm 2.

After running Algorithm 2, the snapshot within the area of interest is recorded. This provides the foundation for detecting locality-aware predicates.

3.4 Detecting Locality-Aware Predicates

The third stage of detecting locality-aware predicates (LAP) is to actually detect the predicate based on the recorded distributed snapshot. Section 3.4.1 formally defines a LAP and gives examples. Section 3.4.2 presents the algorithm for detecting stable LAP, and discusses how it can be adapted for conjunctive LAP and for relational LAP.

3.4.1 Formal Definition

Definition 8. A LAP $Q = (\phi, k, P_r)$ is a predicate ϕ over the states of processes within the local BFST rooted at P_r with height k .

- If ϕ is conjunctive, then LAP Q is conjunctive.
- if ϕ is relational, then LAP Q is relational.

Examples of Conjunctive LAP

- $Q_1 = (\phi, 3, P_r)$, where $\phi = \wedge flag_{P_i}$, for P_i in a height-3 local BFST rooted at P_r .
 Q_1 is true if each process in P_r 's local region with radius 3 has set its flag.
- $Q_2 = (\phi, 5, P_r)$, where $\phi = \wedge terminated_{P_i}$, for P_i in a height-5 local BFST rooted at P_r .
 Q_2 is true if each process in P_r 's local region with radius 5 has terminated.

- $Q_3 = (\phi, 5, P_r)$, where $\phi = \wedge temp_{P_i} > 50$, for P_i in a height-5 local BFST rooted at P_r .

Q_3 is true if each sensor in P_r 's local region with radius 5 has a temperature reading greater than 50°F in a WSN monitored field.

Examples of Relational LAP

- $Q_4 = (\phi, 5, P_r)$, where $\phi = \sum_{P_i \in local\ BFST} token_{P_i} \geq 3$.

Q_4 is true if there are at least 3 tokens among processes within an area of radius 5 from P_r .

- $Q_5 = (\phi, 6, P_r) = average_of_temp_{P_i} \geq 50$, for P_i in the local BFST rooted at P_r .

Q_5 is true if (in a WSN monitored field) the average temperature sensed within an area of radius 6 from P_r is larger than 50°F.

3.4.2 Stable LAP Detection Algorithm

In order to detect a stable LAP $Q = (\phi, k, P_r)$ in the system, a set of process states satisfying Q needs to be found. To achieve this:

1. The set of states needs to form a consistent sub-cut. By running Algorithm 2 from Section 3.3, every process in the local BFST holds a local state which is part of a consistent sub-cut.
2. The predicate ϕ needs to be evaluated over the set of states. To achieve this, we execute Algorithm 3 which collects the set of states recorded in the consistent sub-cut, and then evaluates ϕ over this set.

Algorithm 3 is a convergecast within which the set of states recorded in the consistent sub-cut over the area of interest is collected at P_r using the tree edges in the local BFST. The convergecast uses the *State* message type. Each process P_i in the local BFST maintains the following variables:

- v_i : the variable(s) of the locally recorded snapshot state relevant to the evaluation of ϕ is/are stored;
- V_i : accumulates the snapshot states reported by processes in P_i 's sub-tree within the local BFST;
- $\#children_i$: the number of children nodes in the local BFST; and
- $child_count_i$: the number of children from which a *State* message has been received.

Detection begins at leaf processes which have terminated Algorithm 2. These leaf processes in the local BFST initiates the convergecast by reporting the locally recorded state variable v_i to their parents in a *State* message. When an intermediate node P_i receives a *State* message, it accumulates the contained states from its sub-tree. When a *State* message has been received from all the children and Algorithm 2 has also terminated locally, P_i adds its own local snapshot state v_i to V_i . If P_i is not the initiator P_r , then P_i sends a *State*(V_i) message to its parent in the local BFST. However, if P_i was the initiator P_r , it evaluates the predicate ϕ over the set of states V . The algorithm is listed in Algorithm 3.

Algorithm 3: Stable LAP Detection Algorithm for $Q = (\phi, k, P_r)$ (code for P_i)

```

i. When  $P_i$ , which is a leaf node, terminates Algorithm 2:
1 send  $State(\{v_i\})$  to parent; terminate;

ii.  $P_i$  receives  $State(X)$  from child  $P_j$ :
2  $child\_count_i = child\_count_i + 1$ ;
3  $V_i = V_i \cup X$ ;
4 if  $child\_count_i = \#children_i$  then
5   await (local termination of Algorithm 2);
6    $V_i = V_i \cup \{v_i\}$ ;
7   if  $P_i \neq P_r$  then
8     send  $State(V_i)$  to parent in local BFST; terminate;
9   else
10    evaluate  $\phi(V_i)$ ; terminate;

```

Adaptation to Conjunctive LAP

The local variable v_i can be recorded as a boolean for a conjunctive predicate. The local variable V_i that accumulates the states of processes within the sub-tree rooted at P_i can be represented as a boolean to correspond to the (partial) evaluation of ϕ (over the sub-tree). This is because the evaluation of a conjunctive predicate is based on an *aggregation operation*, namely the AND operator.

- *Aggregation operations* are defined as those operations that are associative, thereby allowing the input to be processed in any order, and do not require all the input to be present before evaluation begins.

The size of a *State* message is thus $O(1)$. The broadcast phase can be terminated at an intermediate node P_i if the local conjunct evaluated over V_i is false. P_i would behave as a leaf node and send a *State* message immediately to its parent. The number of *State* messages is thus between 1 and $n - 1$.

Adaptation to Relational LAP

The local variable V_i may be as large as the number of nodes in the sub-tree rooted at P_i , and hence varies from 1 to n . However, many relational predicates have their evaluations based on *aggregation operations*, e.g., addition, minimum, maximum, and average functions, and hence V_i is of size 1. Only for relational predicates whose evaluation is not based on *aggregation operations*, e.g., the median or the mode of a set of values, the size of V_i could be as large as n . The above observations on the size of V_i also hold for the size of the *State* message. The number of *State* messages is $n - 1$.

Observe that for conjunctive predicates, it was possible to curtail the broadcast and convergecast overhead each time the local conjunct evaluated to false at a process. The same approach does not work for a relational predicate. However, we can cut down the overhead of predicate evaluation. We propose a classification of predicates as follows:

- *Incremental predicate*: This is a predicate whose satisfaction can be determined for at least one input without evaluating the predicate fully over all variables.

All conjunctive LAPs, such as Q_1, Q_2, Q_3 are incremental predicates. Some relational predicates that are non-conjunctive, such as Q_4 , are also incremental predicates.

- *Non-incremental predicate*: This is a predicate whose satisfaction for every input can be determined only after evaluating it fully over all variables.

Some relational predicates, such as Q_5 , are non-incremental predicates.

Depending on the locally recorded snapshot values, the evaluation of relational predicates that are incremental may be determined without considering all the variables. If Algorithm 3 is modified to perform an iterative deepening breadth-first collection of the recorded v_i , i.e., a layer-by-layer reporting of the v_i 's to P_r , the number of messages used is $\Omega(1)$; however, it will use $O(nk)$ number of messages.

3.5 Complexity Analysis

We evaluate the complexities of our algorithms using four metrics:

- message count: count of the total number of messages generated by the algorithm.
- message size: the total size of all the control messages generated by the algorithm. It can be formalized as $\sum_i (\# \text{ type } i \text{ messages} * \text{size of type } i \text{ messages})$.
- storage cost: the space complexity at each process. Since P_r is responsible for storing the final results in Algorithm 3, we do not take P_r into consideration for this measure.
- bandwidth cost: bandwidth usage of a channel measures the total size of messages sent along that channel. The maximum bandwidth usage among all channels in the system is the bandwidth cost of the algorithm.

As mentioned before, all the algorithms share the scale-free feature. The complexities are evaluated within a degree- d bounded network. The results are shown in Table III.

3.5.1 Algorithm 1 (Local BFST)

Message count complexity: The local variable *dist* at each process in the area of interest can hold a value only within the range $[0, k]$. As per Observation 3, the local variable *dist* strictly

TABLE III
COMPLEXITY EVALUATION OF STABLE LAP ALGORITHMS IN A DEGREE- D
BOUNDED NETWORK.

Metric	Local BFST (Algorithm 1)	Consistent Sub-Cut (Algorithm 2)	Stable LAP aggregation based ϕ (Algorithm 3)	Stable LAP non-aggregation based ϕ (Algorithm 3)
Message count	$O(nkd)$	$O(n)$	$O(n)$	$O(n)$
Message size	$O(n^2)$	$O(n^2)$	$O(n)$	$O(nk)$
Storage cost	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Bandwidth cost	$O(n)$	$O(n)$	$O(1)$	$O(n)$

decreases. Each time process P_i 's *dist* decreases, it will send at most $d_i - 1$ *length* messages to its neighbors. Thus, each process can only send *length* messages to its neighbors up to k times. Each *length* message has one *ack* message. This gives a message count complexity of $O(k \sum_{i=1}^n d_i)$. In a degree- d bounded system, this is $O(nkd)$. Further, there are $n - 1$ *terminate* messages.

Message size complexity: The size of *length* control messages is $O(1)$. The sum of sizes of *ack* messages is bounded by $O(nk^2d)$ because the identifier of a node at distance j contributes to the T parameter j times. The size of a *terminate* is n . So the message size complexity is $O(nk^2d + n^2) = O(n^2)$.

Storage cost complexity: During the execution of Algorithm 1, each process has to maintain a *child_list* which is of $O(d)$ size. In addition, at the end of the algorithm, the root of the local BFST broadcasts the identifiers of all the processes in the local BFST using the *terminate*(T)

message. This incurs an $O(n)$ storage cost at each process in the local BFST. So, the total storage cost is $O(n)$.

Bandwidth cost complexity: The value of each process' local variable $dist$ can be only between 0 to k . Each time a process' local $dist$ decreases, it will send a message to all its neighbors. Since $dist$ strictly decreases, each process can only send up to k messages to each neighbor via the same channel. The size of $length$ message is $O(1)$. For the ack message, although its size is $O(n)$, the total size of ack messages sent by a single process is also $O(n)$ since no process identifier will be sent more than once. So the bandwidth cost is $O(n + k) = O(n)$.

3.5.2 Algorithm 2 (Consistent Sub-cut)

Message count complexity: The $INIT$ message is broadcast within the local BFST only once, thus causing a $O(n)$ complexity. The modified *vector counter* technique uses another convergecast (Cvg_Acc_White) and broadcast ($Bcast_Acc_White$), thus causing another $O(n)$ complexity. So the total message count complexity is $O(n)$.

Message size complexity: The $INIT$ control message has size $O(1)$. The two types of control messages – Cvg_Acc_White and $Bcast_Acc_White$ – both have sizes of $O(n)$. So, the message size complexity is $O(n^2)$.

Storage cost complexity: Each process maintains a size- n vector $wmsg_sent_{l_i}$. So the storage cost is $O(n)$.

Bandwidth cost complexity: The $INIT$ message (of size $O(1)$), and the Cvg_Acc_White and $Bcast_Acc_White$ messages (of size $O(n)$ each) traverse the edges of the BFST once. So the bandwidth cost is $O(n)$.

3.5.3 Algorithm 3 (Stable LAP)

Recall that this algorithm works for aggregation based and non-aggregation based LAP. Aggregation based predicates include all conjunctive predicates and some non-conjunctive predicates. Non-aggregation based predicates include the remaining relational predicates.

Message count complexity: To detect the stable LAP, each process in the local BFST sends a *State* message to its parent in the BFST. So the message count complexity is $O(n)$. This result holds for aggregation based and non-aggregation based LAP.

Message size complexity: For aggregation based LAP, the *State* message, which is sent $n - 1$ times, is of size $O(1)$. Thus, the message size complexity for aggregation based LAP is $O(n)$.

For non-aggregation based LAP, for $i \in [0, k - 1]$, the d^{k-i} nodes in the local BFST that are $k - i$ hops away from P_r cumulatively send d^{k-i} *State* messages of size d^i to their parents. Thus, the cumulative size of all the *State* messages is $\sum_{i=0}^{k-1} d^{k-i} d^i = k d^k = kn$. Thus, the message size complexity for non-aggregation based LAP is $O(kn)$.

Storage cost complexity: For aggregation based LAP, the storage cost complexity is $O(1)$ because that suffices to store the local variables v_i , V_i , $num_children$, and $child_count$.

For non-aggregation based LAP, the space at P_i for V_i equals the number of nodes in the sub-tree rooted at P_i . At a distance i from P_r , this is d^{k-i} . In the worst case, this is $O(n)$.

Bandwidth cost complexity: One *State* message is sent on each local BFST channel for both aggregation based LAP and non-aggregation based LAP. The size of the *State* message is $O(1)$ for aggregation based LAP and $O(n)$ for non-aggregation based LAP. Hence, the bandwidth cost complexity for the two classes is $O(1)$ and $O(n)$, respectively.

TABLE IV
COMPLEXITY COMPARISON BETWEEN LAP ALGORITHMS AND SOME EXISTING
ALGORITHMS TO DETECT STABLE PREDICATES.

Metric	Message Count	Message Size	Storage	Bandwidth
Chandy-Lamport (6) (only snapshot recording considered)	$O(N^2)$	$O(N^2)$	$O(N)$	$O(1)$
Mattern's Non-FIFO Snapshot (11) (only snapshot recording considered)	$O(N)$	$O(N^2)$	$O(N)$	$O(N)$
LDP-Basic (4) (recording and predicate evaluation)	$O(d^{k-1})$	$O(kd^{k-1})$	$O(k)$	$O(d^{k-2})$
LDP-Snapshot (4) (recording and predicate evaluation)	$O(d^{2k-2})$	$O(kd^{2k-2})$	$O(k)$	$O(d^{k-1})$
Stable LAP Initialization cost (Algorithm 1)	$O(nkd)$	$O(n^2)$	$O(n)$	$O(n)$
Stable LAP (aggregation based) (Algorithms 2+3)	$O(n)$	$O(n^2)$	$O(n)$	$O(n)$
Stable LAP (non-aggregation based) (Algorithms 2+3)	$O(n)$	$O(n^2)$	$O(n)$	$O(n)$

3.5.4 Comparison with Other Algorithms

From the previous analysis, we can see that all the algorithms proposed in this chapter are indeed scale-free. Taking the dominant complexity of those algorithms, we compare them with the complexities of some existing algorithms to detect stable predicates. Table IV shows the result.

Notice that both Chandy-Lamport's and Mattern's snapshot algorithms have complexities that are affected by N . This is true even when using these algorithms to detect LAP because these algorithms do not have the features that a scale-free solution requires. Further, only the cost of snapshot recording is listed; Mattern's algorithm has an added cost of spanning tree or

ring creation, and both these algorithms have another added cost of global state collection for predicate evaluation, that they do not address.

In a degree- d bounded network, where $n = d^{k-1}$, both *LDP-Basic* and *LDP-Snapshot* have a similar complexity compared to LAP algorithms. However, *LDP-Basic* and *LDP-Snapshot* cannot detect a predicate within a local region that is more complex than a *linear topology* and they cannot work in a system with non-FIFO channels or using multi-hop channels, as outlined earlier in Table I.

The costs of our LAP algorithms can be split into two parts. (1) A one-time initialization cost (of Algorithm 1) for creating a BFST for predicate detection for $Q = (\phi, k, P_r)$. (2) The recurring cost incurred by Algorithms 2 and 3 each time the predicate Q needs to be evaluated.

3.6 Special Cases

Repeated Detection

The algorithms for detecting locality-aware predicates we introduced can also repeatedly detect LAP predicates within the same region. To achieve repeated detection within the same region, Algorithm 1 needs to run only once for constructing the local BFST. Each time a new detection begins, a new consistent sub-cut needs to be constructed. Hence Algorithm 2 will run repeatedly; we can alternate the roles of the two colors or cyclically use two from a set of three colors.

Weaker Stable Predicates

The locality-aware predicates we discussed so far target stable predicates. By detecting a weaker form of stable predicates, we can further cut down the algorithm complexity.

A weaker form of stable predicates, named as *strong stable* predicate, was defined in (41).

A *strong stable predicate* is a stable predicate that, if true on some consistent cut, must remain true on all subsequent consistent cuts. Termination and deadlock are strong stable, though distributed garbage collection is not (41).

Detecting a strong stable predicate requires recording only consistent process states; observing the corresponding consistent channel states is not necessary. So, to detect a locality-aware strong stable predicate, we can simplify the second stage, namely that of constructing a consistent sub-cut. Applying only the *white/red coloring* technique is sufficient for detecting locality-aware strong stable predicates. This reduces the message size complexity of Algorithm 2 to $O(n)$, and its storage and bandwidth complexities to $O(1)$ (refer to Table III). Consequently, the corresponding entries for strong stable LAP (aggregation-based) and strong stable LAP (non-aggregation based) in Table IV will also reduce.

Modeling Area of Interest

If reduction of the initialization algorithm's complexity is needed, we can sacrifice the accuracy of modeling the area of interest. We can use a spanning tree instead of the local BFST, potentially leaving some processes within the area of interest unconsidered, to reduce the complexity of Algorithm 1. Compared with the complexities of Algorithm 1 discussed in Table III, we can achieve $O(nd)$ message count complexity and message size complexity, and $O(1)$ complexity in storage cost and bandwidth cost if we construct only a spanning tree. The changes to Algorithm 1 are as follows. In line (4), replace the test by " $dist = \infty$ "; delete lines (7-8); in line (31), replace "BFST" by "ST".

CHAPTER 4

DETECTING UNSTABLE LOCALITY-AWARE PREDICATES

This chapter is based on our previous publication (30). In this chapter, we further explore the problem of LAP detection. Besides stable predicates, another important type of predicate is the unstable predicate whose satisfaction keeps fluctuating with time. This type of predicate is also very common during the execution of a distributed program. Unstable predicate models the dynamic global properties during the execution of a distributed program. Detecting such predicates is important for various purposes such as monitoring, synchronization, coordination, and debugging. Thus, this makes detecting unstable conjunctive LAPs a relevant problem in large-scale locality-driven networks.

The rest of this chapter is organized as follows. Section 4.1 presents the basic detection algorithm which is scale-free. Section 4.2 presents the encoded vector clock (EVC) technique and shows how we use it to optimize the detection algorithm.

4.1 Detecting Unstable Conjunctive LAP

4.1.1 Establishing the Regional Vector Clock

We already showed in the previous chapter how to model the local region and construct a local BFST to represent it. To detect unstable conjunctive LAPs, we cannot directly apply the detection algorithms (17; 1) within the local region. 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, 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 (1) 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 4.

Algorithm 4: Establishing Regional Vector Clock (Code for P_i in the region)

integer: VID
array of **integer:** V
HashMap of $\langle \mathbf{integer}, \mathbf{integer} \rangle$: 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 (33; 34).

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 $\forall k \in [1 \dots n], V[k] = \max(V[k], U[k]);$
 $V[i] = V[i] + 1$; before delivering the message.

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

4.1.2 Detecting Unstable Conjunctive LAP

With the regional vector clock established, we can run the detection algorithm given as Algorithm 5, based on (1), within the detection region. Process P_r locally maintains n queues, Q_1, Q_2, \dots, 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

Algorithm 5: Detecting Unstable Conjunctive LAP, adapted from (1)

queues for all n processes in the detection region: $Q_1, Q_2, \dots, Q_n \leftarrow \perp$
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* **do**
 6 $newUpdatedQueues = \{\}$;
 7 **for each** $a \in updatedQueues$ **do**
 8 **if** Q_a *is not empty* **then**
 9 $x = \text{head of } Q_a$;
 10 **for** $b = 1 \dots n, b \neq a$ **do**
 11 **if** Q_b *is not empty* **then**
 12 $y = \text{head of } Q_b$;
 13 **if** $\min(x) \not\leq \max(y)$ **then**
 14 add b to $newUpdatedQueues$;
 15 **if** $\min(y) \not\leq \max(x)$ **then**
 16 add a to $newUpdatedQueues$;
 17 **if** $\max(x) < \min(y)$ **then**
 18 add a to $newUpdatedQueues$;
 19 **if** $\max(y) < \min(x)$ **then**
 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;

from all n processes, P_r repeatedly checks the heads of all n queues using the conditions in (Equation 2.2) or (Equation 2.1) in Section 2 to check whether $Possibly(\Phi)$ or $Definitely(\Phi)$ 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.

4.1.3 Tickling at Relevant Communication Events

In Algorithm 5, 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 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.*

Figure 6 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

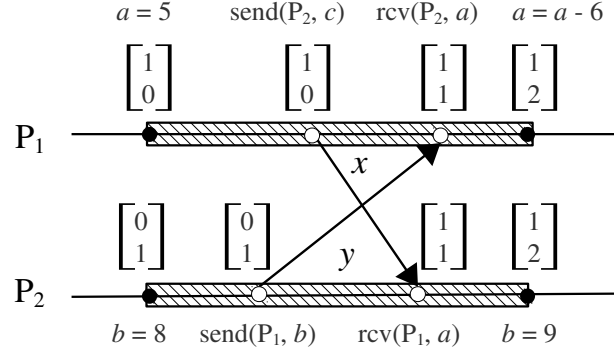


Figure 6. Illustration of the “relevant event” test. Each process has local variables a , b , and c . The conjunctive predicate $\psi = a_1 > 3 \wedge b_2 = 8$ is to be detected. The operations performed at each event are shown. By not ticking the vector clock for message send and receive events that are not relevant, the causal relations between the two intervals x and y at processes P_1 and P_2 are still correctly captured.

$\psi = a_1 > 3 \wedge b_2 = 8$. 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 6. 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 4.2.

4.2 Encoded Vector Clock (EVC) Optimization

Although the solution proposed in Section 4.1 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 (47), 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 (47). 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 4.1. Thus, a regional vector clock containing n elements

$$V = \langle v_1, v_2, \dots, v_n \rangle$$

can be encoded by n distinct prime numbers p_1, p_2, \dots, p_n as:

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

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.

4.2.1 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

$$V_1 = \langle v_1, v_2, \dots, v_n \rangle \text{ and } V_2 = \langle v'_1, v'_2, \dots, v'_n \rangle$$

merging them yields:

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

The encodings of V_1 , V_2 , and U are:

$$Enc(V_1) = p_1^{v_1} * p_2^{v_2} * \dots * p_n^{v_n}$$

$$Enc(V_2) = p_1^{v'_1} * p_2^{v'_2} * \dots * p_n^{v'_n}$$

$$Enc(U) = \prod_{i=1}^n p_i^{\max(v_1, v'_1)}$$

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 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:

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

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

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

Thus,

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

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

TABLE V
CORRESPONDENCE BETWEEN VECTOR CLOCKS AND EVC

Operation	Vector Clock	Encoded Vector Clock
Representing clock	$V = \langle v_1, v_2, \dots, v_n \rangle$	$Enc(V) = p_1^{v_1} * p_2^{v_2} * \dots * p_n^{v_n}$
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 $Enc(V_2) \bmod Enc(V_1) = 0$

The correspondence between the three basic operations of the vector clock and EVC is shown in Table V. These operations using EVC are illustrated in Figure 7. 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 4.1.3. In that case, the EVC timestamp 27000 in Figure 7 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

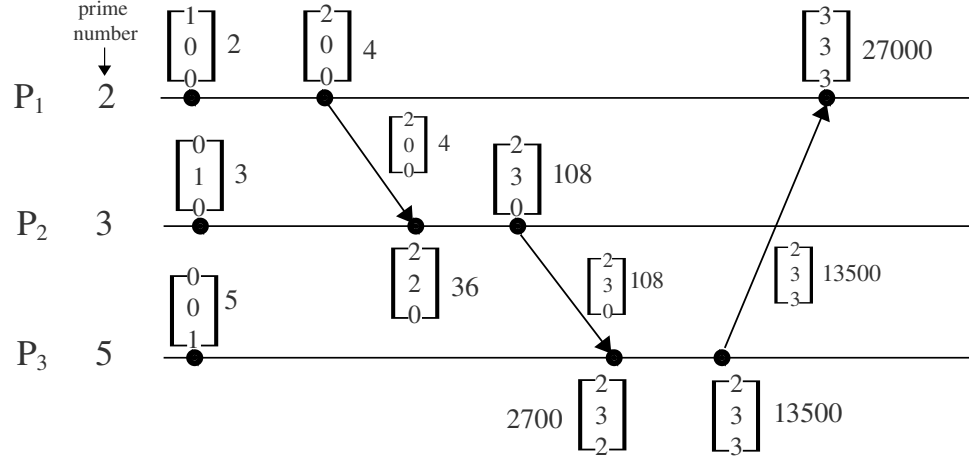


Figure 7. Illustration of using EVC for capturing causal relations. The local prime number for each process is shown beside its ID. The vectors shown in the diagram are only explaining the EVC timestamps. In real scenarios, only the number shown beside each vector is stored and transmitted.

receives. When this happens, P_j simply executes the merge operation by calculating the LCM of two numbers.

Figure 8 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 4.1.3. In that case, the EVC timestamp 72 in Figure 8 now is only 6.

After using EVC, when P_r compares two vector clock timestamps at line 13, 15, 17, and 19 in Algorithm 5, P_r will compare two EVC timestamps using the operation described in Table V. 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.

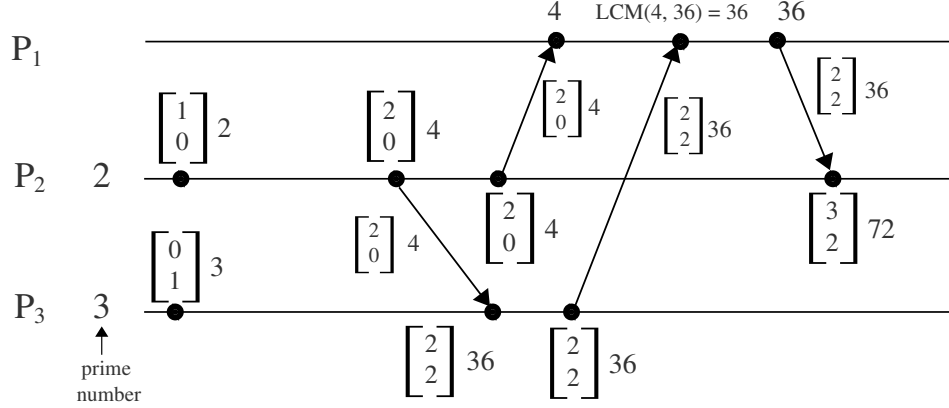


Figure 8. Illustration of using EVC for capturing causal relations within a local region of the network. Here, EVC is built for only P_2 and P_3 . The local prime numbers for those two process are shown beside its ID. Again, the vectors in the diagram are only to help understand the EVC timestamps.

4.2.2 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:

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

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 (48) 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 VI, 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.

TABLE VI
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)

For a system with N processes and assuming the local predicate becomes true at most m times at each process, Algorithm 5 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 5, 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 5 a better scale-free solution for detecting unstable conjunctive LAP.

4.2.3 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 (48) 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 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.

CHAPTER 5

HIERARCHICAL REPEATED DETECTION

This chapter is based on our previous publication (31). In this chapter, we present a decentralized algorithm that detects *Definitely*(Φ) in a large-scale system. This algorithm assumes a pre-constructed spanning tree (1; 49) 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. If one node fails, the detection of the predicate in the remaining processes could be easily resumed because the hierarchical detection manner gives our algorithm the ability to detect a partial predicate of the global predicate. 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 (5). 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.

The rest of the chapter is organized as follows. Section 5.1 presents the hierarchical detection algorithm and its theoretical foundation. Section 5.2 shows how our algorithm deals with node failures and node mobility. Section 5.3 analyses the complexity of the hierarchical detection algorithm.

5.1 Hierarchical Detection

5.1.1 Basic Idea and Challenges

Our hierarchical detection algorithm works as follows. We assume a spanning tree is already constructed in the system (1; 49). The algorithm utilizes this spanning tree to establish a hierarchy for detecting *Definitely*(Φ). 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: (i) how to aggregate a set of intervals, and (ii) how to do repeated detection of *Definitely*(Φ) using aggregated intervals from a lower level in the hierarchy.

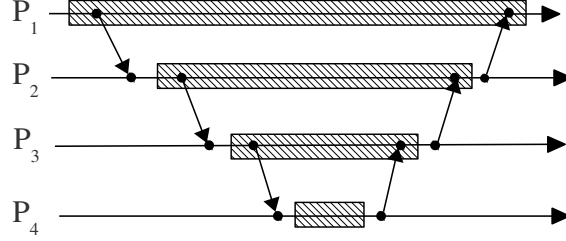


Figure 9. The approach in (17) works only if the intervals are nested.

In (17), 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 (17), 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, \dots, 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 Figure 9. However, such a relation need not always hold when $Definitely(\Phi)$ is satisfied.
2. The approach in (17) 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 Figure 10(a). From Figure 10(b), 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 (17), 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.

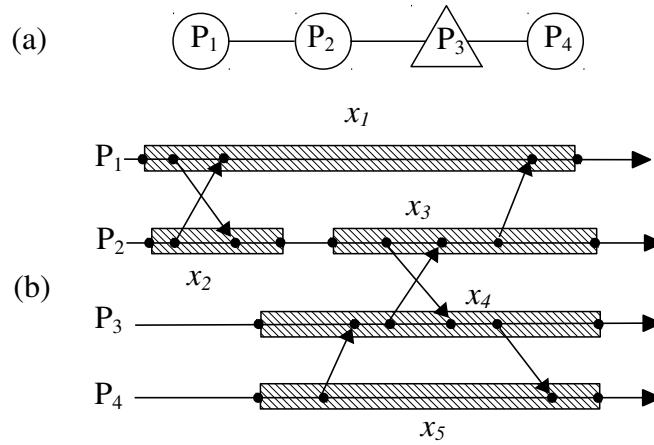


Figure 10. (a) The spanning tree consists of 4 processes. (b) Timing diagram showing the relation between intervals.

Without a proper way to aggregate intervals and without a way to repeatedly detect predicates, the approach given in (17) will fail to detect the predicates at the intermediate nodes as well as at the top of the hierarchy.

5.1.2 Example Scenario of Our Algorithm

In this subsection, still using Figure 10, 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 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.

5.1.3 Aggregation of Intervals to Detect *Definitely*(Φ)

In (17; 43), it was shown that for *Definitely*(Φ) 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*(Φ). We first consider the scenario where *Definitely*(Φ) has been detected in each of the two sets of intervals X and Y and we want to detect *Definitely*(Φ) in $X \cup Y$.

Assume now, we have 4 processes in the system with their timing diagram shown in Figure 11(a). 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.

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

$$\forall i, j \in \{1, 2\}, \min(x_i) < \max(y_j) \wedge \min(y_j) < \max(x_i) \quad (5.1)$$

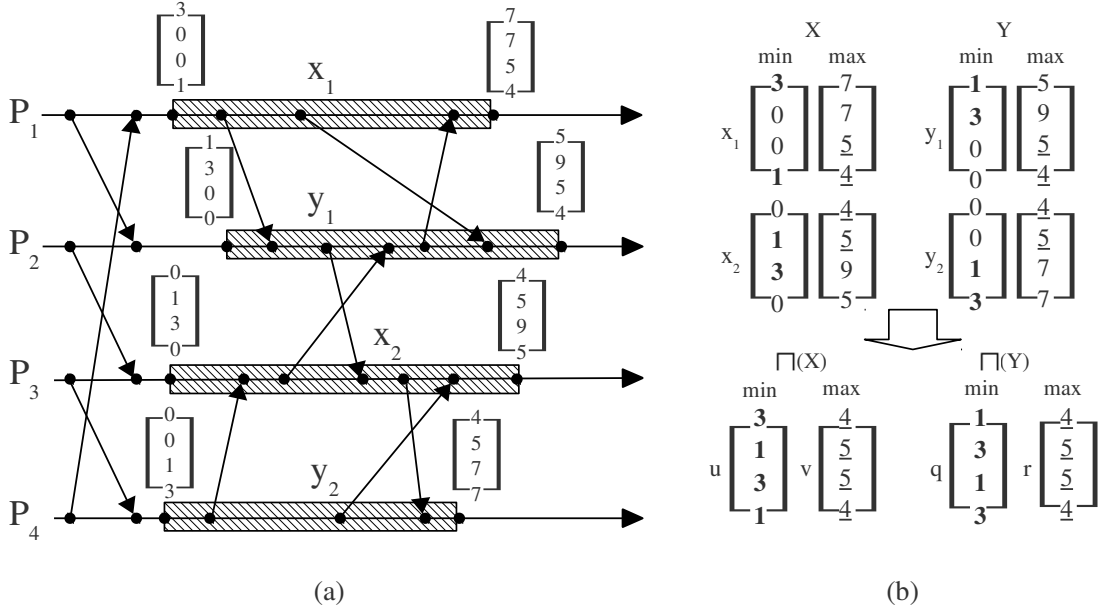


Figure 11. Example showing the aggregation of intervals for detecting $Definitely(\Phi)$. (a) The timing diagram of the system is given. An interval from each process is marked in shade along with the vector clock timestamps identifying the lower and higher bounds. (b) The two sets of intervals X and 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.

From Figure 11(b), 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 Equation 5.1 is equivalent to

$$\forall j \in \{1, 2\}, u < \max(y_j) \quad (5.2)$$

Likewise, taking the component-wise minimum of $\max(y_1)$ and $\max(y_2)$ (illustrated in underline) to form another new vector r , Equation 5.2 is equivalent to $u < r$. The same operations

can also be applied to show the second conjunct of Equation 5.1 using aggregated vectors q and v .

This gives the inspiration to aggregate a set of intervals into a single one in order to detect *Definitely*(Φ) in a larger set of intervals. For sets X and Y in Figure 11, 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 Figure 11(b).

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:

$$\forall i \in [1, n], \min(\sqcap(X))[i] = \max_{x \in X}(\min(x)[i]) \quad (5.3)$$

$$\forall i \in [1, n], \max(\sqcap(X))[i] = \min_{x \in X}(\max(x)[i]) \quad (5.4)$$

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

Theorem 2. *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 $\text{overlap}(X)$ and $\text{overlap}(Y)$, now we have $\text{overlap}(Z)$. \square

Theorem 2 shows that we can aggregate a set of intervals X into a single interval $\sqcap(X)$ which can represent the entire set in detecting $\text{Definitely}(\Phi)$ 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 2 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 2 to scenarios involving more than two sets of intervals.

Lemma 1. *Let X_1, X_2, \dots, 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 $\text{overlap}(Z)$ iff $\wedge_{i=1}^d \text{overlap}(X_i) \wedge \text{overlap}(\sqcap(X_1), \sqcap(X_2), \dots, \sqcap(X_d))$*

Proof. (\Rightarrow) $\wedge_{i=1}^d \text{overlap}(X_i)$ is clearly true since $X_i \subset Z$. Since $\text{overlap}(Z)$, we have $\forall i, j \in [1, d], \text{overlap}(X_i \cup X_j)$. Thus, according to Theorem 2, we have

$$\forall i, j \in [1, d], \text{overlap}(\sqcap(X_i), \sqcap(X_j)).$$

This means, $\forall i, j \in [1, d], \min(\sqcap(X_i)) < \max(\sqcap(X_j))$. Thus, we have

$$\text{overlap}(\sqcap(X_1), \sqcap(X_2), \dots, \sqcap(X_d))$$

(\Leftarrow) Since $\bigwedge_{i=1}^d \text{overlap}(X_i) \wedge \text{overlap}(\sqcap(X_1), \sqcap(X_2), \dots, \sqcap(X_d))$, we have

$$\forall i, j \in [1, d], \text{overlap}(X_i \cup X_j).$$

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 $\text{overlap}(Z)$. \square

For our hierarchical detection algorithm, each process P_i in the spanning tree detects $\text{Definitely}(\Phi)$ 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 ,

$$\sqcap(\sqcap(X), \sqcap(Y)) = \sqcap(X \cup Y) \tag{5.5}$$

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

5.1.4 Repeated Detection

In (5), the author showed how repeated detection can be done in the centralized *Definitely*(Φ) 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 3. *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 Equation 5.3 and Equation 5.4, 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 $\forall x_i, x_j \in X, \forall l \in [1, n], \min(x_i)[l] \leq \max(x_j)[l]$. Thus, $\forall l \in [1, n], \min(\sqcap(X))[l] \leq \max(\sqcap(X))[l]$. 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 Equation 5.3 and Equation 5.4, 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, $\max(\sqcap(X)) < \min(\sqcap(X'))$. \square

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 $\text{succ}(x)$. 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 $\text{overlap}(x_i, \text{succ}(x_j))$ is true. From (5), we know that this is equivalent to

$$\min(\text{succ}(x_j)) < \max(x_i) \tag{5.6}$$

Then, if for all intervals $x_j (j \neq i)$ from the solution set X , Equation 5.6 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

$$\text{remove } x_i \text{ iff } \forall x_j \in X (j \neq i), \min(succ(x_j)) \not\leq \max(x_i) \quad (5.7)$$

Since $\max(x_j) < \min(succ(x_j))$, from (5), we know the test condition in Equation 5.7 can be approximated to

$$\text{remove } x_i \text{ iff } \forall x_j \in X (j \neq i), \max(x_j) \not\leq \max(x_i) \quad (5.8)$$

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 Equation 5.8 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 4. (*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 Equation 5.8 \Rightarrow Equation 5.7, any interval removed using the condition in Equation 5.8 will also satisfy the condition in Equation 5.7. Thus, those intervals cannot be part of any future solution set. Therefore, even if Equation 5.8 is only an approximation, it still guarantees safety. \square

Theorem 5. (*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 Equation 5.8 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 Equation 5.8 holds for any solution set. Thus, Equation 5.8 guarantees liveness. \square

With the safety and liveness of the condition in Equation 5.8 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.

5.1.5 Hierarchical Detection Algorithm

With Theorems 2, 4 and 5, we have the theoretical foundation for the hierarchical detection algorithm outlined in Section 5.1.1. The algorithm is listed in Algorithm 6. 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

Algorithm 6: Hierarchical decentralized detection of conjunctive definitely predicates, adapted from (5) (Code for P_i)

```

number of children:  $l$ 
queue for  $P_i$ :  $Q_0 \leftarrow \perp$ 
queues for children:  $Q_1, Q_2, \dots, Q_l \leftarrow \perp$ 
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          $x = \text{head of } Q_a$ ;
9         for  $b = 0 \dots l (b \neq a)$  do
10          if  $Q_b$  is not empty then
11             $y = \text{head of } Q_b$ ;
12            if  $\min(x) \not\leq \max(y)$  then
13               $\perp$  add  $b$  to  $newUpdated$ ;
14            if  $\min(y) \not\leq \max(x)$  then
15               $\perp$  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        $\perp$  report  $\sqcap(\text{heads of all queues})$  to parent;
21     else
22        $\perp$  report predicate detected;
23   for  $a = 0 \dots l$  do
24      $count = 0$ ;
25     for  $b = 0 \dots l (b \neq a)$  do
26       for  $c = 1 \dots n$  do
27         if  $\max(\text{head}(Q_a))[c] < \max(\text{head}(Q_b))[c]$  then
28            $\perp$   $count++$ ; break;
29     if  $count = l$  then
30        $\perp$  add  $a$  to  $newUpdated$ ;
31   Delete heads of all  $Q_a$  where  $a \in newUpdated$ ;
32    $updatedQueues = newUpdated$ ;

```

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)).

Each time the predicate is detected at some process, Lines (23)-(32) 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 $\forall x_j \in X (j \neq i), \max(x_j) \not\prec \max(x_i)$. 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 $\max(x_j) \not\prec \max(x_i)$ 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 Equation 5.8. Thus, we can safely remove x_i from the corresponding queue. In Algorithm 6, 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.

Although Algorithm 6 has the same basic structure as the centralized algorithm given in (5), it is essentially different. Algorithm 6 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 6 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 2, 4, and 5 together guarantee that Algorithm 6 is correct, meaning that all the predicate occurrences in the system are detected and there are no false alarms.

5.2 Fault-Tolerance

5.2.1 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 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 Figure 12. 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 (50) and Jeon et al. (51). 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.

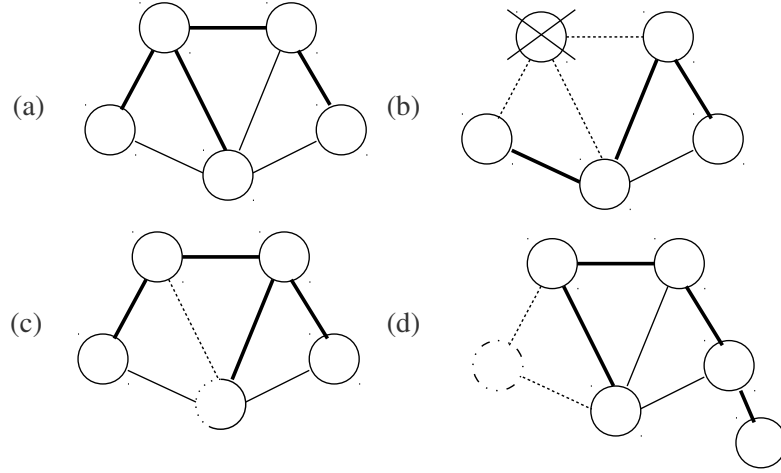


Figure 12. (a) The initial topology of the spanning tree. Bold lines indicate tree edges, while dashed lines indicate disconnected tree edges due to either (b) node crash, (c) decreasing power or (d) node mobility. In order to reconstruct the spanning tree, new tree edges are added.

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.

5.2.2 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 Figure 13 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 Figure 13 (c). The spanning tree is initially constructed as shown in Figure 13 (a). 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 Figure 13 (b).

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

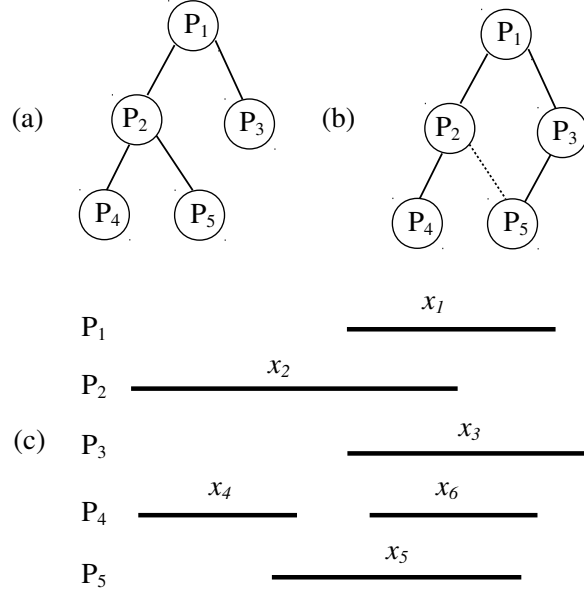


Figure 13. (a) The spanning tree consists of 5 processes. (b) Due to either node mobility or decreasing power of process P_5 , the structure of the spanning tree is changed. (c) Intervals on each process are represented by line segments.

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 .

5.2.3 Algorithm Augmentation for Fault-Tolerance

The addition of fault-tolerance to the hierarchical detection algorithm is listed in Algorithm 7. 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 2, 4, and 5 still ensure the correctness of further detections of the predicate.

Algorithm 7: 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 $flag_j = \text{true};$
 P_i gains a new child P_k :
2 **if** Q_k *exists* **then**
3 $flag_k = \text{false};$
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** $\text{overlap}(x_k, previous)$ **then**
7 send $\cap(x_k, previous)$ 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 ;

Even if multiple occurrences of the failures in Section 5.2.1 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 7, 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 7 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 6. *(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 2 is not impacted because the logic to aggregate intervals stays the same. Theorem 3 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 3 not affected, Theorems 4 and 5 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 2, 4, 5, and 6, 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.

5.3 Complexity

We analyse the complexity of the hierarchical detection algorithm using three metrics: space complexity, time complexity and message complexity, in terms of the following parameters (Notice that $n = d^h$):

- n : the number of nodes in the network
- p : the maximum number of intervals per process
- d : the maximum number of children any process in the spanning tree can have
- h : the height of the spanning tree
- α : the probability that intervals from d children overlap and can be aggregated at one higher level
- m : the maximum number of messages sent by any process.

Table VII summarized the results.

5.3.1 Message Complexity

In the hierarchical detection algorithm, the messages are transmitted along the edges of the spanning tree. For the leaf nodes in the spanning tree, each time an interval completes locally, it sends this interval to its parent. A non-leaf node only sends one aggregated interval to its parent once it detects the predicate within the subtree rooted at itself.

At a leaf node (level 1), all intervals occurring locally are sent to its parent. At level i , the number of aggregated intervals generated by a single process is $d\alpha$ times the number of intervals received from any child in level $i - 1$. This can be justified using the reasoning in (5). Each

TABLE VII
COMPLEXITY COMPARISON BETWEEN HIERARCHICAL DETECTION AND THE
CENTRALIZED REPEATED DETECTION ALGORITHM

	Our Hierarchical Algorithm	Centralized Repeated Detection Algorithm (5)
Space Complexity	$O(pn^2)$ (distributed across all processes)	$O(pn^2)$ (at the sink node)
Time Complexity	$O(d^2pn^2)$ (distributed across all processes)	$O(pn^3)$ (at the sink node)
Message Complexity	pn	$p^{\frac{(d^h-2d)(dh-d-h)-d}{(d-1)^2}}$

time an aggregated interval is generated at a certain process, the predicate is detected within the corresponding subtree. Hence, at level i , $\alpha^{i-1}d^{i-1}p$ number of aggregated intervals will be sent to level $i+1$. Then, we can derive the total number of messages transmitted in the system for hierarchical detection algorithm. For a spanning tree of degree d and height h ,

$$\begin{aligned}
 \text{total \# of msgs} &= \sum_{i=1}^{h-1} d^{h-i} p d^{i-1} \alpha^{i-1} \\
 &= p d^{h-1} \frac{1 - \alpha^{h-1}}{1 - \alpha}
 \end{aligned} \tag{5.9}$$

We now compare with the message complexity of the centralized repeated detection algorithm (5). Notice that each message in hierarchical detection is transmitted only 1 hop, always to the immediate parent. For a message that traverses h hops in a wired or wireless network, it is equivalent to h point-to-point messages, since the communication channels are occupied h

times. Thus, when running the centralized repeated detection algorithm in the same network where the sink collects intervals from the other processes via a spanning tree of degree d and height h , we need to account for the cost coming from each message having to traverse several hops to reach the sink.

In the centralized algorithm (5), messages sent from level i need to traverse $h - i$ hops to reach the sink. Furthermore, each local interval needs to be transmitted all the way to the sink. So at level i , across all nodes at that level in the spanning tree, $pd^{h-i}(h - i)$ messages are generated by the centralized algorithm. We can thus deduce the total number of messages generated by the centralized repeated detection algorithm (5).

$$\begin{aligned} \text{total \# of msgs} &= \sum_{i=1}^{h-1} pd^{h-i}(h - i) \\ &= ph \sum_{i=1}^{h-1} d^{h-i} - p \sum_{i=1}^{h-1} id^{h-i} \end{aligned} \quad (5.10)$$

Let $k = \sum_{i=1}^{h-1} id^{h-i}$, then

$$\begin{aligned} dk &= \sum_{i=1}^{h-1} id^{h-i+1} \\ (d-1)k &= \sum_{i=2}^h d^i + (h-1)d \\ &= \frac{d^2(1 - d^{h-1})}{1 - d} + (h-1)d \\ k &= \frac{d^{h+1} + d^2h - 2d^2 - dh + d}{(d-1)^2} \end{aligned} \quad (5.11)$$

Substituting Equation 5.11 into Equation 5.10, we have

$$\text{total \# of msgs} = p \frac{(d^h - 2d)(dh - d - h) - d}{(d - 1)^2} \quad (5.12)$$

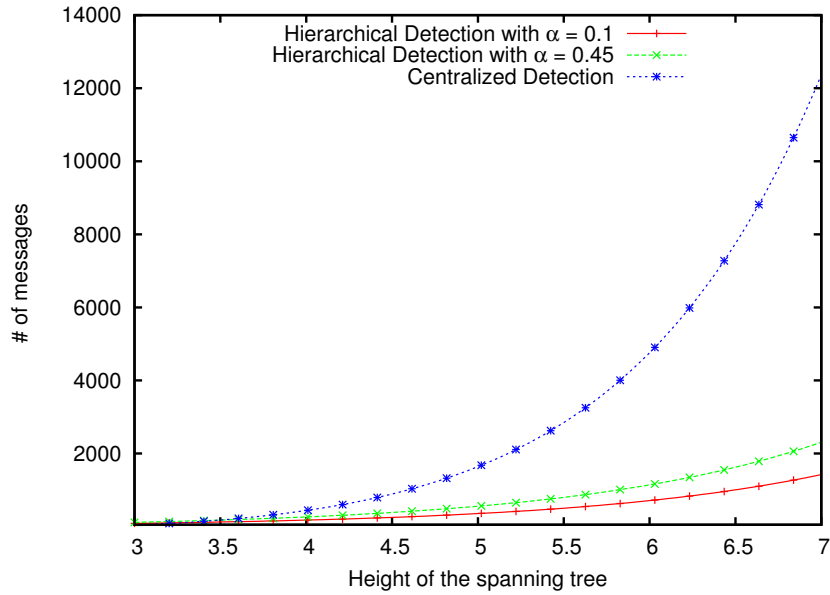


Figure 14. Message complexity comparison between hierarchical and centralized detection, with $d = 2, p = 20$.

Figure 14 and Figure 15 compare the message complexity between hierarchical detection algorithm (Equation 5.9) and the centralized repeated detection algorithm (5) (Equation 5.12) with different parameters. In Figure 14, $d = 2$ and α is set to 0.1 and 0.45. In Figure 15, α takes the same values while d is set to 4. From these two graphs, we can observe that for the same p , the height h and degree d of the spanning tree, or equivalently the size of the network, impacts

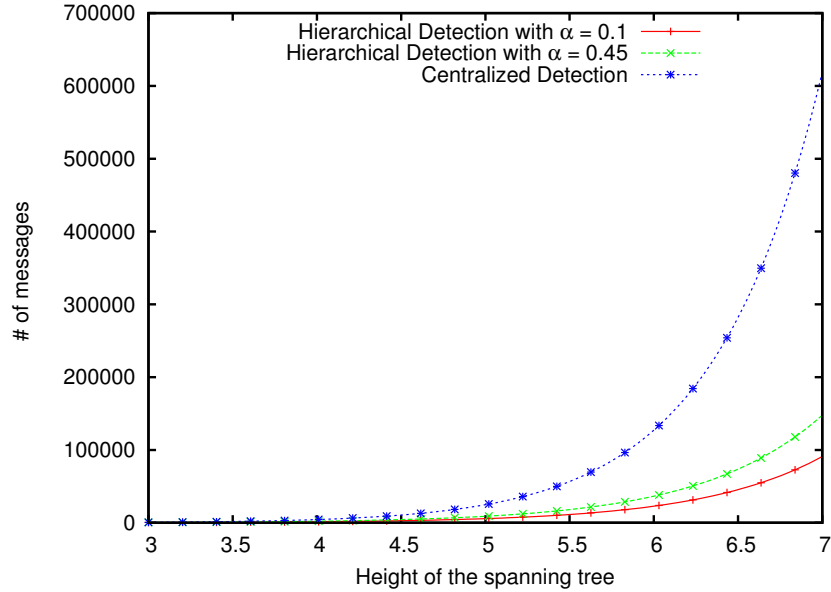


Figure 15. Message complexity comparison between hierarchical and centralized detection, with $d = 4, p = 20$.

the total number of messages. Also, with a smaller α , the number of messages decreases. Furthermore, observe that p is a linear factor in both Equation 5.9 and Equation 5.12. So, if we fix other parameters, as p increases, the total number of messages also increases linearly. Hence, the hierarchical detection algorithm has a better message complexity compared to the centralized repeated detection algorithm, especially when the system is large-scale.

Furthermore, for two networks of the same size (n being the same), we compare the message complexity of hierarchical detection algorithm and the centralized detection algorithm (5) running on different spanning trees generated within the network in Figure 16. For two spanning trees, one with $d = 2$ and $h = 12$ while the other with $d = 8$ and $h = 4$, we observe that they represent networks of the same size since $2^{12} = 8^4 = d^h$. From Figure 16, we can see that,

even if the two networks are of the same size, the one with a larger height of the spanning tree generates more messages. This can be reasoned from the fact that h is a dominant factor in Equation 5.9. Notice that, as far as $h > 2$, the hierarchical detection algorithm is different from the centralized one (5). When $h = 2$, then no aggregation will happen in the network, and the hierarchical detection algorithm becomes centralized. From this observation, we know that the hierarchical detection algorithm will have a better performance when the spanning tree generated in the network is flat, meaning that the spanning tree has a larger degree d and a smaller height h .

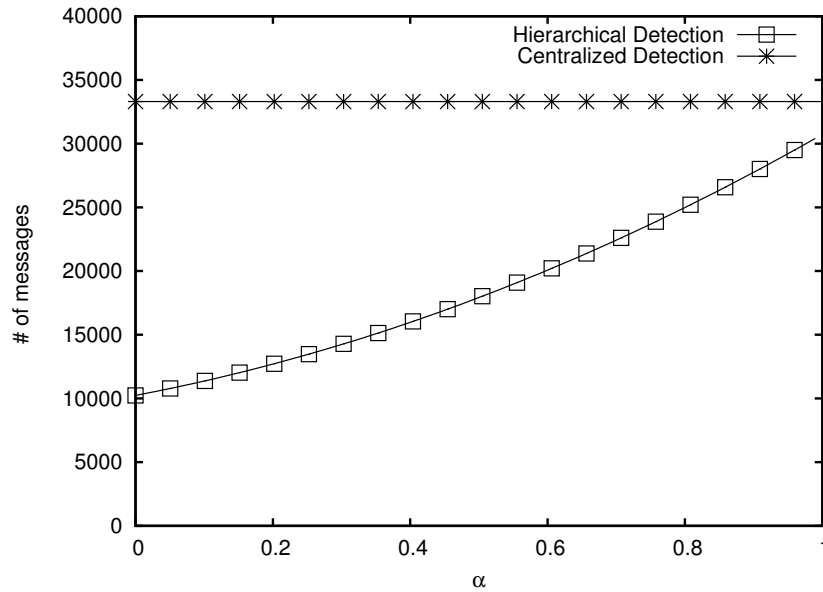


Figure 16. Message complexity comparison of hierarchical detection within networks of the same size but with different spanning tree topologies. Note that $p = 20$.

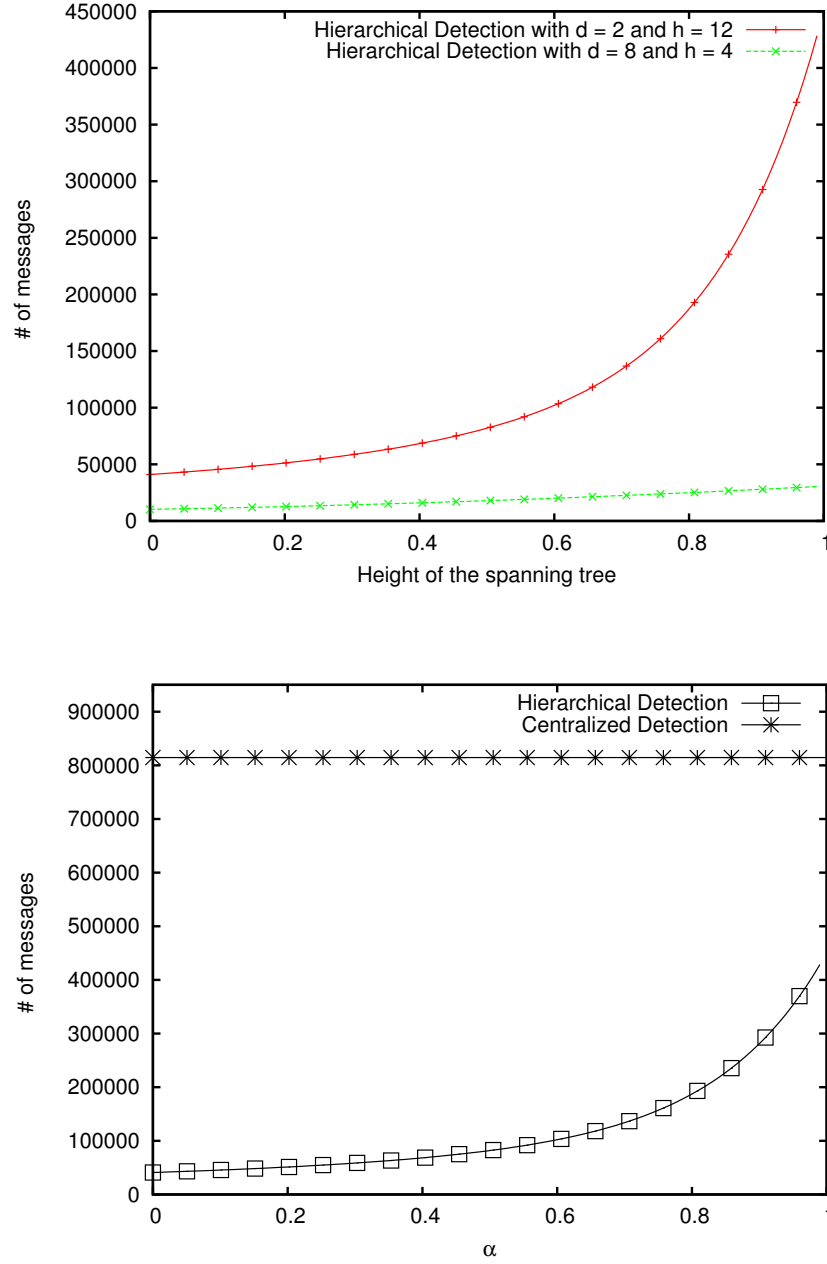


Figure 17. Message complexity comparison between hierarchical detection and centralized detection. (a) $d = 2, h = 12$ (b) $d = 8, h = 4$. Note that $p = 20$.

We also observe from Figure 17 that for the same spanning tree configuration, the centralized algorithm (5) will always generate more messages compared to the hierarchical detection algorithm, no matter what value α takes. This observation further demonstrates that the hierarchical detection algorithm has a better message complexity than the centralized algorithm (5).

5.3.2 Space Complexity

In hierarchical detection, each process other than the leaf nodes only needs to maintain $O(d)$ queues. The number of intervals in the local queue Q_0 is $O(p)$. Also, each process needs to store the aggregate intervals from its $O(d)$ children.

In the worst case, the space complexity is the number of intervals in all the nodes. Equation 5.9 gives the total number of aggregated intervals. Observe that $n = d^h$. Thus, the total number of aggregated intervals is affected by $\frac{1-\alpha^{h-1}}{1-\alpha}$, which is $\sum_{i=1}^{h-1} \alpha^{i-1}$ and is bounded by $h - 1$. As $n = d^h$, we know $\frac{1-\alpha^{h-1}}{1-\alpha}$ is $O(\log(n))$ worst case.

Furthermore, for a particular predicate Φ , α can be treated as a constant since α is only related to the predicate Φ and the execution of the distributed system. Thus $\frac{1-\alpha^{h-1}}{1-\alpha}$ is $O(1)$. $\frac{1-\alpha^{h-1}}{1-\alpha}$ only becomes large when $\alpha \rightarrow 1$ and h is very large. This means that almost all attempts to detect the predicate at every level of the hierarchy will succeed in a large-scale network. This is an impractical assumption. For a practical value of α , even as large as 0.5, $\frac{1-\alpha^{h-1}}{1-\alpha}$ is still ≤ 2 . For impractical values of α , $\frac{1-\alpha^{h-1}}{1-\alpha}$ will be bounded by $\min(h, \frac{1}{1-\alpha})$. From the above analysis, we can conclude that the total number of aggregated intervals stored in all processes is $O(pn)$.

The fact that $\frac{1-\alpha^{h-1}}{1-\alpha}$ is bounded by $\min(h, \frac{1}{1-\alpha})$ also shows that, with a flat spanning tree configuration, the performance of the hierarchical detection algorithm will be better.

In addition, each process will store its local $O(p)$ intervals, and that is an additional $O(pn)$ intervals across the whole network. Since the storage size of both regular intervals and aggregated intervals is $O(n)$, the storage cost of the hierarchical detection algorithm is $O(pn^2)$, distributed across all the nodes in the network.

Compared to the centralized repeated detection algorithm (5), which incurs an $O(pn^2)$ storage cost at the sink/root of the spanning tree, hierarchical detection algorithm does not place all the storage cost at a single process, which makes it suitable for large-scale systems where a single process cannot afford storing all the data.

5.3.3 Time Complexity

From Section 5.3.2, we know that there are $O(pn)$ aggregated intervals and $O(pn)$ non-aggregated intervals stored across all nodes in the network. When running the hierarchical detection algorithm, each node needs to check all intervals stored locally (Lines (1)-(22)) to detect *Definitely*(Φ). Since the total number of intervals is $O(pn)$, and each interval will be compared with $O(d)$ other intervals with each comparison taking $O(n)$ time, Lines (1)-(22) in the hierarchical detection algorithm will incur an $O(dn^2p)$ time complexity distributed across all the nodes in the network. For Lines (23)-(33), each time the predicate is detected at some node, this part of the code will be executed. Since Lines (23)-(33) compare the heads of $O(d)$ queues, each time Lines (23)-(33) execute, they will take $O(d^2n)$ time. Since the total number of aggregated intervals is $O(pn)$ across all nodes, the maximum number of times the predicate

can be detected across all levels in the spanning tree is $O(pn)$. Thus, Lines (23)-(33) incurs an $O(d^2n^2p)$ time complexity across all iterations and all nodes in the network. Furthermore, the augmented code for fault-tolerance will not affect the time complexity. Lines (a)-(e) incur an $O(n)$ time cost which is dominated by Line (1)-(33) and Lines (34)-(35) incur an $O(1)$ cost. Also, the code in Lines (36)-(45) will only be executed if the structure of the spanning tree is changed. So, in total, the time complexity of the hierarchical detection algorithm is $O(d^2n^2p)$, spread across all n nodes. We observe that, with a flat tree, the time complexity of the hierarchical detection algorithm will increase since the degree in the spanning tree becomes larger.

In a large-scale network running the hierarchical detection algorithm, $h > 2$, otherwise the algorithm becomes a centralized algorithm. Since $n = d^h$, we infer that $n > d^2$. Thus, comparing the hierarchical detection algorithm with the centralized repeated detection algorithm, which incurs an $O(pn^3)$ time complexity, the hierarchical detection algorithm has a lower time complexity, especially when h is large. Furthermore, this time complexity is distributed across all nodes, which is not the case in the centralized algorithm.

5.3.4 Cost of Maintaining the Spanning Tree

The cost of maintaining the spanning tree happens at a lower layer in the system. There are many existing works studying the maintenance and reconstruction of a spanning tree (50; 51). The reconstruction mechanism developed by Yang and Fei (50) adopted a proactive approach. In their solution, each node in the spanning tree pre-computes a rescue plan to calculate a MST between its parent and immediate children under certain degree constraint. Once a node fails,

all the affected nodes can directly communicate with their respective parents-to-be to quickly reconstruct the spanning tree. Note that, after the initial pre-computation, only those nodes affected by the failed one need to recalculate the rescue plan.

With such a proactive approach, the spanning tree reconstruction mechanism developed by Yang and Fei (50) shows a very good performance. The reconstruction mechanism will incur a time complexity of $O(d^2 \log(d))$ at each node affected by the failed node. Furthermore, the simulation results show that, with a network of 1600 nodes, the average time to reconstruct the spanning tree is always less than 400 ms. Compared with the duration of the intervals, which are typically seconds, the time spent to reconstruct the spanning tree after a node crashes or moves is very short. The simulation results also show that for reconstructing the spanning tree, each affected node only needs to contact 1 node to find its new parent. This also generates a very low bandwidth cost in the network.

As a conclusion, with a proper spanning tree reconstruction mechanism such as the one developed by Yang and Fei (50), the cost of maintaining the spanning tree is very low. Thus it is reasonable to put the emphasize on the cost of the detection algorithm itself, rather than the cost associated with maintaining the spanning tree.

CHAPTER 6

INSTANTANEOUS DETECTION

In this chapter, we propose an approximation algorithm for detecting predicate satisfactions in physical time. It adopts the hierarchical detection methodology to collaboratively detect predicates relying only on neighborhood communication. Compared with (27), the inaccuracy of our algorithm is bounded only by the message transmission delay over a single hop, which is considerably lower than that of a system broadcast especially when the scale of the network is large. Furthermore, different from (28), our approximation algorithm is based on *Possibly*(ϕ) modality. This enables our algorithm to detect more occurrences of predicate satisfactions in physical time. As a result, our detection algorithm incurs a low time/storage cost at each process and reaches a very high accuracy.

The rest of this chapter is organized as follows. Section 6.1 formally defines the *Instantaneously* modality. Section 6.2 presents the main idea of our approach and gives the detection algorithm. Section 6.3 shows the evaluation of our approach by means of simulations.

6.1 Instantaneously Modality

The *Instantaneously* modality can be better illustrated using the lattice of global states. Below, we consider the same example shown in Figure 1 redrawn in Figure 18 with the current run indicated. Similarly, The timing diagram of this distributed program's execution is shown in Figure 18(a). Event e_i^k denotes the k th event at process P_i . The corresponding lattice of

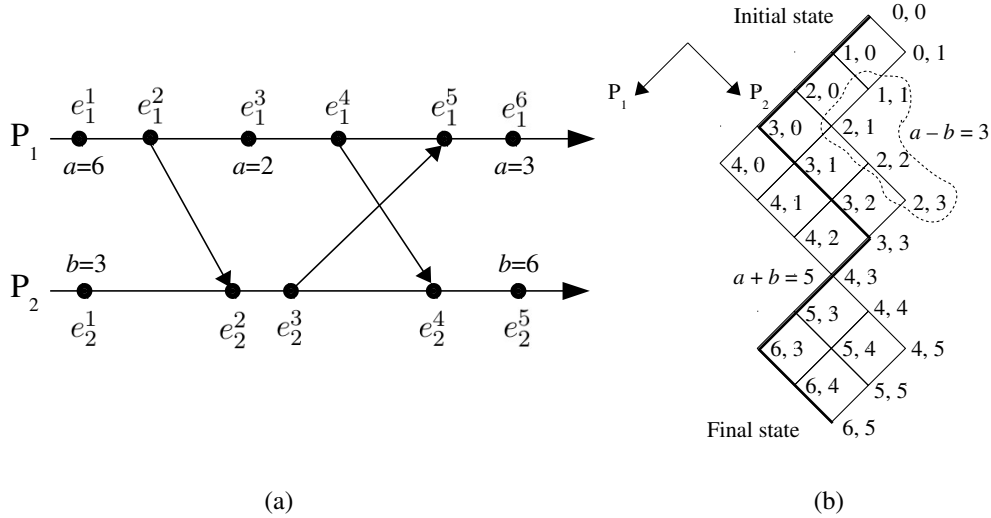


Figure 18. (a) The timing diagram of the execution. (b) The state lattice for the execution. Bold edges indicate current run δ .

global states for the execution is shown in Figure 18(b). Each state is labeled by a tuple (c_1, c_2) , where c_1 and c_2 are the event number at P_1 and P_2 , respectively.

Both *Possibly*(ϕ) and *Definitely*(ϕ) are concerned with all runs of the same distributed program. So, for a predicate $\phi = "a - b = 3"$, when we say *Possibly*(ϕ) holds or *Definitely*(ϕ) does not hold, they are all with respect to the entire state lattice in Figure 18(b). On the other hand, the *Instantaneously* modality focuses on what happened in physical time. The detection of predicate under such modality should test whether the predicate is satisfied only within the sequences of global states that occurred in physical time. Thus, predicate ϕ does not hold under *Instantaneously* modality in the execution depicted in Figure 18 because the current run δ does not traverse any global state in which ϕ is true. With this example, we formally define the *Instantaneously* modality as follows:

Definition 9. *Instantaneously(δ, ϕ): Within the current run δ , there exists a physical time global state of it in which ϕ holds.*

We observe that, without access to a global clock, predicates under the *Instantaneously* modality cannot be detected with a 100% accuracy. In the next section, we present a novel approximation algorithm that detects predicates under the *Instantaneously* modality with a high accuracy.

6.2 Detecting Predicates in Physical Time

6.2.1 Basic Ideas

In (28), the authors approximate predicate detection under the *Instantaneously* modality with detection algorithms for the *Definitely(ϕ)* modality. Although *Definitely(ϕ)* modality is concerned with all runs of the same distributed program, due to its property that a predicate satisfaction under the *Definitely(ϕ)* modality is guaranteed to happen in every run of the distributed program, the predicate satisfactions detected under *Definitely(ϕ)* modality will also occur in physical time. However, this property also imposes the limitation that any predicate satisfactions occurring in physical time that do not satisfy the *Definitely(ϕ)* detection condition (Equation 2.1) will not be detected.

Instead, the approach we adopt is based on *Possibly(ϕ)* modality. This frees the detection algorithm from the limitation imposed by the *Definitely(ϕ)* modality, but introduce its own limitations. Therefore, to deal with these limitations, we propose an innovative hierarchical detection technique for predicate detection under the *Possibly(ϕ)* modality. Hierarchical detection technique is a decentralized way to collaboratively detect predicates among a network span-

ning tree. This technique can reduce the time/storage cost on each process, thus making the algorithm applicable in a resource-constraint system. Furthermore, we give two approximation techniques that work on top of the hierarchical detection algorithm. The first approximation technique is to enable the detection algorithm to perform repeated detection, so our algorithm is capable of providing continuous monitoring in long-running applications. The second one is to focus the detection on the current run δ , so predicate satisfactions that meet the *Possibly*(ϕ) detection condition in Equation 2.2 yet do not occur in physical time are less likely to be reported. In the rest of this section, we will discuss the details of our detection algorithm.

6.2.2 Hierarchical Detection

In this subsection, we show how we decentralize the detection of the *Possibly*(ϕ) modality in a hierarchical fashion. The detection condition for *Possibly*(ϕ) is already given in Equation 2.2. If we know the process at which an interval x occurred, which we denote as $ORIG(x)$, then Equation 2.2 is equivalent to

$$\forall x_i, x_j \in X, \max(x_i)[ORIG(x_i)] > \min(x_j)[ORIG(x_i)], \quad (6.1)$$

We denote this property as *antichain*(X).

Now we assume 4 processes in the network with their timing diagram shown in Figure 19(a). The intervals occurring at each process is marked in shade, and the vector timestamps identifying the lower and upper bound of each interval are illustrated in the figure. Intervals x_1 from

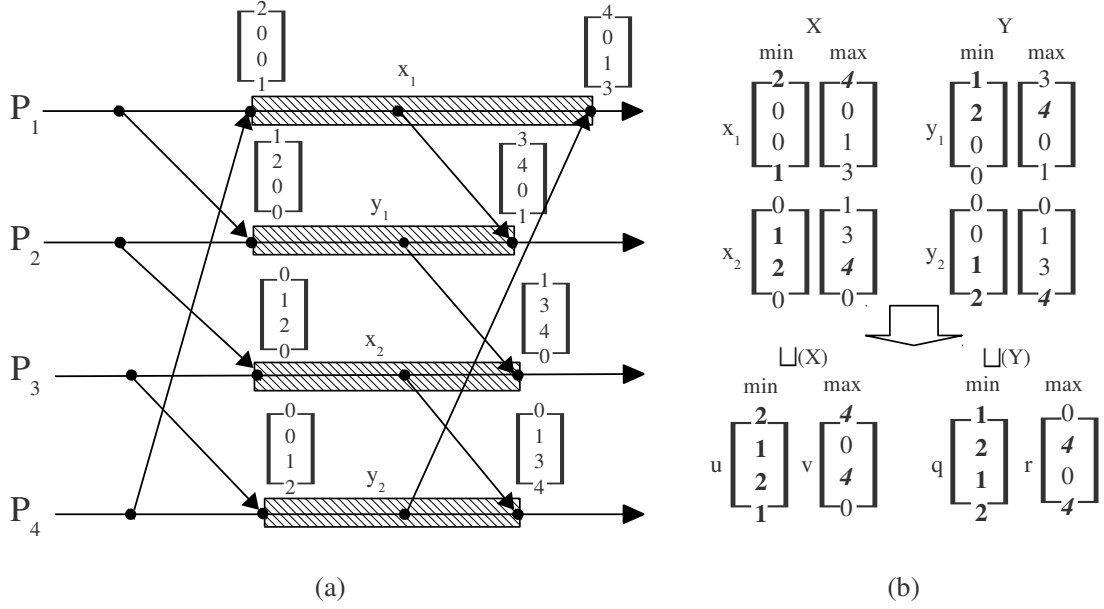


Figure 19. Examples showing the aggregation of intervals for detecting *Possibly*(ϕ). In (a), the timing diagram of the system is given. Intervals from each process is marked in shade along with the vector clock timestamps identifying the lower and higher bounds. In (b), the two sets of intervals X and Y consisting of intervals from (a) are shown. The way to aggregate each set is also illustrated in (b). Notice that, component-wise maximum among all lower bounds in the same set are marked in bold while the element corresponding to $ORIG(x_i)$ or $ORIG(y_i)$ for each interval x_i and y_i is marked in bold and italic.

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 . We can see that both $antichain(X)$ and $antichain(Y)$ are true.

Now, if we want to show that *Possibly*(ϕ) is detected in all 4 processes, we need to show that

$$\begin{aligned} \forall i, j \in \{1, 2\}, \max(x_i)[ORIG(x_i)] &> \min(y_j)[ORIG(x_i)] \wedge \\ \max(y_j)[ORIG(y_j)] &> \min(x_i)[ORIG(y_j)] \end{aligned} \quad (6.2)$$

Observe in Figure 19(b) that, if we take the component-wise maximum of $\min(y_1)$ and $\min(y_2)$ (illustrated in bold) to form a new vector q , we can see that the first conjunct in Equation 6.2 is equivalent to

$$\forall i \in \{1, 2\}, \max(x_i)[\textit{ORIG}(x_i)] > q[\textit{ORIG}(x_i)] \quad (6.3)$$

Furthermore, if we combine the upper bounds of intervals x_1 and x_2 , i.e., $\max(x_1)$ and $\max(x_2)$, into a new vector v by taking $\max(x_1)[\textit{ORIG}(x_1)]$ and $\max(x_2)[\textit{ORIG}(x_2)]$ (illustrated in bold and italic) to fill the corresponding position in v and leave the remaining elements in v as 0, then Equation 6.3 is equivalent to

$$\forall i \in \{1, 2\}, v[\textit{ORIG}(x_i)] > u[\textit{ORIG}(x_i)].$$

Using the same operations, the second conjunct in Equation 6.2 can also be reduced using aggregated vectors u and r .

This example gives us the hint for aggregating a set of intervals when detecting *Possibly*(ϕ). For set X and Y in Figure 19, their aggregated intervals are denoted as $\sqcup(X)$ and $\sqcup(Y)$, respectively. The way to aggregate those two sets is shown in Figure 19(b).

For an aggregated interval $\sqcup(X)$, we define $\textit{ORIG}(\sqcup(X))$, the processes from which this aggregated interval comes from, as the set of processes whose intervals are aggregated by $\sqcup(X)$.

Formally,

$$\textit{ORIG}(\sqcup(X)) = \{\textit{ORIG}(x) | x \in X\}$$

As an example, in Figure 19, $ORIG(\sqcup(X)) = \{P_1, P_3\}$.

With this updated definition of $ORIG(X)$, we formally define $\sqcup(X)$ as

$$\min(\sqcup(X))[i] = \max_{x \in X} \min(x)[i] \quad (6.4)$$

$$\max(\sqcup(X))[i] = \begin{cases} \max(x)[i] : \exists x \in X, \text{ such that } i \in ORIG(x) \\ 0 : \text{otherwise} \end{cases} \quad (6.5)$$

When we are comparing the aggregated intervals with other intervals for the *antichain* property, we need to compare multiple elements rather than one element because for an aggregated interval $\sqcup(X)$, $ORIG(\sqcup(X))$ contains multiple processes. This makes it necessary to define a new relation \sqsupseteq between two intervals x and y , aggregated or not, as $x \sqsupseteq y \equiv \forall i \in ORIG(x), \max(x)[i] > \min(y)[i]$. Thus, the *antichain* property defined in Equation 6.1 is equivalent to $\forall x_i, x_j \in X, x_i \sqsupseteq x_j$.

With this updated definition of *antichain* property and the aggregation function \sqcup , we give the following theorem for decentralizing the detection of *Possibly*(ϕ).

Theorem 7. *Let X, Y and Z be sets of intervals, such that $Z = X \cup Y$. Then $antichain(Z)$ iff $antichain(X) \wedge antichain(Y) \wedge antichain(\sqcup(X), \sqcup(Y))$.*

Proof. (\Rightarrow) $antichain(X)$ and $antichain(Y)$ are clearly true since $X, Y \subseteq Z$. Now consider an interval $x \in X$. Since $antichain(Z)$, $\forall y \in Y, x \sqsupseteq y$. Thus $x \sqsupseteq \sqcup(Y)$. The same deduction applies to any interval $x \in X$. So we have $\forall x \in X, x \sqsupseteq \sqcup(Y)$. Thus $\sqcup(X) \sqsupseteq \sqcup(Y)$. Similarly, $\sqcup(Y) \sqsupseteq \sqcup(X)$. So, we have $antichain(\sqcup(X), \sqcup(Y))$.

(\Leftarrow) From $antichain(\sqcup(X), \sqcup(Y))$ we have $\sqcup(X) \supseteq \sqcup(Y) \wedge \sqcup(Y) \supseteq \sqcup(X)$. For any interval $x \in X$, since $ORIG(x) \subseteq ORIG(\sqcup(X))$, we have $\forall x \in X, x \supseteq \sqcup(Y)$. Also, for any $y \in Y$, we have $\min(y) \prec \min(\sqcup(Y))$. So, $\forall x \in X, \forall y \in Y, x \supseteq y$. The same proof also applies to $\forall x \in X, \forall y \in Y, y \supseteq x$. Since we already have $antichain(X) \wedge antichain(Y)$, we now have $antichain(Z)$. \square

Theorem 7 only deals with the situation containing only 2 interval sets. We can also deduce the following lemma from Theorem 7 to cover general situations where more than 2 interval sets need to be aggregated.

Lemma 2. *Let X_1, X_2, \dots, 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 $antichain(Z)$ iff $\wedge_{i=1}^d antichain(X_i) \wedge antichain(\sqcup(X_1), \sqcup(X_2), \dots, \sqcup(X_d))$*

Proof. (\Rightarrow) $\wedge_{i=1}^d antichain(X_i)$ is clearly true since $X_i \subset Z$. Since $antichain(Z)$, we have $\forall i, j \in [1, d], antichain(X_i \cup X_j)$. Thus, according to Theorem 7, we have

$$\forall i, j \in [1, d], antichain(\cap(X_i), \cap(X_j)).$$

This means, $\forall i, j \in [1, d], \cap(X_i) \supseteq \cap(X_j)$. Thus, we have $antichain(\cap(X_1), \cap(X_2), \dots, \cap(X_d))$

(\Leftarrow) Since $\wedge_{i=1}^d antichain(X_i) \wedge antichain(\cap(X_1), \cap(X_2), \dots, \cap(X_d))$, we have

$$\forall i, j \in [1, d], antichain(X_i \cup X_j).$$

This means, by picking any two intervals y_1, y_2 from Z , it is always true that $y_1 \supseteq \max(y_2)$. 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 $\text{antichain}(Z)$. \square

With Theorem 7 and Lemma 2, we can detect $\text{Possibly}(\phi)$ in a hierarchical manner. For our hierarchical detection algorithm, each process P_i in the spanning tree detects the predicate 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 \sqcup 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 2 ensures that, by testing the *antichain* property on the aggregated intervals, the predicate can be detected within a larger set of intervals. With the level goes higher in the hierarchy, the predicate is also detected within a larger group in the network.

6.2.3 Repeated Detection for $\text{Instantaneously}(\delta, \phi)$

Repeated detection is a feature that enables the predicate detection algorithms to perform continuous monitoring by detecting *every* instance of predicate satisfactions. Basically, repeated detection requires identifying certain removable intervals from a solution set such that these intervals cannot be part of a future solution set. This problem has been studied before for the $\text{Definitely}(\phi)$ modality (5; 31). For hierarchical detection algorithm, repeated detection is also a requirement as it prevents the detection of the predicate at each level in the hierarchy from hanging after the first detection. This is shown in our previous work (31) which presents a hierarchical detection algorithm for the $\text{Definitely}(\phi)$ modality.

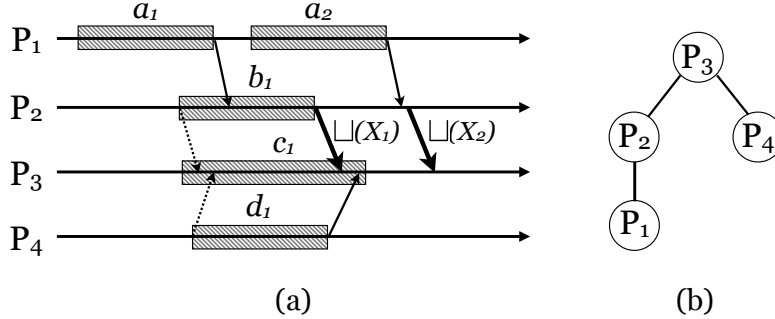


Figure 20. Example illustrating the repeated detection technique in the context of aggregated intervals. (a) Timing diagram showing 4 processes with their corresponding intervals illustrated in shade. (b) Spanning tree consisting of these 4 processes establishes a hierarchy in the network. Note that dotted line indicates an interval start marker, thin straight line indicates an unaggregated interval, and the bold straight line indicates an aggregated interval. Only relevant ones are drawn.

In (5), it has been shown that for repeated detection of the *Definitely*(ϕ) modality, removable intervals x_i in the solution set X must satisfy the condition in Equation 5.8. This condition essentially identifies the set of intervals in the solution set X that can never become part of a future solution in all possible runs of the distributed program. For the *Instantaneously*(δ, ϕ) modality, we need to identify the interval that will not become part of a future solution only in the current run δ . We observe that once the predicate is detected, the interval in the solution set that finishes first in the current run is the one we are looking for. This is illustrated in Figure 20(a). Two intervals appear as overlapped in the horizontal direction if they overlap in physical time. Notice that, in the run depicted in Figure 20(a), interval a_1 finishes first in the solution set $\{a_1, b_1, c_1, d_1\}$. When a_1 finishes, no subsequent intervals on other processes have started yet. Thus, a_1 cannot become part of a future solution set.

This way of identifying removable intervals is essentially equivalent to the one for the *Definitely*(ϕ) modality (Equation 5.8). We observe that, the condition given in Equation 5.8

actually identifies all the intervals x_i in the solution set X that might be the one finishing first in any of all possible runs. Since $Definitely(\phi)$ detects predicate satisfactions that are guaranteed to occur in all runs, all such intervals x_i identified by Equation 5.8 are removable.

Although this idea for performing repeated detection is straightforward, it is not trivial to implement. Due to the lack of global synchronized clocks, there is no way to precisely tell which interval in the solution set finishes first. To address this problem, we propose the first approximation technique. Since immediately after an interval finishes, it either gets transmitted to the upper level in the hierarchy or enters the local queue for detection, we propose to estimate the interval's finishing time by its receiving time, i.e., the time when the interval is received by process in an upper level or enters the local queue. Each process that receives intervals will maintain a queue *Arriv* that keeps track of intervals in the order they are received. Notice how the hierarchical detection algorithm helps to improve the accuracy. In a centralized setting, this approximation technique would require all processes to send the intervals to the fusion server over multiple hops. Such a convergecast will be extremely inaccurate due to the unpredictable delays of message transmission over multiple hops and the scheduling of intermediate processes. With the hierarchical detection algorithm, the intervals only need to be transmitted to each process's immediate parent in the spanning tree. Thus, the message will be transmitted over a single hop and thus significantly improve the approximation technique's accuracy.

Furthermore, to work with the aggregated intervals generated by the hierarchical detection algorithm, this approximation technique also needs to handle the following situation. Since each aggregated interval represents a set of intervals, when performing repeated detection with

aggregated intervals, we need to find the one which contains the interval that finishes first among all the intervals represented by the aggregated intervals to be checked. Take the illustration in Figure 20 as an example. From Figure 20(b), we know that P_2 receives intervals from P_1 and sends aggregated intervals to P_3 . Figure 20(a) shows that the predicate is satisfied twice globally in solution set $\{a_1, b_1, c_1, d_1\}$ and $\{a_2, b_1, c_1, d_1\}$. When performing repeated detection, although $\sqcup(X_2)$ is received by P_3 later than d_1 , $\sqcup(X_2)$'s receiving time should be considered the same as $\sqcup(X_1)$'s receiving time. This is because both $\sqcup(X_1)$ and $\sqcup(X_2)$ aggregate interval b_1 and the interval that finishes first in all the intervals in X_2 must finish earlier than b_1 . So, $\sqcup(X_2)$ finishes no later than $\sqcup(X_1)$.

This approximation technique will generate errors when the network transmission time is high. In such cases, the approximation technique will identify the wrong interval as removable, thus will miss detections of predicate satisfactions occurring in physical time and will produce *false negatives* in the detection results.

With this example, we give the formal algorithm for performing repeated detection in Algorithm 8.

6.2.4 Detection Pruning Technique

The hierarchical detection technique for the *Possibly*(ϕ) modality is still not sufficient for detecting predicates under the *Instantaneously*(δ, ϕ) modality. Since the *Possibly*(ϕ) modality considers all runs of the same distributed program, it is possible that predicate satisfactions meeting the *Possibly*(ϕ) detection condition but not occurring in physical time will be detected. A mechanism that focuses the algorithm's detection on the current run δ is thus necessary.

Algorithm 8: Repeated detection for *Instantaneously*(δ, ϕ) (Code for P_i)

```

arrival queue:  $Arriv \leftarrow \perp$ 

When local interval  $x$  finishes:
1 if  $P_i$  is a leaf node then
2   | set  $x$ 's flag as true;
3   | send  $x$  to  $P_i$ 's parent in the spanning tree;
4 else
5   | set  $x$ 's flag as true;
6   | Enqueue  $x$  onto the local queue corresponding to  $P_i$ ;

On receiving an interval  $x$  from child  $P_j$ :
7 if  $x$ 's flag is set to true then
8   | Append  $x$  at the end of  $Arriv$ ;
9 else
10  | Insert  $x$  at the position of the most recent interval from  $P_j$  with a true flag in  $Arriv$ ;

On detecting the predicate satisfaction in interval set  $X$ :
11 generate aggregated interval  $\sqcup(X)$ ;
12 if all intervals in set  $X$  have their flags set to true then
13   | set  $\sqcup(X)$ 's flag to true;
14   | set the flags of all intervals in set  $X$  to false;
15 else
16   | send  $\sqcup(X)$  to  $P_i$ 's parent in the spanning tree;
17 Delete the head of the interval queue corresponding to  $P_k$  from which interval  $Arriv.getHead()$ 
    is received;

```

We observe that, such excessive detection of predicate satisfactions happens when there is no causal relationships among a set of intervals. For example, in Figure 21(a), there is no message transmission between any 2 of the 3 processes during the time period shown. So, interval x_1 , x_2 and x_3 are not comparable under the causal relationship. However, x_1 clearly does not overlap with the other intervals in the current run δ depicted in the figure. To address this problem, we propose another approximation technique that aims at pruning these excessive detections. Assume in Figure 21(b) that P_2 and P_3 send intervals to P_1 for detection. P_2 and P_3 now send markers to P_1 right after their local intervals start (the dotted line in Figure 21(a)), and P_1

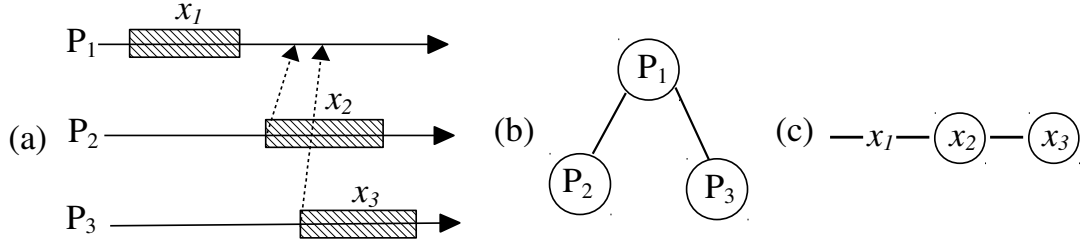


Figure 21. By sending markers right after local interval starts, our detection algorithm is able to prune false positives. (a) P_2 and P_3 send markers to P_1 right after their local intervals start. (b) P_2 and P_3 send intervals to P_1 for detection. (c) P_1 's *Arriv* queue maintaining the orders in which intervals and interval start markers are received at P_1 . Circled entries represent the interval start markers of the corresponding intervals. Since interval x_1 's entry appears before interval x_2 and x_3 's interval start markers, P_1 deduces that x_1 does not overlap with x_2 or x_3 .

receives these markers in the *Arriv* queue (introduced in Section 6.2.3). Then based on the order maintained in the *Arriv* queue, P_1 can tell that those 2 markers are received after its local interval x_1 finishes, and it estimates that x_1 does not overlap with x_2 and x_3 (Figure 21(c)).

These interval start markers essentially add additional relationships between intervals, so the detecting process can tell whether intervals overlap when causal relationships are insufficient. This technique approximates the starting time of a local interval by the receiving time of the corresponding interval start marker. Since the inaccuracy in this approximation technique is again bounded by the single hop transmission time, our approximation still holds a relatively high accuracy.

Furthermore, this detection pruning technique also works well with the aggregated intervals generated in the hierarchical detection algorithms. In the hierarchical detection algorithm, other than parents of leaf nodes, most processes will receive aggregated intervals from its chil-

dren. There is no way to precisely approximate an aggregated interval's start time because it represents a set of intervals. However, we can approximate the aggregated interval's start time with the sender's local interval's start time. For example, assume P_2 in Figure 21(b) is now parent of other processes in the spanning tree and it sends aggregated intervals to P_1 . Then for all aggregated intervals sent from P_2 that aggregates interval x_2 , we approximate their start time by the receiving time of x_2 's interval start marker. This approximation is valid because, among all the intervals aggregated by one such aggregated interval from P_2 , the interval that starts earliest will start no later than x_2 . Additionally, this approach generates only 1 interval start marker for all aggregated intervals that aggregate x_2 . This ensures that the total number of interval start markers is dominated by the number of intervals and aggregated intervals transmitted during the detection.

This approximation technique will generate errors when the network transmission time is high. In such cases, less excessive detections will be pruned and *false positives* will be generated by our algorithm.

With the above example, we give the formal algorithm for detection pruning technique in Algorithm 9.

6.2.5 Integrated Detection Algorithm

With the techniques introduced in the previous subsections, we give the integrated detection algorithm for the *Instantaneously*(δ, ϕ) modality in Algorithm 10. 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 aggregated or non-aggregated intervals, depending on whether the

Algorithm 9: Detection Pruning Technique (Code for P_i)

arrival queue: $Arriv \leftarrow \perp$

When local interval x starts:

1 send an interval start marker to the parent in the spanning tree;

On receiving an interval start marker x :

2 Append x at the end of $Arriv$;

When interval x is being checked for predicate satisfactions:

3 **if** x is a non-aggregated interval **then**

4 $marker = x$'s interval start marker;

5 **if** x is an aggregated interval from P_j containing P_j 's local interval x_j **then**

6 $marker = x_j$'s interval start marker;

7 Remove all intervals which appears in $Arriv$ before $marker$ from their corresponding interval queues;

child is a leaf node. By checking the intervals received (Lines (1)-(18)), each process attempts to detect the predicate within the subtree rooted at itself. Line (7) invokes Lines (3)-(7) in Algorithm 9 to perform the detection pruning technique. Once a solution set is found (Line (19)), the root of the subtree aggregates the set and sends it to its parent (Lines (20)-(21)). 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)-(18)). When the root of the spanning detects a solution set, a satisfaction of the predicate is detected within the whole system (Lines (22)-(23)). Each time the predicate is detected at some process, Line (24) invokes Lines (11)-(17) in Algorithm 8 to perform the repeated detection technique.

6.2.6 Complexity Analysis of the Detection Algorithm

The hierarchical detection algorithm decentralizes the entire detection procedure onto all the processes. Thus it naturally incurs only a small time and storage cost on each process. We

Algorithm 10: *Instantaneously*(δ, ϕ) detection algorithm (Code for P_i)

```

# of children:  $l$ 
queue for  $P_i$ :  $Q_0 \leftarrow \perp$ 
queues for children:  $Q_1, Q_2, \dots, Q_l \leftarrow \perp$ 
set of int:  $updatedQueues, newUpdated \leftarrow \{\}$ 
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       invoke Algorithm 9;
8       if  $Q_a$  is not empty then
9          $x = \text{head of } Q_a$ ;
10        for  $b = 0 \dots l (b \neq a)$  do
11          if  $Q_b$  is not empty then
12             $y = \text{head of } Q_b$ ;
13            if  $x \not\sqsupseteq y$  then
14               $\sqsubset$  add  $b$  to  $newUpdated$ ;
15            if  $y \not\sqsupseteq x$  then
16               $\sqsubset$  add  $a$  to  $newUpdated$ ;
17      Delete heads of all  $Q_c$  where  $c \in newUpdated$  and the corresponding entries in  $Arriv$ ;
18       $updatedQueues = newUpdated$ ;
19      if all queues are non-empty  $\wedge updatedQueues = \emptyset$  then
20        if  $P_i$  has parent in the spanning tree then
21           $\sqsubset$  report  $\sqcup(\text{heads of all queues})$  to parent;
22        else
23           $\sqsubset$  report predicate detected;
24      invoke Algorithm 8;

```

use the following parameters to study the detection algorithm's asymptotic time and storage complexity:

- n : the number of nodes in the network
- p : the maximum number of intervals per process
- d : the maximum number of children any process in the spanning tree can have

It is already shown in (31) that $O(pn)$ aggregated intervals and $O(pn)$ non-aggregated intervals are generated and stored across all nodes in the network. Additionally, each non-leaf process also needs to maintain the arrival queue *Arriv*. Since whenever an interval is removed from the corresponding interval queue, it is also removed from *Arriv*, each process only needs to maintain the most recent received intervals and the corresponding interval start markers for its $O(d)$ children. Thus the arrival queue contains $O(d)$ entries in total, each of size $O(1)$ (only meta-data of the interval is stored in *Arriv*). The storage cost of maintaining these arrival queues is dominated by the cost of storing intervals. So, the total storage cost across all processes is $O(pn^2)$.

Since the total number of intervals is $O(pn)$, and each interval will be compared with $O(d)$ other intervals with each comparison taking $O(n)$ time, Lines (1)-(22) in the hierarchical detection algorithm will incur an $O(dn^2p)$ time complexity distributed across all the nodes in the network. Since Line (23) will be executed each time an aggregated interval is generated, it will be invoked $O(pn)$ times, each time taking only $O(1)$ time. Thus, the total time complexity across all processes is $O(dn^2p)$.

6.3 Evaluation

To evaluate our detection algorithm, we perform a quantitative study of the algorithm's detection accuracy. A parameterized synthetic benchmark is designed and implemented, and some inferences are drawn based on the results of simulation.

6.3.1 Simulator Design

The architecture of the simulator is shown in Figure 22. The entire simulator is composed of 6 components.

- The *simulation framework* is responsible for simulating the processes in the network. It initializes the processes as well as the other components in the simulator using the simulation parameters.
- The *topology creator* is responsible for randomly allocating the processes within a specified area. It creates a random geometric graph using the processes initialized by the *simulation framework* as nodes in the graph. The graph is guaranteed to be connected. The generated graph represents the topology of the network consisting of all the simulated processes.
- The *overlay creator* is responsible for creating an overlay on top of the topology created by the *topology creator*. In our simulation, we create a spanning tree overlay. This constructs the hierarchy used by our repeated hierarchical detection algorithm.
- The *events generator* is responsible for generating the events that occur during the simulation. In our simulation, we do not simulate each process as an independent process/thread, as this will involve inter-process/thread communication and its delays are not easy to con-

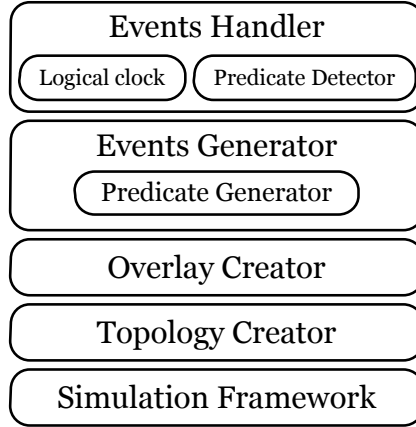


Figure 22. Architecture of the simulator.

trol. In stead, we adopt a global linearization approach, where events are generated before the simulation starts. This approach will be described in detail later. The *events generator* component also has a subsystem, the *predicate generator*. It is responsible for adding events to the events sequence to randomly generate intervals on all processes. These intervals will be processed by the hierarchical detection algorithms to detect predicate within the network.

- Each events generated by the *events generator* will be processed by the *events handlers*. Both the *logical clock* and the *predicate detector* are *events handlers*. The *logical clock* is responsible for ticking the vector clocks at each process when a relevant events is processed. The *predicate detector*, on the other hand, handles events related to intervals and detects predicate within the network.

One important feature of our simulator is that it is capable of simulating the system executions at the granularity of each individual events. This is crucial because detecting predicates is

a very time sensitive task. To simulate it, the events needs to be processed when it occurs so the detection procedure does not get delayed. In our simulation, the *events generator* component maintains a global clock which is different from the vector clocks at each process. When the events are being generated, each of them will be given a timestamp from this global clock. The *events generator* also maintains a priority queue of all the simulated events, which sorts all the events by their timestamp. We define 5 types of events as shown below:

1. **Internal events** represent events happening locally at a process. Each *internal event* is associated with a *source* process indicating the process on which the *internal event* occurs. *Internal events* will tick vector clock at *source* processes.
2. **Msg send events** represent the send events of communication messages in the network. Each *msg send event* is associated with a *source* process and a *destination* process. *Msg send events* will tick vector clock at *source* processes. They are also piggybacked with the vector clocks of the *source* processes after the tick.
3. **Msg recv events** represent the receive events of communication messages in the network. They are similar to *Msg send events* except they tick the vector clocks at *destination* processes.
4. **Int start events** represent the receive events of interval start markers in the network. Each such event is associated with a *source* process indicating the process on which the interval occurs, and a *destination* process indicating the receiver of the marker which is also the *source* process's parent in the spanning tree. *Int start events* do not tick vector clocks.

5. **Int rcv events** represent the receive events of intervals in the network. They are associated with the same information as *int start events*, and they do not tick vector clocks. Note that this type of events represents the receiving events of both unaggregated and aggregated intervals.

Without running the hierarchical detection algorithm, there is no way to generate the *int start events* and *int rcv events* before the simulation starts. However, we can generate these two types of events on the fly and insert them into the events priority queue in the *events generator* when an interval starts or an interval (aggregated or not) is generated. Because the *int start events* or the *int rcv events* will always happen later than the start or generation time of their corresponding intervals, we guarantee that these two types of events can still be processed at the time they occur.

6.3.2 Simulation Parameters

- **Number of processes (n):** To fully evaluate our detection algorithm, it is necessary to simulate a wide range of the number of processes to test the accuracy of our approach against scalability.
- **Simulation time (t):** This parameter controls the simulated duration of the system execution. With a large t , we can generate more events at each process.
- **Mean inter-event time (MIET):** The mean inter-event time is the average period of time between two consecutive events happening in the network. It determines the events generation frequency. The inter-event time is modeled as an exponential distribution about this parameter.

- **Lower bound of interval duration (LID):** This parameter indicates the lower bound of the time duration of any interval occurring at any process.
- **Upper bound of interval duration (UID):** Together with LID, those two parameters specify the range of the time duration of any intervals occurring during the simulation. The actual interval duration is modeled as a uniform distribution within this range. This ensures that the intervals generated during the simulation are of varying lengths. This matches the real scenario.
- **Mean inter-interval time (MIIT):** The mean inter-interval time is the average period of time between two consecutive intervals at any process. This parameter controls the frequency at which intervals occur at each process. The actual inter-interval time is modeled as a normal distribution of the mean.
- **Mean transmission time (MTT):** This parameter indicates the average transmission time of a message over 1 hop in the simulated network. By changing this value, we can simulate networks with a wide range of transmission speed. The actual transmission time is modeled as a normal distribution of the average. A message transmitted over a hops will incur a transmission delay equal to the sum of a samples from this random distribution.

6.3.3 Simulation Result

The intervals in our simulation are generated in a random way on every process. By sampling the interval durations and the inter-interval time based on the simulation parameters, intervals of various lengths are generated at every process with a changing frequency. Since our simulator

can simulate both the physical time and the logical time, the starting and ending events of every interval are timestamped with both physical time and the logical vector clock. Thus, our simulator can precisely tell how many times intervals from all n processes overlap. By gathering the vector clock timestamps of the ending events of these overlapping intervals, we can uniquely identify such a set of n intervals by generating an aggregated vector clock using Equation 6.5. We can then analyse the accuracy of our detection algorithm by comparing the detected predicate satisfactions against the overlapping sets of intervals checked using physical timestamps. Notice that, the physical timestamps are never used in the detection of the predicate satisfactions.

Being an approximation approach, our algorithm may generate both false positives and false negatives as described in Section 6.2.4 and 6.2.3. Thus, we use F-score as a measurement of our algorithm's accuracy. The F-score of the detection results is expressed in terms of both the precision and the recall of the detection results. The precision and recall of the detection results is defined as

$$\begin{aligned} precision &= \frac{\# \text{ true positives}}{\# \text{ true positives} + \# \text{ false positives}}, \\ recall &= \frac{\# \text{ true positives}}{\# \text{ true positives} + \# \text{ false negatives}}. \end{aligned}$$

While measuring the F-score, we weight recall higher than precision to emphasize more on detecting all occurred predicate satisfactions. We use F_2 measure defined as below:

$$F_2 = 5 * \frac{precision * recall}{4 * precision + recall}$$

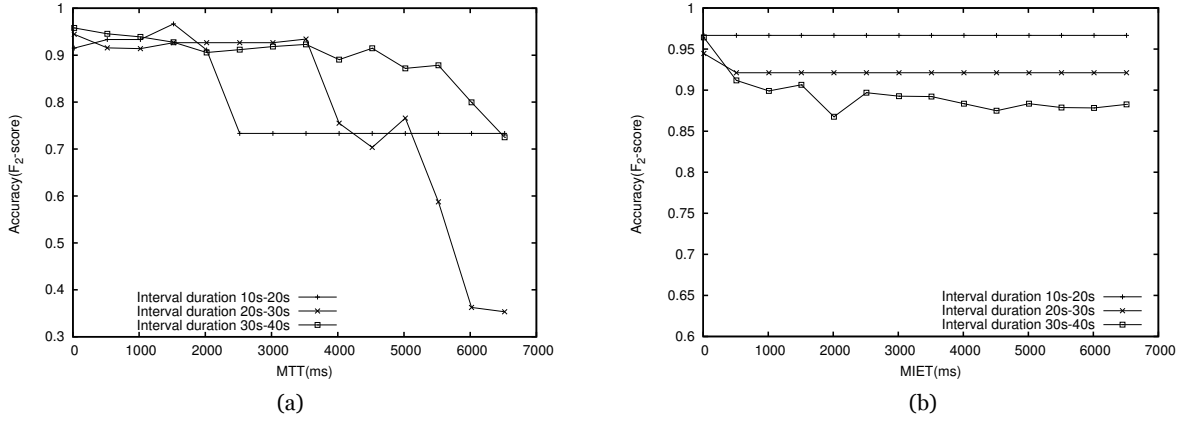


Figure 23. Showing accuracy of the detection method with (a) increasing message transmission time(here $n = 100, t = 1200s, miet = 10ms$), (b) increasing mean inter-event time(here $n = 100, t = 1200s, mtt = 20ms, lid = 20s, hid = 30s$).

We first evaluate the impact of the message transmission time and the interval duration on the detection accuracy. Ideally, with an increasing message transmission time, the accuracy of our detection method should decrease. Also, with the same message transmission time, the longer the interval duration is, the less likely the message transmission delays will impact the detection accuracy. This hypothesis is validated by our simulation as shown in Figure 23(a).

A total of 100 processes are simulated. 3 sets of simulations are carried out with interval durations in the range of 10s - 20s, 20s - 30s, and 30s - 40s, respectively. In each set of simulations, the mean transmission time increases from 20ms to more than 6s, and *MIET* is chosen to be close to the interval durations. Notice that, in order to show the trend of the detection accuracy, we choose the upper bound of the single hop mean transmission time to be 6s. In practice, it is almost impossible to reach such a high value. Each data point in Figure 23 is retrieved by running the simulation with 5 different random seeds and taking the

average value. From Figure 23(a) we can see that, when the mean transmission time is less than $\sim 15\%$ of the interval duration, the detection accuracy keeps at a high percentage. When the transmission time increases beyond 15% , the detection accuracy starts to drop. With a practical value of MTT , our detection algorithm maintains a high detection accuracy ($90\%+$).

Simulations are also carried out to evaluate the effectiveness of the detection pruning technique introduced in Section 6.2.4. The parameter $MIET$ determines the event generation frequency. With a higher frequency, more communication messages are likely to be generated, thus introducing more causal relationships between events and intervals. This can reduce the number of excessive detections described in Section 6.2.4. On the other hand, if the events generation frequency is low, more events and intervals are likely to be incomparable using causal relationships. We run another set of simulations evaluating the impact of the parameter $MIET$ on the detection accuracy. The result is shown in Figure 23(b). It shows that the detection accuracy is not much affected by the increase of $MIET$. Thus, it shows the effectiveness of the detection pruning technique.

As a conclusion, these simulations show that our detection method can achieve a very high ($90\%+$) accuracy when the single hop transmission delay is less than 15% of the interval duration. The simulations also show that the detection pruning technique introduced in Section 6.2.4 can effectively reduce false positives.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this work, we gave the motivations for detecting predicates in large-scale locality-driven networks. We proposed the concept of a locality-aware predicate. This type of a predicate models a predicate in an area of interest. In a large-scale locality-driven network, such as modular robotics or WSNs, the following factors:

1. The interactions being local and driven by neighborhood proximity,
2. The high cost of doing a global predicate detection, and
3. The state of a local region better captures local interactions,

make locality-aware predicate detection a relevant and an interesting problem. We showed how to model the area of interest as a circular region (BFST rooted at P_r), and then proposed Algorithm 1 to efficiently construct the overlay network. This is the first distributed algorithm to construct a BFST within a local region in a graph. We also designed Algorithm 2 to efficiently take a snapshot within the area of interest in a non-FIFO network. This is the first algorithm to construct a consistent sub-cut defined by a region in a larger graph. We then defined two classes of locality-aware predicates: 1) conjunctive LAP, and 2) relational LAP. Finally, we gave an algorithm *Stable LAP Detection* for detecting both classes of LAPs, based on the recorded snapshot. The complexity analysis of all these algorithms showed their performance is *scale-*

free. Hence, these algorithms have great potential for observing local regions within large-scale distributed systems such as modular robotics and WSNs.

Furthermore, we extended the problem of LAP detection to include unstable conjunctive predicates. 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 (1). More importantly, we developed the encoded vector clock (EVC) technique that optimizes the detection algorithm by reducing the storage cost at every process in the whole network. EVC makes detecting unstable conjunctive LAP more practical in large-scale systems.

For detecting unstable predicates within large-scale locality-driven networks, we also proposed the first decentralized hierarchical algorithm that repeatedly detects all occurrences of *Definitely*(Φ) 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 (5), 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.

In addition, we proposed an algorithm that is capable of detecting predicates in the *Instantaneously* modality even when synchronized physical clocks are not available. With a hierarchical detection approach and two approximation techniques, our algorithm reaches a

very high detection accuracy when the single hop transmission delay is much less than the duration of the intervals. In addition, our detection algorithm incurs only a small cost on every process, since the hierarchical detection procedure is decentralized onto all processes in the network. This makes our detection algorithm a natural choice for performing event detections in resource-constraint systems where physically synchronized clocks are not available.

These algorithms we designed in this work provide a suite of tools for predicate detection in large-scale locality-driven networks. As for the future work, this work can be further explored in the following directions.

- Extend the Instantaneous detection algorithm so that it has the similar fault-tolerance feature as the hierarchical repeated detection.
- Perform physical world experiments on the hierarchical repeated detection and the Instantaneous detection algorithms to test their performances in more depth.
- Extend the hierarchical repeated detection methodology and the Instantaneous detection algorithm to also detect unstable relational predicates.

APPENDICES

Appendix A

IEEE LICENSE DOCUMENTS



Copyright
Clearance
Center



RightsLink®

Home

Account Info

Help



IEEE
Requesting
permission
to reuse
content from
an IEEE
publication

Title:	Detecting Unstable Conjunctive Locality-Aware Predicates in Large-Scale Systems	Logged in as: Min Shen
Conference Proceedings:	Parallel and Distributed Computing (ISPD), 2013 IEEE 12th International Symposium on	Account #: 3000753587
Author:	Min Shen; Kshemkalyani, A.; Khokhar, A.	<div style="background-color: #000080; color: white; padding: 2px 10px; border-radius: 3px; display: inline-block;">LOGOUT</div>
Publisher:	IEEE	
Date:	27-30 June 2013	
Copyright © 2013, IEEE		

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

- 1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
- 2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
- 3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

- 1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
- 2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
- 3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.



RightsLink®

Home

Account
Info

Help



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

Author: Shen, M.; Kshemkalyani, A.

Publication: Parallel and Distributed Systems, IEEE Transactions on

Publisher: IEEE

Copyright © 1969, IEEE

Logged in as:

Min Shen

Account #:

3000753587

LOGOUT

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

- 1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
- 2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
- 3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:






- 1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
- 2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
- 3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Appendix B

ELSEVIER PERMISSIONS

The following document is obtained from the ELSEVIER official author web page available at <http://www.elsevier.com/journal-authors/author-rights-and-responsibilities>. It clearly states that, as far as full acknowledgement is given, the author can reuse all or part of a previously published article through Elsevier to be included in a thesis or dissertation.


Type here to search on Elsevier.com
Advanced search
Follow us:    
[Help & Contact](#)

[Journals & books](#)
[Online tools](#)
[Authors, editors & reviewers](#)
[About Elsevier](#)
[Community](#)
[Store](#)

For Authors

- [Journal authors' home](#)
- Author Rights**
- [Ethics](#)
- [Funding body agreements](#)
- [Open access](#)
- [Author services](#)
- [Journal performance](#)
- [Early career researchers](#)
- [Authors' update](#)
- [Book authors' home](#)
- [Getting your paper noticed](#)

Author Rights

Elsevier supports the need for authors to share, disseminate and maximize the impact of their research. We take our responsibility as stewards of the online record seriously, and work to ensure our policies and procedures help to protect the integrity of scholarly works.

Author's rights to reuse and post their own articles published by Elsevier are defined by Elsevier's copyright policy. For our proprietary titles, the type of copyright agreement used depends on the author's choice of publication:

For subscription articles: These rights are determined by a copyright transfer, where authors retain scholarly rights to post and use their articles.

For open access articles: These rights are determined by an exclusive license agreement, which applies to all our open access content.

In both cases, the fundamental rights needed to publish and distribute an article remain the same and Elsevier authors will be able to use their articles for a wide range of scholarly purposes.

Details on how authors can reuse and post their own articles are provided below.

Help and support

For reuse and posting not detailed below, please see our [posting policy](#), or for authors who would like to:

- Include material from other sources in your work being published by Elsevier, please visit: [Permission seeking guidelines for Elsevier authors](#).
- Obtain permission to re-use material from Elsevier books, journals, databases, or other products, please visit: [Obtaining permission to reuse Elsevier material](#)
- Or if you are an Elsevier author and are contacted by a requestor who wishes to re-use all or part of your article or chapter, please also refer them to our [Obtaining Permission to Re-Use Elsevier Material page](#).
- See our FAQ on [posting](#) and [copyright queries](#).
- Contact us directly, please email our [Permissions Help Desk](#).








Author Use	Author Posting	Definitions
------------	----------------	-------------

How authors can use their own journal articles

Authors can use their articles for a wide range of scholarly, non-commercial purposes as outlined below. These rights apply for all Elsevier authors who publish their article as either a subscription article or an open access article.

We require that all Elsevier authors always include a full acknowledgement and, if appropriate, a link to the final published version hosted on Science Direct.

For open access articles these rights are separate from how readers can reuse your article as defined by the author's choice of [Creative Commons user license options](#).

Authors can use either their accepted author manuscript or final published article for:	
	Use at a conference, meeting or for teaching purposes
	Internal training by their company
	Sharing individual articles with colleagues for their research use* (also known as 'scholarly sharing')
	Use in a subsequent compilation of the author's works
	Inclusion in a thesis or dissertation
	Reuse of portions or extracts from the article in other works
	Preparation of derivative works (other than for commercial purposes)

*Please note this excludes any [systematic or organized distribution](#) of published articles.

CITED LITERATURE

1. Kshemkalyani, A. and Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, 2008.
2. Shen, M., Kshemkalyani, A., and Khokhar, A.: Detecting tree distributed predicates. 41st International Conference on Parallel Processing Workshops (ICPPW), pages 598–599, 2012.
3. De-Rosa, M., Goldstein, S., Lee, P., Pillai, P., and Campbell, J.: Programming modular robots with locally distributed predicates. Proceedings of the IEEE ICRA, pages 3156–3162, 2008.
4. De-Rosa, M., Goldstein, S., Lee, P., Campbell, J., and Pillai, P.: Detecting locally distributed predicates. ACM Transactions on Autonomous and Adaptive Systems 6, 2, 6(2):13:1–13:14, 2011.
5. Kshemkalyani, A.: Repeated detection of conjunctive predicates in distributed executions. Information Processing Letters, 111, 9, pages 447–452, 2011.
6. Chandy, K. M. and Lamport, L.: Distributed snapshots: Determining global states in distributed systems. ACM Transactions on Computer Systems 3, 1, pages 63–75, 1985.
7. Lai, T.-H. and Yang, T.: On distributed snapshots. Information Processing Letters, pages 153–158, 1987.
8. Taylor, K.: The role of inhibition in asynchronous consistent-cut protocols. Proc. the 3rd Int’l Workshop on Distributed Algorithms, pages 280–291, 1989.
9. Critchlow, C. and Taylor, K.: The inhibition spectrum and the achievement of causal consistency. Proc. the 9th ACM Symp. Principles of Distributed Computing, pages 31–42, 1990.
10. Helary, J.: Observing global states of asynchronous distributed applications. Proc. the 3rd Int’l Workshop on Distributed Algorithms, pages 45–56, 1989.

11. Mattern, F.: Efficient algorithms for distributed snapshots and global virtual time approximation. J. Parallel and Distributed Computing 18, 4, pages 423–434, 1993.
12. Kshemkalyani, A., Raynal, M., and Singhal, M.: An introduction to snapshot algorithms in distributed computing. Distributed Systems Engineering 2, 4, pages 224–233, 1995.
13. Garg, R., Garg, V., and Sabharwal, Y.: Efficient algorithms for global snapshots in large distributed systems. IEEE Transactions on Parallel and Distributed Systems, 21, 5, pages 620–630, 2010.
14. Kshemkalyani, A.: Fast and message-efficient global snapshot algorithms for large-scale distributed systems. IEEE Transactions on Parallel and Distributed Systems 21, 9, pages 1281–1289, 2010.
15. Tsai, J.: Flexible symmetrical global-snapshot algorithms for large-scale distributed systems. IEEE Transactions on Parallel and Distributed Systems, pages 493–505, 2013.
16. Cooper, R. and Marzullo, K.: Consistent detection of global predicates. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, pages 163–173, 1991.
17. Garg, V. K. and Waldecker, B.: Detection of strong unstable predicates in distributed programs. IEEE Transactions on Parallel & Distributed Systems 7, 12, pages 1323–1333, 1996.
18. Garg, V. K. and Waldecker, B.: Detection of weak unstable predicates in distributed programs. IEEE Transactions on Parallel & Distributed Systems 5, 3, pages 299–307, 1994.
19. Garg, V. and Chase, C.: Distributed algorithms for detecting conjunctive predicates. Proc. 15th IEEE ICDCS, pages 423–430, 1995.
20. Hurfin, M., Mizuno, M., Raynal, M., and Singhal, M.: Efficient distributed detection of conjunctions of local predicates. IEEE Trans. Software Engineering, 24, 8, pages 664–677, 1998.
21. Chandra, P. and Kshemkalyani, A.: Distributed algorithm to detect strong conjunctive predicates. Information Processing Letters, 87, 5, pages 243–249, 2003.

22. Chandra, P. and Kshemkalyani, A.: Data stream based global event monitoring using pairwise interactions. Journal of Parallel and Distributed Computing 68 (6), pages 729–751, 2008.
23. Garg, V. K. and Mittal, N.: On slicing a distributed computation. Proc. IEEE International Conference on Distributed Computing Systems, pages 322–329, 2001.
24. Mittal, N., Sen, A., and Garg, V. K.: Solving computation slicing using predicate detection. IEEE Transactions on Parallel and Distributed Systems (TPDS) 18 (12), pages 1700–1713, 2007.
25. Chauhan, H., Garg, V. K., Natarajan, A., and Mittal, N.: A distributed abstraction algorithm for online predicate detection. 2013.
26. Elson, J. and Kay, R.: Wireless sensor networks: A new regime for time synchronization. ACM SIGCOMM Computer Communication Review 33.1, 2003.
27. Kshemkalyani, A.: Immediate detection of predicates in pervasive environments. Journal of Parallel and Distributed Computing, 72, 2, pages 219–230, 2012.
28. Kshemkalyani, A. and Cao, J.: Predicate detection in asynchronous pervasive environments. IEEE Transactions on Computers 62 (9), pages 1823–1836, 2013.
29. Shen, M., Kshemkalyani, A., and Khokhar, A.: Detecting stable locality-aware predicates. Journal of Parallel and Distributed Computing, 74, 1, 2014.
30. Shen, M., Kshemkalyani, A., and Khokhar, A.: Scale-free detection of unstable locality-aware predicates. 2013 12th International Symposium on Parallel and Distributed Computing (ISPDC), pages 127–134, 2013.
31. Shen, M. and Kshemkalyani, A.: Hierarchical detection of strong unstable conjunctive predicates in large-scale systems. IEEE Transactions on Parallel and Distributed Systems, 10.1109/TPDS.2013.306, 2014.
32. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 7, pages 558–565, 1978.
33. Mattern, F.: Virtual time and global states of distributed systems. Proceedings of the Parallel and Distributed Algorithms Conference, pages 215–226, 1988.

34. Fidge, C.: Logical time in distributed computing systems. IEEE Computer, pages 28–33, 1991.
35. Lai, T. H. and Yang, T. H.: On distributed snapshots. Inf. Process. Lett. 25, 3, pages 153–158, 1987.
36. Kshemkalyani, A. and Wu, B.: Detecting arbitrary stable properties using efficient snapshots. IEEE Transactions on Software Engineering 33, 5, pages 330–346, 2007.
37. Atreya, R., Mittal, N., Kshemkalyani, A., Garg, V., and Singhal, M.: Efficient detection of a locally stable predicate in a distributed system. Journal of Parallel and Distributed Computing, 67, 4, pages 369–385, 2007.
38. Dijkstra, E. and Scholten, C.: Termination detection for diffusing computations. Information Processing Letters (IPL), 11, pages 1–4, 1980.
39. Ho, G. and Ramamoorthy, C.: Protocols for deadlock detection in distributed database systems. IEEE Transactions on Software Engineering, 8, pages 554–557, 1982.
40. Marzullo, K. and Sabel, L. S.: Efficient detection of a class of stable properties. Distributed Computing 8, 2, pages 81–91, 1994.
41. Schiper, A. and Sandoz, A.: Strong stable properties in distributed systems. Distributed Computing 8, 2, pages 93–103, 1994.
42. Cooper, R. and Marzullo, K.: Consistent detection of global predicates. ACM SIGPLAN Notices. Vol. 26., pages 167–174, 1991.
43. Kshemkalyani, A.: Temporal interactions of intervals in distributed systems. Journal of Computer and System Sciences, 52, 2, pages 287–298, 1996.
44. Chandy, K. M. and Misra, J.: Distributed computations on graphs: shortest path algorithms. Communications of the ACM, pages 833–838, 1982.
45. IEEE 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 2012.
46. del Prado Paven, J. and Choi, S.: Link adaptation strategy for IEEE 802.11 WLAN via recorded signal strength measurement. IEEE International Conference on Communications 2003 ICC'03, pages 1108–1113, 2003.

47. Charron-Bost, B.: Concerning the size of logical clocks in distributed systems. Inf. Process. Lett. 39, 1, pages 11–16, 1991.
48. Yen, L.-H. and Huang, T.-L.: Resetting vector clocks in distributed systems. Journal of Parallel and Distributed Computing, 43, 1, pages 15–20, 1997.
49. Gärtner, F.: A survey of self-stabilizing spanning-tree construction algorithms. Swiss Federal Institute of Technology (EPFL), 2003.
50. Yang, M. and Fei, Z.: A proactive approach to reconstructing overlay multicast trees. Proceedings of 23rd INFOCOM, pages 2743–2753, 2004.
51. Jeon, J., Son, S., and Nam, J.: Overlay multicast tree recovery scheme using a proactive approach. Computer Communications, 31 (14), pages 3163–3168, 2008.

VITA

NAME	Min Shen	
EDUCATION	University of Illinois at Chicago	Chicago, IL
	Ph.D. in Computer Science,	Aug. 2009 - Jun. 2014
	Research Area: Distributed System	
	Nanjing University	Nanjing, China
	B.E. in Software Engineering,	Sep. 2005 - Jun. 2009
PUBLICATIONS	<ul style="list-style-type: none"> • M. Shen, A.D. Kshemkalyani, Hierarchical Detection of Strong Unstable Conjunctive Predicates in Large-Scale Systems, In <i>IEEE Transactions on Parallel and Distributed Systems</i>, 2014. • M. Shen, A.D. Kshemkalyani, A. Khokhar, Detecting Stable Locality-Aware Predicates, In <i>Journal of Parallel and Distributed Computing</i>, 74(1), pp. 1971-1983, 2014. • M. Shen, A.D. Kshemkalyani, A. Khokhar, Scale-Free Detection of Unstable Locality-Aware Predicates, In <i>Proc. 12th International Symposium on Parallel and Distributed Computing (ISPDPC)</i>, pp. 127-134, 2013 • M. Shen, A.D. Kshemkalyani, A Fault-Tolerant Strong Conjunctive Predicate Detection Algorithm for Large-scale Networks, In <i>Proc. 27th IEEE International Symposium on Parallel & Distributed Processing Workshops (IPDPSW)</i>, pp. 1460-1469, 2013 • M. Shen, A.D. Kshemkalyani, A. Khokhar, Detecting Tree Distributed Predicates, In <i>Proc. 41st International Conference on Parallel Processing Workshop (ICPPW)</i>, pp. 598-599, 2012 • M. Hefaida, M. Shen, A.D. Kshemkalyani, A. Khokhar, Cross-layer Protocols for WSNs: A Simple Design and Simulation Paradigm, In <i>8th</i> 	

International Wireless Communications and Mobile Computing Conference (IWCMC), pp. 844-849, 2012.

- A.D. Kshemkalyani, A. Khokhar, **M. Shen**, Execution and Time Models for Pervasive Sensor Networks, In *International Journal of Networking and Computing (IJNC)* 2(1), pp. 2-17, 2012
- S. Lee, A.D. Kshemkalyani, **M. Shen**, Performance Evaluation of Incremental Vector Clocks, In *Proc. 10th International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 117-124, 2011.

WORK	Research Assistant,	University of Illinois at Chicago
EXPERIENCE	MTS Intern, VMware, Palo Alto, CA	May. 2013 - Aug. 2013
	SDE Intern, Amazon.com, Seattle, WA	May. 2012 - Aug. 2012
	SDE Intern, Motorola ARC, Schaumburg, IL	May. 2010 - Aug. 2010