ACHIEVING RELIABLE DISTRIBUTED SYSTEMS: THROUGH EFFICIENT RUN-TIME MONITORING AND PREDICATE DETECTION

By

Vidhya Tekken Valapil

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science – Doctor of Philosophy

2020

ABSTRACT

ACHIEVING RELIABLE DISTRIBUTED SYSTEMS: THROUGH EFFICIENT RUN-TIME MONITORING AND PREDICATE DETECTION

By

Vidhya Tekken Valapil

Runtime monitoring of distributed systems to perform predicate detection is critical as well as a challenging task. It is critical because it ensures the reliability of the system by detecting all possible violations of system requirements. It is challenging because to guarantee lack of violations one has to analyze every possible ordering of system events and this is an expensive task. In this report, we focus on ordering events in a system run using HLC (Hybrid Logical Clocks) timestamps, which are $O(1)$ sized timestamps, and present some efficient algorithms to perform predicate detection using HLC. Since, with HLC, the runtime monitor cannot find all possible orderings of systems events, we present a new type of clock called Biased Hybrid Logical Clocks ($BHLC$), that are capable of finding more possible orderings than HLC. Thus we show that $BHLC$ based predicate detection can find more violations than HLC based predicate detection. Since predicate detection based on both HLC and $BHLC$ do not guarantee detection of all possible violations in a system run, we present an SMT (Satisfiability Modulo Theories) solver based predicate detection approach, that guarantees the detection of all possible violations in a system run. While a runtime monitor that performs predicate detection using SMT solvers is accurate, the time taken by the solver to detect the presence or absence of a violation can be high. To reduce the time taken by the runtime monitor, we propose the use of an efficient two layered monitoring approach, where the first layer of the monitor is efficient but less accurate and the second layer is accurate but less efficient. Together they reduce the overall time taken to perform predicate detection drastically and also guarantee detection of all possible violations.

ACKNOWLEDGEMENTS

I would like to thank everyone who has helped and supported me throughout my journey as a Ph.D. student. This journey has helped me grow at a professional and personal level, and I am very grateful for this experience.

First and foremost I would like to thank my advisor Dr.Sandeep Kulkarni for being a patient, motivating and supportive mentor. I have always admired how he can turn every simple idea into something impactful by perceiving it in every different possible aspect. I am thankful for every formal and informal discussion that I have had with him because I have learnt something new from every single conversation. I am very grateful for his continuous support and guidance at every step during the program.

I would like to thank my committee members: Dr. Philip McKinley, Dr. Rajesh Kulkarni and Dr. Eric Torng for their time, insightful comments and critical suggestions. I would like to thank Dr. Torng for his valuable guidance and for also being a great team member in the multiple projects that we have worked together. I would also like to thank my lab mates and project team members, especially Duong Nguyen and Sorrachai Yingchareonthawornchai for being supportive team mates.

Finally, I would like to thank my husband, my parents and my sisters for making this journey possible. I am grateful for the countless meals cooked by my husband and for his support throughout my ups and down in this journey. I would like to thank my father for his constant encouragement to aim higher and for teaching me the importance of continuous learning. I would like to thank my mother and sisters for their emotional support and for giving me strength and confidence every time that I needed it.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

*The growing complexity of software and the increasing number of **interacting** subsystems promises an unending discovery of new vulnerabilities. [34]*

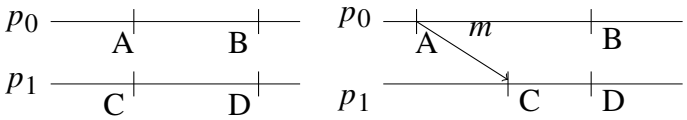
Most real world software systems ranging from wireless sensor network applications to cloud-native applications have the underlying structure of distributed systems, where the system consists of multiple components or processes that communicate with each other and also simultaneously execute their individual tasks to achieve a common goal of the system. These systems are inherently complex, making them a breeding ground for various types of software bugs. It is therefore essential to analyze and evaluate these systems for any violations or bugs to guarantee their correctness and reliability. Specifically, to ensure reliability of a system, one has to make sure that the system never violates its requirements when subject to any type of input or in any environmental setting. To provide such guarantee one has to first (i) observe the system behavior for different inputs and environmental settings, (ii) then analyze the observed behavior to determine if the system ever violates any of its requirements. However, with distributed systems there are several challenges associated with performing steps (i) and (ii), i.e. in observing and analyzing the system's behavior for all inputs.

Need for runtime monitoring. The problem with the first step i.e., in observing system behavior for all types of input or in all possible environmental settings is that it is not always possible to deploy and evaluate the system for all possible inputs or environmental settings. The input space to be considered can be very large thereby making testing for all inputs infeasible. Furthermore, in some scenarios it may not be possible for the system designer or developer to anticipate or predict all the environmental settings or external factors that the system may be subject to precisely. To account for such uncertainty, one can perform **runtime monitoring**, where to ensure reliability the system is equipped with a monitor that is responsible for monitoring the system behavior

during its execution. More specifically, the task of the monitor is to observe the system behavior without interfering with the actual underlying execution and to detect if the system ever violates its requirements during the current run.

Challenges caused by non-determinism in distributed systems. In a distributed system, where there are multiple processes, an **event** at a process corresponds to an execution of some action (at the process) that changes its local state. So local states at the processes can be defined in terms of the events that occur at the processes. For an event e at a process, let s_e denote the corresponding local state at the process, i.e. the local state of the process is s_e after the occurrence of event e. In a distributed system, an event e_1 is said to have **happened-before** another event say e_2, denoted as $e_1 \rightarrow e_2$, if and only if one of the following conditions hold: i) if event e_1 occurred before event e_2 at the same process, ii) if e_1 is a message send event and e_2 is the corresponding message receive event, or iii) if event e_1 happened before event x and event x happened before event e_2. Furthermore, two events e_1, e_2 are **concurrent** (denoted as $e_1||e_2$) if and only if $((e_1 \nrightarrow e_2) \wedge (e_2 \nrightarrow e_1))$. Local states corresponding to concurrent events at the processes are **concurrent states**. A **global state** say g of a distributed system contains a local state per process. The global state g is a **consistent global state** if and only if every local state in it is *concurrent* with every other local state in it. Consistent global states are also referred to as consistent snapshots or consistent cuts. For a distributed system, an observed system behavior i.e., an observed run of the system can contain several consistent global states that the system could have gone through during its observation.

For example, consider the execution of a distributed system shown in figure 1.1 (a) that has two processes p_0 and p_1 that execute asynchronously. During the observed run of the system, let events A, B be the local events that happened consecutively at p_0 and let events C, D be local events that happened consecutively at p_1. If there is no communication between processes p_0 and p_1 before or during events A and C, then events A, C are concurrent events. If s_A and s_C are the local states at processes p_0, p_1 corresponding to events A and C, then s_A, s_C are *concurrent states*. So s_A and s_C together form a consistent global state of the observed distributed system. Furthermore, if the processes p_0 and p_1 never communicated during the entire observed run, then events A and



(a) No communication scenario (b) With communication scenario

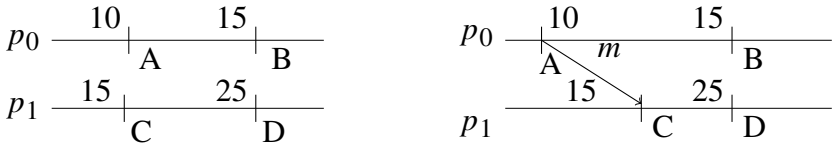
Figure 1.1: Identifying Concurrent events

D, B and C, B and D can also be considered as concurrent events. In other words, these events do not have a happened-before relationship with each other. Thus the set of consistent global states that the system could have gone through are $\{s_A, s_C\}$, $\{s_A, s_D\}$, $\{s_B, s_C\}$, and $\{s_B, s_D\}$. Therefore, while performing the second step of analyzing the observed system behavior, to ensure complete reliability, the runtime monitor has to evaluate *every consistent global state* of the system, to guarantee that the system never violated its requirements at any point during its execution.

The process of identifying all consistent global states and evaluating them is commonly referred to as **predicate detection** in distributed systems. More specifically, in predicate detection the first step is to identify all consistent global states and the second step is to evaluate every consistent global state against a predicate or a condition. This condition can correspond to a representation of a violation (or satisfaction) of some specific system requirement.

Identifying all consistent global states. The total number of consistent global states depends on the **causality relationship**[28] between the events at the processes, i.e. events at different processes that have a happened-before relationship. For example, as shown in figure 1.1 (b), if the event A is a message send event at process p_0 and C is the corresponding message receive event at process p_1, then the consistent global states that the system could have gone through are $\{s_B, s_C\}$ and $\{s_B, s_D\}$, because events A, C and A, D have a happened-before relationship and therefore are not concurrent events.

Eliminating infeasible global states with the help of time. The number of consistent global states can be reduced further if the knowledge of when the events happened in *real time* is available. For instance, let us consider the scenario in figure 1.2 (a) where processes p_0 and p_1 have local physical clocks associated with them and that these clocks are *perfectly synchronized* with each



(a) Concurrency when ($\epsilon = 0$) and when ($\epsilon = 5$) (b) With communication scenario

Figure 1.2: Identifying Concurrent events using Physical Clock timestamps

other. Let the initial states of p_0 and p_1 be I_{p0} and I_{p1} respectively. Now if local events A, B happened at p_0 at local physical time $10, 15$, likewise local events C, D happened at p_1 at local physical time $15, 25$ and if p_0 and p_1 never communicated during the observed run, then we know that the only concurrent events in the observed run are B, C, that have equal physical timestamps. So the first consistent global state of the system is $\{s_A, I_{p0}\}$ at time 10 where an event A at p_0 altered its local state and p_1 is still in its initial state at 10. The next consistent global state is $\{s_B, s_C\}$ at time 15 where events B, C altered the local states at p_0 and p_1 respectively. The final consistent global state is $\{s_B, s_D\}$ since the state of p_0 at time 25 is the same as it was at 15 (because there are no events at p_0 after B). So the consistent global states are $\{s_A, I_{p0}\}$, $\{s_B, s_C\}$ and $\{s_B, s_D\}$. While the presence of perfectly synchronized clocks is beneficial in eliminating some global states as infeasible, they are not available in most real-world distributed systems.

In practice, most distributed systems have **partially synchronized clocks**, where each process has a local physical clock and the clock drift i.e., difference between the clocks of any two processes is bounded by a specified amount, say ϵ. Therefore in this case, events that happened more than ϵ apart from each other cannot be concurrent events. If the clock drift in the distributed system considered in the above example shown in Figure 1.2 (a) is $\epsilon = 5$, then events A and D are clearly not concurrent because they happened more than ϵ apart from each other. Recall that for the corresponding scenario in Figure 1.1 (a) in the absence of real/physical time information $\{s_A, s_D\}$ was considered as a consistent global state because A and D did not have a happened-before relationship. Thus, the total number of consistent global states can be reduced by eliminating global states that are infeasible because the events/states in it are far apart in physical time. This can be achieved even if the local physical clocks at the processes are not perfectly synchronized but

are partially synchronized, by accounting for their clock drift.

Desirable properties of runtime monitors. While runtime monitoring can help ensure reliability of distributed systems, the effectiveness of the monitors depend on their **correctness** and the **cost** associated with monitoring. In other words an **ideal runtime monitor** is a runtime monitor that is correct i.e., reports all bugs in the system and never reports phantom bugs, and also has low overhead or computation cost. To identify and report all bugs the runtime monitor has to identify and evaluate all consistent global states for bugs or violations of system requirements. To avoid reporting phantom or incorrect bugs the runtime monitor should not consider global states that are inconsistent or infeasible. Therefore, the correctness of a runtime monitor directly depends on its ability to eliminate inconsistent/infeasible global states while identifying all consistent global states. To achieve this, based on our discussion, the runtime monitor should account for *causality* among events, *time* at which events happened and *clock drift* between processes.

One of the most commonly used approaches for capturing causality among events is timestamping events using Vector Clocks [18, 36]. A runtime monitor can use vector clock timestamps associated with events to determine if they have a causal relationship. Vector clocks are $O(n)$ sized clocks, where n is the number of processes in the system. However, a problem with vector clocks is that they do not capture real time information associated with the events. On the other hand, one cannot rely solely on physical clocks to eliminate global states that are inconsistent or infeasible, because physical clocks do not capture causality among events. For example, consider the scenario in figure 1.3 where event A is a message send event and C is the corresponding message receive event. In this case the physical clocks at the processes are partially synchronized to be within $\epsilon = 5$ of each other. Observe that the message send event can have a larger timestamp than the receive event because the local physical clock at process p_1 is behind (slower than) the local physical clock at process p_0. Furthermore, with physical clocks, the message send and receive events can also have equal timestamps, so they can be incorrectly considered as concurrent events. As a solution, the runtime monitor could use both vector clock and physical clock timestamps to account for causality and physical time information. However, the problem with this is that vector

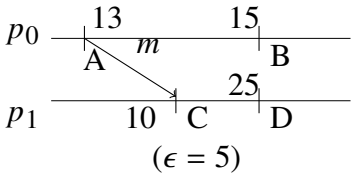


Figure 1.3: Physical Clock timestamps failing to capture causality

clock timestamps are $O(n)$ sized timestamps. For every pair of events, to determine if they have a causal relationship, the runtime time monitor will have to compare two n sized timestamps, this will increase the monitoring/computation overhead.

Efficient but less-accurate runtime monitors that have false negatives. A runtime monitor that guarantees the detection of all consistent global states while eliminating all inconsistent/infeasible global states can suffer from high monitoring overhead. In other words, a runtime monitor that is accurate can be expensive. A runtime monitor that is expensive will not be suitable in scenarios where the cost of monitoring has to be kept low. For instance, in scenarios where the system being monitored is a lightweight distributed application that has time/memory constraints, using a runtime monitor that has high monitoring overhead would not be feasible. In such scenarios, runtime time monitors that are efficient (have less monitoring overhead) but less accurate would be acceptable. For example, a runtime monitor that evaluates only a subset of consistent global states and reports violations identified in those states would be acceptable. Such a monitor will have false negatives, i.e., it will miss some bugs or violations. In Chapter 3, different algorithms that can be used by a runtime monitor for predicate detection are presented. These algorithms have low monitoring overhead but suffer from substantial number of false negatives, i.e., they can miss some bugs or violations. Specifically, these algorithms keep the monitoring cost low by relying on Hybrid Logical Clocks (HLC) [27], which are $O(1)$ sized clocks. To reduce the number of false negatives, an extension of HLC called Biased Hybrid Logical Clocks (BHLC), which are also constant sized clocks, are introduced in Chapter 4. With BHLC, the number of consistent global states considered for predicate detection is increased by creating a bias while capturing causality resulting from inter-process communication.

Accurate but less-efficient runtime monitors. In some scenarios, the accuracy of the runtime monitor may be critical than the associated monitoring cost. In such scenarios, a high monitoring cost may be acceptable as long as the runtime monitor is accurate (i.e., it has no false negatives or false positives). For such scenarios, the monitoring framework presented in Chapter 5 can be used, where the accuracy of the runtime monitor is guaranteed with the help of SMT (Satisfiability Modulo Theories) solvers. However, the time taken by the runtime monitor that uses SMT solvers to identify all consistent global states (without considering inconsistent/infeasible global states) can be high.

Efficient but less-accurate runtime monitors that have false positives. A runtime monitor can also reduce its monitoring overhead by not accounting for all causal relationships among events. Such a monitor will not miss violations/bugs but may identify/report phantom bugs that are not actually present in the system. The first half of Chapter 6 considers such an approach, where the runtime monitor considers all consistent global states during predicate detection by extending an approach from Chapter 3 to account for clock drift. However, the monitor also considers some inconsistent global states by not accounting for causality resulting from inter-process communication. While this approach achieves low monitoring cost it suffers from substantial number of false positives (phantom bugs). A slight modification of the approach that helps in achieving a trade-off between false negatives and false positives while keeping the monitoring cost low is also discussed in the chapter.

Accurate and efficient runtime monitors. For scenarios where an accurate runtime monitor that takes smaller computation time is desired, a hybrid monitoring approach is presented in the second half of Chapter 6. Specifically, the runtime monitor uses a two layered monitoring approach, where an efficient but less accurate monitor is used as the first layer and an accurate but comparatively less efficient monitor (i.e. comparatively expensive monitor) is used as the second layer. The first layer is less accurate because it has false positives, but the accurate monitor in the second layer helps in eliminating the false positives. On the other hand, the first layer acts as a filtering layer and invokes the less efficient monitor in the second layer only when it thinks that a violation or bug

is feasible. Together, they enable detection of violations without false positives or false negatives while requiring only a smaller computation time.

Contributions *(summarized in Table 1.1).*

In this report, a range of monitors that fall in different places on the efficiency and accuracy spectrum are presented.

- Efficient algorithms to perform predicate detection using Hybrid Logical Clocks(HLC)[48] are presented in Chapter 3, where the runtime monitor has low monitoring overhead but can miss violations or bugs.

- An extension of Hybrid Logical Clocks (HLC) called Biased Hybrid Logical Clocks ($BHLC$) is presented in Chapter 4, and it is shown that they can identify more consistent global states than HLC and thereby perform better predicate detection than HLC [44]. BHLC are constant sized clocks like HLC, so the monitoring overhead is kept low, but violations or bugs can still be missed.

- The idea of performing predicate detection using HLC and SMT solvers is presented in Chapter 5, whrere the runtime monitor identifies all possible consistent global states in an observed run, and thereby guarantees reliable predicate detection [45]. However, it is shown that the time taken by the monitor to perform predicate detection is high in some scenarios.

- A runtime monitoring approach that has low monitoring overhead and guarantees the absence of false negatives in presented in Chapter 6. Here, the low monitoring overhead is achieved at the cost of false positives. A modification of the approach that helps in achieving a trade-off between false positives and false negatives is also discussed in the chapter.

- An efficient hybrid monitoring approach that reduces the computation time required by the monitor while ensuring the absence of false positives and false negatives is also presented in Chapter 6.

Chapter	Monitoring Approach	Correctness	Monitoring Overhead
3	Predicate detection using HLC	has false negatives	low - uses $O(1)$ sized clocks
4	Predicate detection using BHLC	has lesser false negatives	low - uses $O(1)$ sized clocks
5	Predicate detection using HLC and SMT solvers	No false positives, No false negatives	high computation time taken by the solver
6	Predicate detection using HLC accounting for clock drift and not accounting for communication based causality	has false positives	low - uses $O(1)$ sized clocks
6	Two layered monitor - combining the above two approaches	No false positives, No false negatives	smaller computation time taken by the solver

Table 1.1: Summary of properties of different monitoring approaches for performing predicate detection in distributed systems

CHAPTER 2

PRELIMINARIES

In this chapter, we will provide an overview of the key concepts that the following chapters rely on. Specifically, the predicate detection algorithms that we will discuss in the following chapters assume that the system model of the underlying distributed system (that they run on) is identical to the model presented in Section 2.1. Algorithms presented in Chapters 3, 5 and 6 assume that the underlying distributed system uses Hybrid Logical Clocks (HLC), so we introduce Hybrid Logical Clocks briefly in Section 2.2. In Section 2.3, we identify the steps involved in performing predicate detection and the need to perform predicate detection. We will also discuss some specific types of predicates, especially ones that the algorithms in the following chapters aim to detect. Finally, in Section 2.4, we discuss about runtime monitoring and the desirable characteristics of runtime monitors.

2.1 System Model

We consider a distributed system that consists of a set of n processes, p_1 to p_n. Each process p_i has its own physical clock, where $1 \leq i \leq n$. We assume that the underlying system guarantees that the local physical clocks of any two processes differ by at most ϵ, the clock drift, by using a clock synchronization protocol such as NTP[38]. The processes communicate via messages. The minimum and maximum message delays between processes are δ_{min} (could be 0) and δ_{max} (could be ∞), respectively. The execution of each process can be denoted as a sequence of events that happen at the process. Each process is associated with a set of variables, and the *state* of a process is identified by the values of its variables. We categorize the events that can happen at each process into three categories: send events, receive events, and local events. A send or a receive event corresponds to the event of a process sending or receiving a message. A local event corresponds to an event at a process where the state of the process may change. The state of a process does not change between any two consecutive events at a process. Each event e at a process i is also

associated with a physical clock value or timestamp $pt.e$, which is the physical clock value of process i when the event e happened.

2.2 Hybrid Logical Clocks

Hybrid Logical Clocks (HLC) [27] are constant sized clocks that have the benefits of both physical clocks and Logical Clocks[28]. With Hybrid Logical Clocks, every process in the system maintains two variables l, c, and together $\langle l, c\rangle$ corresponds to the process' HLC value. The variable l corresponds to the highest physical clock value (of any process) that the process is aware of and c captures causality information. Each process in the system is in charge of updating the values of its own clock variables l and c whenever an event occurs at the process.

With Hybrid Logical Clocks, events and messages in the system have timestamps associated with them. Therefore, when an event e occurs at a process, the process updates its HLC value i.e., the value of $\langle l, c\rangle$, and then assigns the updated HLC value as the timestamp of event e, say $hlc.e$. So the timestamp of event e i.e., $hlc.e$ is of the format $\langle l.e, c.e\rangle$, where $l.e$ is the highest physical clock value that the process is aware of at that moment. The value of $c.e$ is set in such a manner that together $\langle l.e, c.e\rangle$ uniquely identify event e at the process and any causality (happened-before) relationship that event e has with any other prior event is also captured.

Like Logical Clocks, Hybrid Logical Clocks provide the guarantee that for any two events e, f where $(e \rightarrow f)$, i.e., if event e happened before event f, then $hlc.e < hlc.f$. More specifically,

$$(e \rightarrow f) \Rightarrow (l.e < l.f) \vee ((l.e = l.f) \wedge (c.e < c.f)).$$

Therefore, with Hybrid Logical Clocks if two events have exactly equal timestamps, then they are concurrent events,

$$((l.e = l.f) \wedge (c.e = c.f)) \Rightarrow (e||f).$$

Furthermore, since Hybrid Logical Clocks are guaranteed to stay close to physical time, any two events e and f in a system where the clock drift between the physical clocks at the processes is at most ϵ, if $|l.e - l.f| \leq \epsilon$, then e and f could have happened at the same time.

As mentioned earlier, with Hybrid Logical Clocks, events and messages in the system have

timestamps associated with them. Specifically, when a message-send event is assigned an HLC timestamp, the message being sent is also tagged with this timestamp value, i.e. any message has an HLC timestamp associated with it which is the timestamp of the corresponding message send event. The algorithm for Hybrid Logical Clocks is as shown in Algorithm 1.

Algorithm 1 Hybrid Logical Clocks (HLC) Algorithm from [27]

Send/Local Event at process p_i

1: $l'.i := l.i$
2: $l.i := max(l'.i, pt.i)$ //tracking maximum time event, $pt.i$ is physical time at p_i
3: If $(l.i = l'.i)$ then $c.i := c.i + 1$ //tracking causality
4: Else $c.i := 0$
5: Timestamp the event (and the message for send event) with $l.i$, $c.i$

Receive Event of message *m at process* p_i

6: $l'.i := l.i$
7: $l.i := max(l'.i, l.m, pt.i)$ // $l.m$ is l value in the timestamp of the message received
8: If $(l.i = l'.i = l.m)$ then $c.i := max(c.i, c.m) + 1$
9: Elseif $(l.i = l'.i)$ then $c.i := c.i + 1$
10: Elseif $(l.i = l.m)$ then $c.i := c.m + 1$
11: Else $c.i := 0$
12: Timestamp event with $l.i$,$c.i$

2.3 Predicate Detection

Predicate detection has several applications in testing and debugging of distributed systems [5, 29, 25, 31], where the system behavior is analyzed to determine if a specific condition or predicate becomes true. Predicate detection generally involves (i) computation of all possible consistent global states and (ii) verifying if a condition or a predicate $\mathcal{P}$ (often representing a violation) becomes true in any of the consistent global states. As stated in the Introduction, a *consistent global state* of a distributed system is one that contains a local state per process (i.e., a global state of the system) and every local state is concurrent with every other local state in it. Consistent global states are also referred to as *consistent snapshots* or *consistent cuts*. Predicate $\mathcal{P}$ is a condition defined over the variables of more than one process, so we also refer to it as a global predicate. We refer to a consistent global state where $\mathcal{P}$ is true as a **valid snapshot** of the system.

Computing all possible consistent global states is essential, because even if the system did not

actually go through a computed consistent global state during the observed system run, it indicates a potential that a violation can occur in a future run of the system. We refer to bugs that become visible only when events occur at the processes in a specific relative order and do not show up in other cases as *latent concurrency bugs*. Computing all possible consistent global states is essential in identifying such latent bugs in the system. Computing consistent global states can also be useful in recovery, where a system has to recover to a consistent global state in the past where there was no violation i.e. a valid state of the distributed system before the occurrence of the actual violation or failure.

Detection of some violations in distributed systems may not require computation of global states and can be detected by local checking. For instance, some violations can be detected locally by a process in the system [2], based on its knowledge of its own state and/or based on its knowledge of the state of its neighbouring process. In this report, we will focus on the detection of predicates that require computation of all possible consistent global states. Furthermore, predicates or conditions that represent a violation or a bug can involve state information or (values of) variables related to all or a subset of the processes in the distributed system. In this report, we will focus on the detection of specific types of predicates namely Weak Conjunctive Predicates (WCP) and Arithmetic Predicates.

A weak conjunctive predicate (WCP) is true in an observed run of the system, if there exists a consistent global state in the run, where the corresponding local predicate or condition becomes true at all processes in the system [24]. In other words, conjunctive predicates are those that can be defined as a conjunction of local predicates, i.e., they are predicates of the form $\bigwedge_{j=1}^{n} pr_j$, where pr_j is the local predicate at process p_j and n is the number of processes in system. Local predicates are conditions defined in terms of the local state or local variables associated with the processes in the system. So the value of pr_j can depend on local variables at process p_j.

Classic problems like identifying the lack of a leader or mutual exclusion in distributed systems can be formulated as conjunctive predicates. For example, for mutual exclusion, where a critical resource can be accessed strictly by at most one process in the system at any time, this condition can be represented as a conjunction of local predicates, where the local predicate would correspond

to the fact that if a process is accessing the resource or not. More specifically, if each process p_i in the system has a boolean variable cs_i and the value of cs_i is true only if process p_i is accessing the critical resource, then mutual exclusion with respect to the resource can be represented as $\exists(p_i, p_j) : \{(i \neq j) \wedge (1 \leq i, j \leq n) \wedge (cs_i \wedge cs_j)\}$. In other words, the condition evaluates if there are two processes in the system that are accessing the critical resource. Thus if there is a consistent global state in an observed run, where this predicate or condition becomes true, then there is a possible violation of mutual exclusion, i.e., it indicates a possibility that more than one process was accessing the critical resource simultaneously.

Arithmetic predicates are predicates of the form $f(x_1, \ldots, x_n) \leq C$, where C is a constant, x_j is a variable at process p_j, and f is an arbitrary arithmetic function. Problems like overall resource usage, network density, aggregated parametric analysis, etc., can be expressed as arithmetic predicates. For example, consider a system of n sensors, p_1 to p_n, distributed in a radioactive environment. Let us consider that it is a violation if the overall radiation sensed by the sensors becomes more than a specific limit say C. Then this violation can be expressed as $(\sum_{i=1}^{n} s_i.r_i) \geq C$, where r_i is the radiation sensed by sensor p_i and s_i is the sensitivity of the sensor p_i. So if there exists a consistent global state where this condition becomes true, then it indicates a possible violation.

2.4 Runtime Monitoring

Distributed systems have several sources of non-determinism like non-deterministic order of events, clock drift, unanticipated environmental settings or inputs, etc. So runtime monitoring of distributed systems is essential to ensure their correctness and reliability. Runtime monitoring can be performed in an offline or an online setting. When performed in an offline setting, various aspects of the system behavior are logged during its execution and after the actual execution of the system the log is analyzed by the monitor to detect violations of the system requirements. When runtime monitoring is performed in an online setting, the system is equipped with a monitor that observes the system behavior as its executes and analyzes the observed behavior simultaneously to

detect any violations of the system requirements.

The most desirable characteristics of runtime monitors are (i) non- intrusiveness (ii) efficiency and (iii) timeliness. While non-intrusiveness and timeliness are applicable only to online monitors, efficiency applies to both online and offline runtime monitors. Specifically, an online runtime monitor is non-intrusive if it does not interfere with the underlying system execution during the monitoring process. An efficient monitor is one that does the task of monitoring with low computation overhead. Finally, the usefulness and effectiveness of an online runtime monitor depends on how quickly it can detect a violation, i.e. smaller the duration between when the monitor detected the violation and when the violation actually happened in the system the better the effectiveness and usability of the monitor. In the next chapter, we will discuss specific algorithms that aid in performing non-intrusive, timely, and efficient monitoring of distributed systems to perform predicate detection using Hybrid Logical Clocks.

CHAPTER 3

EFFICIENT PREDICATE DETECTION USING HYBRID LOGICAL CLOCKS

In this chapter, we present several algorithms for performing efficient predicate detection using Hybrid Logical Clocks. Specifically, in Section 3.1, we first discuss how the monitor can identify consistent global states in an observed run using the HLC timestamps assigned to the events at the processes. Then in Section 3.2, we briefly present the specific format in which the processes are required to report information related to their events to the monitor to aid the monitoring task. Followed by this, in Section 3.3 we present some offline and online monitoring algorithms that utilize the information reported by the processes to perform detection of conjunctive predicates. In Section 3.4, to reuse the algorithms presented in Section 3.3 to perform detection of arithmetic predicates, we identify the modifications required in terms of the format in which events are reported by the processes and the data-structures associated with the algorithms. Finally, we conclude the chapter by discussing the advantages and limitations of performing predicate detection using Hybrid Logical Clocks.

3.1 Identifying consistent global states using Hybrid Logical Clocks

With Hybrid Logical Clocks, as discussed in Section 2.2, two events with equal timestamps are concurrent events, i.e. for any two events e and f,

$$((l.e = l.f) \wedge (c.e = c.f)) \Rightarrow (e||f).$$

Local states of the processes corresponding to concurrent events are concurrent local states. A consistent global state is a set of local states, one state per process, where each local state is concurrent with every other local state in it. Therefore, in a distributed system, where events are timestamped using Hybrid Logical Clocks, a consistent global state can be formed by putting together events at the processes that have exactly equal HLC timestamps. Thus, for an HLC timestamp say t, putting together the events of all the processes with that timestamp t forms a consistent global state. For example, to obtain a consistent global state corresponding to an

HLC timestamp say $\langle l = 3, c = 2\rangle$, one has to identify the event of each process with largest HLC timestamp value say $t_1 \leq \langle l = 3, c = 2\rangle$ and put them together. So if a process does not have an event with the exact timestamp $\langle l = 3, c = 2\rangle$, but has events with timestamps $\langle l = 1, c = 0\rangle$,$\langle l = 1, c = 1\rangle$,$\langle l = 3, c = 0\rangle$,$\langle l = 3, c = 1\rangle$ and $\langle l = 4, c = 0\rangle$, then the local state corresponding to the event with timestamp $\langle l = 3, c = 1\rangle$ is chosen. This is because the local state of the process clearly did not change in between timestamps $\langle l = 3, c = 1\rangle$ and $\langle l = 4, c = 0\rangle$ (no other events in that duration), so the local state of the process at timestamp $\langle l = 3, c = 2\rangle$ should be same as that at timestamp $\langle l = 3, c = 1\rangle$. This can also be viewed as creating a phantom local event at that process with timestamp $\langle l = 3, c = 2\rangle$ where the local state of the process is same as it was at $\langle l = 3, c = 1\rangle$.

3.2 Predicate Detection Framework

To identify concurrent events in the system, the monitor needs information regarding the events at all the processes and their associated timestamp information. Therefore to aid the monitor in performing predicate detection, the processes in the system report their events and their associated HLC timestamp information to the monitor. Furthermore, to evaluate the predicate or condition of interest in the computed consistent global states, the monitor has to know the local state information at the processes (values of the local variables, especially those that are required to evaluate the global predicate) corresponding to the computed global state. For instance, in the mutual exclusion violation detection example discussed in Section 2.3, to compute the value of the predicate: $\exists(p_i, p_j) : \{(i \neq j) \wedge (1 \leq i, j \leq n) \wedge (cs_i \wedge cs_j)\}$ in a consistent global state say corresponding to HLC timestamp t, the monitor has to know the value of the variable cs_i at every process i at timestamp t. So when a process p_i in this system reports an event to the monitor, it also reports the latest value of its variable cs_i after the occurrence of the event. Thus, in general, for the purpose of monitoring, when events occur, the processes in the distributed system report the events with their associated timestamps, and also any other required local state information to the monitor depending upon the predicate being monitored. In the case of offline runtime monitoring, these

pieces of information are recorded or logged for offline analysis in the future, rather than being reported to the monitor in real time.

3.2.1 Reporting information required for the detection of weak conjunctive predicates

Let us consider the generalized form of weak conjunctive predicates (WCPs) presented in Section 2.3, where each process p_j in the system has a local predicate pr_j associated with it and $\bigwedge_{j=1}^{n} pr_j$ denotes a weak conjunctive predicate. If the goal of the monitor is to detect weak conjunctive predicates of the form $\bigwedge_{j=1}^{n} pr_j$, then it is sufficient for the processes in the system to only report when the value of the local predicate pr changes, rather than reporting all types of events at the processes. Therefore, whenever the value of the local predicate pr_j at a process p_j changes, the process has to update its local Hybrid Logical Clock value and report the current value of the local predicate pr_j along with the updated Hybrid Logical Clock value. Here since pr represents a condition, its value is either true or false at any instant. We will refer to instants at which the value of the local predicate pr changes as ***change points***.

Note that in this scenario, the goal of the monitor is to find a common HLC timestamp t where the corresponding values of the local predicates at all the processes are true. In the rest of this chapter, we will refer to this generalized form of weak conjunctive predicates while discussing the various monitoring algorithms to perform detection of WCPs.

Remark 1. *Different monitoring techniques may require the processes to report different information. Since monitoring algorithms in this chapter utilize Hybrid Logical Clocks, and with Hybrid Logical Clocks two events with exact same timestamps are concurrent (guaranteed to not have a happened before relationship), it is sufficient to report the timestamps and not the details related to the type of the event (local event, message send or receive event). In Chapter 5, we will see an example where the processes report event details like type of the event to the monitor to help identify all possible consistent global states by eliminating global states where the events have a happened before relationship.*

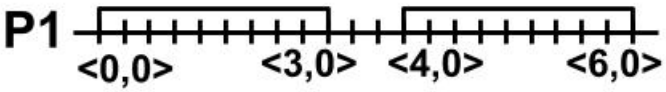


Figure 3.1: Intervals of process p_1 timestamped using HLC

In online runtime monitoring, the reporting-messages from the processes can arrive at the monitor in any order. Therefore, rather than reporting individual *change points*, reporting is done in an interval format. More specifically, for detection of WCP, each process p_j reports to the monitor the intervals when the local predicate pr_j is true. For instance, a process in the system reports the interval $[\langle l_1, c_1\rangle; \langle l_2, c_2\rangle)$, where $\langle l_1, c_1\rangle$ is the timestamp of the change point when the local predicate first became true and continued to be true until $\langle l_2, c_2\rangle$, i.e., $\langle l_2, c_2\rangle$ is the HLC timestamp of the change point when the local predicate became false. In other words, the local predicate was true at any HLC timestamp t s.t.$\langle l_1, c_1\rangle \leq t < \langle l_2, c_2\rangle$. Note that the interval is left-closed and right-open, so it includes the starting timestamp $\langle l_1, c_1\rangle$ but does not include the ending timestamp $\langle l_2, c_2\rangle$. Thus, if we have an interval ending at time t and another interval beginning at time t (corresponding to a different process), in a sorted list of intervals reported by all processes at the monitor, the ending event will occur before the begin event. If multiple interval endpoints (corresponding to different processes) start at the same time, their ordering can be arbitrary. The same observation applies to interval endpoints that end at the same time.

We illustrate reporting and timestamping in Figure 3.1, where for simplicity we set the c values in the HLC timestamps to be 0. In this figure, process p_1 has two intervals during which its local predicate was true. Process p_1 reports the first interval as $[\langle 0, 0\rangle; \langle 3, 0\rangle)$, i.e., the local predicate was first true at p_1 when its Hybrid Logical Clock value was $(l = 0, c = 0)$ and became false at $\langle l = 3, c = 0\rangle$. Similarly p_1 reports the next interval as $[\langle 4, 0\rangle; \langle 6, 0\rangle)$.

Consider the case where the processes report individual change points rather than intervals. For example, let us consider that reports of change points from a process get delivered at the monitor in this order $\langle 0, 0\rangle$, $\langle 6, 0\rangle$, $\langle 3, 0\rangle$, $\langle 4, 0\rangle$. The monitor will incorrectly assume that the predicate

Section	Algorithm	Running Time	Assumption
3.3.1	Naive	$O(In)$	Offline
3.3.2	$C\text{-}point$ Tree	$O(I \log I)$	-
3.3.3	Naive with Minheap	$O(I \log n)$	Local Ordering Property
3.3.3	Naive with Δ-Bounded Message Delay	$O(I \log(\Delta n))$	Message Delivery Property

Table 3.1: Summary of Conjunctive Predicate Detection Algorithms based on HLC

holds at this process for the interval $[\langle 3,0\rangle, \langle 4,0\rangle)$ after receiving the change point $\langle 6,0\rangle$, i.e., until it receives the change point $\langle 3,0\rangle$. The situation becomes even more complex when considering intervals from multiple processes. Thus, we require that the processes report change points as a pair to the monitor. So the goal of the monitor is to find an overlap in the reported intervals i.e., *to find a common HLC timestamp* $\mathbf{t}$ *that is in some interval reported by each process in the system.*

An issue with this reporting mechanism is that if the local predicate stays true forever at any process, then the corresponding interval is never reported, until the end of the system execution. To avoid stalling the detection task under such scenarios, we will split such long intervals by creating artificial endpoints at thresholds and report them as smaller intervals.

3.3 Algorithms for Detecting Conjunctive Predicates

In this section, we will consider various algorithms that perform conjunctive predicate detection using Hybrid Logical Clocks. The first algorithm in Section 3.3.1 is applicable in an offline setting where the intervals from all the processes are logged and available upfront for analysis by the monitor. The remaining algorithms are applicable in an offline or an online setting, i.e., they can handle the scenario where the intervals are not available ahead of time and are reported by the processes in real-time. The summary of the algorithms discussed in this section is provided in Table 3.1.

3.3.1 Naive Algorithm

In this section, we present our first, naive, algorithm for detecting weak conjunctive predicates. This algorithm works in an offline setting, so we assume that each process reports all of its intervals in a sorted order to the monitor when the process terminates. As noted in the previous section, the goal of the monitor is to find a timestamp t that is included in some interval reported by each of the n processes. Given this as the goal, we can reduce the problem to finding the maximum overlap among all the intervals. If this overlap is n, then we know that there is a timestamp where the global predicate is true. Moreover, this algorithm immediately extends to the more general case where we want to evaluate whether pr_j is true for some fixed size subset of processes. This algorithm is based on a key observation that the value of the maximum overlap among the intervals would get updated only at the endpoints of the intervals.

Outline of the algorithm. Our algorithm processes each change point one at a time, in order, from the smallest timestamp to the largest timestamp, maintaining the current overlap value which is initialized to 0. Recall that each interval has two change points, a left endpoint and a right endpoint. If the current change point is a left endpoint, we add +1 to the current overlap and check if the overlap is n, in which case we report that the conjunctive predicate is satisfied. If the current change point is a right endpoint, we add -1 to the current overlap. A pseudo-code description is provided as Algorithm 2.

Algorithm 2 Naive Algorithm

```
1:  loop until there are no unprocessed change points
2:      t := smallest timestamp among unprocessed change points
3:      X := set of change points with timestamp t
4:      p[t] := 0 // net effect at t, initialized to 0
5:      foreach cp ∈ X
6:          p[t] := p[t] + p[cp]
7:      overlap_count := overlap_count + p[t]
8:      if overlap_count = number_of_processes then
9:          Report conjunctive predicate detected
10:     end if
11: end loop
```

Illustration of Algorithm 2. We illustrate this algorithm using the example in Figure 3.2, where timestamps are simplified to non-negative integers. Our algorithm first processes the change point a with timestamp 0 (this is the value of l, considering $c = 0$ for the sake of simplicity). Since a is a left endpoint, the value of $overlap\_count$ increases from its initial value of 0 to 1. Our algorithm then processes the change point b with timestamp 1, and the value of $overlap\_count$ increases to 2. Then the algorithm processes change points c and d together with common timestamp 2. We compute $p[2] = p[c] + p[d] = 0$ because $p[c] = +1$ and $p[d] = -1$, so $overlap\_count = 2 + p[2] = 2$. The naive algorithm then processes change point e with the timestamp 3, and the value of $overlap\_count$ increases to 3. Since this satisfies the condition of $overlap\_count = number\_of\_processes$, a conjunctive predicate is detected and reported the monitor.

Advantages and disadvantages of the algorithm. The naive algorithm is reasonably efficient with a worst case complexity of $O(In)$ in a system of n processes and I reported intervals, and is very easy to understand. However, it has two key disadvantages. First, it requires all the events or change points to be available at the beginning. If we try to use this in an online setting where the events of different processes are reported at different speeds, then it may not work correctly. For instance, consider the example in Figure 3.2, if $[d, h)$ is reported after $[e, g)$ so that the change point d is processed after the change point g, then the algorithm will fail to detect that the global predicate is true at time 4. Second, while its worst case complexity is polynomial, there is a linear dependence on n, the number of processes. We focus on alleviating these deficiencies in our next algorithm in Section 3.3.2.

3.3.2 C-point tree Algorithm

In Section 3.3.1, we described a naive algorithm with complexity $O(In)$ that can be used for offline detection of WCPs, where all the intervals are available in advance. In this section, we focus on adapting that algorithm so that (1) it can be used online, where the monitor receives each interval in a possibly unsorted manner, and (2) its complexity does not directly depend on n, the number of

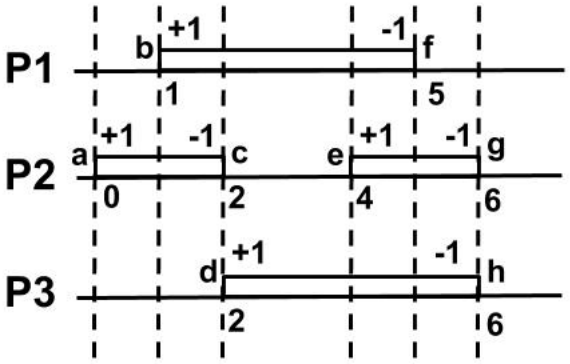


Figure 3.2: Conjunctive Predicate Detection when Local Predicates are true for some duration of time

processes.

Change-point tree (*C-point* tree). The key notion behind this algorithm is to use an augmented tree from [15], that we refer to as a change point tree (*C-point* tree), to keep track of the change points reported by the processes. The *C-point* tree differs from other augmented trees like interval or segment trees in that nodes represent change points rather than intervals. Specifically, the *C-point* tree maintains the change points in a balanced binary tree sorted by HLC timestamp which allows us to insert or delete change points efficiently. The timestamp of a node is denoted $t[node]$. For any *C-point* tree T with node x, let $T(x)$ denotes the subtree of T rooted at node x. Each node x is augmented with three additional attributes that allows us to calculate the maximum overlap efficiently. These values are $p[x]$, which is $+1$ or -1 depending on whether x is a left or right endpoint of an interval, $v[x]$ which denotes the sum of all p values of nodes in $T[x]$, and $m[x]$, a comparatively complex attribute. Let w be any node in $T(x)$; then $sum_x(w)$ is the sum of all p values of nodes in $T(x)$ with timestamps at most $t[w]$. Then $m[x] = \max_{w \in T(x)} sum_x(w)$. It follows that, at any time, $m[root]$ gives the maximum overlap that occurs at any timestamp for the intervals and change points that have been processed.

Outline of the algorithm. We now illustrate how the algorithm operates, which basically involves how we update the tree when a change point is inserted. We start with an empty *C-point* tree T. Intervals arrive at the monitor in any order, which inserts the left and right endpoints into T. Let x be a change point to be inserted into T. We set $p[x]$ based on whether the *node* corresponds

to a left or right endpoint. We then insert x into T. We defer setting $v[x]$ or $m[x]$ until x has been inserted.

After insertion, which may involve re-balancing T, we must update $v[y]$ and $m[y]$ for each node y whose subtree was modified during the insertion. Because we are working with a balanced binary search tree, this is at most $O(\log I)$ nodes. We update $v[y]$ and $m[y]$ for the affected nodes in a bottom up fashion as follows.

$$v[y] = v[left\_y] + p[y] + v[right\_y]$$

$$m[y] = max \begin{cases} m[left\_y], \\ \textbf{//max is in the left subtree of y} \\ v[left\_y] + p[y], \\ \textbf{//max is at y} \\ v[left\_y] + p[y] + m[right\_y], \\ \textbf{//max is in the right subtree of y} \end{cases}$$

For the above formula, we do need to handle the special case where y has no left child, in which case $m[y]$ is the max of the second and third cases. After all the nodes have been updated, we check $m[root]$ to determine if the global predicate has been satisfied. For efficiency, if multiple change points have the same timestamp, we maintain a single node for that timestamp. Pseudo-code for the algorithm with this optimization is presented as Algorithm 3.

Illustration of Algorithm 3. We revisit the example in Figure 3.2 to illustrate Algorithm 3. Let us consider that the intervals are reported to the monitor in the following order: $[a, c)$, $[b, f)$, $[d, h)$, $[e, g)$. After the first interval is reported, two nodes are inserted into the tree, $(P2, 0, +1)$ for change point a and $(P2, 2, -1)$ for change point c. Here the node contains process id, (simplified) HLC timestamp associated with the change point and the value associated with the endpoint i.e., +1 or -1. The resulting tree, with nodes for change points a and c inserted first, is shown in Figure 3.3 (a).

When $[b, f)$ is reported to the monitor, it inserts two nodes $(P1, 1, +1)$ and $(P1, 5, -1)$ corre-

Algorithm 3 Algorithm for inserting a change point into a *C-point* tree

Given a new change point from process p_j:

1: $node\ x := create\_node(j, p\_clock, p[x])$ //if x is left endpoint $p[x] = +1$, $p[x] = -1$ otherwise
2: **If** node x (with p_clock) exists in the tree already **then**
3: Append j to process id of the existing node,
Add value of $p[x]$ to existing $p[x]$
// values of $v[x]$ and $m[x]$ will be updated below
4: **Else**
5: $insert\_node(x)$ // maintains balanced tree
6: X^* = { y | the modification or insertion of node x caused a change at node y or some descendant of node y }
// node x is in X^*
7: **for** each node y in X^* in a bottom up fashion **do**
8: $v[y] := v[l] + p[y] + v[r]$
//l is $y \rightarrow left\_node$, r is $y \rightarrow right\_node$
//if l or r does not exist then $v[l/r] = 0$,$m[l/r] = 0$
9: $m\_left := m[l]$
10: $m\_node := v[l] + p[y]$
11: $m\_right := v[l] + p[y] + m[r]$
12: $temp := max(m\_node, m\_right)$
13: **If** l exists, **then** $m[y] := max(m\_left, temp)$
14: **Else** $m[y] = temp$
//$m[y]$ must include one timestamp
15: **end for**
16: **if** $m[root] = n$ **then**
17: $max\_pred\_set := trace\_back(m[root])$
18: **end if**

sponding to b and f and updates attributes m and v of the nodes affected by the insertion as shown in Figure 3.3 (b). We mark the affected nodes by a star to identify nodes that need to recompute their values for m and v. When $[d, h)$ is reported to the monitor, since the tree has an existing node $(P2, 2, -1)$ with the timestamp value 2, the monitor updates the node $(P2, 2, -1)$ to $((P2, P3), 2, 0)$ (Lines 2 to 3 in Algorithm 3); note that the value of p increases from -1 to 0. Then the monitor creates and inserts node $(P3, 6, -1)$ for change point h resulting in the tree shown in Figure 3.3 (c). Finally, when $[e, g)$ is reported to the monitor, the insertion of node $(P3, 4, +1)$ for the change point e will make $m[root]$ to be 3 (equal to the number of processes) and the monitor can report that the global predicate was satisfied. The final tree after the change point g gets processed, results

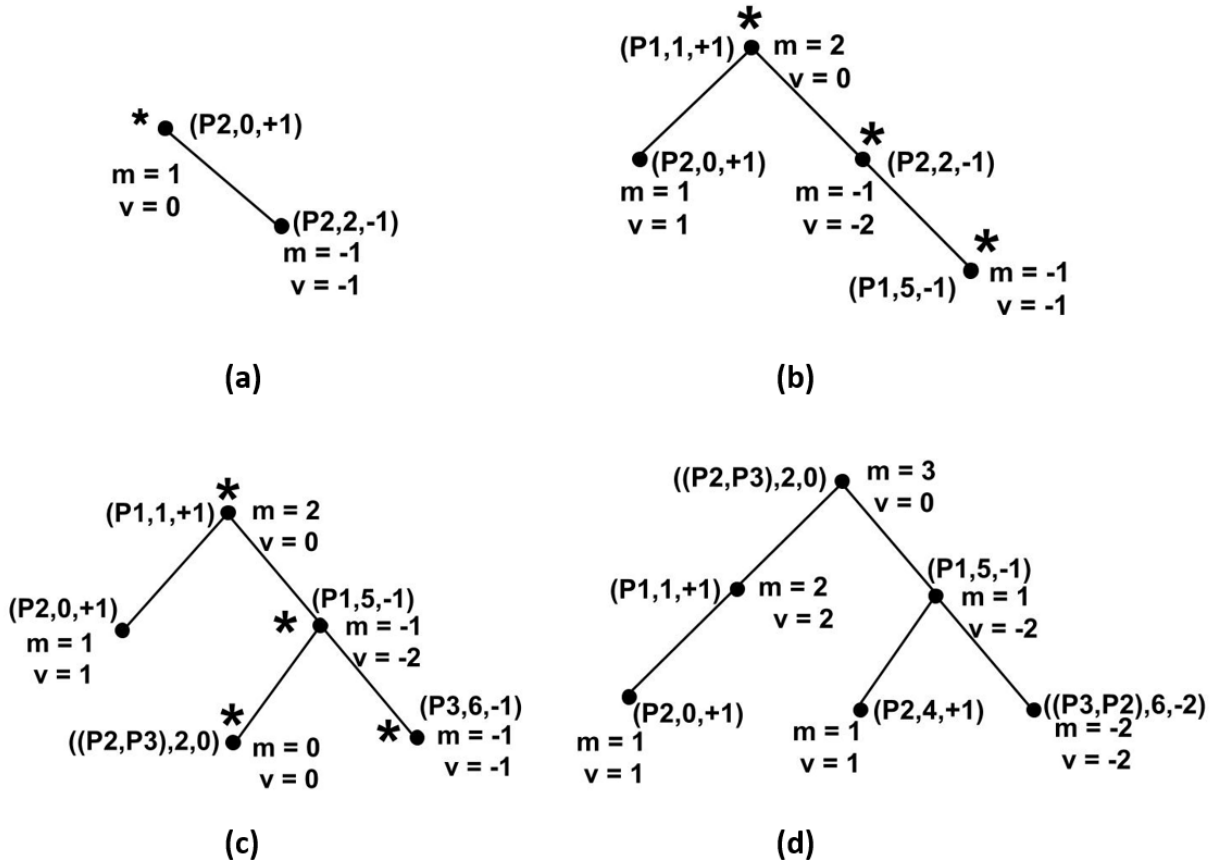


Figure 3.3: An Illustrative Example for Algorithm 3

in existing node $(P3, 6, -1)$ getting updated to $((P2, P3), 6, -2)$, as shown in Figure 3.3 (d).

Advantages and disadvantages of the algorithm. This algorithm has several advantages over the naive algorithm in the previous section. First, it can be used in an online setting at runtime and is robust against messages arriving out of order and arbitrarily large delays as long as the delays are finite. Whenever all the necessary intervals arrive, thereby allowing the detection of the global predicate, this information will be available at the root node. In fact, once the intervals have been inserted, detection of conjunctive predicate just means checking if $m[root] = n$, which requires $O(1)$ time. The main drawback with this algorithm is that I may be large; in fact, $I \log I$ may be larger than In. Unfortunately, the running time cannot be improved without making some additional assumptions, because we must sort the change points when there is no guarantee about the ordering of intervals, and sorting I elements requires at least $\Omega(I \log I)$. We next explore some

possible improvements when we make some additional assumptions.

3.3.3 Monitoring with report/message delivery guarantees

In this section, we return to the naive algorithm from Section 3.3.1 and show that we can perform runtime monitoring and reduce the worst case time complexity if we add some assumptions about *message delivery to the monitor*. We consider two types of message delivery assumptions: (1) local ordering property and (2) bounded message delay. We first focus on the **local ordering property** and then show how to adapt this algorithm to leverage bounded message delay. So let us consider that the following is guaranteed:

Intervals reported by a process arrive in a FIFO manner at the monitor.

In this case the only sorting that is required is sorting intervals across the processes. So we extend Algorithm 2, by utilizing a min-heap of size n to perform this sorting. Specifically, all unprocessed change points from different processes are buffered in separate lists. The min-heap is filled by fetching change points from these lists of sorted intervals. Specifically, the min-heap contains at most n change points, one unprocessed change point per process. Each of these change points is the smallest unprocessed change point at the corresponding process, and this is straightforward because of the FIFO guarantee among intervals from the same process. The extended algorithm is as shown in Algorithm 4. This algorithm first checks if the min-heap is full (one change point per process) and only if it is full it extracts the change point with the smallest timestamp. Rest of the processing is same as in Algorithm 2. Observe that the size of the heap is always at most n. Therefore to process each change point, it takes $O(1)$ time to find the minimum in the heap, $O(\log n)$ time to remove this minimum from the heap and $O(\log n)$ to insert the next change point into the heap. Hence, the complexity to process one change point is $O(\log n)$. Since the number of change points is $2I$ (two change points or endpoints per interval), the overall complexity is $O(I \log n)$.

Advantages and disadvantages of this algorithm. This algorithm has two key advantages: it has $O(I \log n)$ worst case complexity, so it is efficient, and it can be used in both offline and

Algorithm 4 Online min-heap algorithm based on local ordering property.

```
1: H := min-heap containing first endpoint of each process
2: overlap_count := 0
3: loop
4:     p := ExtractMin(H)
5:     overlap_count := overlap_count + p.val
6:     if overlap_count = number_of_processes then
7:         Report conjunctive predicate detected
8:     end if
9:     Insert(H, next endpoint from p.process_id)
10:    // wait if the next endpoint is not available yet.
11: end loop
```

online settings. It has the disadvantage that it requires the intervals from each process to arrive in FIFO order at the monitor. The online version has the disadvantage that if some process takes a long time to report an interval, the monitor must buffer change points reported by other processes. Such a delay may occur if the local predicate at that process remains false for a longer duration or because of message delays.

Bounded Message Delay. Now we will consider an alternative assumption i.e. we assume that the messages are delivered at the monitor with bounded delay. Specifically, we introduce the following bounded message delay property.

At time t, the monitor has received any message that was sent by time $t - \Delta$.

Recall that Algorithm 4 maintains the unprocessed change points for each process in a sorted list and a newly reported change point can just be appended to the end of the appropriate list. Given that the change points may not arrive in order, we modify the management of these lists by utilizing another min-heap.

At any point in time t, the monitor maintains a latest timestamp value $max(t)$. This is initially set to 0. We maintain the invariant that all change points with timestamps at most $max(t) - \Delta$ have been placed into the sorted lists for their respective processes (and perhaps subsequently entered into the min-heap where points are actually processed). The remaining change points are stored in a single min-heap of unplaced change points, where the heap order is still based on timestamps.

When the monitor receives a new interval message, it checks to see if the message results in an update of $max(t)$. If so, it updates $max(t)$, adds the two new change points to its min-heap of unplaced change points, and then extracts the change point with the minimum timestamp, say x, if that change point has timestamp $t(x) \leq max(t) - \Delta$ and places x into the corresponding process's sorted list of change points. This extraction continues until the minimum timestamp change point x has timestamp $t(x) > max(t) - \Delta$. The rest of the algorithm is same as Algorithm 4.

The running time for this algorithm is identical to that of Algorithm 4, except for the min-heap required to process unplaced change points. We observe that this min-heap for unplaced points must have a bounded size as follows. The timestamp of every change point on a given process is unique. Also, in HLC [27], the timestamp is of the form $\langle l, c \rangle$, where l captures the most recent physical time that the change point was aware of and c is a bounded counter. In [27], it has been shown that the theoretical maximum value of c is $O(n)$. (In practice, this value remains less than 5.) Hence, the number of change points reported by a process in Δ time units is $O(\Delta n)$. And, the total number of change points reported by all the processes in Δ time units is $O(\Delta n^2)$. Thus, the min-heap for unplaced change points has a size of at most $O(\Delta n^2)$ and so the maximum time for any insertion or deletion is at most $O(\log \Delta n)$, giving a total additive cost of $O(I \log \Delta n)$.

To use this algorithm in an online setting, we must ensure that the messages not only reach the monitor in a timely manner, but that each process provides frequent updates. Specifically, we must ensure that each process p_j sends at least one message to the monitor within each closed interval of length Δ. Without this condition, even with bounded message delay, the monitor cannot be certain that it has up to date information on process p_j and cannot detect predicate satisfaction in a timely manner.

Advantages and disadvantages of this algorithm. This algorithm has two advantages: it has $O(I \log \Delta n)$ worst case complexity, so it is efficient, and it can be used in both offline and online settings. It has the disadvantage that it requires intervals from each process to arrive at the monitor with bounded delay. The online version also has the disadvantage that if the local predicate at some process stays true for a long time, then the monitor may not have up to date information

about that process. So to handle this, the processes in the system are required to send frequent updates, specifically within each closed interval of length Δ.

3.4 Detecting Arithmetic Predicates

In this section, we discuss the modifications required to the reporting framework and to the algorithms presented in Section 3.3, to use them in detecting arithmetic predicates. For arithmetic predicates, we assume that each process p_j has a variable x_j and we wish to test some inequality $f(x_1, \ldots, x_n) \geq C$. For the purpose of our discussion, we will focus on the detection of arithmetic predicates of the type $(\sum_{j=1}^{n} c_j.x_j) \geq C$, where x_j is an integer variable of process p_j, c_j and C are constants.

A process p_j can report changes in the value of x_j individually or in an interval format. Specifically, if the variable x_j increases from 4 to 7 and then to 6, process p_j can report the change to 7 as an $incval$ of 3 with its corresponding timestamp, and then report the change to 6 as $incval = -1$ with its corresponding timestamp. The second alternative is to impose an interval structure on the change points by having the process report consecutive change points in overlapping pairs. So if the value of variable x_j changes from 4 to 7 and then to 6, p_j reports the pair of values $(4, 7)$ and their corresponding timestamps, when it encounters the change to 7. It then reports the pair of values $(7, 6)$ and the corresponding timestamps, when it encounters the change to 6.

Offline Arithmetic Predicate Detection. Recall that in an offline setting all the change points are available up front. If the changes points are available upfront, but are not in a sorted order, then we can run any of our presented algorithms, at worst case with an $O(I \log I)$ additive cost for sorting them and then executing the algorithms. This applies regardless of the reporting mechanism. If we assume that the points from each process are reported in order, then the algorithms can be applied with no additional $O(I \log I)$ cost. Furthermore, with arithmetic predicates if multiple change points have the same timestamp, then we must process all the change points for a given timestamp before we can assert the predicate is true at that timestamp.

Online Arithmetic Predicate Detection. The reporting mechanism plays a critical role in

performing online arithmetic predicate detection. If processes report individual change points rather than intervals, then without any assumptions about message delivery, the resulting detection can be incorrect. For instance, if the change points are delivered in an incorrect order, any of the discussed monitoring algorithms can make incorrect conclusions. Therefore, we must either assume the interval-based reporting mechanism or make some additional assumption about the message order or message delivery. We consider both the options below.

Local Ordering Property. If we assume that each process delivers messages in order to the monitor, as in the first case in Section 3.3.3, then we can use either reporting mechanisms and employ Algorithm 4. The key observation is that when we process a change point, we know that it is the change point with the minimum timestamp. As was the case in Section 3.3.3, the online monitor may stall and require large buffers if some processes are slow in delivering messages to the monitor. Also, as noted above, we cannot report predicate satisfaction at timestamp t until all change points with timestamp t have been processed.

Bounded Message Delay Property. If we assume that each process delivers messages with bounded delay to the monitor, as in the second case in Section 3.3.3, then we can use either reporting mechanisms if the monitor employs the modified algorithm from Section 3.3.3. To ensure timely and correct predicate detection, we do need to further require frequent updates from the processes to the monitor.

Change Points Delivered in Pairs. If we assume that the change points are delivered in pairs i.e., as intervals to the online monitor, we can reuse Algorithm 4 with only a slight modification without assuming anything about message order or message delay. Specifically, for each process p_j, the online monitor has to store p_j's unprocessed messages and maintain a variable $val(j)$, which is the current maximum validated timestamp at p_j. The idea behind $val(j)$ is that $val(j)$ is the largest timestamp such that the monitor knows the exact value of x_j for all the timestamps in the interval $[0, val(j))$. We maintain the invariant that all the change points of process p_j with timestamp up to $val(j)$ are placed into process p_j's sorted list of change points (and perhaps subsequently entered into the min-heap). Initially $val(j)$ is undefined and each process sends out

a special message with timestamp 0 and the initial value of x_j.

When the monitor receives a new message from process p_j, it checks to see if the first change point in the message has timestamp identical to $val(j)$ or if it is the special initialization message. If so, then it can update $val(j)$ and add the corresponding change point into p_j's sorted queue. It then repeatedly checks its received messages to determine if the next message in order has already been received, in which case it updates $val(j)$ and adds the next change point into p_j's sorted queue until this cannot continue. Otherwise, the message is added to the unprocessed messages from p_j.

The rest of the Algorithm 4 can be used as it is. Therefore, the extra time required is for processing and managing the unsorted messages. We can maintain this list using a min-heap, which would require at most $O(\log I)$ time for each insertion and deletion, thereby adding an $O(I \log I)$ complexity or cost to the running time of the algorithm in the worst case. This assumes a worst case, where each process's min-heap is large. In practice, this seems unlikely and the algorithm will likely run more quickly.

3.5 Advantages and Limitations of HLC based Predicate Detection

Predicate detection using HLC has low overhead because HLC timestamps are $O(1)$ sized timestamps. Furthermore, based on the property of HLC which guarantees that events with equal HLC timestamps are concurrent, identifying a common timestamp t, where the local predicates at all the processes are true can be done efficiently and in a timely manner using the algorithms presented in this chapter. However, predicate detection based on HLC has the key disadvantage that it can result in false negatives, i.e., it can fail to identify some possible valid instances were the predicate was satisfied. This is because Hybrid Logical Clocks only guarantee one way causality, i.e., for any two events e, f, $(e \rightarrow f) \Rightarrow \{(l.e < l.f) \vee ((l.e = l.f) \wedge (c.e < c.f))\}$. The reverse implication is not guaranteed, i.e., $\{(l.e < l.f) \vee ((l.e = l.f) \wedge (c.e < c.f))\} \nRightarrow (e \rightarrow f)$. This extends to the fact that, if events have equal timestamps, they are guaranteed to be concurrent events, but concurrent events are not guaranteed to have equal timestamps. Due to this lack of

information, Hybrid Logical Clocks fail to identify all possible consistent global states and thereby fail to identify all instances of predicate satisfaction. Therefore in the next chapter, we will focus on improving the ability of Hybrid Logical Clocks to perform predicate detection. We will do so by modifying some aspects of Hybrid Logical Clocks and thereby introducing a new type of clock called Biased Hybrid Logical Clocks (BHLC).

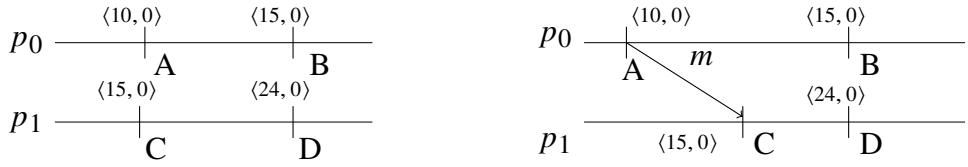
CHAPTER 4

IMPROVED PREDICATE DETECTION USING BIASED HYBRID LOGICAL CLOCKS

As discussed in the previous chapter, predicate detection based on HLC timestamps have low overhead, but do not guarantee detection of all possible instances of predicate satisfaction. This is mainly because like Logical Clocks, Hybrid Logical Clocks also guarantee only one way causality. Therefore in this chapter, we will focus on extending the notion behind Hybrid Logical Clocks and introduce a new type of clock called Biased Hybrid Logical Clocks($BHLC$), that aim to reduce the number of missed instances of predicate satisfaction in a system. Specifically, in Section 4.1 we will introduce the idea behind $BHLC$ and present the algorithm. In Section 4.2, we consider some extensions of $BHLC$. In Section 4.3, we will present the approach to perform predicate detection using $BHLC$. In Section 4.4, we will study the effectiveness of $BHLC$ in performing predicate detection, especially by comparing it with HLC based predicate detection. Finally, in Section 4.5, we will conclude by discussing the benefits and limitations of performing predicate detection using $BHLC$.

4.1 Biased Hybrid Logical Clocks

As discussed in Section 2.3, the first step in predicate detection is identifying all possible consistent global states. The key to identifying all possible global states is the ability to find all possible concurrent events across processes in the system. Recall that this is because local states corresponding to concurrent events at the processes are concurrent local states and every local state in a consistent global state is concurrent with every other local state in it. Vector clocks (VC) [18, 36] which are $O(n)$ sized clocks have this ability to identify all possible concurrent events in the system. However, Vector Clocks do not account for physical time information, so two events that are concurrent based on their VC timestamps may not be actually concurrent if they are far apart in physical time. This can lead to false positives while performing predicate detection using VC, i.e., some identified instances of predicate satisfaction may be invalid. Hybrid



(a) No communication scenario (b) With communication scenario

Figure 4.1: Identifying Concurrent events using HLC timestamps

Vector Clocks [46], that extend the notion of VC, overcome this issue because they account for physical time information. However, like VC timestamps, HVC timestamps are also $O(n)$ sized timestamps in the worst case. Therefore, the monitoring overhead is high when using VC or HVC timestamps for predicate detection. As discussed in Section 3.5, predicate detection using HLC timestamps have low overhead because they are constant sized clocks, but the monitor may suffer from false negatives. In other words, HLC based predicate detection may fail to detect some instances of predicate satisfaction. Thus, it is desirable to have clocks, that are of ***constant size*** like Hybrid Logical Clocks, but that ***result in reduced number of false negatives*** when used to perform predicate detection.

The issue with using HLC timestamps to perform predicate detection is that when events across processes are ordered based on HLC timestamps, one cannot identify all possible concurrent events. This leads to false negatives. For example, consider the two scenarios in Figure 4.1, where events A, B, C, and D have the exact same HLC timestamps in Figure 4.1 (a) and Figure 4.1 (b). Let us consider that the maximum clock drift $\epsilon = 15$. Clearly, HLC timestamps do not help in distinguishing between the two cases considered in the figure. In the first scenario, the set of possible concurrent events are (A, C), (A, D), (B, C) and (B, D). In the second scenario, the set of possible concurrent events are (B, C) and (B, D). However, based on the HLC timestamps, the only conclusion that one can make in both the cases is that B and C are concurrent events because they have equal HLC timestamps. The order between events (A, C), (A, D), and (B, D) that have unequal HLC timestamps, remains unclear. Based on this observation, we will now extend Hybrid Logical Clocks to introduce a new type of clock called Biased Hybrid Logical Clocks($BHLC$), that

aim to identify more concurrent events than HLC.

4.1.1 Hybrid Logical Clocks - Naive Algorithm

We start with the naive version of Hybrid Logical Clocks (from [27]) shown in Algorithm 5. The naive version maintains a single clock variable l at all processes rather, than maintaining l and c.

Specifically, in naive Hybrid Logical Clocks algorithm, each process p_j maintains a variable $l.j$ that captures the logical time associated with an event. Intuitively, $l.j$ corresponds to a logical clock that is subject to the constraint that $l.j$ is always at least as large as $pt.j$, the physical time at p_j. Hence, for a send event, rather than just increasing $l.j$ by 1, we set $l.j$ to be $max(l.j + 1, pt.j)$. And, for a receive event, instead of setting $l.j$ to be $max(l.j + 1, l.m + 1)$, we set it to $max(l.j + 1, l.m + 1, pt.j)$. Thus, the naive Hybrid Logical Clocks algorithm is as shown in Algorithm 5.

Algorithm 5 Naive Hybrid Logical Clocks Algorithm from [27]

At node j

1: Initially $l.j := 0$

Send/Local event

2: $l.j := max(l.j + 1, pt.j)$

3: Timestamp with $l.j$

Receive event of message ***m***

4: $l.j := max(l.j + 1, l.m + 1, pt.j)$

5: Timestamp with $l.j$

Key properties of naive HLC are:

$$(e \rightarrow f) \Rightarrow (l.e < l.f) \qquad \text{and} \qquad (l.e = l.f) \Rightarrow (e||f)$$

A key disadvantage of the naive HLC algorithm is that the drift between $l.j$ and $pt.j$ could grow unbounded. Whereas, in the HLC algorithm (c.f. Algorithm 1 in Section 2.2), instead of adding 1 to $l.j$ or $l.m$ when an event is created, the value of l can remain unchanged. Since this can create the possibility of two successive events on a process (or a send event and the corresponding receive event) to have the same timestamp, an additional variable $c.j$ is maintained. $c.j$ is a counter

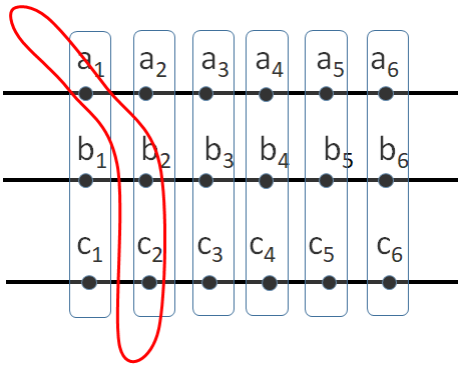


Figure 4.2: A sample execution in a system of three partially synchronized processes.

(shown to be provably bounded). Thus, HLC timestamps which are of the form $\langle l, c \rangle$, satisfy the following property,

$$(e \rightarrow f) \Rightarrow (l.e < l.f) \vee ((l.e = l.f) \wedge (c.e < c.f))$$

The Biased Hybrid Logical Clocks algorithm proposed in this chapter is based on the naive version of Hybrid Logical Clocks described in Algorithm 5. Hybrid Logical Clocks and *naive* Hybrid Logical Clocks have the same capability in terms of being able to detect the satisfaction of a given global predicate.

4.1.2 Idea of using a Bias

Consider the system execution shown in Figure 4.2, where there are three processes, and six events per process. We assume that the physical clock value of a process increases by 1 between every two events. Furthermore, assume that the maximum clock drift ϵ is 10 and that there are no messages in this execution. This implies that $\langle a_i, b_j, c_k \rangle$ is a possible consistent global state for any $1 \leq i, j, k \leq 6$. In such a system, let the predicate of interest be $\bigwedge_{q=1}^{n} pr_q$, where pr_q is a local predicate at process p_q.

In this execution, since there are no messages, all the events are local events, so the value of $l.e$ (in the HLC timestamp of e) for an event e is same as the physical time of the process when event e happened. Thus the consistent global states where HLC can detect if $\wedge pr_q$ is true include $\{\langle a_1, b_1, c_1 \rangle\ \langle a_2, b_2, c_2 \rangle\ \langle a_3, b_3, c_3 \rangle\ \langle a_4, b_4, c_4 \rangle, \langle a_5, b_5, c_5 \rangle, \langle a_6, b_6, c_6 \rangle\ \}$, i.e., only

global states containing events with equal HLC timestamps. Even though there are several other consistent global states, HLC does not have enough information to help the monitor in identifying them. For instance, HLC does not allow us to conclude that a_1, b_2 and c_2 are possibly concurrent events. Now, let us change the implementation of **receive** in Algorithm 5 as follows:

Upon receiving a message m, to create a receive event b,

$l.j = max(l.j + 1, l.m\mathbf{+ 2}, pt.j)$

$l.b = l.j$

Observe that in this implementation, we have a bias for messages. For local and message send events, we still add one to differentiate the new event from the previous event on the process. However, for message receive, we added 2 (instead of 1 as done in Algorithm 5, line 4). This modified implementation of naive Hybrid Logical Clocks will guarantee that if e and f are events on two different processes, then $(e \rightarrow f) \Rightarrow (l.e + 1 < l.f)$. It follows that, if e and f are events on two different processes and if $|l.e - l.f| \leq 1$ then e and f are concurrent events. Recall that if the timestamping was based on the original naive Hybrid Logical Clocks algorithm, then the only guarantee was $(l.e = l.f) \Rightarrow (e||f)$.

From this discussion, it follows that with this new implementation, global states such as $\langle a_1, b_2, c_2 \rangle$ (cf. Figure 4.2) are also identified as consistent global states and the monitor can then evaluate them for predicate satisfaction, i.e., evaluate if the given predicate $\wedge pr_q$ is true in the consistent global state. We view the above implementation as a solution with a bias value of 2. The default implementation in Section 4.1.1 corresponds to a bias of 1. We also denote it as an *unbiased* implementation.

With a bias of 2, in Figure 4.2, we can see that there is an increased potential to find a consistent global state where the given predicate is true. Furthermore, the effectiveness of predicate detection in the scenario considered in Figure 4.2 will increase as the value of bias increases. In particular, if we use a bias of 6 then all possible consistent global states will be identified by the monitor.

We note that the addition of bias is not free of cost. In particular, there is a potential that some consistent global states that were detected with bias of 1 (i.e., with Algorithm 5 in Section 4.1.1) are not detected by an algorithm with a higher bias value. As an illustration, consider the scenario in Figure 4.3. Here, process p_j has received several messages, and others did not receive any messages. In this case, $l.j$ becomes significantly higher than $l.h$. Hence, even with the increased drift permitted between $l.j$ and $l.h$, it is possible that some consistent global state detected by naive HLC will not be detected by an algorithm with a higher bias.

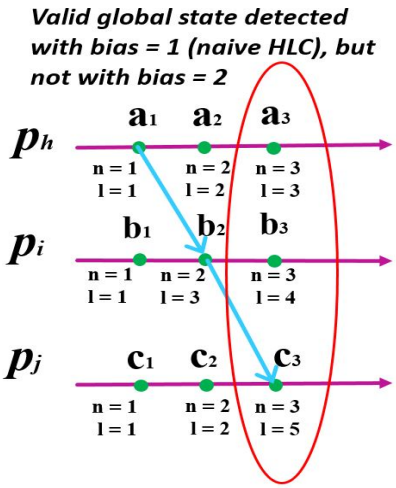


Figure 4.3: A scenario with non-uniform message distribution (n corresponds to the naive HLC value with bias=1, l corresponds to the modified naive HLC value with bias=2).

From the above discussion, we can see that the notion of introducing a bias for messages has the potential to increase the effectiveness of monitoring, without affecting the overall complexity of the detection algorithm. However, there is a potential of missing some consistent global states found by the unbiased algorithm.

4.1.3 Biased Hybrid Logical Clocks Algorithm

Our first algorithm for Biased Hybrid Logical Clocks ($BHLC$) is based on the idea discussed in Section 4.1.2. Specifically, each process p_j maintains a variable $l.j$ to keep track of its (biased) clock value. This value is initialized to 0. We also utilize physical clock $pt.j$ for process p_j. This value is updated automatically. As far as our algorithm is concerned, it is a read-only value.

However, the underlying system will ensure that the clocks of any two processes differ by at most ϵ. As an input, $BHLC$ takes a parameter B, that denotes the bias value.

When a new event occurs at process p_j, $l.j$ is updated. The updated value is assigned as a timestamp to the new event. If the event is a message send event or a local event, then the algorithm works exactly the same as the naive HLC algorithm. It increases the value of $l.j$ by 1 and sets it to a value that is at least as large as $pt.j$. If it is a message send event, then the message is tagged with the updated value of $l.j$. If the event is a message receive event, where message m is received, then $l.j$ is set to $max(l.j\mathbf{+1}, l.m\mathbf{+B}, pt.j)$. In other words, the algorithm biases its clock to be at least $l.m + B$. Thus, the algorithm is as shown in Algorithm 6.

Algorithm 6 Algorithm $BHLC$ with Input Parameter B

At node j
1: Initially $l.j := 0$
2: Set B to the input bias value
Send/Local event
3: $l.j := max(l.j + 1, pt.j)$
4: Timestamp with $l.j$
Receive event of message ***m***
5: $l.j := max(l.j + 1, l.m + B, pt.j)$
6: Timestamp with $l.j$

From this algorithm, we can show that the following properties are satisfied.

Lemma 1. *Let e and f be events on two different processes and let $l.e$ and $l.f$ be the timestamps assigned to them by Algorithm 6. Then, we have*

$$(e \rightarrow f) \quad \Rightarrow \quad (l.e + B \leq l.f)$$

Proof. This proof follows from Line 5 of Algorithm 6. □

Lemma 2. *Let e and f be events on two different processes and let $l.e$ and $l.f$ be the timestamps assigned to them by Algorithm 6. Then, we have*

$$(|l.e - l.f| < B) \quad \Rightarrow (e||f)$$

Proof. We consider two cases: $l.e \geq l.f$ and $l.f \geq l.e$. In the first case, clearly 'e happened before f' is false. Also, since $l.e - l.f < B$, i.e., $l.f + B > l.e$. From Lemma 1, 'f happened before e' is also false. Thus, $e||f$. And, the analysis of the second case is identical.

□

4.2 Extensions of $BHLC$

4.2.1 Extension 1: Multiple Simultaneous Instances of $BHLC$

Algorithm of Biased Hybrid Logical Clocks takes B as a parameter. We can run two versions of this algorithm, say with $B = B_1$ and $B = B_2$. Observe that if any one of them allows the monitor to conclude that two events are concurrent, then they are indeed concurrent. However, if we run two versions of the same algorithm, it would increase the storage and computational cost.

4.2.2 Extension 2: Algorithm $BHLC_r$: Resetting clocks at cut-points

Based on the analysis of the scenario in Figure 4.3, we can see that Biased Hybrid Logical Clocks will work effectively if the length of the computation is small so that the number of messages received by different processes are close. Therefore, we consider two extensions to deal with these issues.

First, to deal with long computations, we introduce the notion of (periodic) cut-points with length C. Thus, the first interval is $\langle 0..C-1\rangle$. The next interval is $\langle C..2C-1\rangle$, and so on. Whenever, the clock of a process reaches its cut-point, we increase its l value to a large enough value that is guaranteed to not occur before that cut-point. This is straightforward to achieve because the computation length between cut-points has a fixed length. Note that this would create a problem in terms of comparing events, when event on one process is just before the cut-point and an event on another process is just after the cut-point. To deal with this issue, we can maintain another clock which resets at $\frac{1}{2}C$, $\frac{3}{2}C$, $\frac{5}{2}C$ and so on. As discussed in Section 4.2.1, even if one of these clocks

allow the monitor to conclude concurrency of two events, then they are indeed concurrent events. The resulting algorithm is referred to as $BHLC_r$.

4.2.3 Extension 3: Algorithm $BHLC_a$: Adjusting message rate

Figure 4.3 also suggests that biased clocks would work most effectively if the number of messages received by each process is roughly the same. If the underlying system does not have such a message distribution, then we can achieve this by allowing a process to pretend to receive *fake messages*. This can be achieved as follows: Let $x(t)$ denote the number of messages that are expected to be received by a process by time t. If the actual number of messages received by a process is smaller, then the process pretends to receive a message (and updates its clock value). The resulting algorithm is denoted as $BHLC_a$. We use $BHLC_{ra}$ to denote the algorithm that uses both extensions 2 and 3.

4.3 Predicate Detection using Biased Hybrid Logical Clocks

In this section, we will discuss the algorithm or approach to perform detection of conjunctive predicates in a system that uses Biased Hybrid Logical Clocks to timestamp its events. Specifically, let us consider that every process p_i in the system has a boolean variable v_i and the goal is to detect if v_i becomes true at all processes simultaneously, $\wedge v_i$ $(1 \leq i \leq n)$. When an event occurs at a process, the process updates its Biased Hybrid Logical Clock value and then timestamps the event with the updated Biased Hybrid Logical Clock value and the current physical clock value. Initially, v_i is false at every process p_i.

Reporting. Let e and f denote (successive) events where v_i becomes true and false respectively. Let $\langle b.e, pt.e\rangle$ denote the value of $BHLC$ and physical clock timestamp of event e. Likewise, let the timestamp of f be $\langle b.f, pt.f\rangle$. Thus, v_i is true in the interval $[\langle b.e, pt.e\rangle, \langle b.f, pt.f\rangle)$. Hence, process p_i creates a candidate $[\langle b.e, pt.e\rangle, \langle b.f, pt.f\rangle)$ and adds it to its queue. The monitoring process uses these queues for the detection of conjunctive predicate i.e., in this instance to detect if the variable v was true at all the processes at a common point in time.

Predicate Detection. The monitor forms a global state of the system by picking one candidate per process. It then evaluates that for any two candidates in the global state, say $[\langle b.e_i, pt.e_i\rangle$, $\langle b.f_i, pt.f_i\rangle)$ and $[\langle b.e_j, pt.e_j\rangle$, $\langle b.f_j, pt.f_j\rangle)$, if there is some point in time within a candidate interval that is possibly concurrent with some point in time within the other candidate interval. This is achieved by checking if $((|b.f_i - b.e_j| < B) \vee (|b.f_j - b.e_i| < B)) \wedge ((|pt.f_i - pt.e_j| \leq \epsilon) \vee (|pt.f_j - pt.e_i| \leq \epsilon))$. If this evaluates to true for every pair of candidates in the global state, then it is a consistent global state where the conjunctive predicate is satisfied. When the monitor detects such a consistent global state, it reports a satisfaction of conjunctive predicate $\wedge v_i$ $(1 \leq i \leq n)$.

4.4 Experimental Setup

In our simulations, we considered a system of 10 independent processes, where each process had a physical clock and a biased clock associated with it. We treated a clock tick to be 0.1ms. Each simulation run was for a total of 100 (virtual) seconds, i.e., each process incremented the physical clock from 0 to $1,000,000$. Every process p_i had a boolean variable v_i associated with it. Whenever v_i is eligible for a change, process p_i changed v_i's value with probability β. Once the value of v_i changes, this value remains unchanged for a minimum length of time $interval$ before becoming eligible to change again. The default set of parameters that we used are clock drift $\epsilon = 10$ms (100 clock ticks), message delay $\delta = 1$ms (10 clock ticks), $\beta = 10\%$ (the expected time before the variable v_i becomes true is 1ms), local predicate (in this case v_i) stays true for just 0.1ms ($interval = 1$ clock tick) and an average communication frequency of 1000 messages per second (10% chance of sending a message every clock tick). To compare the effectiveness of biased clocks under different configurations, we varied one parameter at a time.

Our experiments focused on identifying the number of consistent global states or consistent cuts where the given predicate is true. Subsequently, we determined how many of these consistent global states or consistent cuts are found by HLC/$BHLC$. In this sense, our analysis is independent of the predicate being considered. However, if we change the predicate under consideration, we will

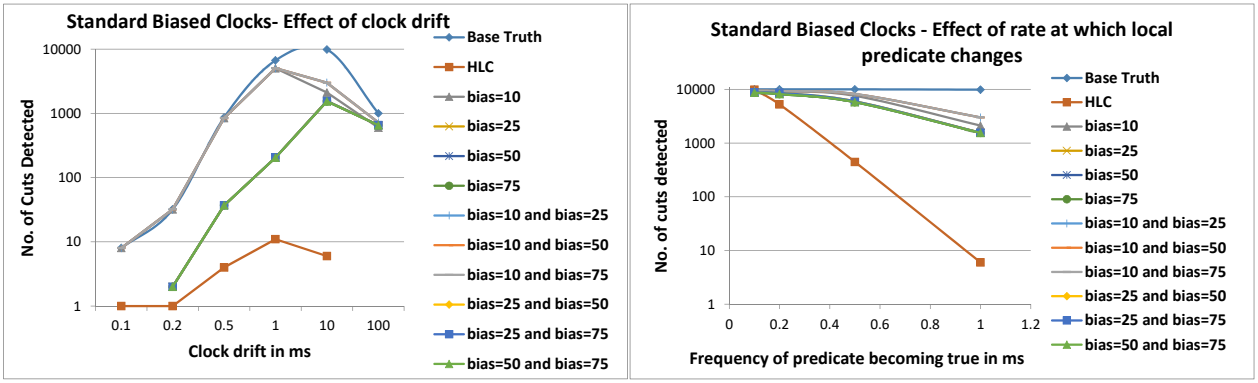
need to change the algorithm used in detecting it. However, the comparison of consistent global states or consistent cuts identified by Hybrid Logical Clocks and Biased Hybrid Logical Clocks will remain unaffected.

4.4.1 Effectiveness of $BHLC$ under different system parameters and Bias B

In this section, we analyze the effect of varying system parameters namely clock drift, communication frequency, message delay, local predicate rate and the amount of bias (B) on the ability of the monitor in detecting satisfaction of predicates using Biased Hybrid Logical Clocks and Hybrid Logical Clocks.

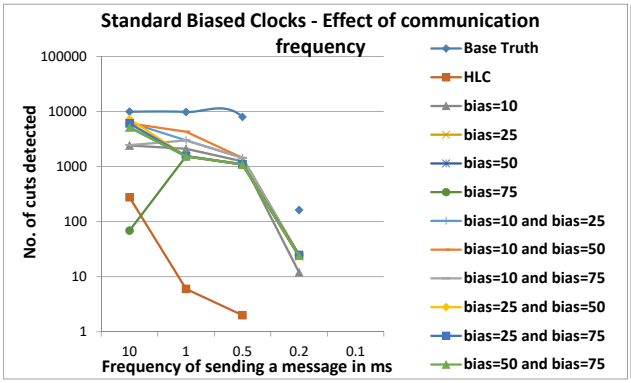
Varying Clock Drift (ϵ). To analyze the effect of varying clock drift or clock synchronization of the processes in the system, we used the default set of system parameters and varied the clock drift ϵ. Specifically, we considered clock drifts of 0.1ms, 0.2ms, 0.5ms, 1ms, 10ms and 100ms. In Figure 4.4a, we observe that the monitor using Biased Hybrid Logical Clocks with any value of B performs **orders of magnitude better** than the monitor using Hybrid Logical Clocks. In terms of clock synchronization, a monitor that uses Biased Hybrid Logical Clocks implementation performs better when the processes in the system are more tightly synchronized with each other. More specifically, when the processes in the system are tightly synchronized with each other, detection using Biased Hybrid Logical Clocks with $B = 10$ detects almost all consistent global states or cuts where the predicate is true.

Varying local predicate rate (β). Starting from the default experimental setup, we considered different rates at which the local predicate becomes true, starting from the case where $\beta = 1$ (means that the local predicate is always true), i.e., in our example, when variable v_i is always true, to $\beta = 0.1$ (probability that v_i is true is 0.1). The observed effect is as shown in Figure 4.4b. As β decreases, the number of consistent global states or cuts satisfying the predicate in the system decrease. As expected, all the methods of detection detect fewer cuts with smaller β. The earlier observation that the monitor using Biased Hybrid Logical Clocks performs better than the monitor using Hybrid Logical Clocks continues to hold for Figure 4.4b. Again, detection using

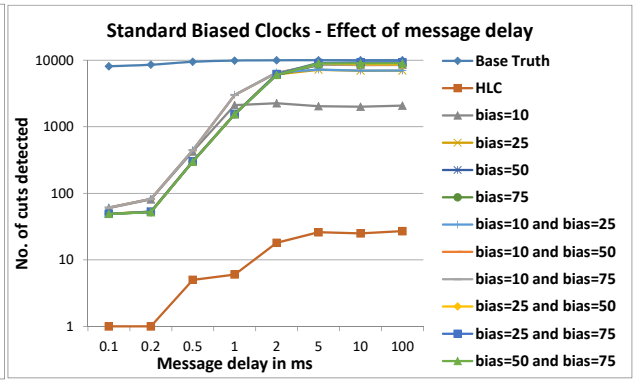


(a)

(b)



(c)



(d)

Figure 4.4: a) Effect of varying clock drift, (b) Effect of varying the rate at which local predicate becomes true, (c) Effect of varying frequency of sending a message, (d) Effect of varying message delay on Standard Biased Hybrid Logical Clocks

Biased Hybrid Logical Clocks with $B = 10$ and a simultaneous implementation with $B = 10$ and $B = 75$ perform the best. More specifically, simultaneous implementation with $B = 10$ and $B = 75$ identifies approximately 70% of the actual cuts.[1]

Varying communication frequency (α). Starting from the default experimental setup, we varied α from 0.1 (roughly 100 messages per second) to 1 (roughly 10000 messages per second). As shown in Figure 4.4c, all detection methods detect fewer consistent global states or cuts as the communication frequency increases; as the number of messages increase, there are fewer concurrent events.

We observe that the monitor using Biased Hybrid Logical Clocks continues to perform better than the monitor using Hybrid Logical Clocks. On an average, the monitor using Biased Hybrid Logical Clocks detects about 50% of the total cuts. In general, predicate detection based on Biased Hybrid Logical Clocks performs better if communication frequency is low. This is expected, given that biased clocks were motivated by what happens when message the communication frequency is very low. Hence, one may consider allowing for higher bias to improve performance. However, very high bias also means longer jumps in Biased Hybrid Logical Clocks. In turn, this may also result in rejection of more consistent global states. From Figure 4.4c, we find that monitor using Biased Hybrid Logical Clocks with $B = 25$ and $B = 50$ works best.

Varying Message Delay (δ). For this case, we considered message delays of 0.1ms, 0.2ms, 0.5ms, 1ms, 2ms, 5ms, 10ms and 100ms. Remaining parameters were the same as in the case of the default setup. From Figure 4.4d, we can observe that with increase in message delay, predicate detection using Biased Hybrid Logical Clocks performs significantly better. When the message delay is small, the monitor using Biased Hybrid Logical Clocks detects less than 10% of the actual consistent cuts in the system. However, as the message delay increases the predicate detection rate of the monitor using Biased Hybrid Logical Clocks improves rapidly to 80%, specifically when message delay $\geq$ 2ms.

[1]Note that for the sake of comparison the analysis was done over the same set of execution traces and when we considered multiple bias amounts, common consistent global states were counted only once.

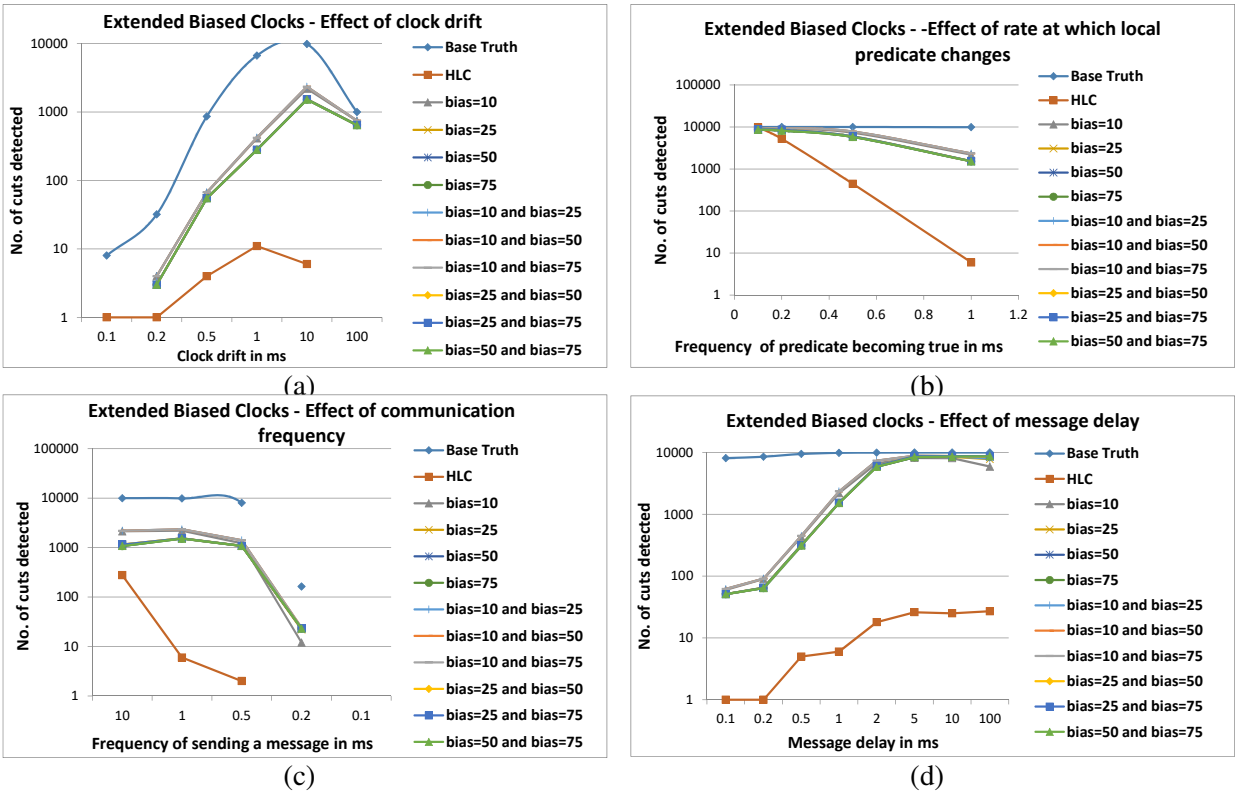
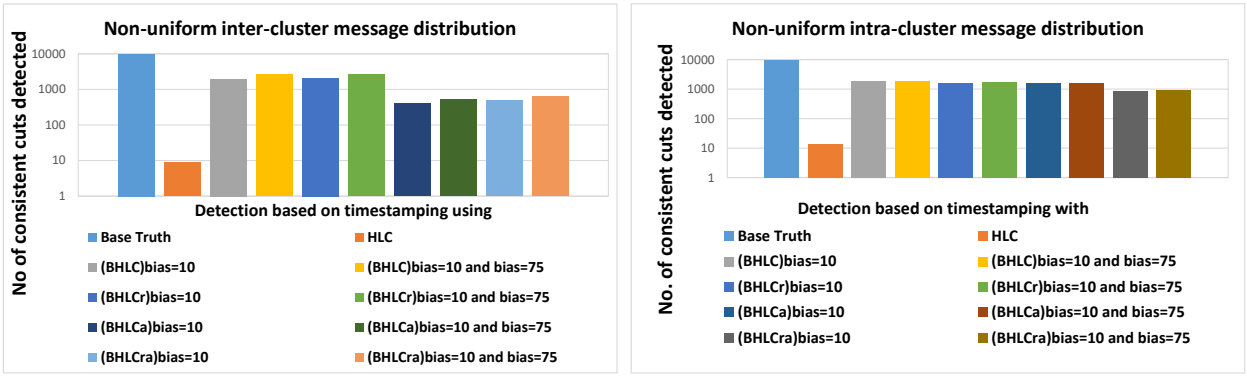


Figure 4.5: (a)Effect of varying clock drift, (c) Effect of varying frequency of sending a message and (b) Effect of varying the rate at which local predicate becomes true and (d) Effect of varying Message delay on Extended biased clocks with reset every 100ms.

4.4.2 Effectiveness of $BHLC_r$

In this section we analyze the effectiveness of $BHLC_r$, where clocks at the processes are reset periodically to overcome the issue of biased clocks growing far apart over time.

We perform the same set of experiments presented in Section 4.4.1 using $BHLC_r$ where clocks are reset every 1000 clock ticks, i.e. every 100 ms. We vary one system parameter at a time and present the results in Figure 4.5. We observe that the detection capability of the monitor using $BHLC_r$ is similar to the detection capability of the monitor using standard Biased Hybrid Logical Clocks.



(a) (b)

Figure 4.6: No. of violations (consistent cuts where the predicate is true) detected in (a)Non-uniform Message distribution 1 and (b)Non-uniform Message distribution 2

4.4.3 Effectiveness of $BHLC$ under Non-uniform Message Distribution

In this section we analyze the effectiveness of $BHLC$ and its extensions in a system with non-uniform message distribution. Specifically, we focus on the extension of $BHLC$ discussed in Section 4.2.3, where processes compensate for non-uniform message distribution.

We consider a scenario where we partition the set of 10 processes into two groups (5 in each). The first group (processes 1-5) receive messages at twice the rate of the second group (processes 6-10). Hence, processes 6-10 compensate by adding twice as much bias in the receive statement. In other words, if we instantiate $BHLC$ with $B = 10$, processes 1-5 add 10 on Line 5 and processes 6-10 add 20. In this scenario, we consider two sub-cases: (1) processes in one group only communicate among themselves (cf. Figure 4.6b) and (2) processes choose their destination randomly and, hence, they could send messages to processes in the other group (cf. Figure 4.6a). For this version ($BHLC_a$), we utilize the following observation to decide if two events are related by happened-before relation.

$$e \longrightarrow f \quad \text{iff} \quad \begin{cases} l.e + 10 < l.f & f \text{ is on processes } 1..5 \\ l.e + 20 < l.f & f \text{ is on processes } 6..10 \end{cases}$$

From Figures 4.6a and 4.6b, we find that $BHLC$, $BHLC_a$ and $BHLC_{ra}$ work better than HLC. However, $BHLC_a$ and $BHLC_r$ do not provide the desired improvement. Rather, (standard) $BHLC$ works better. In part, this happens because $BHLC_a$ does an abrupt jump of size $2B$ for processes

6..10. This abrupt jump makes it harder to find concurrent events at processes 1..5. That said, the addition of bias improves the predicate detection capability when compared with unbiased implementation (which corresponds to HLC).

4.5 Advantages and Limitations of performing Predicate Detection using $BHLC$

Predicate detection using Biased Hybrid Logical Clocks has low overhead like Hybrid Logical Clocks because they are also $O(1)$ sized clocks. Our analysis shows that with biased clocks presented in this chapter, the chances of finding the predicate of interest being true is substantially higher than detection using Hybrid Logical Clocks. On an average, in our experiments, monitors using biased clocks were able to find 100-200 times as many instances where the global predicate is true when compared to monitors using Hybrid Logical Clocks. This result was observed to be true for different communication frequencies, message delays, clock drifts and frequencies of the local predicate being true. Furthermore, for many scenarios, Biased Hybrid Logical Clocks was able to find more than half of the instances where the given predicate was true. Given that Biased Hybrid Logical Clocks are $O(1)$ sized clocks and predicate detection using Biased Hybrid Logical Clocks finds a substantial fraction of instances where the given predicate is true, we expect that Biased Hybrid Logical Clocks will provide an inexpensive and effective way to perform better predicate detection.

Though the number of concurrent events that monitors using Biased Hybrid Logical Clocks miss to identify is substantially smaller than the number of concurrent events that monitors using Hybrid Logical Clocks may miss, monitors using Biased Hybrid Logical Clocks still do not guarantee detection of all possible consistent global states. Therefore predicate detection using Biased Hybrid Logical Clocks can still miss to identify instances of predicate satisfaction. If we want to detect all possible instances where a global predicate is *true* without finding any phantom instances, then $O(n)$ sized clocks are necessary. Therefore in the next chapter an alternative monitoring approach is discussed, where the monitor continues to use Hybrid Logical Clocks based event-timestamps but

achieves guaranteed detection of all instances of predicate satisfaction with the help of Satisfiability Modulo Theories (SMT) Solvers.

CHAPTER 5

RELIABLE PREDICATE DETECTION USING SMT SOLVERS

As discussed in Chapters 3 and 4, though predicate detection using Hybrid Logical Clocks and Biased Hybrid Logical Clocks are efficient in terms of monitoring overhead, they fail to identify all possible instances of predicate satisfaction. Therefore in this chapter a new monitoring approach of performing predicate detection using Hybrid Logical Clocks and Satisfiability Modulo Theories(SMT) solvers is discussed. Specifically, in Section 5.1, a brief discussion about SMT solvers and how SMT solvers can be used to perform predicate detection is presented. In Section 5.1.1, the specific set of information that has to be reported by the processes in the system to the monitor, to aid the predicate detection task is identified. In Section 5.2, the details of how the monitor uses the reported information and provides corresponding inputs to the solver to identify the instances of predicate satisfaction is discussed. The experimental setup and results from evaluating the effectiveness of this approach are presented in Section 5.3. Finally, in Section 5.4, the advantages and disadvantages of performing predicate detection using Hybrid Logical Clocks and SMT solvers are discussed.

5.1 Predicate Detection using SMT Solvers

As discussed in Section 2.3, performing predicate detection to detect all possible instances where a predicate is *true* is critical in identifying latent concurrency bugs in distributed systems. Specifically, the first step in predicate detection i.e., identifying all possible consistent global states is critical in the detection of latent concurrency bugs. This is because latent concurrency bugs do not manifest themselves in all runs of the system, they are visible only if the events across the processes occur in a specific relative order. Also, as discussed in Section 2.3, the total number of possible consistent global states to be evaluated for predicate detection increases exponentially as the number of processes and events per process increase. In this chapter we will discuss how the monitor can use SMT solvers to handle this explosion and guarantee detection of all possible

instances of predicate satisfaction accurately.

Generally, an SMT solver takes a formula and a list of constraints as its input. The solver then identifies whether the formula can be satisfied while simultaneously satisfying all the constraints. If satisfiable, it produces a satisfying variable assignment for the formula. Otherwise, it reports that the formula cannot be satisfied. In our case, we use SMT solvers to perform predicate detection as follows. First, we develop a formula to represent the violation of a safety specification in the system. This formula is developed once for the system. Then during runtime, we propose a lightweight method for the monitor to learn the system events and generate corresponding constraints that define the partial order on the system events (that capture all causal relationships between events) that any valid ordering of events must follow. The SMT solver then determines if there is a valid ordering of events that would lead to violation of the safety specification. The lightweight method for accepting system events and for generating constraints that define the partial order on system events is presented in Section 5.2. On the whole, the goal of the monitor is to help the solver in finding a consistent global state where the predicate of interest $\mathcal{P}$ is satisfied, by providing constraints that help the solver in eliminating invalid choices i.e., eliminate invalid global states or consistent global states where $\mathcal{P}$ does not hold true.

Relying on SMT solvers for runtime monitoring has several advantages. The most important advantage is the correctness. Since an SMT solver evaluates all possible combinations of variables before declaring the formula as unsatisfiable, it guarantees the correctness of the monitor; i.e., it will not miss a violation and it will not identify phantom violations.

5.1.1 Reporting

As mentioned in the beginning of the chapter, the overall approach still uses Hybrid Logical Clocks. Specifically, processes in the system timestamp their events and messages using Hybrid Logical Clocks. The system model is same as the model presented in Section 2.1. For the sake of discussion, once again we consider that every process p_i in the system has a boolean variable v_i and the goal is to detect if v_i becomes true at all processes simultaneously, $\wedge v_i$ $(1 \leq i \leq n)$. Next we describe

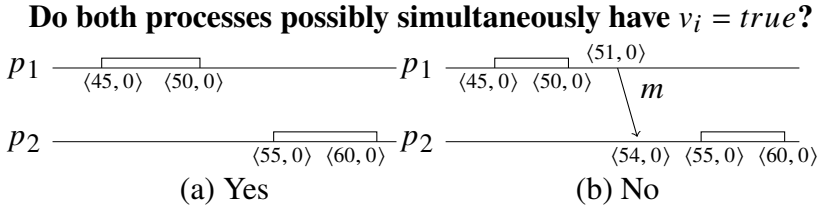


Figure 5.1: Example (a) there are four variable events and no messages. Due to clock drift, it is possible that both processes simultaneously had $v_i = true$ if $\epsilon > 5$. Scenario (b) has the same four variable events plus a message m. Because of the message, the two processes cannot have had $v_i = true$ simultaneously regardless of ϵ.

how each process reports changes in the values of its variables and information about inter-process communication to the monitor.

5.1.2 Reporting a change in variable value

Every time when the value of the variable v_i changes, process p_i updates it HLC value and sends a message with three pieces of information to the monitor: the previous value of v_i, the HLC timestamp of the previous variable update event, and the current HLC timestamp associated with the new variable update event. The two timestamps are sent as an interval that includes the left endpoint but excludes the right endpoint. Note that here the variable v_i corresponds to the local predicate.

We assume that process p_i starts with the Hybrid Logical Clock value of $\langle 0, 0\rangle$ and initially $v_i = False$. The information related to the new variable update event (containing the current value of v_i) will be captured in the next report sent to the monitor when the next variable update event occurs. Providing the previous value of v_i and the associated HLC timestamp allows the monitor to process reports correctly even if they arrive out of order, though out of order reports may delay detection of predicate satisfaction. To illustrate these variable update event reports, consider the run of the program in Figure 5.1 (a) or (b), where the duration for which v_i is true is marked as intervals. Process p_1 sends two variable update-event reports to the monitor. The first report has $v_1 = False$, $[\langle 0, 0\rangle, \langle 45, 0\rangle)$ and is sent at $\langle 45, 0\rangle$. The second report has $v_1 = True$, $[\langle 45, 0\rangle, \langle 50, 0\rangle)$ and is

sent at $\langle 50, 0\rangle$. Likewise process p_2 sends two variable update-event reports to the monitor. The first report has $v_2 = False$, $[\langle 0, 0\rangle, \langle 55, 0\rangle)$ and the second report has $v_2 = True$, $[\langle 55, 0\rangle, \langle 60, 0\rangle)$.

5.1.3 Reporting Message Events

The process that receives a message from another process is responsible for reporting both the send and receive events to the monitor. Specifically, the receiver process reports four things to the monitor: the sender process ID and the HLC timestamp for the send event (information that is included in the message by the sender process before sending the message), the receiver process ID, and the HLC timestamp for the receive event. For example, in Figure 5.1 (b), process p_2 sends a message to the monitor with the sender ID p_1, the send event timestamp $\langle 51, 0\rangle$, the receiver process ID p_2, and the receive event timestamp $\langle 54, 0\rangle$.

5.2 Monitoring setup

Now we will see how the monitor generates corresponding constraints to define the partial order of the events that any event ordering should follow and the formula representing the predicate to be detected, to the SMT solver, to perform predicate detection. The basic setting is that for each process p_i, the monitor uses three variables: v_i, l_i, and c_i that correspond to a variable value and the associated HLC timestamp for that variable value. The formula and the constraints that the monitor creates are satisfiable if there is a way to assign values to all $3n$ variables such that the formula and constraints are simultaneously satisfied. The intuition behind a satisfying variable assignment for v_i, l_i, and c_i is that they should correspond to a consistent global state where the formula/predicate is satisfied. The monitor adds several constraints to ensure that only consistent global states will satisfy the SMT formula, some of which are *static constraints*, that do not depend on the actual run. Others are *dynamic constraints*, that depend on the events and their causal relationship observed during the actual run.

Clock Synchronization Constraints. First the monitor enforces the clock synchronization requirement of a consistent global state, i.e., event/states in a global state should be concurrent with

each other so they should be within $epsilon$ of each other. Specifically, the value of l_i for each process p_i must be at most ϵ apart from the value of l_j of every other process p_j. The monitor enforces this by adding the following static constraint:

$$\forall i, j \quad 1 \leq i < j \leq n : |l_i - l_j| \leq \epsilon$$

Communication Constraints. Next the monitor enforces all communication requirements of a consistent global state that capture causality. Specifically, if process p_i sends a message at time $\langle l_s, c_s \rangle$ to process p_j, which receives the message at time $\langle l_r, c_r \rangle$, then if process p_j's timestamp in the consistent global state is at least $\langle l_r, c_r \rangle$ which means process p_j has received the message, then p_i's timestamp in the consistent global state should be greater than $\langle l_s, c_s \rangle$, which means that p_i has sent the message. This is to ensure that if the message receive event is recorded in the global state, then the corresponding message send event should also be recorded in it. Thus, for each message reported to the monitor, the monitor adds the following constraint:

$$(\langle l_j, c_j \rangle \geq \langle l_r, c_r \rangle) \Rightarrow (\langle l_i, c_i \rangle > \langle l_s, c_s \rangle)$$

These are dynamic constraints as we need one for every inter-process message. Continuing with the example discussed in Figure 5.1 (b), when the monitor receives the details of the message m from the receiver process p_2, it adds the following constraint:

$$(\langle l_2, c_2 \rangle \geq \langle 54, 0 \rangle) \Rightarrow (\langle l_1, c_1 \rangle > \langle 51, 0 \rangle)$$

Variable Update Event Constraints. The monitor also adds constraints to ensure that variable v_i takes on the correct value corresponding to the consistent global state or consistent snapshot. This is ensured by adding one constraint per variable update-event reported to the monitor. Specifically, if process p_i sends a variable update event report $v_i = val$, $[\langle l_1, c_1 \rangle, \langle l_2, c_2 \rangle)$, then the monitor adds the constraint:

$$(\langle l_i, c_i \rangle \geq \langle l_1, c_1 \rangle) \wedge (\langle l_i, c_i \rangle < \langle l_2, c_2 \rangle) \quad \Rightarrow \quad v_i = val$$

Predicate Constraints. Finally, to ensure that the predicate being monitored is satisfied at the consistent snapshot or consistent global state, the predicate is also provided to the solver by the monitor. This is a static formula that depends only on the n v_i variables. For example, if the predicate being monitored requires that all values of v_i are true simultaneously, then it would be

captured by providing $\bigwedge v_i$ as a formula to the solver. If v_i is an integer variable at process p_i and if the goal is to check that the sum of v_i across the n processes is at least 10, then it would be captured by providing $\sum v_i \geq 10$ as a formula to the solver.

5.3 Experimental Results

5.3.1 Experimental Setup

In our simulation, we used a system of 10 independent processes. In one set of experiments, we use a synthetically generated workload, where the process variable v_i changes value randomly; we considered v_i as a boolean variable and as a integer variable. In another set of experiments, we considered the case of an exclusive access to a shared resource in a time division multiplexing protocol based application, that has a timing error, that can potentially cause two processes to simultaneously access the shared resource. We ran our experiments for one second of (virtual) time, where the processes reported events as described in Section 5.1.1. The monitor generated SMT constraints and formula as described in Section 5.2. The SMT solver Z3[16] was then invoked on the SMT constraints/formula. In our simulation, the SMT solver was invoked periodically (period chosen to be 1s). It could also be changed so that it is invoked when a new event occurs (or when a given threshold number of events occur).

The default parameters were a clock tick of 0.01ms, a clock drift ϵ = 10ms (1000 clock ticks), message delay δ = 1ms (100 clock ticks), β = 1% (the expected time before the variable v_i becomes true is 1ms), $interval$ = 0.1ms (variable v_i stays true for 10 clock ticks) and an average communication frequency of 1000 messages per second (1% chance of sending a message every clock tick). Among all the experiments performed, the predicate of interest was satisfiable approximately 70% of the time. Since we avoid generating instances where the satisfaction of the predicate of interest is *too easy*, we do not observe a clear pattern that indicates a correlation between the time taken by Z3 and whether the predicate of interest is satisfiable or not, so we omit discussion of whether the given predicate is satisfiable.

Synthetic workload. In our synthetic workload, the v_i variables were either boolean variables

or integer variables. When they were integer variables, we restricted them to $\{0, 1\}$. In both the cases, whenever v_i is eligible for a change, process p_i changed v_i's value with probability β. Once the value of v_i changes, this value remains unchanged for a minimum length of time $interval$ before becoming eligible to change again. When v_i was a Boolean variable, we considered three different predicates: the conjunctive predicate $\bigwedge v_i$ that requires all v_i variables to be true simultaneously, the exactly 5 predicate $|\{v_i = true\}| = 5$, that requires exactly 5 v_i variables to be true simultaneously, and at least 5 predicate, $|\{v_i = true\}| >= 5$, that requires at least 5 v_i variables to be true simultaneously. When v_i was an integer variable, we considered two predicates $\Sigma v_i = 5$ and $\Sigma v_i \geq 5$ that were equivalent to the exactly 5 and at least 5 boolean predicates, respectively.

Exclusive access workload. We considered an application that uses time division multiplexing protocol, where each process accessed the shared data in its time slot, which had a length of 100ms and that the clock drift was at most 10ms. We assumed that each process started accessing the data at the start of its time slot. To ensure that there is no simultaneous access, each process stopped accessing 10ms before the end of its time slot. For example, process 1 accessed the shared data in the interval $[0ms, 90ms)$, process 2 accessed it in the interval $[100ms, 190ms)$, and so on. We introduced a chance of error where each process held on to its access for an extra 1ms with a probability of 10%, which means that the process p_i and process p_{i+1} might simultaneously access the data. For this experiment, v_i was a boolean variable that indicated when process p_i accessed the shared data, and the predicate to be detected was that whether two v_i variables might be true simultaneously. Next, we will identify how one can interpret the results of our experiments.

5.3.2 Interpreting the Experimental Results

There are two approaches for implementing run-time monitors; a standalone approach where a monitoring process is independent of the application processes, which is how we have described the monitor process so far, and a combined approach where the monitor runs on the same machines as the processes and uses a certain fraction of the resources from those machines. We now describe how to interpret our Z3 timing results using these two perspectives. Recall that we run the process

for one second in all experiments.

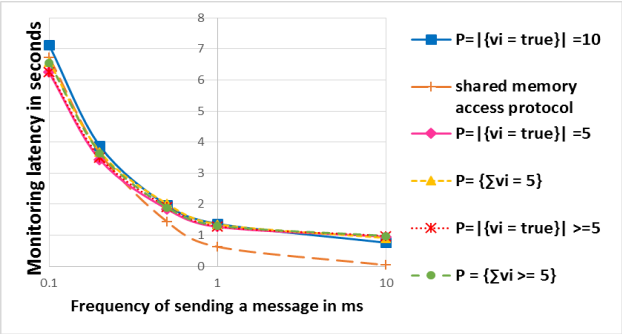
Let us start with the standalone monitor. If the monitoring time is at most one second, a single monitor running on the given environment (Windows 8.1 on 2.19 GHz Intel(R) Core(TM) i5 and 8.00 GB RAM) would be sufficient with a latency of at most $1s$. If the monitoring time is more than one second, say two seconds, then we need two machines and two instances of Z3. If two monitors are used then it could be achieved by sending events at odd time (first, third, fifth second) to the first monitor and sending events at other times to the second monitor. Some overlap may be necessary to ensure that events that span across boundary are recorded correctly. In general, if the Z3 monitoring time (time required for solving the SMT problem) is r seconds, then we need $\lceil r \rceil$ machines and $\lceil r \rceil$ instances of Z3 to keep pace. Note that we can reduce the machine requirements and latency by getting a more efficient machine or finding a more efficient SMT solver.

Let us now consider the combined approach. In this case, if monitoring one second of execution time on 10 processors takes r seconds, then each process would need to devote roughly $r \times 10\%$ of its resources to the monitor to ensure that the monitoring process can keep up with the application. We can view this as either needing a $r \times 10\%$ more efficient machine or that $\frac{r \times 10}{100 + r \times 10}$ of its resources are devoted to monitoring, meaning that the application itself will slow down due to monitoring.

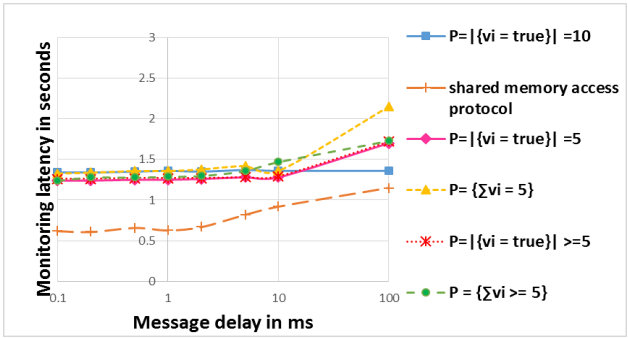
In our experiments we observed that in general for a run that is 1 second long, the monitor took 1 to 2 seconds to perform predicate detection. Therefore to use SMT solvers to perform predicate detection in an online runtime monitoring setup, where the monitor has to catch up with the system being monitored, one may have to choose one of the two implementation techniques discussed above.

5.3.3 Effect of Communication Frequency

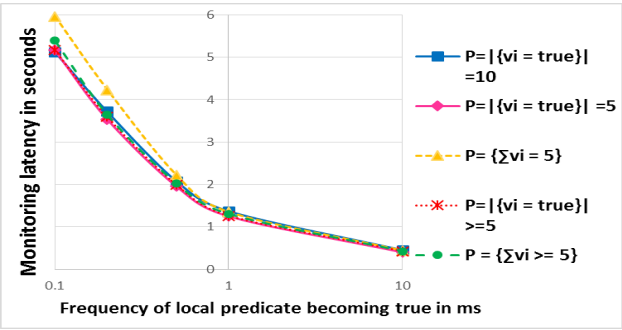
We first discuss how the inter-process communication frequency affects the time required for monitoring. Figure 5.2 (a) summarizes these results. We used our default parameters except that we varied communication frequency from an average of 100 messages per second (0.1% chance of sending a message to another process every clock tick) to an average of 10,000 messages per



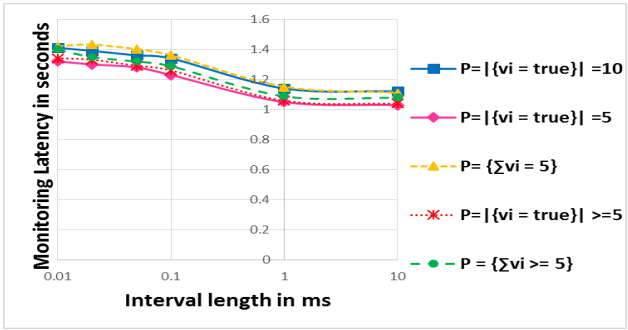
(a) Effect of communication frequency



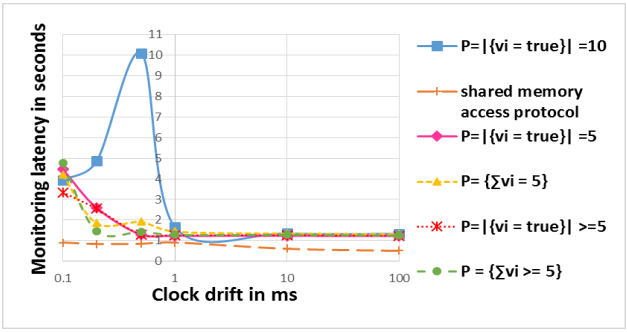
(b) Effect of change in message delay δ



(c) Effect of rate at which the local predicate changes



(d) Effect of duration for which the local predicate stays unchanged



(e) Effect of change in clock drift ϵ

Figure 5.2: Analysis of the role of system parameters on monitoring latency

second (10% chance of sending a message to another process every clock tick). We can observe that as the communication frequency decreases, the time for verification also decreases. This holds for all the predicates that we studied. Also, monitoring the faulty shared memory access protocol required less time than monitoring the synthetic workloads.

5.3.4 Effect of Communication Latency

Next we analyze how inter-process communication latency affects the time required for monitoring. Figure 5.2 (b) summarizes these results. We used our default parameters except that we varied communication latency from 0.1ms to 100ms. We can observe that communication latency has a small effect on the time required for monitoring. For all predicates considered, the monitoring time increases with an increase in communication latency, but by at most half a second even when the latency increases from 0.1ms to 100ms.

5.3.5 Effect of Variable Stability

Next, let us consider how variable stability affects the time required for monitoring. Note that there are two parameters that affect variable stability in the synthetic workload experiments: β which is the probability of changing the variable value at a given time and $interval$ which determines how long the variable value will remain stable after a change. We used our default parameters except that we first varied β from 0.1% (the expected time before the variable becomes true is 10ms) to 10% (the expected time before the variable becomes true is 0.1ms) in one set of experiments and varied $interval$ from 0.01ms (1 clock tick) to $10ms$ (1000 clock ticks) in the other. Figure 5.2 (c) summarizes the results where we varied β and Figure 5.2 (d) summarizes the results where we varied $interval$. We see that more variable stability leads to faster monitoring. As we decrease the probability of changing variable value or increase the stable interval time, Z3 monitoring time drops.

5.3.6 Effect of Clock Drift

Next, we analyze how the clock drift ϵ affects the time required for monitoring. Figure. 5.2 (e) summarizes these results. We used our default parameters except that we varied the clock drift ϵ from 0.1ms to 100ms. We can see that unlike other parameters, clock drift does not have a monotonic affect on monitoring time. For some predicates such as conjunctive predicates, the time for monitoring first increases as ϵ increases and then decreases as ϵ increases further. While we do not know the exact reason for this, we suspect the following is true. We are looking for a valid global snapshot where the given predicate is true, which in some sense requires examining ϵ-length intervals in the execution. The number of ϵ-length windows is inversely proportional to ϵ. The number of events within an ϵ length window and thus the complexity of the window is proportional to ϵ. Thus, there are competing pressures making the exact complexity a complicated function of ϵ.

5.4 Advantages and Limitations of predicate detection using SMT solvers

Unlike the solutions presented in Chapters 3 and 4, using SMT solvers to perform predicate detection has no false negatives (and no false positives). Also, the field of SMT solvers is an active field where new advances result in more efficient solvers. Thus, over time, runtime monitors based on SMT solvers will be able to monitor more complex systems. Furthermore, predicate detection using SMT solvers can be used in an online and offline runtime monitoring setup. However, though runtime monitoring using SMT solvers guarantees detection of all instances of predicate satisfaction, they can take more time for detection when compared to the solutions presented in Chapters 3 and 4. As discussed in Section 2.4, timeliness is an important characteristic for online runtime monitors. Especially in an online runtime monitoring setup, if the solvers are not quick enough in solving the input constraints i.e., in performing predicate detection, then the detection latency gets accumulated over time and can cause the monitor to lag behind the system being monitored, making the overall solution not usable in an online setting. Therefore, in the next chapter a two layered monitoring approach is presented, where the predicate detection approach

using SMT solvers discussed in this chapter is used as the second layer of the monitor. We will see how the time taken by the monitor for detection can be decreased drastically with the help of a first layer, which acts as a filtering layer, that reduces the number of times the solver gets invoked in the second layer during the detection process.

CHAPTER 6

EFFICIENT AND RELIABLE PREDICATE DETECTION USING TWO LAYERED MONITORING

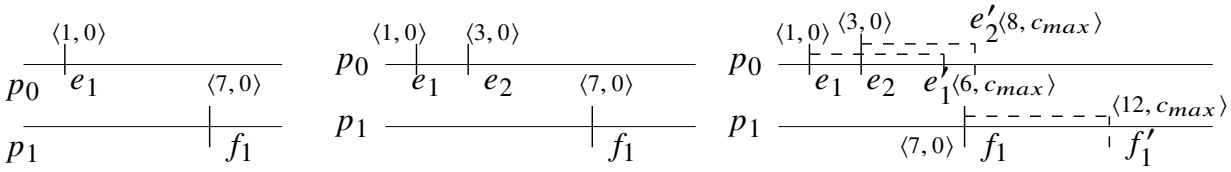
In this chapter, first an approach to modify the HLC based monitoring algorithms from Chapter 3, that have false negatives, into monitoring algorithms that do not have false negatives is discussed in Section 6.1.2. The modification preserves efficiency but introduces potentially many false positives. The extension can handle predicates encountered in practice (conjunctive, arithmetic, violation of mutual exclusion), etc. Section 6.1.3 focuses on how the monitor can be used in scenarios where one would like to reduce the number of false positives, but can afford some false negatives, by modifying a single parameter γ in this extended detection approach. The effectiveness of the monitoring approach is then evaluated with the help of experiments in Section 6.1.4, where the number of false positives/negatives reported by the the monitor when used in detecting conjunctive predicates are analyzed.

In the later half of the chapter, a two-layered monitoring algorithm that combines the algorithm that uses HLC with parameter γ from Section 6.1.3 with the monitoring algorithm from Chapter 5 that uses SMT solvers to perform predicate detection is presented in Section 6.2. This two layered monitoring approach eliminates all false positives and, depending on γ, eliminates many or all false negatives, while performing detection in a timely manner with small monitoring overhead. The effectiveness of the two-layered monitoring approach is evaluated in Section 6.2.1 by computing the time taken to detect violations of mutual exclusion in an application that uses time division multiplexing.

6.1 Predicate Detection with HLC: Trade-off in False Positives and Negatives

6.1.1 False Negatives with HLC

The key property used by the monitors to perform predicate detection using HLC (presented in Chapter 3) is that the state of each process does not change between events. For example, consider



(a) Monitor considers the state at $\langle 1, 0\rangle$

(b) Monitor considers the state at $\langle 3, 0\rangle$ only

(c) With ϵ-extension: Monitor considers the state at $\langle 1, 0\rangle$ and $\langle 3, 0\rangle$

Figure 6.1: Analyzing the state of p_0 considered by the monitor when evaluating the global predicate at $\langle 7, 0\rangle$. ($\epsilon = 5$ in the system) (a) & (b) consider a monitor that uses HLC, (c) considers a monitor that uses HLC with ϵ-extension.

the execution in Figure 6.1(a) and the timestamp $\langle 7, 0\rangle$. The monitor (that uses HLC) needs to evaluate the predicate at both the processes at $\langle 7, 0\rangle$. For process p_1, this is trivial. For process p_0, the monitor evaluates the state at $\langle 7, 0\rangle$ by evaluating p_0's global state at $\langle 1, 0\rangle$; this works because the state of the process does not change without an event and there is no event for p_0 between $\langle 1, 0\rangle$ and $\langle 7, 0\rangle$. Essentially, an HLC based monitor computes the consistent global state or consistent global snapshot corresponding to every event e using the closest preceding event f (i.e., event with the largest timestamp less than or equal to $hlc.e$) on every other process. Stated another way, this HLC monitor does two things when processing an event e_1 on process i: (1) it *ends* the current state of process i that was in effect before $hlc.e_1$ and (2) *introduces* a new state for process i that is in effect from $hlc.e_1$ until the $hlc.e_2$ where e_2 is the next event on process i.

A disadvantage in performing predicate detection using HLC is that it suffers from false negatives. In particular, lets reexamine what happens when it computes the global snapshot for a given timestamp t for an event e on process i. When it considers a process $j \neq i$, an HLC based monitor will only use the state of process j from the most recent preceding event f (i.e., the event with largest HLC timestamp that is less than or equal to $t = hlc.e$). This is not a problem if event f occurs more than ϵ before t, but it is problematic if f occurs within ϵ of t.

For example, consider Figure 6.1(b) where $\epsilon = 5$, with a new event e_2 on p_0 at timestamp $\langle 3, 0\rangle$ which is within ϵ of $\langle 7, 0\rangle$. When the HLC monitor computes the global snapshot at timestamp $\langle 7, 0\rangle$, for process p_0, it will only consider the state of process p_0 after event e_2 even though the

period prior to event e_2 also lies within ϵ of $\langle 7, 0\rangle$ and thus is concurrent with event f_1. To ensure that there are no false negatives, the monitor needs to consider all possible states of p_0 that can be concurrent with t at p_1; namely any state of p_0 within ϵ of $\langle 7, 0\rangle$ should also be considered.

6.1.2 Eliminating False Negatives with ϵ extension

As discussed in Section 6.1.1, HLC based monitor misses consistent snapshots consisting of events e_1 and f_1 where there exists an event e_2 such that $hlc.e_1 < hlc.e_2 < hlc.f_1$ and $l.f_1 - l.e_2 < \epsilon$. To overcome this limitation, we introduce the notion of ϵ-extension, where the effect of event e_1 (in the previous sentence) is extended by ϵ. Specifically, for an event e if $hlc.e$ is $\langle l.e, c.e\rangle$, we define $hlc.e + \epsilon$ to be $\langle l.e + \epsilon, c_{max}\rangle$, where c_{max} is the maximum possible value of c. Extending by ϵ leads to multiple possible values for a process where the extended intervals overlap.

To illustrate this, let x_i be a variable of process i used in predicate $\mathcal{P}$ that is being monitored, and suppose x_i is set to v_0 at timestamp t_0 and then changed to v_1 at timestamp t_1. As we observed earlier, the HLC monitor would use the event at timestamp t_1 to perform two actions: (1) remove v_0 as the value of x_i and (2) add v_1 as the value of x_i. With ϵ-extension, at timestamp t_1, we only perform the second action of adding v_1 as a possible value for x_i. At timestamp $t_1 + \epsilon$, we perform the first action of removing v_0 as a possible value of x_i. This extends the effective interval where x_i has value v_0 from $[t_0, t_1)$ to $[t_0, t_1 + \epsilon)$. Note that we explicitly must consider multiple possible values, in this case v_0 and v_1, for x_i for any snapshot in the time interval $[t_1, t_1 + \epsilon)$. Stated more generally, with ϵ-extension, we treat each event e as two events, e and e', where event e has its original timestamp t and event e' has timestamp $t + \epsilon$. At event e, we perform the second action where we add a new possible value for a variable. At event e', we perform the first action where we remove a possible value for a variable.

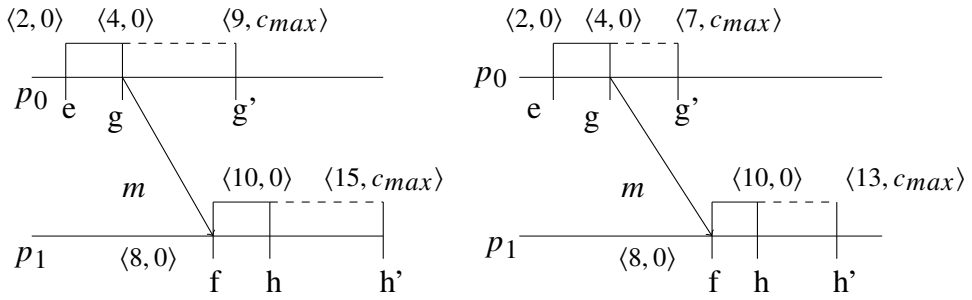
As an illustration, consider the execution in Figure 6.1, where we have events e_1, f_1 and e_2 as shown. Effect of event e_1 would be in the interval $[\langle 1, 0\rangle, \langle 3, 0\rangle)$ (c.f. Figure 6.1b). The ϵ-extension (c.f. Figure 6.1c) will change it to interval $[\langle 1, 0\rangle, \langle 8, c_{max}\rangle)$. The effect of event e_2 is in the interval $[\langle 3, 0\rangle, \langle \infty, -\rangle)$. Thus, in the interval between $\langle 3, 0\rangle$ and $\langle 8, c_{max}\rangle$, we consider

both states of p_0 corresponding to events e_1 and e_2.

While having to consider multiple values will increase complexity and decrease efficiency, we expect that the number of values are likely to be small given that we are only extending effective intervals by ϵ. For example, in the above scenario, multiple values of x_i are only considered in the interval $[t_1, t_1 + \epsilon)$. Also, it is expected that such multiple values would need to be considered only for a small subset of processes. For example, not all processes would have multiple permitted values in the interval $[t_1, t_1 + \epsilon)$.

Furthermore, for many types of predicates, it is possible to eliminate multiple values altogether. For example, consider the conjunctive predicate $\mathcal{P} = \bigwedge \mathcal{P}_i$. Clearly, for any interval where $\mathcal{P}_i$ can be both true and false, we can just record that $\mathcal{P}_i$ is true. For example, if $\mathcal{P}_i$ is set to true at t_0 and then set to false at t_1, then in the interval $[t_0, t_1 + \epsilon)$, we would only record that $\mathcal{P}_i$ is true and only begin recording that $\mathcal{P}_i$ is false at timestamp $t_1 + \epsilon$. Beyond conjunctive predicates, this also applies to predicates of the form $\Sigma x_i > C$ and $\Sigma x_i < C$. In the first case, we choose the maximum value of x_i; in the second case, we choose the minimum value of x_i when multiple values are possible. For predicates of the form $x_1 < x_2$, we can choose the largest value of x_1 and smallest value of x_2. The first case $\Sigma x_i > C$ includes violation of mutual exclusion, where $x_i = 1$ means process i is accessing the shared resource and we set $C = 1$.

Observe that the ϵ-extension does not account for messages. Therefore, it will lead to false positives. Furthermore, instead of detecting the given predicate $\mathcal{P}$, if we detect a slightly weaker predicate, it will increase the rate of false positives. Allowing false positives will enable us to detect more complex predicates. For example, if the predicate we want to detect is $C_1 < \Sigma x_i < C_2$, we can split this predicate into two separate predicates, one for the upper bound and one for the lower bound, both of which we can detect efficiently. We may increase the false positive rate with respect to the original predicate if there are no choices of x_i that simultaneously satisfy both bounds. We discuss in Section 6.1.5 how the false positive rate can be effectively managed.



(a) With ϵ-extension ($\epsilon = 5$) (b) With γ-extension ($\gamma = 3$)

Figure 6.2: Reducing false positives with γ-extension

6.1.3 Reducing False Positives with γ-extension

The ϵ-extension eliminates false negatives at the cost of introducing false positives. In particular, the ϵ-extension allows the state of a process to be extended by ϵ into the future even if causality shows that such an extension is impossible. As an illustration, in Figure 6.2(a), where the maximum clock drift is $\epsilon = 5$, event e happened before event g which happened before event f. However, the approach of ϵ-extension would incorrectly allow the state of p_0 at $\langle 2, 0\rangle$ to be considered by the monitor at timestamp $\langle 8, 0\rangle$. In other words, the monitor considers e and f to be concurrent even though they are not, resulting in a false positive.

If the intervals in Figure 6.2(a) were extended for a shorter duration, say $\gamma < \epsilon$, then the rate of false positives will decrease at the cost of reintroducing false negatives. For example, for $\gamma = 3$, the state of p_0 at $\langle 2, 0\rangle$ will never be considered at timestamp $\langle 8, 0\rangle$ (c.f. Figure 6.2(b)). Thus, it will not be part of the false positives. However, for $\gamma = 3$, in the absence of communication (even when $e \nrightarrow f$), the snapshot consisting of e and f will be discarded, thereby resulting in a false negative.

The parameter γ provides a mechanism to control the rate of false positives and false negatives. In the next section, we analyze the false positives/negatives for different values of γ and other system parameters. Specifically, we evaluate the effectiveness of performing predicate detection using γ-extension through experimental analysis; that is, we compute the precision (1−false positive rate) and recall (1 − false negative rate) of a monitor that uses γ-extension to perform conjunctive predicate detection.

6.1.4 Analyzing False Positives and Negatives with γ extension

6.1.4.1 Experimental Setup

To analyze the effectiveness of using HLC with γ-extension in performing predicate detection, we simulate a distributed system of 10 processes. Although this analysis is for the case of conjunctive predicates, we note that the approach is general enough to be applied for other predicates. We use conjunctive predicates because evaluating the ground truth (i.e., identifying all valid snapshots in the system) is feasible using [20]. And, this ground truth is essential to compute the number of false positives and negatives reported by the monitor.

In this work, each process has a physical clock pt and a hybrid logical clock associated with it. Each process i is also associated with a boolean variable v_i. The processes execute in a round robin fashion and each process executes a million times. Each time a process executes it advances its physical clock with a certain probability such that the physical clock value of any two processes differ by at most ϵ. When a process advances its clock, it sends a message with a probability α to a uniformly randomly chosen process. Based on the value of δ-message delay and time at which a message was sent, the process receives any message that was sent to it if it is ready to be delivered. If the value of the variable v_i is false the process sets its value to true with a probability β. If v_i is true, it remains true for a duration of length ℓ (counted in terms of physical clock of i). Then, it is set to false. The process updates its hybrid logical clock value every time it sends or receives a message, and whenever it changes the value of v_i based on the HLC algorithm (Algorithm 1) presented in Section 2.2.

Each process i reports every duration for which v_i was true as an interval to a common monitoring process. Each interval consists of the HLC timestamp when v_i became true and the HLC timestamp when v_i became false after that. The monitor applies γ-extension by adding γ to the interval-end timestamp (of every interval) since it is always advantageous to choose v_i to be true whenever there is a choice. The monitor then reports all snapshots where $\bigwedge v_i$ becomes true (i.e., when γ extended intervals of all processes overlap). The monitor can use any algorithm discussed in Chapter 3 to

compute the overlap after the γ-extension.

We extend the predicate detection algorithm in [20] to identify the ground truth with two modifications. First, we replace Vector Clocks with Hybrid Vector Clocks(HVC)[46] to account for clock synchronization. Second, we modified the algorithm so that it continues the detection process until all valid snapshots are identified. Specifically, after finding a valid snapshot, we look for the next non-overlapping valid snapshot.

We compute the number of true positives, false positives, and false negatives by comparing the results returned by both monitoring solutions. For a valid snapshot identified by the modified HVC based algorithm if the monitor using HLC with γ-extension is unable to detect that snapshot (or another overlapping snapshot), then it is a false negative (i.e., the monitor using HLC with γ-extension missed to identify this snapshot). If the monitor using HLC with γ-extension finds a snapshot but it (or another overlapping snapshot) is not reported by the HVC based algorithm then it is a false positive. And, if the snapshot is reported by both then it is a true positive.

6.1.4.2 Observation

To analyze the effectiveness of γ-extension, we computed the true positives, false positives and false negatives reported by the monitor under different system settings as we varied γ used by the monitor. Our precision and recall results are displayed in Tables 6.1[1] and 6.2, respectively.

Precision of the monitor is computed as the ratio of the number of *valid* snapshots (i.e., consistent global states where $\mathcal{P}$ is true) detected by the monitor to the total number of snapshots reported by it. In the table we provide the actual ratio i.e., number of valid snapshots divided by the total number of snapshots reported in brackets, right next to the precision. **Recall** of the monitor is computed as the ratio of the number of valid snapshots detected by the monitor to the number of actual valid snapshots in the system. In the table we provide the actual ratio i.e. number of valid snapshots detected divided by the number of actual snapshots in the system in brackets, right

[1]We report the precision as NA when the monitor does not detect any snapshots and therefore detects no valid snapshots and the precision would be $0/0$. This only happens for small γ (for example $\beta = 0.02, \gamma = 0.1 * \epsilon$).

next to the recall. We computed the precision and recall of the monitor under different system parameters namely β - the rate at which the local predicate becomes true at a process, α - rate at which a process sends a message, ϵ - clock drift, δ - message delay, ℓ - duration for which the local predicate remains true at a process and n - number of processes.

Effect of β, ℓ and n. We observe that precision and recall increase with increase in β and ℓ, because as β or ℓ increase they increase the probability of the local predicate being true in a consistent snapshot. We also observe that for a smaller n (c.f. Tables 6.1f and 6.1g) the monitor has a better precision and recall than for a higher n, because for an increased number of processes the probability of a conjunctive predicate ($\bigwedge \mathcal{P}_i$) being true in a snapshot decreases.

Effect of α and δ. We observe that the precision decreases with increase in α and decrease in δ. This observation is compatible with our discussion in Section 6.1.3. Specifically, in Section 6.1.3, we considered the scenario where a snapshot or global state that is not a consistent due to inter-process communication (i.e., due to causality between the events in the snapshot) is incorrectly declared as a consistent snapshot/consistent global state by the monitor using γ-extension. As α increases - communication increases and as δ decreases - messages are not delayed (causality has immediate effect), therefore the number of false positives increase and precision of the monitor decreases. On the other hand, recall increases with increase in α and decrease in δ. This is because in general an increase in α and decrease in δ reduces the chance of a snapshot (or global state) being a consistent snapshot (consistent global state), thereby reducing the number of actual valid snapshots in the system. A decrease in the number of actual valid snapshots in the system (denominator in recall) increases the recall.

No. of Valid Snapshots/Total Snapshots Detected (Default values: n = 10, ϵ = 100, α = 0.1, δ = 10, ℓ = 1, β = 0.02)

Precision	β = 0.02	β = 0.025	β = 0.03	β = 0.035	β = 0.04	β = 0.045
γ = 0.10 * ϵ	NA	NA	0.333 (1/3)	1.000 (4/4)	0.739 (17/23)	0.929 (26/28)
γ = 0.15 * ϵ	0.500 (1/2)	0.000 (0/3)	0.529 (9/17)	0.650 (13/20)	0.797 (102/128)	0.743 (113/152)
γ = 0.20 * ϵ	0.500 (2/4)	0.286 (2/7)	0.511 (46/90)	0.582 (46/79)	0.732 (341/466)	0.742 (339/457)
γ = 0.25 * ϵ	0.200 (3/15)	0.190 (4/21)	0.477 (126/264)	0.466 (110/236)	0.688 (778/1131)	0.706 (771/1092)
γ = 0.50 * ϵ	0.077 (44/572)	0.070 (42/604)	0.192 (699/3635)	0.192 (722/3756)	0.383 (2756/7188)	0.384 (2780/7246)
γ = 0.75 * ϵ	0.025 (71/2827)	0.024 (70/2931)	0.107 (809/7575)	0.109 (844/7721)	0.304 (2877/9473)	0.305 (2905/9518)
γ = ϵ	0.013 (71/5651)	0.013 (71/5665)	0.088 (813/9208)	0.092 (850/9252)	0.292 (2879/9855)	0.295 (2907/9859)

(a) Varying β - rate at which the local predicate becomes true at a process

Precision	α = 0.01	α = 0.1	α = 0.2
γ = 0.10 * ϵ	1.000 (9/9)	NA	NA
γ = 0.15 * ϵ	0.913 (21/23)	0.500 (1/2)	NA
γ = 0.20 * ϵ	0.911 (51/56)	0.500 (2/4)	NA
γ = 0.25 * ϵ	0.901 (136/151)	0.200 (3/15)	0.000 (0/1)
γ = 0.50 * ϵ	0.830 (1373/1655)	0.077 (44/572)	0.006 (1/164)
γ = 0.75 * ϵ	0.688 (3348/4867)	0.025 (71/2827)	0.001 (1/1131)
γ = ϵ	0.541 (4141/7657)	0.013 (71/5651)	0.000 (1/3084)

(b) Varying α - rate at which processes send messages

Precision	ϵ = 100	ϵ = 1000
γ = 0.10 * ϵ	NA	0.077 (76/993)
γ = 0.15 * ϵ	0.500 (1/2)	0.076 (76/1000)
γ = 0.20 * ϵ	0.500 (2/4)	0.076 (76/1000)
γ = 0.25 * ϵ	0.200 (3/5)	0.076 (76/1000)
γ = 0.50 * ϵ	0.077 (44/572)	0.076 (76/1000)
γ = 0.75 * ϵ	0.025 (71/2827)	0.076 (76/1000)
γ = ϵ	0.013 (71/5651)	0.076 (76/999)

(c) Varying ϵ-clock drift

Precision	δ = 1	δ = 5	δ = 10	δ = 50	δ = 100	δ = 1000
γ = 0.10 * ϵ	NA	NA	NA	NA	NA	NA
γ = 0.15 * ϵ	0.000 (0/5)	0.500 (1/2)	0.500 (1/2)	1.000 (1/1)	NA	1.000 (1/1)
γ = 0.20 * ϵ	0.000 (0/10)	0.286 (2/7)	0.500 (2/4)	1.000 (5/5)	NA	1.000 (4/4)
γ = 0.25 * ϵ	0.054 (2/37)	0.111 (3/27)	0.200 (3/15)	0.800 (8/10)	0.909 (10/11)	1.000 (8/8)
γ = 0.50 * ϵ	0.010 (7/678)	0.042 (25/602)	0.077 (44/572)	0.752 (415/552)	0.846 (452/534)	0.871 (452/519)
γ = 0.75 * ϵ	0.002 (7/2911)	0.011 (31/2864)	0.025 (71/2827)	0.565 (1535/2717)	0.800 (2176/2719)	0.794 (2144/2701)
γ = ϵ	0.001 (8/5768)	0.006 (31/5578)	0.013 (71/5651)	0.382 (2116/5536)	0.679 (3737/5502)	0.693 (3759/5426)

(d) Varying δ - message delay

Precision	ℓ = 1	ℓ = 10	ℓ = 20	ℓ = 50	ℓ = 100	ℓ = 1000
γ = 0.10 * ϵ	NA	0.500 (1/2)	0.923 (12/13)	0.859 (140/163)	0.887 (524/591)	0.908 (1337/1472)
γ = 0.15 * ϵ	0.500 (1/2)	0.600 (6/10)	0.765 (26/34)	0.801 (233/291)	0.860 (762/886)	0.892 (1653/1854)
γ = 0.20 * ϵ	0.500 (2/4)	0.576 (19/33)	0.675 (54/80)	0.756 (356/471)	0.829 (1035/1249)	0.867 (1984/2289)
γ = 0.25 * ϵ	0.200 (3/15)	0.476 (40/84)	0.570 (102/179)	0.704 (533/757)	0.776 (1295/1669)	0.832 (2309/2775)
γ = 0.50 * ϵ	0.077 (44/572)	0.183 (191/1041)	0.275 (434/1581)	0.421 (1325/3150)	0.546 (2419/4430)	0.629 (3451/5483)
γ = 0.75 * ϵ	0.025 (71/2827)	0.070 (261/3744)	0.126 (551/4373)	0.262 (1555/5934)	0.388 (2664/6868)	0.492 (3701/7525)
γ = ϵ	0.013 (71/5651)	0.041 (264/6368)	0.082 (566/6898)	0.201 (1568/7812)	0.320 (2687/8394)	0.431 (3723/8639)

(e) Varying ℓ- duration for which the local predicate remains true

No. of Valid Snapshots/Total Snapshots Detected (Default values: n = 5, ϵ = 100, α = 0.1, δ = 10, ℓ = 20, β = 0.004)

Precision	β = 0.02	β = 0.01	β = 0.005	β = 0.004	β = 0.002
γ = 0.10 * ϵ	0.953 (784/823)	0.915 (483/528)	0.700 (7/10)	1.000 (2/2)	NA
γ = 0.15 * ϵ	0.937 (1145/1222)	0.904 (768/850)	0.667 (12/18)	0.667 (2/3)	NA
γ = 0.20 * ϵ	0.913 (1548/1696)	0.869 (1059/1218)	0.710 (22/31)	0.667 (2/3)	NA
γ = 0.25 * ϵ	0.893 (1977/2214)	0.847 (1409/1663)	0.596 (28/47)	0.600 (3/5)	NA
γ = 0.50 * ϵ	0.751 (3856/5133)	0.666 (3011/4518)	0.376 (86/229)	0.286 (12/42)	0.000 (0/2)
γ = 0.75 * ϵ	0.628 (4532/7222)	0.535 (3634/6789)	0.182 (141/776)	0.135 (26/192)	0.333 (1/3)
γ = ϵ	0.556 (4681/8426)	0.462 (3756/8123)	0.104 (184/1766)	0.067 (38/566)	0.071 (1/14)

(f) Varying β - rate at which the local predicate becomes true at a process

Precision	α = 0.01	α = 0.025	α = 0.05
γ = 0.10 * ϵ	1.000 (2/2)	1.000 (2/2)	0.667 (2/3)
γ = 0.15 * ϵ	1.000 (3/3)	1.000 (3/3)	0.667 (2/3)
γ = 0.20 * ϵ	1.000 (3/3)	1.000 (3/3)	0.667 (2/3)
γ = 0.25 * ϵ	1.000 (6/6)	0.833 (5/6)	0.667 (4/6)
γ = 0.50 * ϵ	0.850 (34/40)	0.590 (23/39)	0.390 (16/41)
γ = 0.75 * ϵ	0.631 (113/179)	0.425 (74/174)	0.271 (49/181)
γ = ϵ	0.493 (276/560)	0.310 (174/562)	0.157 (90/572)

(g) Varying α - rate at which processes send messages

Table 6.1: Percentage of Valid Snapshots out of all snapshots detected during Conjunctive Predicate Detection using γ-extension. Each entry corresponds to precision (No. of Valid Snapshots detected/Total Snapshots Detected).

No. of Valid Snapshots detected/No. of Valid Snapshots in the system(Default values: n = 10, ϵ = 100, α = 0.1, δ = 10, ℓ = 1, β = 0.02)

Recall	β = 0.02	β = 0.025	β = 0.03	β = 0.035	β = 0.04	β = 0.045
γ = 0.10 * ϵ	0.000 (0/71)	0.000 (0/71)	0.001 (1/813)	0.005 (4/850)	0.006 (17/2879)	0.009 (26/2907)
γ = 0.15 * ϵ	0.014 (1/71)	0.000 (0/71)	0.011 (9/813)	0.015 (13/850)	0.035 (102/2879)	0.039 (113/2907)
γ = 0.20 * ϵ	0.028 (2/71)	0.028 (2/71)	0.057 (46/813)	0.054 (46/850)	0.118 (341/2879)	0.117 (339/2907)
γ = 0.25 * ϵ	0.042 (3/71)	0.056 (4/71)	0.155 (126/813)	0.129 (110/850)	0.270 (778/2879)	0.265 (771/2907)
γ = 0.50 * ϵ	0.620 (44/71)	0.592 (42/71)	0.860 (699/813)	0.849 (722/850)	0.957 (2756/2879)	0.956 (2780/2907)
γ = 0.75 * ϵ	1.000 (71/71)	0.986 (70/71)	0.995 (809/813)	0.993 (844/850)	0.999 (2877/2879)	0.999 (2905/2907)
γ = ϵ	1.000 (71/71)	1.000 (71/71)	1.000 (813/813)	1.000 (850/850)	1.000 (2879/2879)	1.000 (2907/2907)

(a) Varying β - rate at which the local predicate becomes true at a process

Recall	α = 0.01	α = 0.1	α = 0.2
γ = 0.10 * ϵ	0.002 (9/4141)	0.000 (0/71)	0.000 (0/1)
γ = 0.15 * ϵ	0.005 (21/4141)	0.014 (1/71)	0.000 (0/1)
γ = 0.20 * ϵ	0.012 (51/4141)	0.028 (2/71)	0.000 (0/1)
γ = 0.25 * ϵ	0.033 (136/4141)	0.042 (3/71)	0.000 (0/1)
γ = 0.50 * ϵ	0.332 (1373/4141)	0.620 (44/71)	1.000 (1/1)
γ = 0.75 * ϵ	0.809 (3348/4141)	1.000 (71/71)	1.000 (1/1)
γ = ϵ	1.000 (4141/4141)	1.000 (71/71)	1.000 (1/1)

(b) Varying α - rate at which processes send messages

Recall	ϵ = 100	ϵ = 1000
γ = 0.10 * ϵ	0.000 (0/71)	1.000 (76/76)
γ = 0.15 * ϵ	0.014 (1/71)	1.000 (76/76)
γ = 0.20 * ϵ	0.028 (2/71)	1.000 (76/76)
γ = 0.25 * ϵ	0.042 (3/71)	1.000 (76/76)
γ = 0.50 * ϵ	0.620 (44/71)	1.000 (76/76)
γ = 0.75 * ϵ	1.000 (71/71)	1.000 (76/76)
γ = ϵ	1.000 (71/71)	1.000 (76/76)

(c) Varying ϵ-clock drift

Recall	δ = 1	δ = 5	δ = 10	δ = 50	δ = 100	δ = 1000
γ = 0.10 * ϵ	0.000 (0/8)	0.000 (0/31)	0.000 (0/71)	0.000 (0/2116)	0.000 (0/3737)	0.000 (0/3759)
γ = 0.15 * ϵ	0.000 (0/8)	0.032 (1/31)	0.014 (1/71)	0.000 (1/2116)	0.000 (0/3737)	0.000 (1/3759)
γ = 0.20 * ϵ	0.000 (0/8)	0.065 (2/31)	0.028 (2/71)	0.002 (5/2116)	0.000 (0/3737)	0.001 (4/3759)
γ = 0.25 * ϵ	0.250 (2/8)	0.097 (3/31)	0.042 (3/71)	0.004 (8/2116)	0.003 (10/3737)	0.002 (8/2759)
γ = 0.50 * ϵ	0.875 (7/8)	0.806 (25/31)	0.620 (44/71)	0.196 (415/2116)	0.121 (452/3737)	0.120 (452/3759)
γ = 0.75 * ϵ	0.875 (7/8)	1.000 (31/31)	1.000 (71/71)	0.725 (1535/2116)	0.582 (2176/3737)	0.570 (2144/3759)
γ = ϵ	1.000 (8/8)	1.000 (31/31)	1.000 (71/71)	1.000 (2116/2116)	1.000 (3737/3737)	1.000 (3759/3759)

(d) Varying δ - message delay

Recall	ℓ = 1	ℓ = 10	ℓ = 20	ℓ = 50	ℓ = 100	ℓ = 1000
γ = 0.10 * ϵ	0.000 (0/71)	0.004 (1/264)	0.021 (12/566)	0.089 (140/1568)	0.195 (524/2687)	0.359 (1337/3723)
γ = 0.15 * ϵ	0.014 (1/71)	0.023 (6/264)	0.046 (26/566)	0.149 (233/1568)	0.284 (762/2687)	0.444 (1653/3723)
γ = 0.20 * ϵ	0.028 (2/71)	0.072 (19/264)	0.095 (54/566)	0.227 (356/1568)	0.385 (1035/2687)	0.533 (1984/3723)
γ = 0.25 * ϵ	0.042 (3/71)	0.152 (40/264)	0.180 (102/566)	0.340 (533/1568)	0.482 (1295/2687)	0.620 (2309/3723)
γ = 0.50 * ϵ	0.620 (44/71)	0.723 (191/264)	0.767 (434/566)	0.845 (1325/1568)	0.900 (2419/2687)	0.927 (3451/3723)
γ = 0.75 * ϵ	1.000 (71/71)	0.989 (261/264)	0.973 (551/566)	0.992 (1555/1568)	0.991 (2664/2687)	0.994 (3701/3723)
γ = ϵ	1.000 (71/71)	1.000 (264/264)	1.000 (566/566)	1.000 (1568/1568)	1.000 (2687/2687)	1.000 (3723/3723)

(e) Varying ℓ- duration for which the local predicate remains true

No. of Valid Snapshots detected/No. of Valid Snapshots in the system(Default values: n = 5, ϵ = 100, α = 0.1, δ = 10, ℓ = 20, β=0.004)

Recall	β = 0.02	β = 0.01	β = 0.005	β = 0.004	β = 0.002
γ = 0.10 * ϵ	0.167 (784/4681)	0.129 (483/3756)	0.038 (7/184)	0.053 (2/38)	NA
γ = 0.15 * ϵ	0.245 (1145/4681)	0.204 (768/3756)	0.065 (12/184)	0.053 (2/38)	NA
γ = 0.20 * ϵ	0.331 (1548/4681)	0.282 (1059/3756)	0.120 (22/184)	0.053 (2/38)	NA
γ = 0.25 * ϵ	0.422 (1977/4681)	0.375 (1409/3756)	0.152 (28/184)	0.079 (3/38)	NA
γ = 0.50 * ϵ	0.824 (3856/4681)	0.802 (3011/3756)	0.467 (86/184)	0.316 (12/38)	NA
γ = 0.75 * ϵ	0.968 (4532/4681)	0.968 (3634/3756)	0.766 (141/184)	0.684 (26/38)	1.000 (1/1)
γ = ϵ	1.000 (4681/4681)	1.000 (3756/3756)	1.000 (184/184)	1.000 (38/38)	1.000 (1/1)

(f) Varying β - rate at which the local predicate becomes true at a process

Recall	α = 0.01	α = 0.025	α = 0.05
γ = 0.10 * ϵ	0.007 (2/276)	0.011 (2/174)	0.022 (2/90)
γ = 0.15 * ϵ	0.011 (3/276)	0.017 (3/174)	0.022 (2/90)
γ = 0.20 * ϵ	0.011 (3/276)	0.017 (3/174)	0.022 (2/90)
γ = 0.25 * ϵ	0.022 (6/276)	0.029 (5/174)	0.044 (4/90)
γ = 0.50 * ϵ	0.123 (34/276)	0.132 (23/174)	0.178 (16/90)
γ = 0.75 * ϵ	0.409 (113/276)	0.425 (74/174)	0.544 (49/90)
γ = ϵ	1.000 (276/276)	1.000 (174/174)	1.000 (90/90)

(g) Varying α - rate at which processes send messages

Table 6.2: Percentage of Valid Snapshots detected during Conjunctive Predicate Detection using γ-extension out of all Valid Snapshots in the system. Each entry corresponds to recall (No. of Valid Snapshots detected/No. of Valid Snapshots in the system)

Effect of γ. We observe that irrespective of the underlying system setting the precision of the monitor decreases and the recall increases as the value of γ increases. For example in Table 6.1a, consider the case where $\beta = 0.045$, as the value of γ increases from $0.1 * \epsilon$ to ϵ, the precision of the monitor drops from 0.929 to 0.295.On the other hand, in Table 6.2a, for the case $\beta = 0.045$, as the value of γ increases from $0.1 * \epsilon$ to ϵ the recall of the monitor increases from 0.009 to 1. Thus if γ is set to be a smaller fraction of ϵ then the monitor will have higher precision and lower recall. On the other hand if γ is set to be closer to ϵ then the monitor will have lower precision and higher recall. Observe that one can choose γ such that both precision and recall of the monitor are reasonable and not too low. For example, in Table 6.1f for $\beta = 0.02$ when γ is set to $0.5 * \epsilon$ the precision is 0.751 and the corresponding recall in Table 6.2f is 0.824.

While the observed precision values are low, observe that the number of actual valid snapshots in the system for some of the settings are very high. For example, in Table 6.1a for $\beta = 0.045$, $\gamma = \epsilon$, precision is 0.295, and the number of actual valid snapshots in the system is 2907 (c.f. Table 6.2a, $\beta = 0.045$, $\gamma = \epsilon$, denominator). This corresponds to the case where the monitor has a low precision in a system that has several bugs. In such a scenario, bugs in the system could be related to a common problem and detecting a few bugs and fixing the cause could eliminate majority of the bugs in the system. Also, on the other hand if a system has so many bugs one should be able to detect them by generic testing.

6.1.5 Implications of False Negatives/Positives

From Tables 6.1a and 6.2a, we see that the use of γ-extension can suffer from high false positives and/or false negatives. We now discuss how we can cope with false positives and false negatives.

We first consider the case where we only have false positives. We expect that the violations are likely to be rare. This is due to the fact that we generally deploy programs that are *mostly correct*. As an illustration, suppose that the likelihood of an error in a given time unit (say 1 $second$) is 0.1% and the false positive rate is 90%. That means that the likelihood that the monitor reports an error in the given time unit is 1%. In this setting, we can use a second monitor that is accurate

but expensive to analyze these computation slices (corresponding to 1% of the computation) to determine which errors are real. The key observation is that, in this case, 99% of the computation would not be analyzed with the expensive monitor. We use this idea to develop a two-layered monitor in Section 6.2.

We next consider the case of false negatives. While false negatives are not ideal, in many situations, they may be acceptable. For example, if most false negatives are latent errors (rarely manifest in practice but can be detected with an accurate monitor), even a 90% false negative rate (i.e., only 1 in 10 latent errors is identified) means the error, on average, would take roughly 10 times as long to detect. Furthermore, we may resort to using a monitor that has false negatives when the resource requirements of more accurate monitors, such as the one from [20] which requires $O(n)$ size clocks, make deploying the more accurate monitor practically impossible. In such circumstances, we must use the most accurate monitor that can be deployed.

6.2 Two-Layered Monitoring Approach

Predicate detection using SMT solvers discussed in Chapter 5 guarantees the absence of false positives and false negatives, but the drawback of this monitoring approach is the high computation time required to perform the detection. If the monitor takes too long, it cannot be used in an online setting as it will not be able to keep up with the application. In this section, we present our two-layered monitoring approach that is both accurate and efficient. The first layer uses HLC based monitor with γ-extension, and the second layer uses SMT solvers. We invoke the SMT based monitor (which is accurate but inefficient) only if the HLC based monitor with γ extension (which is inaccurate but efficient) identifies the possibility that the predicate $\mathcal{P}$ of interest is likely to be true. Specifically, the combined monitor works as follows:

- Similar to the monitoring approach discussed in Section 5.1, each process reports changes of variables involved in predicate $\mathcal{P}$ to the monitor. It also reports to the monitor the timestamps of messages that are sent and received.

- HLC based monitor with γ-extension uses the information about variable changes to deter-

mine if $\mathcal{P}$ is true. Note that if $\gamma = \epsilon$, this approach suffers from only false positives. For $\gamma < \epsilon$, it may suffer from false positives and negatives.

- The SMT based monitor creates constraints as discussed in Section 5.1. These constraints are partitioned into *windows* based on the timestamps of the corresponding events (message send/receive events and events where the value of a variable changes). For example, if w is the length of the window, then the windows correspond to timestamps, $[0..w+\epsilon]$, $[w..2w+\epsilon]$, etc. The overlap of ϵ is added to ensure that we do not miss snapshots that cross window boundaries. In this chapter, we use $w = \epsilon$.

- If the monitor based on HLC with γ-extension finds a consistent snapshot where $\mathcal{P}$ is satisfied, then the monitor invokes the SMT solver on corresponding window of constraints. Otherwise, the constraints from that window are discarded.

- We batch the windows on which the SMT solver is invoked. This means the SMT solver is invoked less often but has to deal with multiple windows at once.

The recall of the two-layered monitor depends on the value of γ used in the filtering layer. Based on our discussion in Section 6.1.4.2, if $\gamma = \epsilon$ in the filtering layer, then the two-layered monitor will have perfect recall. If the false positive rate of ϵ-extension is too high thereby resulting in an inability to run the second layer of the monitor efficiently, we may have to use $\gamma < \epsilon$ thereby sacrificing perfect recall. In this case, the two-layered monitor may suffer from false negatives.

Irrespective of the value of γ used, the two-layered monitor will have perfect precision, i.e. no false positives, because the solver in the second layer will verify and eliminate all false positives. In general, using HLC with γ-extension as a filtering layer will reduce the number of times the monitor has to invoke the solver. This is due to the fact that if the filtering layer does not detect a violation in a window, then the monitor does not have to invoke the solver to check that window.

To analyze the effect of the two-layered approach, we compare it with the single-layered approach in Chapter 5.

6.2.1 Evaluating Efficiency of the Two Layered Monitor

6.2.1.1 Application based on time division multiplexing.

To evaluate the effectiveness of the two-layered monitor, we use it to monitor an application similar to the exclusive access application considered in Section 5.3.1. The application uses time division multiplexing to ensure exclusive access to a shared resource. The application has the same setup discussed in Section 6.1.4.1, except for the parameters β and ℓ. In this application, the value of the variable v_i at process i denotes if the process i is accessing the shared resource ($v_i = true$) or not ($v_i = false$).

The basic intuition of time division multiplexing is as follows. Process 0 accesses the resource in interval $[0, T)$, process 1 accesses it in the interval $[T, 2T)$, and so on. After the last process accesses the resource, process 0 can access it again. And, the cycle repeats.[2]. To account for clock drift and avoid simultaneous access, the access-windows are changed to $[0, T - \epsilon)$, $[T, 2T - \epsilon)$ and so on. However, we introduce an error so that the time to release the resource is changed with probability of 10%. This will cause some processes to continue accessing the resource while the next process in the sequence starts accessing the resource. Thus, the predicate $\mathcal{P}$ of interest captures that two or more processes are accessing the resource simultaneously.

6.2.1.2 Two-layered Monitoring Setup

In the two layered monitoring setup, we treat every 100 windows (recall that each window w is of duration ϵ) of the application run as a *batch*. At the end of every batch, the monitor first performs predicate detection using HLC with γ-extension by processing all the intervals[3] reported by the processes in the current batch. For every snapshot detected using HLC with γ-extension, the monitor marks the corresponding window in the batch. Then the monitor invokes the solver to process all the *marked* windows in the batch. Specifically, the monitor generates constraints

[2]The application considered in Section 5.3.1 allowed every process to access the resource exactly once, i.e., without a cyclic repetition.

[3]Recall that each interval reported by a process i corresponds to a duration for which v_i was true

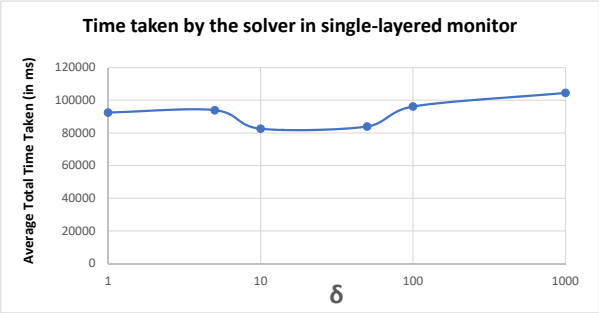
corresponding to all the marked windows in the batch and feeds it to the solver. It records the total time taken by the solver to evaluate them. The monitor then invokes the solver (again) to process all the windows (marked and unmarked) in the batch and records the total time taken by the solver.

We apply the two-layered monitor in the time division multiplexing protocol based application varying the parameters α, δ, ϵ in the application and γ in the monitor where the monitor is trying to detect if the processes ever violate the exclusive access requirement i.e., if two or more processes access the shared resource simultaneously. For each setting, we obtain the overall time taken by the solver to check all the windows in each batch (this corresponds to the time taken by the solver in the single-layered monitor in Chapter 5) and the overall time taken by the solver to check only the windows marked by HLC with γ-extension in each batch (this corresponds to the time taken by the solver in the two-layered monitor). We also computed the time taken for predicate detection using HLC with γ-extension in the filtering layer and observed that it takes around 2 to 3 seconds.

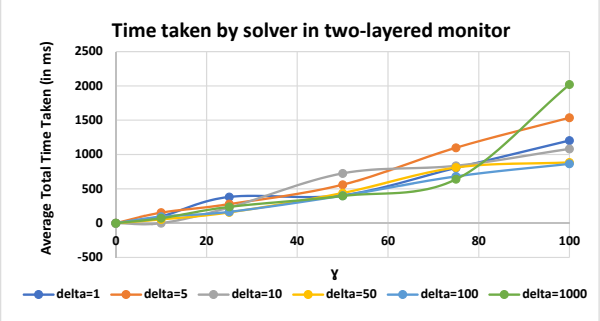
6.2.1.3 Experimental Results

We present our experimental results in Figure 6.3. The graphs on the left (Figures 6.3a, 6.3c, 6.3e) present the time taken (in milliseconds) by the solver in the single-layered monitor in Chapter 5. For the same settings, we present the corresponding total time taken (in milliseconds) by the solver in the two-layered monitor in the graphs on the right (Figures 6.3b, 6.3d, 6.3f).

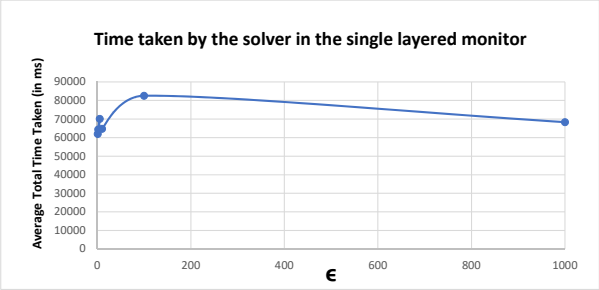
We note that when $\gamma = \epsilon$, the single layered monitor and the two-layered monitor are both completely accurate with no false positives or false negatives. We observe that the two-layered monitor is much more efficient. For example, when $n = 10, \delta = 10, \epsilon = 100, \alpha = 0.1$ in the application in 6.2.1.1, the two layered monitor takes 2 seconds for the first layer and 1.1 seconds for the second layer. By contrast, the single layered monitor takes 81 seconds. On average, the two-layered monitor reduces the cost by 90%, with a minimum savings of 85% across all our parameter settings. Furthermore, the maximum time required for the second layer across all our parameter settings is only 2.5 seconds which is comparable to the time required for the first layer. This ensures that the second layer will not become the bottleneck.



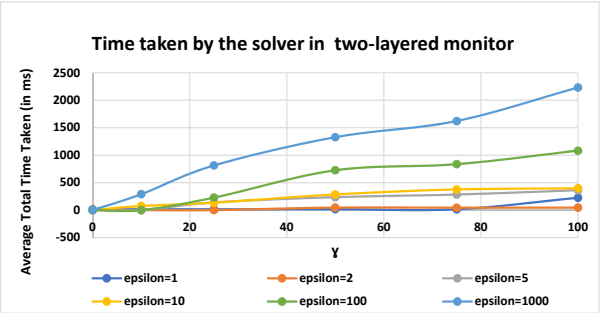
(a) Varying δ and γ



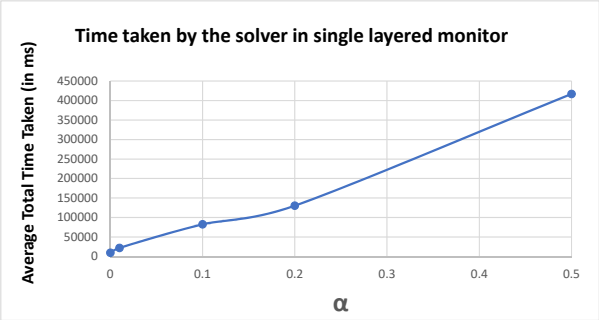
(b) Varying δ and γ (Time taken by the first layer was approximately 2 seconds)



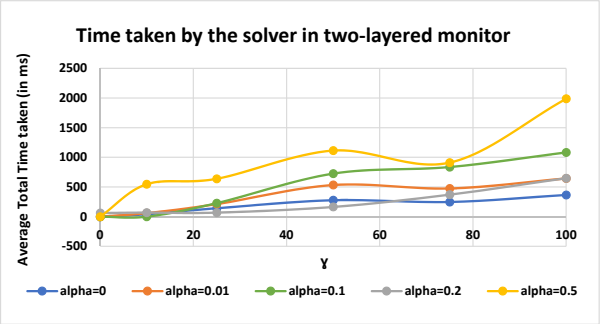
(c) Varying ϵ and γ



(d) Varying ϵ and γ (Time taken by the first layer was approximately 3 seconds)



(e) Varying α and γ



(f) Varying α and γ (Time taken by the first layer was approximately 3 seconds)

Figure 6.3: Time taken by the SMT Solver to detect violations of mutual exclusion in a time division multiplexing protocol when processing all windows vs windows marked by γ-extension. γ is varied as fractions of ϵ (taking floor value if the fraction is not an integer). Time is measured in milliseconds. Default values: n=10, $\epsilon = 100$, $\alpha = 0.1$, $\delta = 10$.

A γ extension ($\gamma < \epsilon$) should be used only if the time required for the ϵ-extension monitor is still too high, as it will allow us to reduce the time of monitoring further at the cost of false negatives. We explore how decreasing γ reduces the time required by the second layer. The three graphs in Figures 6.3b, 6.3d, 6.3f show a roughly linear decrease in running time for the second layer as a function of γ across all three parameters α, β, and δ. Details of the increase in false positives/negatives are in Tables .1b and .2b in the Appendix. Note that false positives will be removed by the second layer. In general, we observe that the time taken for the two-layered monitoring approach decreases when the message frequency decreases, clock drift decreases, or when the message delay increases.

We note that we considered the problem of conjunctive predicate monitoring in Section 6.1.4 as it forms a basis of all predicates. We considered mutual exclusion in this section as it occurs more frequently in practice. Furthermore, conjunctive predicates cannot be configured easily to have a nonzero but small number of bugs. However, we find similar results in the case of conjunctive predicates when the number of bugs is small. Specifically, we observed the same trend when the two-layered monitor was deployed in the context of conjunctive predicates when the number of valid snapshots in the system was small (<50). For example, for the setting $n = 5, \alpha = 0.1, \ell = 20, \beta = 0.004$ where the number of valid snapshots in the system was 38, the time taken by the solver in the single-layered monitor was 16.8s, whereas in the two-layered monitor it took 3.2s. (We note that having more than 50 bugs in the runs considered here would mean the application is not ready to be deployed; it needs to be tested thoroughly, not monitored at runtime.)

6.3 Advantages and Limitations of predicate detection using γ-extension and two-layered monitoring

In this chapter, HLC based predicate detection (monitoring) with the use of γ-extension was first introduced. A key advantage of this monitor is its simplicity and reduced overhead. Specifically, it permits the use of just $O(1)$ sized timestamps. Also, it permits monitoring of complex predicates.

A key disadvantage of HLC based monitors was that they suffered from false negatives. In

partially synchronous systems (that rely on clocks being synchronized within ϵ), we can eliminate the false negatives at the cost of permitting false positives by using ϵ-extension. By choosing a γ extension ($\gamma < \epsilon$), we can obtain a trade-off between false positives and false negatives. We combined the HLC based monitor with γ-extension and the SMT based monitor in Chapter 5 to obtain efficient and accurate monitors. When we set $\gamma = \epsilon$, these monitors have no false positives or false negatives (i.e., they identify all bugs without identifying any phantom bugs). Furthermore, the time required for monitoring is significantly smaller (85-95% less) than the monitoring solution discussed in Chapter 5.

In the event that the computation time is still higher than anticipated, γ-extension ($\gamma < \epsilon$) can help. Specifically, γ-extension reduces the number of false positives thereby reducing the instances when the SMT solver is invoked. The cost of this is that the monitor will suffer from false negatives. However, if we are dealing with bugs that stay latent for a long time before they result in an actual problem, false negatives may be acceptable when the other option is an inability to monitor. For example, for the application in 6.2.1, when $\alpha = 0.1, n = 10, \delta = 10, \epsilon = 100$, if we choose $\gamma = 0.25 * \epsilon$, then the cost of monitoring becomes 228 ms (when compared with 1.1s for ϵ-extension). In this scenario, the false negative rate is 0.627, i.e., roughly every 3 bugs out of 8 would be identified.

While our approach is designed for partially synchronous systems, it works for asynchronous ($\epsilon = \infty$) systems as well. Specifically, the notion of γ-extension can be applied for any γ, $\gamma < \epsilon$. The number of expected false negatives in such a setting can be characterized by the results in [47]. While this approach would have some false negatives, the overall cost of it would be significantly lower as it relies on $O(1)$ HLC timestamps than $O(n)$ Vector Clock timestamps.

CHAPTER 7

RELATED WORK

7.1 Detection of different types of predicates

Predicate detection in distributed systems has been studied extensively [24, 40, 14, 20, 21, 1] in the past. Predicate detection algorithms focusing on detection of specific types or classes of predicates exist in literature [24, 10, 11]. Specifically, a predicate is referred to as a strong predicate if all serializations or ordering of events (across process) contain a global state where the predicate becomes true. A predicate is weak if only some serializations of events contain a global state where the predicate becomes true. Algorithms for detection of strong and weak unstable predicates were presented in [19] and [24] respectively. A predicate is referred to as an unstable predicate if it can become false after holding true for some time and stable predicates are predicates that never become false once they become true. Predicate detection algorithms focusing on stable predicates were discussed in [10, 42]. In [19], Waldecker and Garg also presented an algorithm that focuses on the detection of strong linked predicates, where linked predicates are predicates that involve multiple cuts or global states.

Temporal logic predicates are predicates that involve temporal operators like EF, EG and AG. In distributed systems, an observed run can correspond to several serializations of events, i.e., events in the run can be ordered in several different ways. If each serialization is viewed as a possible path that the system could have taken during execution, then temporal operators help in specifying requirements in terms of the paths that the system can take. Algorithms for detecting temporal logic predicates in distributed systems were presented in [40, 41, 4].

In [11], Chase and Garg classified predicates into linear and semi-linear predicates. They showed that for linear predicates efficient algorithms for detecting the earliest global state that satisfies the predicate exist, whereas for semi-linear predicates some global state where the predicate is satisfied can be detected. In [11], an algorithm for the detection of predicates of the type $x_1 + x_2 + \cdots + x_N < k$,

called bounded sum predicates (similar to arithmetic predicates in Chapter 3), was also presented. For predicate detection in [10, 42] processes in the system send their local state information or local snapshot to other processes in a sequential or coordinated fashion to compute the global snapshot or global states. Predicate detection algorithms in [24, 19, 11, 41] rely on the use of Vector Clocks.

While Vector Clocks are widely used for predicate detection [24, 20, 22]. A disadvantage of using Vector Clocks is that each vector clock timestamp is of size $O(n)$. One can use Optimal Vector Clocks [50] or Hybrid Vector Clocks [46] to reduce the size of the timestamps. But, the size still remains $O(n)$ in the worst case. Another disadvantage of Vector Clocks is that they do not account for real time information. Capturing real time information can be beneficial in identifying when the violation happened in real time. Also, as discussed in Chapter 1, capturing real/physical time information helps in eliminating global states that contain events that are far apart in physical time as infeasible states, i.e., the system could not have gone through these global states during system execution. Specifically, if the clock drift in the system is bounded by ϵ, then two events that seem to be concurrent based on their vector clock or logical clock timestamps are not concurrent events if they happened more than ϵ apart in physical time. Therefore, predicate detection using vector clocks can result in false positives.

7.2 Accounting for real time information

To capture real time information, algorithms can use physical timestamps assigned by local physical clocks at the processes in a distributed system. However, local physical clocks at processes in distributed systems are not always perfectly synchronized. Local physical clocks can be synchronized using NTP (Network Time Protocol)[38], GPS, WWV radio [33], GOES (Geostationary Operational Environment Satellites)[39], etc. The clock drift associated with these synchronization solutions range from 0.1 ms to 10 ms. Since solutions that guarantee almost no clock drift have higher associated cost or are not suitable in most practical settings, synchronization solutions that guarantee non-zero but small clock drift are used more commonly in practice. In other words, most real-world distributed systems are partially synchronized, i.e., local physical clocks at processes

have bounded clock drift ϵ, $\epsilon > 0$. Therefore, with partially synchronous systems, one has to account for clock drift when capturing and utilizing real/physical time information for predicate detection. All the predicate detection algorithms presented in this report capture physical time information and clock drift between processes. Specifically, all the algorithms presented in this report use Hybrid Logical Clocks or an extension of it, and the value of l in Hybrid Logical Clocks is always guaranteed to stay close to (i.e., within ϵ-clock drift of) the underlying physical clock value.

Predicate detection using physical clocks while also accounting for clock drift was discussed in [43]. Specifically, in [43], Stoller proposed the use of physical clocks to detect global predicates, where for a system of n processes, if the inter-event spacing is greater than the clock drift, then the total number of possible states to be evaluated is $O(En)$, where E is maximum number of events at any process. The predicate detection algorithms considered in this report relied on Hybrid Logical Clocks (or an extension of it) which are also constant sized clocks like physical clocks. However, algorithms considered in this report do not make any assumptions about the inter-event spacing. In terms of complexity, in Chapter 3, several algorithms with comparable complexities ranging from $O(I \log(\Delta n))$ to $O(In)$, where I is number of reported intervals and Δ is the bounded message delay, were presented.

7.3 Trading off false negatives/positives for efficiency

Several existing monitoring techniques use the notion of allowing false negatives for efficiency. Specifically, in [17], they perform optimistic hybrid analysis. In hybrid analysis, checks proven by static analysis are not checked during dynamic analysis. In [17], they show that by carefully making the static analysis unsound i.e., by allowing false negatives, they can speed up the dynamic analysis. Specifically, they make the static analysis unsound by making some assumptions (consider some likely invariant) that may or may not be true. However, they check the assumptions during dynamic analysis to ensure that they are actually true, if not then the analysis is rolled back and traditional hybrid analysis is performed i.e., without allowing false negatives during static analysis.

Monitoring approaches that use techniques like sampling [35] and computation slicing [23] aim

to improve efficiency of the monitor by analyzing only selected portions of program executions. In [35], they show that it is possible to sample a multi-threaded program at a low frequency, and yet, find infrequently occurring data races. They found that even when sampling at a small rate (for instance 2% of data accesses) they were able to find a lot (70%) of the data races. Essentially, they allow false negatives (for instance 30% of the data races) to achieve efficiency. In [49], they use adaptive tracking for efficient detection of data races. To eliminate false positives they switch the monitoring granularity from object level (for example tracking locksets associated with arrays) to field level (tracking locksets associated with array elements) only if the object level analysis reports a warning.

7.4 Adaptive runtime monitoring

Monitors in [32] and [37] adapt to overhead-budgets or timing/memory constraints. Specifically, in [32], they discuss policies that help decide what monitoring events/operations should be skipped/dropped while monitoring to reduce the monitoring overhead. They skip some monitoring operations (leading to false negatives) to stay within predefined overhead budget while maximizing coverage (reduce false negatives) within the budget.

In [37], they propose a control-theoretic monitoring solution that focuses on improving memory utilization of the monitor and reducing detection latency. They reduce monitor jitters by dynamically buffering and reporting events to the monitor periodically. Specifically, in [37], they propose an approach where the monitor uses a feedback loop to control how frequently the events are reported to the monitor. This helps the monitor to adapt to the frequency of events in system. They also propose the use of dynamically-sized buffers, where the size of the buffers that stores events between monitor invocations changes to adapt to demand. Our two-layered monitoring approach in Chapter 6 can be combined with the approach in [37]. Specifically, our two-layered monitor has a fixed polling period (batches of fixed-length) and we consider unbounded buffers. We can extend our approach to let the polling periods and the buffer sizes to be dynamic. On the other hand, we could also extend our approach to use a feedback loop to learn about the frequency of

events, and dynamically turn off the second layer of the monitor if the frequency is high at the cost of false positives/negatives (depending on γ), and turn it back on when the frequency is not too high. While the monitoring solutions provided in this report do not dynamically adapt to timing/memory constraints or adjust monitoring operations to stay within predefined overhead-budgets, we presented different monitoring approaches that one can choose from to suit their monitoring budget and coverage.

CHAPTER 8

FUTURE WORK

In this chapter, we will discuss some of our potential future research directions. Specifically, in Section 8.1.1, we will discuss about improving the performance of runtime monitors that use SMT solvers, presented in Section 5. Furthermore, in Section 8.1.2, we discuss about the applicability of these monitors in detecting different types of predicates. In Section 8.2, we will discuss the use of other techniques or models instead of SMT solvers, to aid the monitor in performing predicate detection. In Section 8.3, we discuss about detection of violations that do not reveal themselves in all runs, but when the messages in the system are received by the processes in a specific order. Lastly, in Section 8.4, we discuss about defining the problem of predicate detection as a Constraint Satisfaction Problem (CSP) and solving it using a constraint programming optimizer.

8.1 Extensions of runtime monitoring using SMT solvers

8.1.1 Online monitoring with multiple monitors.

As discussed in Chapter 5, runtime monitoring using SMT solvers guarantee detection of all possible instances of predicate satisfaction, but if implemented in an online runtime monitoring setup, the monitor may fail to keep up with the system being monitored. Therefore, using multiple monitors to share the monitoring load during runtime monitoring can help in reducing the lag. Specifically, since our experiments in Chapter 5 showed that the monitor took 1-2 seconds to perform predicate detection in a system run that was 1s long, using two monitor instances that invoke independent SMT solver instances can be helpful in such scenarios. In other words, periodically the processes in the system will have to switch the monitor that they report to. Such a monitoring approach can be effective if at least one of the two monitors is always readily available to process the newly reported window of events.

8.1.2 Detecting different types of predicates.

Multi-variable predicates. The predicates that we considered so far were conjunctive predicates of the form $\bigwedge_{j=1}^{n} pr_j$ and arithmetic predicates of the form $f(x_1, \ldots, x_n) \leq C$. However, violations in several practical distributed systems may be represented as predicates that contain more than one variable. For example, let us consider a set of autonomous robots in motion, and let us say that it is a violation if these robots come too close to each other. In this case, the corresponding predicate of interest could be a distance function $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} < D$, where (x_1, y_1), (x_2, y_2) correspond to the locations of two different robots. To detect such predicates, first the reporting mechanism has to be modified. Specifically, the processes have to report the values of x and y whenever either of them get updated. The reports can be in an interval format or can be reported as individual change points depending upon the frequency of the updates, i.e., in this case depending upon the speed of the robots. Algorithm 3 in Chapter 3 cannot be used in detecting such predicates. Algorithm 6 in Chapter 4 can be used to identify consistent global states (made up of possibly concurrent locations of the robots), and one can evaluate the predicate using the values of x and y corresponding to those consistent global states. However, as discussed in Section 4.5, $BHLC$ based predicate detection can miss to identify some possible consistent global states leading to false negatives. So we would like to utilize the monitoring mechanism presented in Chapter 5 using SMT solvers. As mentioned earlier, the processes will have to report when either of the variables x, y get updated and the monitor will have to generate corresponding constraints.

The predicate being detected will also have to be provided by the monitor to the SMT solver. Furthermore, to evaluate the monitoring approach under difficult scenarios of multi-variable predicates, we would like to consider predicates with more than two variables per process and would like to analyze the efficiency of the monitor as the number of variables being handled increases.

Fixed-length predicates. We would also like to determine the applicability of the online monitoring approach using SMT solvers in detecting predicates that are satisfied for a specific duration of time. For instance, reconsider the example of autonomous robots, and let us consider that the robots are designed for collision avoidance and prevent such situations by moving away

from each other. So let us consider that it is a violation only if the robots come too close to each other and do not move away, i.e., if the distance between them becomes smaller than a specific limit and stays that way for a longer duration. To handle this scenario, the monitor will have to provide the SMT solver more information (constraints), to help in identifying valid instances of predicate satisfaction.

Temporal Logic Predicates. As discussed in Chapter 7, temporal logic predicates are predicates defined over different paths that a distributed system could have taken during an observed execution or run. If the temporal logic predicate involves detecting if *there exists* **(EF)** a path where a specific condition or predicate *finally* became true then algorithms presented in this report can be used to detect the predicate. If the temporal logic predicate involves finding if *all paths* **(AF or AG)** satisfied a specific condition or predicate then algorithms based on Hybrid Logical Clocks may not be suffice because they do not capture enough information required to enumerate all possible serializations of events (i.e., all possible paths). However, the combination of HLC with ϵ-extension and SMT solvers discussed in Chapter 6 can be used to detect such predicates. Specifically, the HLC with ϵ-extension approach can help in detecting parts of the predicate that do not require exploring all paths of execution and the SMT solver can be invoked whenever all paths are required to be analyzed.

8.2 Considering alternatives for SMT solvers.

Binary Decision Diagrams [7], that represent a function as a graph, have been widely used in performing formal verification, logic synthesis and test generation[8]. Specifically, an ordered binary decision diagram (OBDD) contains one non-terminal node for every variable in the function and two terminal nodes: 0 and 1. A path from the root of an OBDD to the terminal 1 corresponds to a satisfying variable assignment for the corresponding function. We would like to use BDDs to aid the monitor in performing predicate detection. Specifically, like the runtime monitoring approach using SMT solvers, presented in Chapter 5, we would like to perform runtime monitoring using BDDs. For this, the processes in the system will have to report the same set of events to

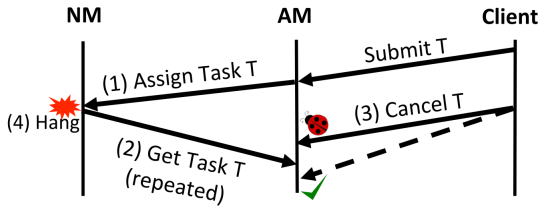


Figure 8.1: Violation revealed when messages get delivered in a specific order [30]

the monitor, as those discussed in Section 5.1.1. The monitor will have to create the same set of constraints, as those discussed in Section 5.2. However, rather than generating corresponding SMT constraints, the monitor will have to create an OBDD corresponding to the formula say F, where F is the conjunction of all the constraints. If the derived OBDD does not have a path from its root to the terminal 1, then it indicates that the predicate was never satisfied. If the derived OBDD has a path from its root to the terminal 1, then it corresponds to an instance of predicate satisfaction, i.e. a consistent global state where the predicate of interest is true. Furthermore, variables in our work are not strictly boolean variables, so we will have to use extensions of BDD namely MTBDD [13, 3], BMDs [9] or Hybrid Decision Diagram [12] to handle arithmetic properties.

8.3 Detecting latent concurrency bugs under uncertainty in communication delay

Consider the scenario in Figure 8.1 from [30], where a violation is revealed only when the messages in the system get delivered in a specific order. This calls for the need to analyze a given system run, for possibilities or outcomes when the messages in the observed run get delivered early or get delayed. Furthermore, the local states at the processes, after message reception, may depend upon the message receive events. So the local state of a process, at any point in time, may depend on the number of messages received by the process till that point. We would like to perform runtime monitoring and evaluate an observed run of a distributed system, to detect any such violations, by considering the uncertainty in communication delay and corresponding local states of the processes. We aim to use the monitoring approach presented in Chapter 5, based on SMT solvers, to achieve this. However, in this scenario the monitor will have to provide additional

or modified constraints to the SMT solver to account for the uncertainty factor in terms of message delay and values of the variables at the processes, that may depend on the messages received by the processes so far.

8.4 Predicate Detection as a Constraint Satisfaction Problem

Constraint Satisfaction Problems (CSP) are combinatorial optimization problems and several real world problems like scheduling, sequencing, resource allocation, etc., can be expressed as CSPs. Formally, a CSP [6] consists of

- a set of variables $X = \{x_1, \cdots, x_n\}$;
- for each variable x_i, a finite set D_i of possible values (its domain);
- a set of constraints restricting the values that the variables can simultaneously take.

A solution to a CSP is a set of variable assignments, for the variables in X, from their domain, that satisfies the set of constraints. The problem can also have an optional objective function, defined in terms of the variables in X, in which case the algorithm solving the CSP may aim to find an optimal solution with respect to that function. Constraint programming, which is mainly based on logic programming and graph theory, is widely used to solve CSPs. Constraint programming algorithms, represent the search space as a tree and traverse them to find feasible or optimal solutions. Some of the common techniques used by constraint programming algorithms to solve CSPs are backtracking, forward checking and MAC (maintaining arc consistency) [6].

In our work, first we would like to define the problem of predicate detection to detect all possible instances of violations as a CSP. Specifically, a runtime monitor that recieves reports from the processes would define the problem as a CSP by creating constraints similar to those discussed in Section 5.2. Then the monitor can use any of the existing CSP solving approaches to find a satisfying variable assignment, and this would correspond to an instance of predicate satisfaction. We plan to use ILOG CP Optimizer, a constraint programming optimizer by IBM to solve the CSP, i.e., predicate detection in this scenario. This tool benefits from several search techniques

that are dynamically changed during the search by adapting to the problem at hand. It uses search techniques like large neighborhood search, genetic algorithms, etc. [26]. We would like to evaluate the timeliness of this approach, i.e, we would like to know if the monitor can perform predicate detection based on this approach in a timely manner. This would depend on the time taken for all the sub-tasks namely representing the problem as a CSP, invoking the optimizer and the time taken by the optimizer to find the solution.

CHAPTER 9

CONCLUSION

Performing runtime monitoring to detect bugs or violations in distributed systems is critical to ensure their correctness and reliability. However, analyzing an observed run of a distributed system to identify violations is challenging due to the exponential number of consistent global states that are possible in a run. To identify all possible consistent global states in a run, one will have to order the events in the run using $O(n)$ sized timestamps, where n is the number of processes in the system. In other words, $O(n)$ is an unavoidable cost if the goal is to identify all possible consistent global states and evaluate them to thereby identify all possible instances of predicate satisfaction, that indicate possible violations. However, runtime monitoring of distributed systems to perform predicate detection with less overhead, is possible, if one does not require to identify all possible instances of predicate satisfaction. Specifically, by using Hybrid Logical Clocks, which are $O(1)$ sized clocks, one can perform predicate detection at a lower cost. In this report, we discussed some efficient algorithms to help the monitor in performing predicate detection using Hybrid Logical Clocks, in a timely and non-intrusive manner at a low cost.

Another approach to perform predicate detection with low overhead is by using Biased Hybrid Logical Clocks, which are also $O(1)$ sized clocks. Since monitors that use Biased Hybrid Logical Clocks miss to identify fewer consistent global states than monitors that use Hybrid Logical Clocks, predicate detection using Biased Hybrid Logical Clocks can identify 100-200 times the number of instances of predicate satisfaction (i.e., violations) identified using Hybrid Logical Clocks. While Biased Hybrid Logical Clocks successfully improve upon the ability of Hybrid Logical Clocks to perform predicate detection, they still do not guarantee detection of all instances of predicate satisfaction.

To achieve detection of all instances of predicate satisfaction the monitor can use Hybrid Logical Clocks with ϵ-extension, where ϵ is the maximum clock drift in the system. Essentially, with ϵ-extension the effect of each event at a process is extended for an ϵ duration to compensate

for the clock drift. As a result, for each event (or corresponding local state) at a process, the number of events (local states) at other processes that are considered as concurrent events (states) by HLC is increased with ϵ-extension. This helps in detecting all possible consistent global states. While monitoring using HLC with ϵ-extension can guarantee detection of all instances of predicate satisfaction, it also introduces false positives, i.e., detects phantom instances. Specifically, the length of the extension (of the effect of events) has direct impact on the number of false positives. Therefore, the monitor can reduce the false positives by reducing the amount by which it extends the effect of the events, essentially by extending for a duration γ, smaller than ϵ. But this will re-introduce some eliminated false negatives (i.e., the monitor will again fail to identify some instances of predicate satisfaction). On the whole, the monitor can achieve a trade-off between false positives and false negatives by altering γ-the amount by which it extends the effect of events in an observed run.

To achieve detection of all instances of predicate satisfaction using $O(1)$ sized clocks like Hybrid Logical Clocks and to avoid detecting phantom instances one can use SMT solvers. Specifically, while Hybrid Logical Clocks do not capture all the necessary information essential to identify all possible consistent global states, SMT solvers can help in identifying those instances that can be missed by a pure HLC based detection solution. However, the timeliness of monitors that use SMT solver based detection approach depends on the time taken by the solver to solve the problem. In some scenarios the solver can take longer to solve the problem thereby causing the monitor to lag and fall behind the application that it is monitoring.

The timeliness of the monitors that rely on SMT solvers can be improved by using the HLC with ϵ-extension approach as a filtering layer. Specifically, the monitor can choose to invoke the SMT solver only if the HLC with ϵ-extension approach indicates a violation. This reduces the number of times the monitor invokes the solver, thereby reducing the overall monitoring time substantially. On the other hand, even if the violation identified using HLC with ϵ-extension approach is a false positive, the SMT solver can verify and eliminate it. Therefore, a two layered monitor that uses HLC with ϵ-extension and SMT solvers can guarantee timeliness and accuracy.

Thus, while choosing or designing a runtime monitor, to perform predicate detection and identify instances of violation in an observed run, one may have to choose between the desirable characteristics of runtime monitors namely non-intrusiveness, timeliness, low overhead and correctness/accuracy based on the resource limitations of the application being monitored.

APPENDIX

APPENDIX

EXPERIMENTAL DATA

A.1 Precision and Recall results for application in 6.2.1

Tables .1 and .2 show the two-layered monitor's precision and recall values when deployed in the application presented in Section 6.2.1. The corresponding time taken by the solver in each of these settings were presented in Figures 6.3b, 6.3d, 6.3f.

No. of Valid Snapshots / Total Snapshots Detected (Default values: n = 10, ϵ = 100, α = 0.1, δ = 10)

Precision	δ **= 1**	δ **= 5**	δ **= 10**	δ **= 50**	δ **= 100**	δ **= 1000**
γ **= 0.10 *** ϵ	1 (1/1)	1 (2/2)	NA	1 (1/1)	1 (1/1)	1 (1/1)
γ **= 0.25 *** ϵ	1 (4/4)	0.666667 (2/3)	1 (3/3)	1 (2/2)	1 (2/2)	1 (3/3)
γ **= 0.50 *** ϵ	1 (5/5)	0.571429 (4/7)	0.888889 (8/9)	1 (6/6)	1 (4/4)	1 (5/5)
γ **= 0.75 *** ϵ	0.5 (5/10)	0.357143 (5/14)	0.727273 (8/11)	1 (10/10)	1 (8/8)	1 (8/8)
$\gamma = \epsilon$	0.333333 (5/15)	0.263158 (5/19)	0.571429 (8/14)	1 (11/11)	1 (9/9)	1 (11/11)

(a) Varying δ - message delay

Precision	α **= 0**	α **= 0.01**	α **= 0.1**	α **= 0.2**	α **= 0.5**
γ **= 0.10 *** ϵ	1 (1/1)	1 (1/1)	NA	1 (1/1)	1 (3/3)
γ **= 0.25 *** ϵ	1 (4/4)	1 (3/3)	1 (3/3)	1 (1/1)	1 (4/4)
γ **= 0.50 *** ϵ	1 (6/6)	1 (6/6)	0.888889(8/9)	1 (2/2)	0.666667 (4/6)
γ **= 0.75 *** ϵ	1 (7/7)	1 (6/6)	0.727273(8/11)	0.4 (2/5)	0.666667 (4/6)
$\gamma = \epsilon$	1 (10/10)	0.888889(8/9)	0.571429(8/14)	0.222222(2/9)	0.5 (4/8)

(b) Varying α - rate at which processes send messages

Precision	ϵ **= 1**	ϵ **= 2**	ϵ **= 5**	ϵ **= 10**	ϵ **= 100**
γ **= 0.10 *** ϵ	1 (1/1)	NA	NA	1 (2/2)	NA
γ **= 0.25 *** ϵ	1 (1/1)	NA	1 (5/5)	1 (4/4)	1 (3/3)
γ **= 0.50 *** ϵ	1 (1/1)	1 (3/3)	1 (8/8)	1 (8/8)	0.888889 (8/9)
γ **= 0.75 *** ϵ	1 (1/1)	1 (3/3)	1 (9/9)	1 (11/11)	0.727273 (8/11)
$\gamma = \epsilon$	1 (14/14)	1 (3/3)	1 (12/12)	1 (11/11)	0.571429 (8/14)

(c) Varying ϵ - clock drift

Table .1: Percentage of Valid Snapshots out of all snapshots detected during mutual exclusion detection using γ-extension.

No. of Valid Snapshots detected / No. of Valid Snapshots in the system (Default values: n = 10, ϵ = 100, α = 0.1, δ = 10)

Recall	δ **= 1**	δ **= 5**	δ **= 10**	δ **= 50**	δ **= 100**	δ **= 1000**
γ **= 0.10*** ϵ	0.2 (1/5)	0.4 (2/5)	0 (0/8)	0.090909 (1/11)	0.111111 (1/9)	0.090909 (1/11)
γ **= 0.25*** ϵ	0.8 (4/5)	0.4 (2/5)	0.375 (3/8)	0.181818 (2/11)	0.222222 (2/9)	0.272727 (3/11)
γ **= 0.50*** ϵ	1 (5/5)	0.8 (4/5)	1 (8/8)	0.545455 (6/11)	0.444444 (4/9)	0.454545 (5/11)
γ **= 0.75 *** ϵ	1 (5/5)	1 (5/5)	1 (8/8)	0.909091 (10/11)	0.888889 (8/9)	0.727273 (8/11)
$\gamma = \epsilon$	1 (5/5)	1 (5/5)	1 (8/8)	1 (11/11)	1 (9/9)	1 (11/11)

(a) Varying δ - message delay

Recall	α **= 0**	α **= 0.01**	α **= 0.1**	α **= 0.2**	α **= 0.5**
γ **= 0.10 *** ϵ	0.1 (1/10)	0.125(1/8)	0 (0/8)	0.5 (1/2)	0.75 (3/4)
γ **= 0.25 *** ϵ	0.4 (4/10)	0.375(3/8)	0.375(3/8)	0.5 (1/2)	1 (4/4)
γ **= 0.50 *** ϵ	0.6 (6/10)	0.75 (6/8)	1 (8/8)	1 (2/2)	1 (4/4)
γ **= 0.75 *** ϵ	0.7 (7/10)	0.75 (6/8)	1 (8/8)	1 (2/2)	1 (4/4)
$\gamma = \epsilon$	1 (10/10)	1 (8/8)	1 (8/8)	1 (2/2)	1 (4/4)

(b) Varying α - rate at which processes send messages

Recall	ϵ **= 1**	ϵ **= 2**	ϵ **= 5**	ϵ **= 10**	ϵ **= 100**
γ **= 0.10 *** ϵ	0.071429(1/14)	0 (0/3)	0 (0/12)	0.181818(2/11)	0 (0/8)
γ **= 0.25 *** ϵ	0.071429(1/14)	0 (0/3)	0.416667(5/12)	0.363636(4/11)	0.375 (3/8)
γ **= 0.50 *** ϵ	0.071429(1/14)	1 (3/3)	0.666667(8/12)	0.727273(8/11)	1 (8/8)
γ **= 0.75 *** ϵ	0.071429(1/14)	1 (3/3)	0.75 (9/12)	1 (11/11)	1 (8/8)
$\gamma = \epsilon$	1 (14/14)	1 (3/3)	1 (12/12)	1 (11/11)	1 (8/8)

(c) Varying ϵ - clock drift

Table .2: Percentage of Valid Snapshots detected during mutual exclusion detection using γ-extension out of all Valid Snapshots in the system.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Ranganath Atreya, Neeraj Mittal, Ajay D. Kshemkalyani, Vijay K. Garg, and Mukesh Singhal. Efficient detection of a locally stable predicate in a distributed system. *Journal of Parallel and Distributed Computing*, 67(4):369–385, 2007.

[2] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset (extended abstract). In *Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29 - October 1, 1994, Proceedings*, pages 326–339, 1994.

[3] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[4] Andreas Bauer and Yliès Falcone. Decentralised ltl monitoring. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 85–100, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[5] Janusz Borkowski, Damian Kopanski, and Marek Tudruj. Implementing control in parallel programs by synchronization-driven activation and cancellation. In *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings.*, pages 316– 323, 03 2003.

[6] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

[7] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[8] Randal E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1995, San Jose, California, USA, November 5-9, 1995*, pages 236–243, 1995.

[9] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits using binary moment diagrams. *STTT*, 3(2):137–155, 2001.

[10] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[11] Craig M. Chase and Vijay K. Garg. Efficient detection of restricted classes of global predicates. In Jean-Michel Hélary and Michel Raynal, editors, *Distributed Algorithms*, pages 303–317, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[12] Edmund M. Clarke, Masahiro Fujita, and Xudong Zhao. Hybrid decision diagrams. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1995, San Jose, California, USA, November 5-9, 1995*, pages 159–163, 1995.

[13] Edmund M. Clarke, Kenneth L. McMillan, Xudong Zhao, Masahiro Fujita, and Jerry Chih-Yuan Yang. Spectral transforms for large boolean functions with applications to technology mapping. *Formal Methods in System Design*, 10(2/3):137–148, 1997.

[14] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. *SIGPLAN Not.*, 26(12):167–174, December 1991.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 348–362. ACM, 2018.

[18] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[19] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, 1996.

[20] Vijay K. Garg and Craig M. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proceedings of the 15th International Conference on Distributed Computing Systems, Vancouver, British Columbia, Canada, May 30 - June 2, 1995*, pages 423–430, 1995.

[21] Vijay K. Garg, Craig M. Chase, J. Roger Mitchell, and Richard B. Kilgore. Detecting conjunctive channel predicates in a distributed programming environment. In *28th Annual Hawaii International Conference on System Sciences (HICSS-28), January 3-6, 1995, Kihei, Maui, Hawaii, USA*, pages 232–241, 1995.

[22] Vijay K. Garg and Rohan Garg. Parallel algorithms for predicate detection. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, ICDCN '19, page 51–60, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Vijay K. Garg and Neeraj Mittal. On slicing a distributed computation. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), Phoenix, Arizona, USA, April 16-19, 2001*, pages 322–329. IEEE Computer Society, 2001.

[24] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(3):299–307, 1994.

[25] Yongjian Hu and Iulian Neamtiu. Static detection of event-based races in android apps. *ACM SIGPLAN Notices*, 53:257–270, 03 2018.

[26] IBM. Constraint programming. https://developer.ibm.com/docloud/documentation/optimization-modeling/cp/[Online].

[27] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *18th International Conference on Principles of Distributed Systems OPODIS 2014*, volume 8878, pages 17–32, 2014.

[28] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[29] Eryk Laskowski, Marek Tudruj, Richard Olejnik, and Damian Kopanski. Dynamic load balancing based on applications global states monitoring. In Nicolae Tapus, Dan Grigoras, Rodica Potolea, and Florin Pop, editors, *IEEE 12th International Symposium on Parallel and Distributed Computing, ISPDC 2013, Bucharest, Romania, June 27-30, 2013*, pages 11–18. IEEE, 2013.

[30] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. Dcatch: automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 7:1–7:14, 2018.

[31] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. Fcatch: Automatically detecting time-of-fault bugs in cloud systems. *ACM SIGPLAN Notices*, 53:419–431, 03 2018.

[32] D. Lo, T. Chen, M. Ismail, and G. E. Suh. Run-time monitoring with adjustable overhead using dataflow-guided filtering. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 662–674, 2015.

[33] Michael Lombardi and Glenn Nelson. Wwvb: A half century of delivering accurate frequency and time by radio. *Journal of research of the National Institute of Standards and Technology*, 119:25–54, 03 2014.

[34] L Ma, S Mandujano, G Song, and P Meunier. Sharing vulnerability information using a taxonomically-correct, web-based cooperative database. In *CERIAS Tech Report*, 03 2001.

[35] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 134–143. ACM, 2009.

[36] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[37] Ramy Medhat, Borzoo Bonakdarpour, Deepak Kumar, and Sebastian Fischmeister. Runtime monitoring of cyber-physical systems under timing and memory constraints. *ACM Trans. Embedded Comput. Syst.*, 14(4):79:1–79:29, 2015.

[38] David L. Mills. Internet time synchronization: the network time protocol. *IEEE Trans. Communications*, 39(10):1482–1493, 1991.

[39] A Schwalb, D Cotter, and J Hussey. Goes (geostationary operational environmental satellite)-next overview. Technical report, NATIONAL ENVIRONMENTAL SATELLITE DATA AND INFORMATION SERVICE WASHINGTON DC, 1985.

[40] Alper Sen and Vijay K. Garg. Detecting temporal logic predicates on the happened-before model. In *16th International Parallel and Distributed Processing Symposium (IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, 2002.

[41] Alper Sen and Vijay K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In Marina Papatriantafilou and Philippe Hunel, editors, *Principles of Distributed Systems*, pages 171–183, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[42] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *ICDCS*, 1986.

[43] Scott D. Stoller. Detecting global predicates in distributed systems with clocks. *Distributed Computing*, 13(2):85–98, 2000.

[44] Vidhya Tekken Valapil and Sandeep S. Kulkarni. Biased clocks: A novel approach to improve the ability to perform predicate detection with O(1) clocks. In *Structural Information and Communication Complexity - 25th International Colloquium, SIROCCO 2018, Ma'ale HaHamisha, Israel, June 18-21, 2018, Revised Selected Papers*, pages 345–360, 2018.

[45] Vidhya Tekken Valapil, Sorrachai Yingchareonthawornchai, Sandeep S. Kulkarni, Eric Torng, and Murat Demirbas. Monitoring partially synchronous distributed systems using SMT solvers. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, pages 277–293, 2017.

[46] Sorrachai Yingchareonthawornchai, Sandeep S. Kulkarni, and Murat Demirbas. Analysis of bounds on hybrid vector clocks. In *OPODIS 2015, December 14-17, 2015, Rennes, France*, pages 34:1–34:17, 2015.

[47] Sorrachai Yingchareonthawornchai, Duong N. Nguyen, Vidhya Tekken Valapil, Sandeep S. Kulkarni, and Murat Demirbas. Precision, recall, and sensitivity of monitoring partially synchronous distributed systems. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, 2016, Madrid, Spain, Sept 23-30*, volume 10012, pages 420–435. Springer, 2016.

[48] Sorrachai Yingchareonthawornchai, Vidhya Tekken Valapil, Sandeep S. Kulkarni, Eric Torng, and Murat Demirbas. Efficient algorithms for predicate detection using hybrid logical clocks. In *Proceedings of the 18th International Conference on Distributed Computing and Networking, Hyderabad, India, January 5-7, 2017*, page 10, 2017.

[49] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 221–234. ACM, 2005.

[50] X. Zheng and V. Garg. An optimal vector clock algorithm for multithreaded systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2188–2194, 2019.