Nguyen, Duong; Yingchareonthawornchai, Sorrachai; Tekken Valapil, Vidhya; Kulkarni, Sandeep S.; Demirbas, Murat

Precision, recall, and sensitivity of monitoring partially synchronous distributed programs

Published in:
Distributed Computing

DOI:
10.1007/s00446-021-00402-w

Published: 01/10/2021

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Please cite the original version:
Nguyen, D., Yingchareonthawornchai, S., Tekken Valapil, V., Kulkarni, S. S., & Demirbas, M. (2021). Precision, recall, and sensitivity of monitoring partially synchronous distributed programs. *Distributed Computing*, *34*(5), 319-348. https://doi.org/10.1007/s00446-021-00402-w

Distributed Computing manuscript No.
(will be inserted by the editor)

Precision, Recall, and Sensitivity of Monitoring Partially Synchronous Distributed Programs

Duong Nguyen · Sorrachai Yingchareonthawornchai · Vidhya Tekken Valapil · Sandeep S. Kulkarni · Murat Demirbas

Received: date / Accepted: date

Abstract Distributed programs are often designed with implicit assumptions about the underlying system. We focus on assumptions related to clock synchronization. When a program written with clock synchronization assumptions is monitored to determine if it satisfies its requirements, the monitor should also account for these assumptions precisely. Otherwise, the monitor will either miss potential bugs (false negatives) or find bugs that are inconsistent with these assumptions (false positives). However, if assumptions made by the program are implicit or change over time and are not immediately available to the monitor, such false positives and/or negatives are unavoidable. This paper characterizes precision (probability that the violation identified by the monitor is valid) and recall (probability that the monitor identifies an actual violation) of the monitor based on the gap between clock synchronization assumptions made by the program/application and the clock synchronization assumptions made by the monitor. Our analysis is based on the development of an analytical model for precision, recall, and sensitivity of monitors detecting conjunctive predicates. We validate the model via simulations and experiments on the Amazon Web Services (AWS) platform.

Duong Nguyen
E-mail: nguye476@cse.msu.edu

Vidhya Tekken Valapil
E-mail: tekkenva@cse.msu.edu

Sandeep S. Kulkarni
E-mail: sandeep@cse.msu.edu
Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, U.S.A.

Sorrachai Yingchareonthawornchai
E-mail: sorrachai.yingchareonthawornchai@aalto.fi
Department of Computer Science, Aalto University, Espoo, Finland.

Murat Demirbas
E-mail: demirbas@buffalo.edu
Department of Computer Science and Engineering, The State University of New York at Buffalo, Buffalo, New York, U.S.A.

1 Introduction

1.1 Role of Clock Synchronization in Distributed Programs

Consider a simple problem of *ordered-access* (a simplified version of mutual exclusion or time-division multiplexing (TDM)) where we have two processes and a shared resource, and it is required that process P1 accesses the resource before P2. One straightforward approach to achieve such ordered-access is as shown in Figure 1a where process P1 sends a message to P2 when it is done. Process P2 waits until this message is received. Then, it can access the resource.

An advantage of the solution in Figure 1a is that it works even if the time taken by P1 to access the resource varies arbitrarily. However, an important disadvantage of the solution is that it requires *explicit* communication between P1 and P2. If such communication is difficult/slow (due to the network being large, mobile, or ad-hoc), implementing the solution in Figure 1a would be inefficient.

An alternate approach is as shown in Figure 1b, where P1 guarantees that it will complete its resource access within 5 ms. So P2 will have to wait for only 5 ms and can then access the resource. An advantage of this solution shown in Figure 1b is that it does not require explicit communication. Hence, this solution is beneficial if P1 and P2 are mobile in nature (e.g., drones or robots) and permitting them to talk with each other is

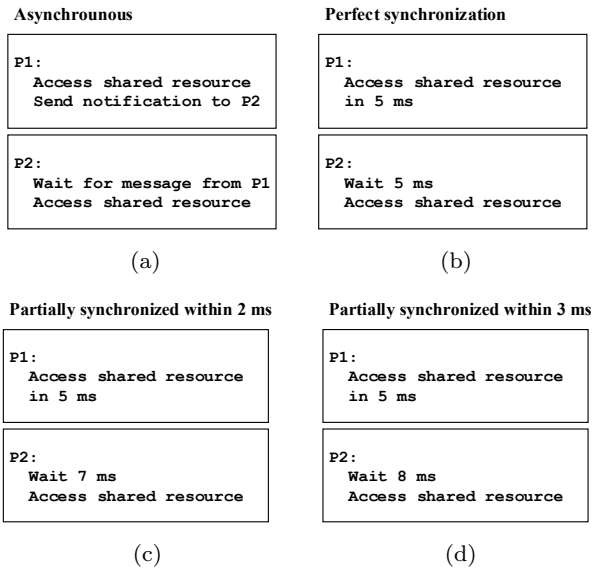


Fig. 1: Solution for the ordered-access problem in a distributed program with 2 processes when their clocks are (Figure 1a) asynchronous, (Figure 1b) perfectly synchronous, (Figure 1c) partially synchronized within 2 *ms*, (Figure 1d) partially synchronized within 3 *ms*. For simplicity, assume that P1 and P2 begin simultaneously.

challenging/expensive. In particular, after P1 and P2 have synchronized their clocks with each other, they can schedule their activities without explicit communication at every step. Permitting such implicit communication is likely to be significantly more efficient in such programs rather than requiring explicit communication.

However, a problem with the solution in Figure 1b is that the clocks of P1 and P2 may advance at a slightly different rate. Specifically, if the clock of P2 advances faster than that of P1, then it is possible that P1 and P2 may access the resource simultaneously.

If the clock skew between P1 and P2 is arbitrary then the only choice is to use the program in Figure 1a. However, suppose that we also knew that the clock skew is bounded by some value, we can utilize it in the program. As an illustration, if the clocks differ by at most 2 ms, then we can make P2 to wait for 7 ms (cf. Figure 1c). The program in Figure 1c guarantees that even if the clocks of P1 and P2 differ by up to 2 ms, P2 will access the resource only after P1. Similarly, if the clocks differed by at most 3 ms then the program in Figure 1d would achieve the same result.

1.2 Implicit Clock Synchronization Assumptions in Distributed Programs

Observe that if the programmer writes the code in Figure 1c (respectively, 1d), the programmer is making an implicit assumption that clocks differ by at most $2ms$ (respectively $3ms$). While examples in Figure 1 are straightforward, we note that programs that utilize time/timeouts (e.g., [5,11,9,10]) make similar assumptions about clock skew and/or message delays. Implicit clock synchronization assumptions are also used in programs where communication between processes is challenging (e.g., drones), and enabling the processes to take independent actions in a pre-determined manner is desirable. Figure 1 illustrates one such scenario where actions need to be taken one after another. Another common scenario is when actions of two processes need to be (almost) simultaneous. Examples of such predetermined independent simultaneous actions include two sensors taking pictures (or recording other measurements) simultaneously (coordinated without communication) so that the centralized manager can combine and interpret them later effectively.

Also, in some applications the actual clock synchronization assumption may depend upon intermittent feedback from the underlying system. For example, Network Time Protocol (NTP) [16] provides information about the clock difference in the system (with respect to the reference NTP server) at the given instant. In other words, the clock synchronization assumption of the application may be dynamic, i.e., it may change over time.

1.3 Analyzing Programs with Implicit Clock Synchronization Assumptions

As it is well known, writing distributed programs is challenging and error-prone. Hence, developing techniques for evaluating their correctness is critical. These techniques include static program analysis (e.g., model checking) and runtime analysis (e.g., runtime monitoring). Our paper focuses on the latter.

The correctness of runtime monitors depends on their knowledge of the properties of the programs being monitored. Especially, it is crucial that the monitor is aware of the clock synchronization assumptions made by the program. For example, if we use a monitor that assumes that the clocks are *not synchronized*, then that monitor will declare a violation of ordered-access for the program in Figure 1c. This is because the monitor thinks that the resource-access at P1 and P2 can possibly happen simultaneously (i.e. concurrently [13]). On the other hand, a programmer who designed

this program with the assumption that the clock skew is at most $2ms$ will consider the prompted violation to be a false positive. This is because the programmer knows that the instant at which P1 accesses the resource is strictly earlier than the instant at which P2 accesses the resource due to the wait enforced at P2. Thus knowing the program's clock synchronization assumption (as well as other assumptions made by the application programmer) helps the monitor avoid false positives/negatives.

As discussed in Section 1.2 above, the program in Figure 1c makes an implicit assumption that clocks are synchronized to be within $2ms$. These implicit clock synchronization assumptions (or assumptions that change over time, discussed in Section 1.2) make monitoring even more challenging, as the monitor may not be aware of the exact values of these implicit assumptions made by the program. In these scenarios, there can be a mismatch between the assumptions made by the program and the assumptions made by the monitor. This mismatch can lead to either false positives (the monitor flags a violation that did not occur) or false negatives (the monitor misses reporting a valid violation that actually occurred).

For example, if the program in Figure 1c is monitored by a monitor that assumes that the clocks are synchronized to be within $3ms$ then it will lead to false positives, where the monitor reports a violation when P1 accesses the resource until time 5 and P2 accesses the resource at time 7.

On the other hand, if the application developer assumes that the clocks differ by at most $3ms$ but *incorrectly* develops the program in Figure 1c (that accounts only for clock skew up to $2ms$) then the developed program may violate ordered-access. (Observe that the program in Figure 1c is correct only if clocks differ by at most $2ms$.) If the monitor that is monitoring this (incorrectly developed) program assumes that the clock skew is at most $3ms$ then it will correctly report every such violation. However, if the monitor relied on the assumption that the clock skew is at most $2ms$ then it will miss all violations. If the monitor relied on a clock skew of $2.5ms$ it will miss some (but not all) violations.

Based on the above discussion, the goal of the paper is to analyze false positives and false negatives for the case where the clock synchronization assumption of the application/program (denoted as $\boldsymbol{\epsilon_{app}}$) differs from the clock synchronization assumption of the monitor (denoted as $\boldsymbol{\epsilon_{mon}}$). Specifically, for such monitors, we want to compute **precision** and **recall** which are defined next (cf. Figure 2).

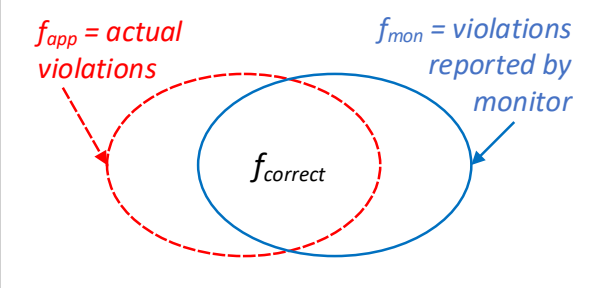


Fig. 2: Illustration of *precision* and *recall*: $f_{correct}$ are all actual violations reported by the monitor.

Definition 1 *Let f_{app} be all actual violations by the application/program, f_{mon} be all the violations reported by the monitor, and $f_{correct}$ be all actual violations correctly reported by the monitor. Then,*

$$precision = \frac{f_{correct}}{f_{mon}}$$

$$recall = \frac{f_{correct}}{f_{app}}$$

$$false\ positive\ rate = 1 - precision$$
$$false\ negative\ rate = 1 - recall$$

1.4 Contributions of the Paper

In this paper, we analyze the precision and recall of runtime monitors when there is a mismatch between the clock synchronization assumption made by the application and that made by the monitor. In particular, we consider the following scenarios:

- **Asynchronous monitors**: We consider the case where the application assumes that clocks are partially synchronized (ϵ_{app} is finite) but the monitor is not aware of this synchronization and assumes no bound for clock synchronization ($\epsilon_{mon} = \infty$). Since the recall of such monitors (denoted as asynchronous monitors) is always 1 (every actual violation is detected and reported by asynchronous monitors), we identify their precision under different parameters (specified in Section 3.1).
- **Partially synchronous monitors**: We consider the case where both the application and monitor assume that clocks are partially synchronized but the value of ϵ_{app} differs from ϵ_{mon}. We identify the precision and recall of such monitors under different parameters.
- **Quasi-synchronous monitors**: We consider the case where we use quasi-synchronous monitors. While we define quasi-synchronous monitors in Section 5, we view them as the *dual* of asynchronous

monitors. Specifically, while asynchronous monitors have a recall of 1 (and a precision of less than 1), quasi-synchronous monitors have a precision of 1 (every violation reported by quasi-synchronous monitors is an actual violation) and have a recall of less than 1. We focus on analyzing their recall under different parameters.
- **Analytical/Simulation/Experimental Results:** For each of these settings, we present an analytical model to predict the precision and recall of the monitors, as well as results from simulations and experiments in Amazon Web Services (AWS) environment to validate our analytical models.

1.5 Organization of the Paper

In Section 2, we present our computational model. In this section, we also define the notion of happened before, concurrency, global snapshots, and their extensions in applications that rely on clock synchronization. Section 3 investigates the precision and sensitivity of asynchronous monitors in monitoring applications that rely on bounded clock synchronization assumptions. In Section 4, we analyze the precision, recall, and sensitivity of the monitor when the clock synchronization assumptions made by the application and the monitor differ. Section 5 focuses on the effectiveness of quasi-synchronous monitors. We discuss related work in Section 6. Finally, we summarize the results and their implications in Section 7 and conclude in Section 8.

2 Preliminaries

2.1 Distributed Program Model

We consider a distributed program that consists of a set of $\boldsymbol{n}$ processes that communicate via messages. Each process has a local clock. We assume that the processes use a protocol such as Network Time Protocol (NTP) [16] to ensure that their local clocks are synchronized to be within $\boldsymbol{\epsilon}$ of each other. Any message sent in the program is received no earlier than $\boldsymbol{\delta_{min}}$ time and no later than $\boldsymbol{\delta_{max}}$ time. We denote such a program as **$\langle \epsilon, \delta_{min}, \delta_{max} \rangle$-program**. We also use the abbreviated notion of **$\langle \epsilon, \delta \rangle$-program**, where **$\delta$ denotes the *minimum* message delay and the maximum message delay is ∞.**

Observe that this modeling is generic enough to model programs in asynchronous systems ($\epsilon = \infty, \delta_{min} = 0, \delta_{max} = \infty$), programs in purely synchronous systems ($\epsilon = 0, \delta_{min} = 0, \delta_{max} = 0$), and programs in partially synchronous systems ($0 < \epsilon, \delta_{min} < \infty$). In this paper, we focus on monitoring programs that rely on a clock synchronization assumption. In other words, the $\langle \epsilon, \delta \rangle$-programs considered in this paper are programs in partially synchronous systems.

Table 1: List of important symbols used in the paper. These symbols/notations are defined in Sections 1.3, 2, and 3.1

Symbol	Meaning
α	Communication rate between processes
β	The frequency of local predicates becoming true
δ	Message delay
δ_{min}	Minimum message delay
δ_{max}	Maximum message delay (∞, if unspecified)
ϵ	Clock skew (synchronization error)
ϵ_{app}	Synchronization error assumed by the application/program
ϵ_{mon}	Synchronization error assumed by the monitor
ℓ	Interval length: duration for which a local predicate stays true
n	Number of processes

For convenience, Table 1 lists some important symbols and what they mean in the paper. Other symbols that are used locally are defined in the sections using them.

2.2 Extended Happened-before ($\rightarrow^{\epsilon}$) and Concurrency ($||^{\epsilon}$)

In a $\langle \epsilon, \delta_{min}, \delta_{max} \rangle$-program, each process is associated with a set of variables. Execution of this program consists of a set of events that are either send events, receive events, or local events. Send/Receive events correspond to messages sent/received by the process. Local events at a process (may) change the values of the variables of the process. Each event e is associated with $\boldsymbol{phy.e}$ that corresponds to the **physical time** (the reading of the local clock of the process where e occurred) when e occurred.

Let e and f be two events in $\langle \epsilon, \delta_{min}, \delta_{max} \rangle$-program. First, we recall Lamport's definition of causality relation [13] as follows: we say that e happened before f (denoted as $e\ hb\ f$) if and only if

- e and f are events at the same process and e occurred before f, or
- e is a send event and f is the corresponding receive event, or

– there exists an event g such that e *hb* g and g *hb* f.

Two events e and f are concurrent, denoted as $e \parallel f$, if and only if both e *hb* f and f *hb* e are false.

Based on Lamport's definition, we define the **relation $\rightarrow^{\epsilon}$** where $e \rightarrow^{\epsilon} f$ if and only if

– e and f are events on the same process and $phy.e < phy.f$, or
– e is a send event and f is the corresponding receive event, or
– $phy.e + \epsilon < phy.f$, or
– there exists an event g such that $e \rightarrow^{\epsilon} g$ and $g \rightarrow^{\epsilon} f$

Note that the definition of $\rightarrow^{\epsilon}$ is just an extension of Lamport's happened-before relation (*hb*) with the stipulation that the clocks of two processes never differ by more than ϵ. We can also extend the generic concurrency relation $\parallel$ to define **relation $\parallel^{\epsilon}$**, such that $e \parallel^{\epsilon} f$ if and only if $(\neg(e \rightarrow^{\epsilon} f) \ \wedge \neg(f \rightarrow^{\epsilon} e))$.

2.3 State of a Process

The goal of the monitor is to determine if the program reaches (has the potential to reach) an undesirable state. Hence, next, we define the notion of program state (which is obtained from the states of its processes).

The **(local) state of a process** is identified by the values of variables used by that process. The values of these variables may change due to local events at the process. (For the sake of simplicity, we are assuming that the state does not change at message send/receive. If there is a need to change the state at send or receive, we split the send event into a local event that changes the state and a message transmission. Likewise, we split the receive event into message reception and a local event that changes the state.)

It is straightforward that if e and f are consecutive local events on process i with physical time $phy.e$ and $phy.f$ then the state of process i remains unchanged in the interval $[phy.e, phy.f)$. Thus, we can create additional local events (where nothing happens) during this interval if needed. For example, in Figure 3, $e2$ and $e3$ are two consecutive events on process $p0$ that happened at physical time 15 and 20, respectively. We assume that there are *pseudo* events at physical time 16, 17, 18, 19 where the state of $p0$ is the same as that in $e2$.

Finally, since each process state corresponds to an event (or the newly added pseudo event), we can extend relations $\rightarrow^{\epsilon}$ and $\parallel^{\epsilon}$ to states as well. Specifically, two local states that are mutually concurrent (as defined in [13]) and that are within ϵ of each other are said to be **ϵ-consistent** with each other.

2.4 Monitoring

In this paper, we focus on monitors that aim to detect conditions or violations represented in the form of **weak conjunctive predicates** [8] . Specifically, we consider identifying if a given **predicate $\mathcal{P}$** becomes true in the program, where $\mathcal{P}$ is of the form $\mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \cdots \wedge \mathcal{P}_n$ and $\boldsymbol{\mathcal{P}_i}$ is a **local predicate** or condition at process i. The truth value of $\boldsymbol{\mathcal{P}_i}$ is determined by variables in process i. If a predicate is a conjunction of local predicates of less than n processes, it is a **partial conjunctive predicate**.

Satisfaction of $\mathcal{P}$ corresponds to the case where we identify a consistent global snapshot (g) of the program where $\mathcal{P}$ is true. A **global snapshot** (also called global state) g is of the form $\{ls.i |\ ls.i$ is a local state at process $i\ \}$. A global snapshot is called a consistent global snapshot if for any two local states $ls.i$ and $ls.j$ in it, $ls.i \parallel^{\epsilon} ls.j$. In other words, a consistent global snapshot consists of local states (also called local snapshots) of all processes such that (1) any two local states are within ϵ of each other and (2) any two local states are mutually concurrent as defined by Lamport's happened-before relation [13]. We denote such a snapshot as **$\boldsymbol{\epsilon}$-consistent snapshot**.

Thus, satisfaction of $\mathcal{P}$ corresponds to the case where we identify a consistent global snapshot (g) that consists of a set of mutually concurrent local states, where for each process i, predicate $\mathcal{P}_i$ is true in the local state $ls.i$ of the process.

For the sake of simplicity of the discussion, we define the term ***hb*-consistent** for the case where $\epsilon = \infty$. In other words, ***hb*-consistent snapshots** are consistent snapshots of programs in asynchronous systems.

From these definitions, we observe

Observation 1 *If a snapshot g is ϵ_1-consistent then it is ϵ_2-consistent if $\epsilon_2 \geq \epsilon_1$*

Observation 2 *If a snapshot g is ϵ-consistent then it is hb-consistent.*

When determining consistent global snapshots, the monitor uses the clock synchronization assumption ϵ_{mon}. This means that the monitor detects ϵ_{mon}-consistent snapshots. On the other hand, ϵ_{app}-consistent snapshots are the actual snapshots that could have occurred during the execution of the application program, where ϵ_{app} is the synchronization error assumed by the application. If $\epsilon_{app} \leq \epsilon_{mon}$, by Observation 1, the set of all ϵ_{app}-consistent snapshots will be a subset of ϵ_{mon}-consistent snapshots . Therefore, $f_{correct} = f_{app}$ and, by Definition 1 the monitor has $recall = 1$ (no false negatives). Similarly, if $\epsilon_{app} \geq \epsilon_{mon}$, the set of all ϵ_{mon}-consistent snapshots is a subset of

ϵ_{app}-consistent snapshots. Therefore, $f_{correct} = f_{mon}$ and $precision = 1$ (no false positives). So we have:

Observation 3 *If $\epsilon_{app} \leq \epsilon_{mon}$, the recall of the monitor is 1.*

Observation 4 *If $\epsilon_{app} \geq \epsilon_{mon}$, the precision of the monitor is 1.*

2.5 Predicate Detection Algorithm by Garg and Chase [8]

The predicate detection algorithm used by the monitors in this paper is based on the algorithm by Garg and Chase [8] for detecting conjunctive predicates in programs in asynchronous systems ($\epsilon = \infty$). In this section, we briefly describe the predicate detection algorithm by Garg and Chase.

In [8], besides the n application processes, there are a set of n monitoring processes, which are collectively called **the monitor**, that perform the task of detecting when predicate $\mathcal{P}$ is satisfied. Each time the local predicate at process i becomes true, process i sends a message containing its state and the Vector Clock timestamp [7,15] of when its local predicate $\mathcal{P}_i$ became true to its corresponding monitoring process. Such a message is called a **candidate**. From the streams of candidates provided by the application processes, the monitor runs the predicate detection algorithm as follows:

S1. The monitor forms the initial global snapshot using the first candidates from each application process.
S2. If the global snapshot is *hb*-consistent, the monitor moves to step S3. If the global snapshot is not *hb*-consistent, the monitor moves to step S4.
S3. The global snapshot is *hb*-consistent and the monitor reports a satisfaction instance of predicate $\mathcal{P}$ and terminates.
S4. Since the global snapshot is not *hb*-consistent, there exist at least two candidates not concurrent with each other. Let $cand_i$ denote the candidate from process i currently being used in the global snapshot. There must exist two processes i and j such that $cand_i$ *hb* $cand_j$. The monitor replaces the current candidate with the next candidate from process i and then moves to step S2.

The presence of the monitor may also interfere with the execution of the application processes (e.g. transmission of information to the monitor, competing for resources in case the monitor shares physical machines with application processes). However, this paper focuses on the accuracy (precision/recall) of the monitor and not on the monitor's interaction with the application. We refer to [20] for an evaluation of the impact caused by similar monitors on the performance of the application processes. Since performance is not the focus of this paper, our results are unaffected by the choice of predicate detection algorithm used by the monitor.

3 Precision and Sensitivity of Asynchronous Monitors

In this section, we evaluate the precision and sensitivity of asynchronous monitors (that assume no bound for clock synchronization, $\epsilon = \infty$) when monitoring $\langle \epsilon, \delta \rangle$-programs (that assume bounded clock synchronization). Specifically, this section answers the following question: *Suppose we utilize an asynchronous monitor (designed for monitoring programs in asynchronous systems) in monitoring programs in partially synchronous systems, what is the precision, recall, and sensitivity of the monitor.* While the notion of sensitivity is explained in detail in Section 3.3.2, intuitively it characterizes how the precision/recall of the monitor changes when other parameters change. In Section 3.1 we present the parameters used in the analytical model. In Section 3.2 we discuss the predicate detection algorithm used by the monitor. Section 3.3 presents the analytical model to compute precision of asynchronous monitors in $\langle \epsilon, \delta \rangle$-programs. We validate the analytical model using simulations and experiments in Sections 3.5-3.6 based on the setup in Section 3.4.

3.1 Parameters for Analytical Model

Since we analyze the effectiveness of asynchronous monitors in $\langle \epsilon, \delta \rangle$-program, ϵ and δ are two parameters in our analytical model. We now identify other parameters considered in our model. These parameters are summarized in Table 1.

To generate the analytical model and simulations, we view the application/program execution as a run, where each process runs independently with respect to the constraint that clocks of two different processes never differ by more than ϵ. At each clock tick of the process [1], it sends a message with probability $\boldsymbol{\alpha}$. The destination is determined uniformly randomly from the remaining processes. With the constraint δ (message delay), the delivery time for this message is determined.

[1] We provide an interpretation of a clock tick in terms of actual elapsed time in Section 3.3.3. Furthermore, the exact discretization of the clock has a negligible effect on the computed precision/recall.

Thus, at each clock tick, the process also checks if there is a message intended to be received at that time. If so, it receives that message.

At each clock tick, each process i also changes its local predicate $\mathcal{P}_i$ to true with probability $\boldsymbol{\beta}$. We consider two types of runs: (1) point-based, where the decision to change $\mathcal{P}_i$ at each clock tick is independent of the decision at another clock tick, and (2) interval-based, where once the local predicate becomes true, it remains true for an interval of duration $\boldsymbol{\ell}$. And only after time ℓ, process i uses probability β to make its local predicate $\mathcal{P}_i$ true. We denote global predicates in the former runs as **point-based predicates** and in the latter runs as **interval-based predicates**. Finally, the number of processes n is also used in the analytical model.

3.2 Modification of Predicate Detection Algorithm in [8]

In this section, we describe how we extend the predicate detection algorithm by Garg and Chase (discussed in Section 2.5) for asynchronous monitors. We note that these extensions are also used by the partially synchronous monitors in Section 4.

The first modification, which enables the monitors to detect interval-based predicates, is that our predicate detection algorithm accounts for ℓ, the duration (or interval) for which the local predicates remain true. For example, consider the scenario in Figure 3, where the local predicate at process p0 becomes true at event $e1$ and continues to be true till event $e3$ and the local predicate at process p1 becomes true at event $f1$ and continues to be true till event $f2$. Their corresponding time intervals can be denoted as $[5, 20]$ and $[20, 35]$ at p0 and p1, respectively. Here if a snapshot containing events $e1$ and $f1$ is reported by the monitor, the snapshot may not seem to be ϵ-consistent (i.e. it may look like a false positive) if $\epsilon < 15$, because $e1$ and $f1$ are 15 time units apart from each other. However the local predicate at p0 is true from event $e1$ till event $e3$ at $t = 20$, so events $e3$ and $f1$ form a valid snapshot even if $\epsilon = 0$.

We handle the intervals where the local predicates are true as follows: Processes report intervals as candidates to the monitor. We run the algorithm in [8] with the initial events in the intervals. When a consistent snapshot is found in Step 3 (cf. Section 2.5), we obtain the candidates that correspond to an *hb*-consistent snapshot. Next, we use all pseudo events in the corresponding candidate intervals. If any of these pseudo events could lead to an ϵ-consistent snapshot, the monitor reports that snapshot.

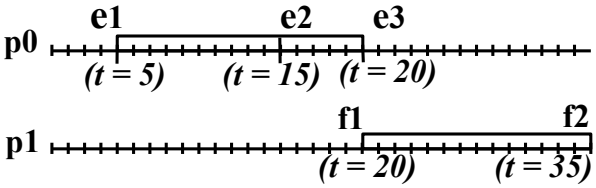


Fig. 3: Interval-based predicate detection. $\epsilon = 10$. Events e_1 and f_1 are not ϵ-consistent. e_3 and f_1 are ϵ-consistent.

The second modification deals with the scenario where the monitor in [8] terminates once it finds the snapshot where the given conjunctive predicate $\mathcal{P}$ is true. At this point, instead of terminating, our monitor starts to identify the next snapshot where $\mathcal{P}$ is true. In particular, the candidate with the smallest physical time in the current snapshot is replaced by the next candidate (from the same process), and the monitor moves to step S2 of the algorithm (cf. Section 2.5).

We note that since the above modifications are straightforward, the correctness of predicate detection in [8] is preserved.

3.3 Analytical Model

When a monitor designed for programs in asynchronous systems is used for programs in partially synchronous systems, it can result in false positives. In this section, we develop an analytical model to address the following question:

> **Problem Statement 1:** If we use a monitor that is designed for a program in asynchronous systems and apply it for an $\langle \epsilon, \delta \rangle$-program, what is the likelihood that it would result in a false positive?

Recall that a false positive occurs if the snapshot identified by the monitor is infeasible in an $\langle \epsilon, \delta \rangle$ program because the events in the snapshot are too far (more than ϵ) apart in physical time.

3.3.1 Precision

Recall that every snapshot found by an asynchronous monitor is an *hb*-consistent snapshot. An asynchronous monitor has recall = 1 because every ϵ-snapshot is *hb*-snapshot as stated in Observation 2. If an *hb*-consistent snapshot is also ϵ-consistent, the asynchronous monitor is correct (true positive). In the following results, we compute the precision of an asynchronous monitor when monitoring an $\langle \epsilon, \delta \rangle$-program.

For simplicity of analysis, in the analytical model, we ignore messages. Essentially, this means that α is 0 or very small. With simulation and experiments, we find that this assumption is reasonable in practice. In other words, in most settings, α has a small effect on the precision of the monitor. We discuss the impact of α in more detail in Section 3.5.3.

Theorem 1 *For an $\langle \epsilon, \delta \rangle$-program with n processes, where the local predicate $\mathcal{P}_i$ becomes true with probability β and remains true for duration ℓ, the probability that an hb-consistent snapshot where $\bigwedge_i \mathcal{P}_i$ is true is also ϵ-consistent is*

$$\phi(\epsilon, n, \beta, \ell) = (1 - (1 - \beta)^{\epsilon+\ell})^{n-1}$$

Proof Given a distributed program, for a long execution trace of the distributed program, it follows that the precision (true positive rate) will eventually converge to some value by the law of large numbers in probability theory.

Without loss of generality, assume that process 0 is one of the first processes to have their local predicates become true in the *hb*-consistent snapshot. Recall that $\mathcal{P}_i$ denotes the local predicate at process i. Let 0 be the time when $\mathcal{P}_0$ becomes true. Define the random variable x_i as the first time since time 0 that $\mathcal{P}_i$ $(1 \leq i \leq n-1)$ becomes true. Since the probability for $\mathcal{P}_i$ to be false at each clock tick is $1-\beta$, the probability that $\mathcal{P}_i$ does not become true before time t ($\mathcal{P}_i$ is false from time 0 to $t-1$) is $P(x_i \geq t) = (1-\beta)^t$. The probability of the complement event, i.e. $\mathcal{P}_i$ becomes true before time t, is $P(x_i < t) = 1 - (1-\beta)^t$.

Recall that for interval-based predicate, when a local predicate becomes true it remains true for a duration ℓ. Therefore, $\mathcal{P}_0$ remains true until time $\ell - 1$. For the *hb*-consistent global snapshot to be ϵ-consistent in interval-based predicate, any $\mathcal{P}_i$ $(1 \leq i \leq n-1)$ must become true before time $\epsilon + \ell$ so that the local state at process i is still ϵ-consistent with the local state at process 0. Furthermore, since every $\mathcal{P}_i$ becomes true no earlier than time 0, the local states of the processes will be mutually ϵ-consistent as long as the local predicates at the processes become true before time $\epsilon + \ell$. Therefore, we have:

$$\begin{aligned}\phi(\epsilon, n, \beta, \ell) &= \textstyle\prod_{i=1}^{n-1} P(x_i < \epsilon + \ell) \\ &= (1 - (1-\beta)^{\epsilon+\ell})^{n-1}\end{aligned}$$

□

Since point-based predicate is a special case of interval-based predicate where $\ell = 1$, the precision of an asynchronous monitor detecting point-based predicate in an $\langle \epsilon, \delta \rangle$-program denoted as $\phi(\epsilon, n, \beta) = \phi(\epsilon, n, \beta, 1)$, is provided in Corollary 1:

Corollary 1 *For point-based predicate, the probability of an hb-consistent snapshot being ϵ-consistent is*

$$\phi(\epsilon, n, \beta) = \phi(\epsilon, n, \beta, 1) = (1 - (1-\beta)^{\epsilon+1})^{n-1}$$

The formula in Theorem 1 suggests that the precision of the monitor will decrease when n (the number of processes) increases. The precision will increase when ℓ (the interval length for which local predicates remain true) increases or β (the probability that a local predicate becomes true) increases. Since the false positive rate = 1 − precision, the effects of these factors are reversed for the false positive rate.

3.3.2 Sensitivity

In addition to the value of the precision of the monitor, we want to evaluate how the value of precision changes with the value of ϵ, as it will characterize the sensitivity of the monitor if the precise value of ϵ is not known. Since the first derivative [2] of $\phi(\epsilon, n, \beta, \ell)$ with respect to ϵ (ϕ' – which is always positive – cf. Figure 4a) measures how the precision ϕ changes as the value of ϵ changes, the sensitivity of the monitor's precision is identified by this first derivative of ϕ – the monitor is sensitive (not sensitive, respectively) when the value of ϕ' is high (small, respectively) corresponding to the solid (dashed, respectively) portion of the red line in Figure 4a.

Figure 4b shows the first (ϕ'), second (ϕ'') and third (ϕ''') derivative of precision function ϕ with respect to ϵ. We note that the y-axis of ϕ' is on the left side of Figure 4b and the y-axis of ϕ'' and ϕ''' is on the right side of the graph. The two values ϵ_{p_1} and ϵ_{p_2} on the x-axis correspond to the values of ϵ where $\phi''' = 0$ (i.e. $\phi'''(\epsilon_{p_1}) = \phi'''(\epsilon_{p_2}) = 0$). On the curve of ϕ', A = $(\epsilon_{p_1}, \phi'(\epsilon_{p_1}))$ and B = $(\epsilon_{p_2}, \phi'(\epsilon_{p_2}))$. On the curve of ϕ'', C = $(\epsilon_{p_1}, \phi''(\epsilon_{p_1}))$ and D = $(\epsilon_{p_2}, \phi''(\epsilon_{p_2}))$. From this figure, we observe that the curve for ϕ' can be partitioned into three phases from left to right:

(1) The first phase on the left where ϕ'' (the first derivative of ϕ') is positive and increases from 0 until it reaches the maximum (point C when $\epsilon = \epsilon_{p_1}$). In this phase ϕ' increases from 0 at an accelerating rate but its value is small.

[2] We use ϕ', ϕ'', and ϕ''' to denote the first ($\frac{\partial \phi}{\partial \epsilon}$), second ($\frac{\partial^2 \phi}{\partial \epsilon^2}$), and third ($\frac{\partial^3 \phi}{\partial \epsilon^3}$) partial derivative of ϕ with respect to ϵ.

(2) The second phase at the middle where ϕ'' decreases from its positive maximum (point C) to 0 and keeps decreasing to its negative minimum (point D) when $\epsilon = \epsilon_{p_2}$. Thus, ϕ' increases (from point A) at a slower rate until it reaches its maximal value, then decreases to point B but its value is still high.
(3) The last phase on the right where ϕ'' is negative and increases from its minimum (point D) to 0. Accordingly, ϕ' keeps decreasing at a slower rate until it vanishes.

The region of high sensitivity is associated with the middle phase where the value of ϕ' is high (this region is highlighted by solid lines in Figure 4). The boundary point where the transition between the phases of ϕ' occurs are called the inflection points of ϕ', denoted as ϵ_{p_1} and ϵ_{p_2}. They are the points where ϕ'' (the green line in Figure 4b) changes its direction, or ϕ''' (the purple line in Figure 4b) equals 0.

The following lemma identifies the position of the two inflection points of ϕ'.

Lemma 1 (Inflection points) *Assume that $n > 1$. For interval-based predicates, the two inflection points ϵ_{p_1} and ϵ_{p_2} (with $\epsilon_{p_1} < \epsilon_{p_2}$) of the function $\frac{\partial\phi(\epsilon,n,\beta,\ell)}{\partial\epsilon}$ are*

$$\epsilon_{p_1} = \log_{(1-\beta)}\left(\frac{3n-4+\sqrt{5n^2-16n+12}}{2(n-1)^2}\right) - \ell$$
$$\epsilon_{p_2} = \log_{(1-\beta)}\left(\frac{3n-4-\sqrt{5n^2-16n+12}}{2(n-1)^2}\right) - \ell$$

Proof An inflection point of the slope $\frac{\partial\phi(\epsilon,n,\beta,\ell)}{\partial\epsilon}$ is where its second derivative (which is the third derivative of $\phi(\epsilon,n,\beta,\ell)$) equals 0.

With $B = 1-\beta$, we obtained the following partial derivatives of $\phi(\epsilon,n,\beta,\ell)$ with respect to ϵ:

$$\begin{aligned}
\phi &= (1-B^{\epsilon+\ell})^{n-1}\\
\phi' &= -(n-1)\ln B(1-B^{\epsilon+\ell})^{(n-2)}B^{\epsilon+\ell}\\
\phi'' &= -(n-1)(\ln B)^2(1-B^{\epsilon+\ell})^{(n-3)}\times\\
&\quad (B^{\epsilon+\ell} - nB^{2(\epsilon+\ell)} + B^{2(\epsilon+\ell)})\\
\phi''' &= -(n-1)(\ln B)^3(1-B^{\epsilon+\ell})^{(n-4)}B^{\epsilon+\ell}\times\\
&\quad [(n^2-2n+1)B^{2(\epsilon+\ell)} - (3n-4)B^{\epsilon+\ell} + 1]
\end{aligned}$$

We know that $0 < B = 1-\beta < 1$ (since $0 < \beta < 1$) and $n > 1$. Therefore $\phi''' = 0$ iff

$$(n^2-2n+1)B^{2(\epsilon+\ell)} - (3n-4)B^{\epsilon+\ell} + 1 = 0$$

With $y = B^{\epsilon+\ell}$, we have

$$(n^2-2n+1)y^2 - (3n-4)y + 1 = 0$$

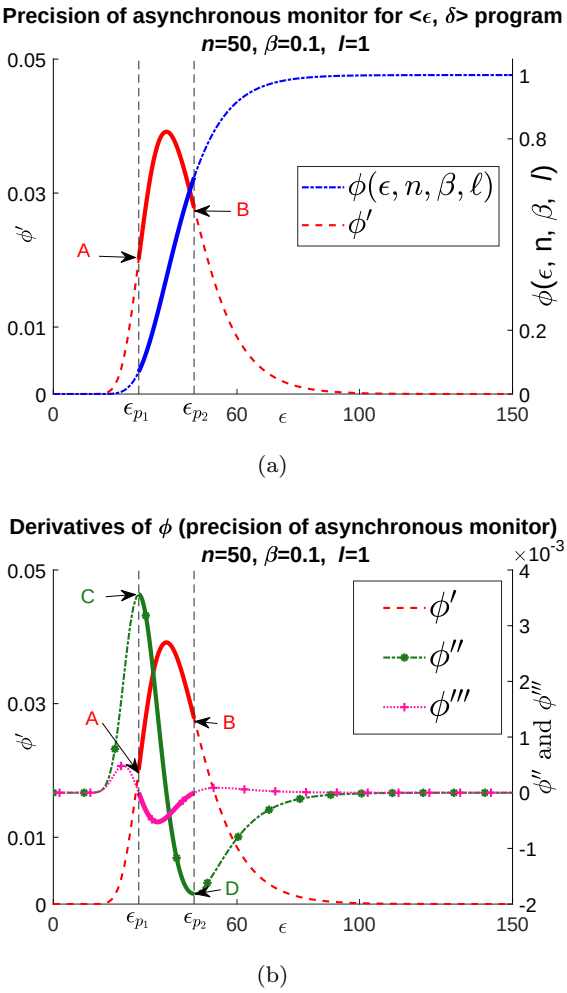


Fig. 4: A sample graph of $\phi(\epsilon,n,\beta,\ell)$ – the precision function of asynchronous monitor for $\langle\epsilon,\delta\rangle$-program – and its derivatives with respect to ϵ. Figure 4a: ϕ and its first derivative. Figure 4b: the first, second, and third derivatives of ϕ. The region between two inflection points (ϵ_{p_1} and ϵ_{p_2}) are highlighted with solid lines.

The solutions of this quadratic equation are

$$\{y_1, y_2\} = \frac{3n-4\pm\sqrt{5n^2-16n+12}}{2(n-1)^2}$$

By substituting $\epsilon = log_B y - \ell$ and $B = 1-\beta$, we obtain the values of ϵ_{p_1} and ϵ_{p_2} as stated in Lemma 1. □

In case of point-based predicate where $\ell = 1$, the inflection points of $\frac{\partial\phi(\epsilon,n,\beta)}{\partial\epsilon}$ are provided in Corollary 2:

Corollary 2 (Point-based inflection points) *Assume that $n > 1$. For point based predicates, the two inflection points ϵ_{p_1} and ϵ_{p_2} (with $\epsilon_{p_1} < \epsilon_{p_2}$) of the*

function $\frac{\partial\phi(\epsilon,n,\beta)}{\partial\epsilon}$ *are*

$$\epsilon_{p_1} = \log_{(1-\beta)}(\frac{3n-4+\sqrt{5n^2-16n+12}}{2(n-1)^2}) - 1$$
$$\epsilon_{p_2} = \log_{(1-\beta)}(\frac{3n-4-\sqrt{5n^2-16n+12}}{2(n-1)^2}) - 1$$

The two inflection points ϵ_{p_1} and ϵ_{p_2} partition the domain of ϵ into three ranges: $(0, \epsilon_{p_1})$, $[\epsilon_{p_1}, \epsilon_{p_2}]$, and (ϵ_{p_2}, ∞). In the ranges $(0, \epsilon_{p_1})$ and (ϵ_{p_2}, ∞), the rate of change of precision (i.e., ϕ') is small and the precision (i.e., ϕ) is relatively insensitive to changes in ϵ. In the range $[\epsilon_{p_1}, \epsilon_{p_2}]$, ϕ' is high and the value of ϕ changes significantly as ϵ changes. In other words, except in the range $[\epsilon_{p_1}, \epsilon_{p_2}]$, we can compute the precision of the asynchronous monitor with only approximate knowledge of ϵ. Specifically, if we believed that the value of ϵ equals x, but the actual value of ϵ is $x \pm \Delta x$ and $[x, x \pm \Delta x]$ does not overlap with $[\epsilon_{p_1}, \epsilon_{p_2}]$, then the precision computed by the model for $\epsilon = x$ would be close to the actual precision for $\epsilon = x \pm \Delta x$. As a result, if the high sensitivity range $[\epsilon_{p_1}, \epsilon_{p_2}]$ is small, it is more likely to get a good estimate of the precision of the asynchronous monitor. The next theorem shows that the *relative gap* between the two boundaries of this range approaches zero as the number of processes n increases.

Theorem 2 (Relative high sensitivity range) *The relative width of the high sensitivity range (the relative difference between phase transition* ϵ_{p_1} *and post-phase transition* ϵ_{p_2}*) of an asynchronous monitor when monitoring an* $\langle\epsilon, \delta\rangle$*-program with* n *processes approaches* 0 *as* n *increases (and this is independent of* β*, the probability of a local predicate becoming true):*

$$\lim_{n\to\infty} \frac{\epsilon_{p_2} - \epsilon_{p_1}}{\epsilon_{p_1}} = 0$$

Proof The equation in Theorem 2 is equivalent to $\lim_{n\to\infty} \frac{\epsilon_{p_2}}{\epsilon_{p_1}} = 1$. Since $\beta < 1$ and $n > 1$, we observe that both ϵ_{p_1} and ϵ_{p_2} are non-zero and differentiable with respect to n. By l'Hôpital's rule, we have

$$\lim_{n\to\infty} \frac{\epsilon_{p_2}}{\epsilon_{p_1}} = \lim_{n\to\infty} \frac{(\epsilon_{p_2})'}{(\epsilon_{p_1})'}$$

where $(\epsilon_{p_1})'$ and $(\epsilon_{p_2})'$ are the first derivatives of ϵ_{p_1} and ϵ_{p_2} with respect to n, respectively. Denote $A = \sqrt{5n^2-16n+12}$, we obtained $(\epsilon_{p_1})'$ and $(\epsilon_{p_2})'$ (we have omitted the intermediate steps for brevity) as below:

$$(\epsilon_{p_2})' = \frac{-6An + 10n^2 - 38n + 10A + 32}{2\ln(1-\beta)A(n-1)(3n-4-A)}$$
$$(\epsilon_{p_1})' = \frac{-6An - 10n^2 + 38n + 10A - 32}{2\ln(1-\beta)A(n-1)(3n-4+A)}$$
$$\lim_{n\to\infty} \frac{\epsilon_{p_2}}{\epsilon_{p_1}} = \lim_{n\to\infty} \frac{(\epsilon_{p_2})'}{(\epsilon_{p_1})'} =$$
$$\lim_{n\to\infty} \left(\frac{(-6An + 10n^2 - 38n + 10A + 32)}{(-6An - 10n^2 + 38n + 10A - 32)} \frac{(3n-4+A)}{(3n-4-A)} \right)$$

Substitute A with $\sqrt{5n^2-16n+12}$, then multiply both the numerator and the denominator of the first fraction with $\frac{1}{n^2}$, and of the second fraction with $\frac{1}{n}$, we have:

$$\lim_{n\to\infty} \frac{\epsilon_{p_2}}{\epsilon_{p_1}} = \lim_{n\to\infty} \frac{(\epsilon_{p_2})'}{(\epsilon_{p_1})'} = \lim_{n\to\infty} \Bigg(\frac{-6\sqrt{5-\frac{16}{n}+\frac{12}{n^2}} + 10 - \frac{38}{n} + \frac{10}{n}\sqrt{5-\frac{16}{n}+\frac{12}{n^2}} + \frac{32}{n^2}}{-6\sqrt{5-\frac{16}{n}+\frac{12}{n^2}} - 10 + \frac{38}{n} + \frac{10}{n}\sqrt{5-\frac{16}{n}+\frac{12}{n^2}} - \frac{32}{n^2}} \times \frac{(3-\frac{4}{n}+\sqrt{5-\frac{16}{n}+\frac{12}{n^2}})}{(3-\frac{4}{n}-\sqrt{5-\frac{16}{n}+\frac{12}{n^2}})} \Bigg)$$
$$= \frac{(-6\sqrt{5}+10)\times(3+\sqrt{5})}{(-6\sqrt{5}-10)\times(3-\sqrt{5})} = \frac{-8\sqrt{5}}{-8\sqrt{5}} = 1.$$

□

3.3.3 Illustration and Interpretation of the Analytical Results

In this subsection, we illustrate how the analytical results above can be used in practice. In order to do so, we need to convert between the parameter values in practice and the corresponding values in the analytical model. In particular, we consider the conversion of these parameters: $\alpha, \beta, \delta, \ell, \epsilon$.

The conversion of these parameters is based on an implicit parameter: clock granularity – the time duration of each clock tick. The clock granularity is a free parameter with only one constraint: in our model, since the completion of every operation takes one or multiple clock ticks, the clock granularity should be reasonably small.

Consider a distributed program with $n = 5$ processes where their local predicates become true every 10 ms and remain true for 1 ms, the processes send out messages every 2 ms on average, and the messages are delivered after 5 ms. Let clock granularity be 1 tick per µs. Since the local predicates become true every 10,000 clock ticks and remain true for 1,000 clock ticks, we

have $\beta = 0.0001$ and $\ell = 1,000$. The average communication frequency is one message sent every 2,000 clock ticks and $\alpha = 0.0005$. The message delay is $\delta = 5,000$ clock ticks. Using the formula in Lemma 1, we obtain two inflection points $\epsilon_{p_1} = 4,452$ clock ticks (4.45 ms) and $\epsilon_{p_2} = 21,272$ clock ticks (21.27 ms). Our analytical model predicts that the monitor precision will be low if the application assumption ϵ_{app} is less than or equal 4 ms, and high if the application assumption ϵ_{app} is higher than or equal 22 ms. Suppose $\epsilon_{app} = 4$ ms (4,000 clock ticks), using the formula in Theorem 1, we calculate that the precision of the asynchronous monitor is 0.02. If $\epsilon_{app} = 30$ ms (30,000 clock ticks), the monitor precision is 0.83.

To understand the notion of sensitivity, consider the case where the exact value ϵ_{app} is not known. If, however, we know that ϵ_{app} is between 30 ms and 40 ms, then we can compute the monitor precision to be between 0.83 and 0.94; were ϵ_{app} instead between 10 ms and 20 ms, then the monitor precision would be between 0.20 and 0.59. In other words, in $[\epsilon_{p_1}, \epsilon_{p_2}]$ the change in precision is large.

Moreover, the relative width of the high sensitivity range $\frac{\epsilon_{p_2} - \epsilon_{p_1}}{\epsilon_{p_1}} = 3.78$. When the number of processes n increases to 500 (other parameters are unchanged), the two inflection points shift upward to 51.5 ms and 70.7 ms while the relative width of the high sensitivity range reduces to 0.37 (cf. Table 2), which is comparable with the results in Theorem 2.

In Table 2, we also change the clock granularity to 100 µs and to 1 ns, and observe that the calculated values are practically unchanged. We note that the analytical model is designed with the granularity as small as possible. However, even when the granularity is 1 µs or 100 µs, the predicted values of precision and ϵ_{p_1}, ϵ_{p_2} remain (essentially) unchanged.

Input parameters					Calculation			
n	granularity	β	ℓ	ϵ_{app}	ϵ_{p_1}	ϵ_{p_2}	$\frac{\epsilon_{p_2} - \epsilon_{p_1}}{\epsilon_{p_1}}$	precision
	µs	per ms	ms	ms	ms	ms		
5	1	0.1	1	4	4.45	21.27	3.78	0.02
5	1	0.1	1	10	4.45	21.27	3.78	0.20
5	1	0.1	1	20	4.45	21.27	3.78	0.59
5	1	0.1	1	30	4.45	21.27	3.78	0.83
5	1	0.1	1	40	4.45	21.27	3.78	0.94
500	1	0.1	1	80	51.51	70.74	0.37	0.86
5	100	0.1	1	30	4.43	21.16	3.78	0.83
5	0.001	0.1	1	30	4.45	21.27	3.78	0.83

Table 2: Illustration of using the analytical model to predict the precision and inflection points of asynchronous monitors.

3.4 Experimental and Simulation Setup

We validated our analytical model using experiments and simulations. The code and the raw results are available at [18]. The parameters ϵ and δ in the experiments are measured from the system and the network, while in the simulations they are configured as input parameters. The values of α, β, ℓ, n are configured as input parameters in both the experiments and the simulations as described below. When running experiments, it is difficult to control the exact value of α, β or ℓ. We describe how we identify these parameters for AWS experiments later in section 3.6.

For the experiments, we implemented a distributed application and a monitor to detect satisfaction of a conjunctive predicate in the application. The application consists of a main loop in which each process (1) sends a message with some probability, (2) receives any messages sent to it, (3) sets the local predicate to be true with a certain probability, where it stays true for a certain duration and (4) sleeps for duration *lsct* (local sleep computation time) to simulate local computation. Since a predicate is inherently true for some duration of time that is longer than a clock tick in the experiment, it is not possible to conduct point-based experiments (where it is required that when a local predicate is set to true it stays true for one clock tick). However, by setting *lsct* as small as possible, the interval-based experiments approximate point-based scenarios. If there is communication in the middle, the interval will be split as required by the algorithm in [8]. The monitor implements the conjunctive predicate detection algorithm described in [8]. We use 5 Amazon AWS EC2 t2.micro machines located at different regions (Ohio-USA, North Virginia-USA, California-USA, Oregon-USA, and Central Canada). The machines run Ubuntu 18.04 operating system. The clock skew between the AWS machines synchronized using NTP protocol [16] (measured by `ntpq -p`) is between 17 ms and 20 ms. In our experiment, the average message delays between AWS regions and within an AWS region (measured by `ping` utility) are 29 ms and 0.6 ms, respectively.

Besides the experiments, we also use simulations due to several benefits: (1) the simulations support both

point-based and interval-based scenarios while point-based predicates are not feasible in experiments, (2) simulations allow control of network latency δ, (3) we observe that the simulation results are consistent with the experimental results while it is easier to deploy and faster to obtain results from the simulations than from the experiments.

In our simulations, in a step, with a certain probability, a process chooses to advance its clock as long as the synchrony requirement is not violated. If a process does not advance its clock at the given step then nothing happens at that process. By allowing a subset of processes to take action in one step, we are able to create scenarios where clocks of different processes advance at different speeds.

When a process increments its clock, it decides if the local predicate is true with probability β. Depending on the type of detection i.e. point-based or interval-based, the local predicate will remain true for just one instant (one clock tick) or for a duration of time ℓ. Furthermore, when a process advances its clock, it can choose to send a message to a randomly selected process with probability α. The delivery time of this message will be determined by δ.

During a simulation/experiment run, we identify f_{mon}, the number of snapshots identified by the asynchronous monitor algorithm in [8], and f_ϵ, the number of snapshots that are also ϵ-consistent. Then, the precision of the monitor is $\frac{f_\epsilon}{f_{mon}}$ and the false positive rate is $1 - \frac{f_\epsilon}{f_{mon}}$.

For the simulation length, we run until each process advances its clock to $100,000$ so that the false positive rate (and the precision) stabilizes. In particular, when a new snapshot is identified by the algorithm in [8] indicating that the given conjunctive predicate is possibly true, the snapshot may or may not be consistent with synchrony requirements (i.e., may or may not be a false positive). Hence, initially, the false positive rate varies substantially. However, when sufficiently many snapshots are identified over time the false positive rate converges to a stable value. To validate this, we considered how the false positive rates vary over time in different simulations. Figure 5 shows the results for different values of ϵ while n, δ, α and β are fixed. From these results, we find that the false-positive rate stabilizes fairly quickly. When we vary n, δ, α and β, we also observed a similar stabilizing pattern. Whereas in the experiments, we observed that the false positive rates converge to stable values after running for 10 minutes.

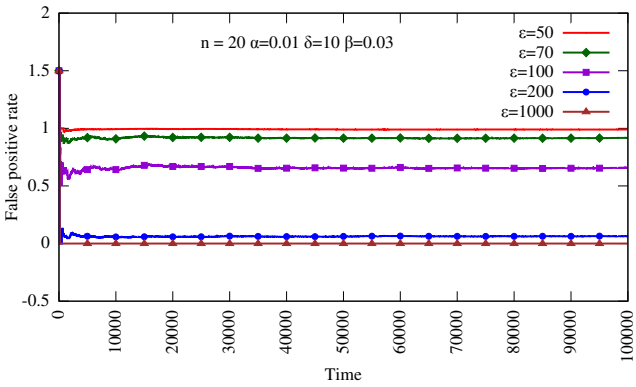


Fig. 5: (Simulation) Point-based asynchronous monitors: convergence of false positive rates over time. Since precision = 1 − false positive rate, the precision also converges.

3.5 Simulation Results

3.5.1 Point-based Predicate Simulations

Comparison of analytical model and simulation results. The analytical model in section 3.3 predicts that the false positive rate of an asynchronous monitor has the shape of a logistic function. In Figure 6a, we plot the graph of false positive rate (which is $1 - \phi(\epsilon, n, \beta)$) with $n = 20, \beta = 0.1$ (the continuous red curve). We run simulations with the same n, and β but vary the values of α, δ, ϵ, and plot simulation results in the same figure. As shown in Figure 6a, the simulation results (the continuous curves with different markers) agree with the analytical results (the continuous curve in red).

Sensitivity of the false positive rate of asynchronous monitors to changes in ϵ. The blue dotted line in Figure 6a is the derivative of a simulation result (with $\alpha = 0.1, \delta = 100$), which illustrates how sensitive the precision (i.e. 1 - false positive rate) of an asynchronous monitor is with respect to changes in the clock skew ϵ. We observe that the values of ϵ can be divided into 3 ranges: a brief range of high false positives to the left when ϵ is small ($\epsilon < \epsilon_{p_1}$), a range of low false positives to the right when ϵ is large ($\epsilon > \epsilon_{p_2}$), and a short high sensitivity range $[\epsilon_{p1}, \epsilon_{p2}]$ in the middle where small changes in the clock skew ϵ changes the false positive rates significantly, as anticipated by the analytical model.

Effect of α and δ. In Figure 6a we consider the false positive rates for $n = 20, \beta = 0.10$. We consider different values of $\alpha = 0.05$, 0.1 and $\delta = 10$, 100 and compare the simulation results with the analytical model. The simulation results validate the analytically

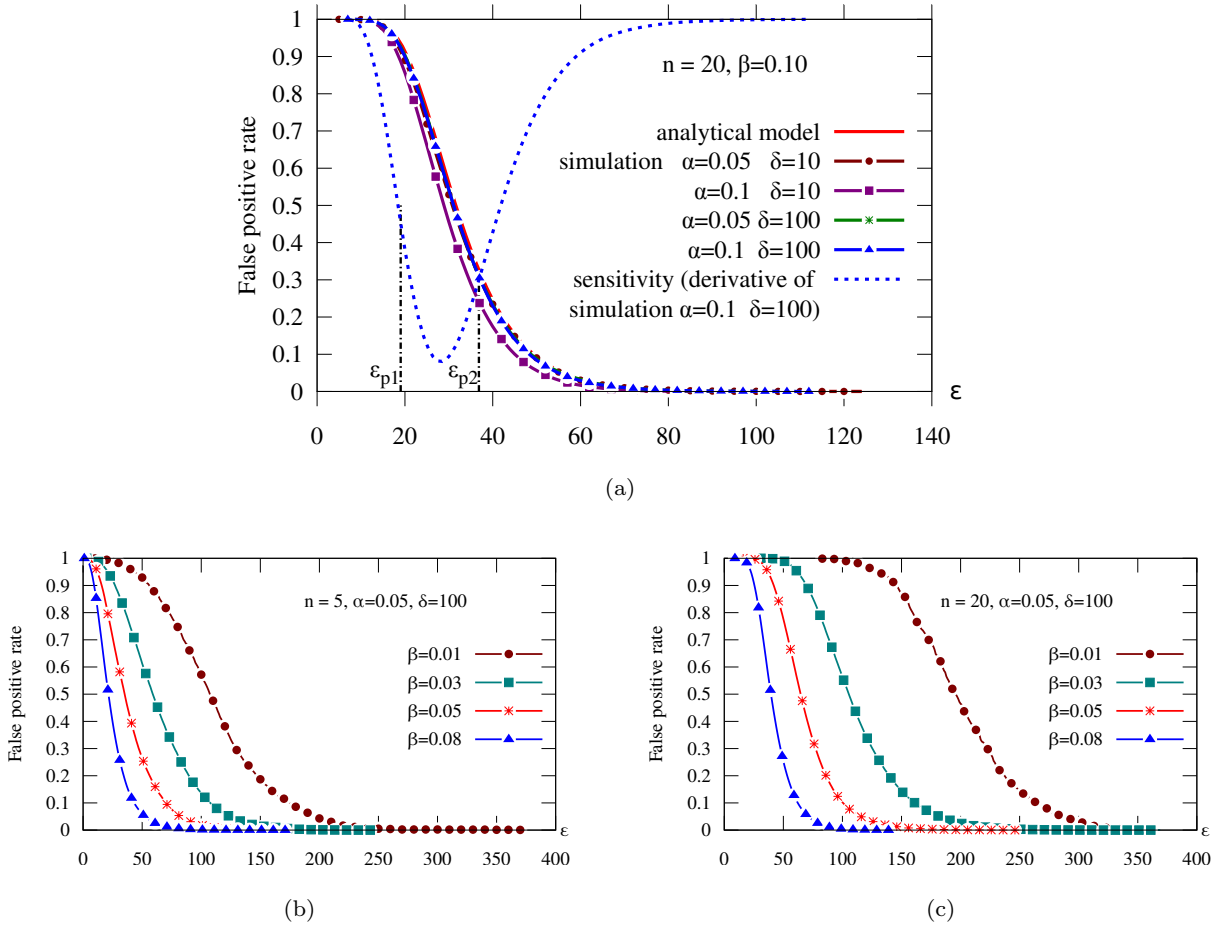


Fig. 6: (Simulation) Point-based asynchronous monitors. Figure 6a: Comparison of analytical and simulation results, sensitivity of false positive rate to changes in ϵ, the independence of false positive rate from α and δ. Figures 6b, 6c: Impact of β and n on false positive rates

computed false positive rate. Also, we find that the false positive rate is (almost) independent of α and δ.

Effect of β. In the analytical model, when the local predicates at the processes become true rarely (value of β is close to 0), the predicted false positive rate is 1. And, as the local predicates become true frequently (β approaches 1), the false positive rate approaches 0 (the false positive rate is 1 − precision). We validate this result with Figure 6c. When considering a network of 20 processes, and β is small, say 0.01, the false positive rate at $\epsilon = 120$ is 97%. By contrast, if β is increased to 0.05 and 0.08 then the false positive rate at $\epsilon = 120$ decreases to 4% and 0.08% respectively.

Effect of n. The analytical model predicts that when n increases, the false positive rate increases. This predication is confirmed in Figures 6b and 6c. Let $\beta = 0.01$, when n is small, say 5, the false positive rate at $\epsilon = 120$ is 38.2%. If n is increased to 20 then the false positive rate increases to 97%.

3.5.2 Correlated Point-based Predicate Simulations

In the analytical model in Section 3.3 and in the simulation in Section 3.5.1, it was assumed that the probability of a local predicate $\mathcal{P}_i$ being true was independent of local predicate $\mathcal{P}_j$ being true for any two different processes i and j. In this section, we consider the case where the probability of the local predicate being true on different processes is correlated. While we analyze some specific approaches to consider correlation below, we note that our analysis technique is useful for several other correlations as well.

In the first correlation called *PMA* (Positively correlated with MAjority), the processes are divided into 2 groups. Each process in the first group of size $G_1 < n$ generates local predicates independently with the same base rate β at each clock tick. A process in the second group of size $G_2 = n - G_1$ has 2 possibilities: either (1) with probability P_{dep}, it follows the majority of the first group or (2) with probability $P_{ind} = 1 - P_{dep}$,

it chooses the truth value independently by itself with rate β. The values of G_1, G_2, P_{dep} (thus, P_{ind} as well) are configurable.

In a rough estimation of the false positive rate in the *PMA* model, we observe that given that the local predicates in group G_1 are close enough in a snapshot, the chance for the snapshot to be a false positive would depend on whether the local predicates in the second group G_2 are close enough to the first group G_1 or not. This would, in turn, depend on cases where local predicates in the group G_2 are independently generated (if they are dependently generated, they would be close to the predicates of the first group). The probability that the instant at which the local predicate independently generated by a process in group G_2 becomes true is at least t apart from the instant at which the local predicate becomes true for a process in G_1 follows a geometric distribution which is $(1-\beta_{ind})^t$ where $\beta_{ind} = P_{ind} * \beta$. Hence, the probability that all processes of the second group G_2 are within the ϵ distance from the first group is roughly $\phi_{PMA}(\epsilon, n, \beta) = 1 - (1 - (1-\beta_{ind})^{\epsilon})^{|G_2|}$, where $|G_2|$ is the number of processes in the group G_2. As shown in Figure 7a, when half of the processes are in G_2 and $P_{ind} = 0.5$ (thus $\beta_{ind} = \frac{\beta}{2}$), the monitor precision for *PMA* is $\phi_{PMA}(\epsilon, n, \beta) \approx \phi(\epsilon, \frac{n}{2}, \frac{\beta}{2})$, which fits with the simulation.

The above analysis was for the case where $|G_2| = \frac{n}{2}$ and $P_{ind} = 0.5$. We note that we have performed similar analysis for other values of G_2 and P_{ind} and the simulation results matched the expected analytical results. For example, when $(|G_2|, P_{ind}) = (\frac{2}{3}n, 0.5)$, it corresponds to $\phi(\epsilon, \frac{2}{3}n, \frac{\beta}{2})$. And, when $(|G_2|, P_{ind}) = (\frac{n}{4}, 0.25)$, it corresponds to $\phi(\epsilon, \frac{n}{4}, \frac{\beta}{4})$.

We also consider other correlation models for local predicates of the processes such as *HNMA* (Half Negatively correlated with MAjority) and *PMAJ* (Positively correlated with MAjority up to index J). The *HNMA* model is the similar to *PMA* where $G_1 = \frac{n}{2}, P_{dep} = 0.5$ with one exception: processes in the second group would follow the minority of the first group. In the *PMAJ* model, process 0 chooses whether its local predicate is true with probability β. The truth value of local predicate at every remaining process is correlated with local predicates in its preceding processes (w.r.t. process ID). In particular, each process j will follow the majority of its preceding processes (i.e. processes $0, ..., j-1$) with probability of 0.5; with probability of 0.5, it will change its local predicate on its own with probability β. As shown in Figure 7b, 7c, there are parameters that help our analytical model to estimate the simulation results of these correlation models under different parameter settings (e.g. n, β). For example, when we replace (n, β) in the analytical model (Section 3.3, Corollary 1) by $(n, \frac{\beta}{2})$ (respectively, $(\frac{n}{4}, \frac{\beta}{2})$), the false positive rate predicted by the analytical model matches with the false positive rate of the *HNMA* (respectively, *PMAJ*) correlation model obtained by simulations.

3.5.3 Interval-based Predicate Simulations

Point-based scenarios could be generalized to interval-based scenarios where local predicates are true for a certain interval of time, ℓ. In any practical program, the value of local predicate $\mathcal{P}_i$ is not changing at every clock tick. Thus, for these programs, local predicates are expected to be true for an interval.

Simulation results in Figure 8 show that the false positive rate for interval-based predicate detection increases when β decreases (Figure 8b), or n increases (Figure 8d), or interval length ℓ decreases (Figure 8e). The false positive rate is independent of δ (Figure 8c). These observations are compatible with the analytical model for asynchronous monitors.

For simplicity, we omitted the effect of α in Theorem 1. We discuss it here. Revisiting the proof of Theorem 1, let 0 be the time at which $\mathcal{P}_0$ became true. Let e denote this event. As done in the proof of Theorem 1, let e be the earliest event in the *hb*-consistent snapshot. Let $t_1 \geq 0$ be the time when $\mathcal{P}_1$ became true. Let f denote the corresponding event. Theorem 1 evaluated the case where $t_1 \leq \epsilon$ given that $t_1 \geq 0$. If the message rate is 0 then the value of t_1 in the *hb*-consistent snapshot can be any non-negative value. On the other hand, if the message rate is very high, t_1 would need to be smaller, as for larger values of t_1 the probability of e happened before f increases. In other words, as α increases, the chances of t_1 being closer to 0 increases. This increases the probability of the *hb*-consistent snapshot also being ϵ-consistent (i.e. the precision is increased). On the other hand, as required in [8], sending of a message causes intervals to be split. The splitting of intervals reduces the *effective* length of ℓ. As discussed in the analytical and simulation results above, reduced ℓ could reduce the precision. In practice, we observe that the overall effect of α is small (cf. Figure 8a).

3.6 AWS Based Experimental Results

Figure 9a presents our experimental results for interval-based asynchronous monitors on Amazon AWS platform. To compare with the simulation and the analytical model, we need to identify the values of $\alpha, \beta, \delta, \ell$, and ϵ. We achieve this as follows. Value of α is determined by evaluating the number of messages sent in a given time interval; α is chosen so that the number

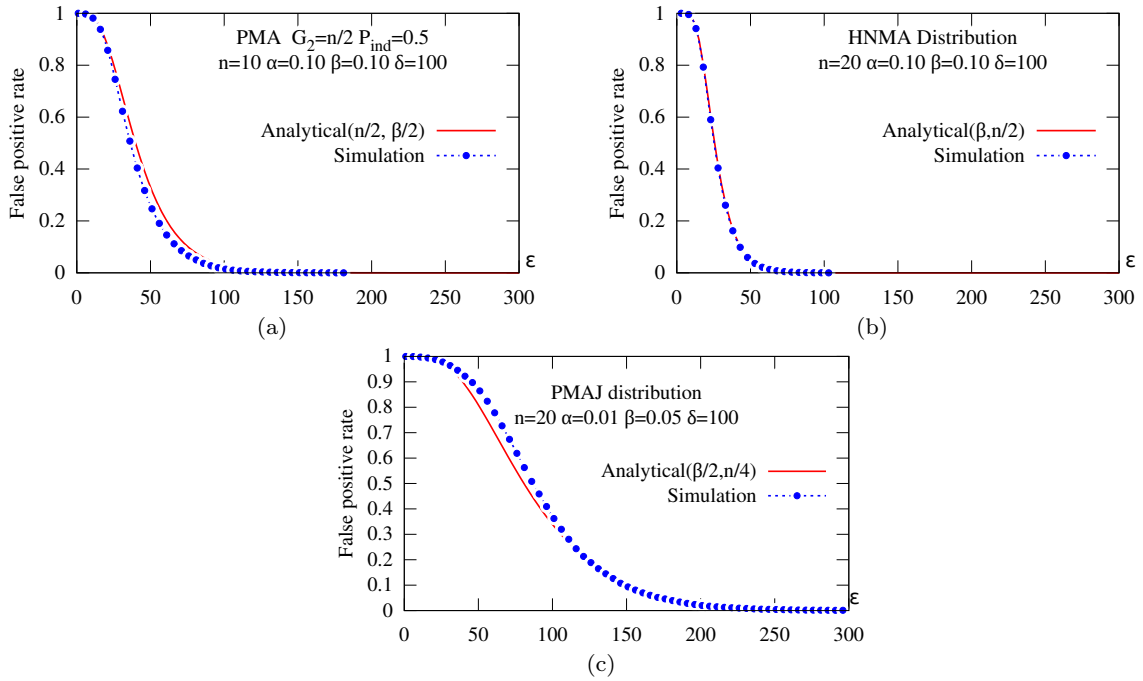


Fig. 7: (Simulation) Asynchronous monitors: false positive rates in some correlated point-based scenarios

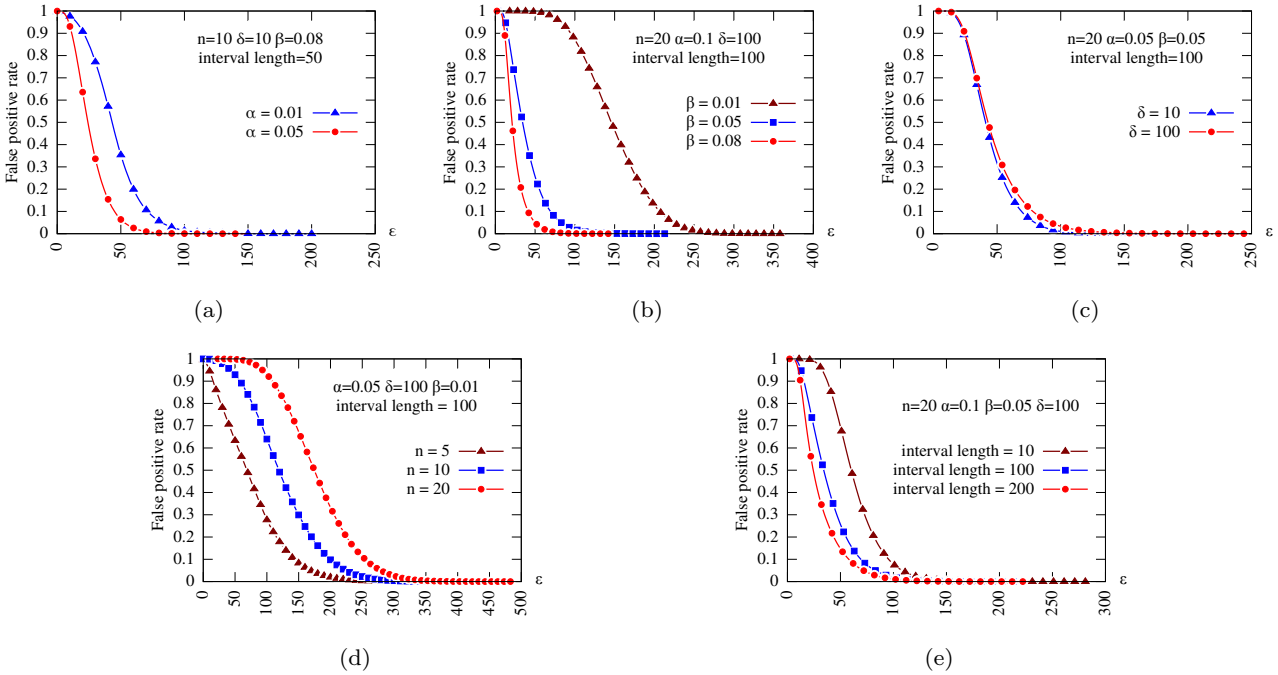


Fig. 8: (Simulation) Interval-based asynchronous monitors: Impact of α, β, δ, n, and ℓ on false positive rate.

of expected messages in that interval matches the actual number of messages sent during the experiment. Likewise, β is computed by observing the number of times the local predicate became true in the given interval during the experiment and choosing β such that it matches this number. Value of δ is obtained by values reported by ping service. And, we measured the average time for which the predicate was actually true and determine the (effective) value of ℓ. Once we obtained the run, we evaluated it by identifying which snapshots

would have been reported for different values of clock skew ϵ.

As shown in Figure 9a, the prediction based on our analytical model is compatible with AWS experimental results and simulation results. The dotted line in Figure 9a is the derivative with respect to ϵ of the false positive rate obtained in AWS experiments, which informs the sensitivity of asynchronous monitor precision with respect to changes in clock skew. We also observe the high sensitivity range $[\epsilon_{p1}, \epsilon_{p2}]$ where a small change in ϵ causes a large difference in the precision of the monitors.

We also note that the derivative in Figure 9a has been smoothed using Bézier curve since the original derivative of experimental data contains noises. In Figure 9b, we compare the original (non-smoothed) derivative (the green dotted curve) and the smoothed derivative (the red curve) and observe that the smoothed version represents the rate of change in the experimental data (the blue curve) well.

The sensitivity of asynchronous monitor precision to different factors such as α, β, δ, n, ℓ is illustrated in Figure 10. To quantify the impact of δ, we run one set of experiments where all machines are located in the same AWS region (the average latency is 0.6 ms) and another set of experiments where all machines are placed in different AWS regions (the average latency is 29 ms). We observe that the false positive rate of an asynchronous monitor increases when β decreases (Figure 10b), or n increases (Figure 10d), or ℓ decreases (Figure 10e). The false positive rate is independent of δ (Figure 10c) and *lsct* (Figure 10f). These results are compatible with the analytical model and simulation results. Figure 10a shows that α has a neutral impact on the false positive rate.

Note that *lcst* corresponds to the scenario where the process is only performing local computations. We can view process execution to consist of (1) interactive phase (where it is involved in inter-process communication) and (2) local phase (where it does not send/receive any messages). While such phases are not part of the analytical model, experimental result shows that having such phases does not affect the overall precision/recall of the monitor.

4 Precision, Recall, and Sensitivity of Partially Synchronous Monitors

In this section, we focus on the following problem:

> **Problem Statement 2:** Suppose we designed a monitor for an $\langle \epsilon_{mon}, \delta_1 \rangle$-program and applied it in an $\langle \epsilon_{app}, \delta_2 \rangle$-program, then what is the precision and recall of the monitor?

In Section 4.1, we discuss the predicate detection algorithm used by partially synchronous monitors. Section 4.2 presents the analytical model for the precision and recall of partially synchronous monitors in $\langle \epsilon_{app}, \delta_2 \rangle$-program. Sections 4.3-4.5 present the simulation and experiment results. For partially synchronous monitors, the parameters used by the analytical model and the experimental and simulation setup are the same as those for asynchronous monitors (cf. Sections 3.1, 3.4).

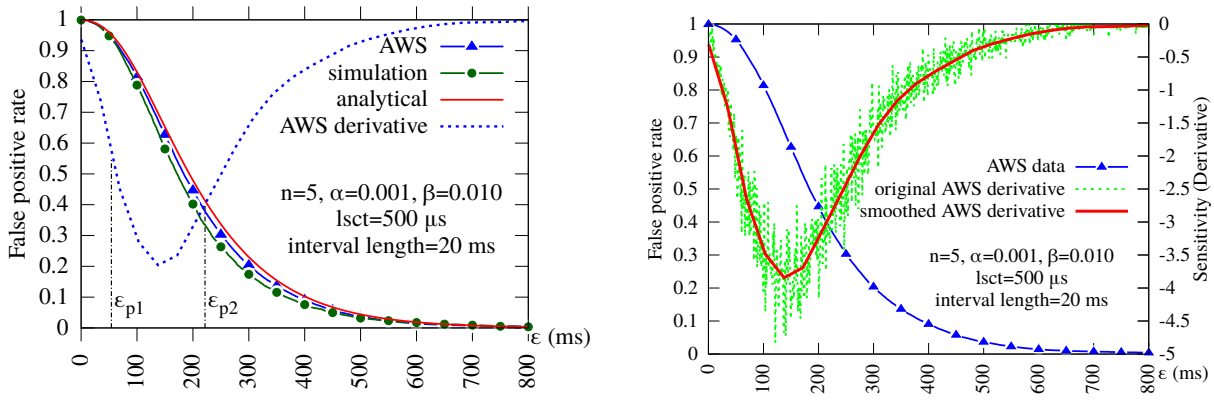
4.1 Modification of Predicate Detection Algorithm

The predicate detection algorithm used by partially synchronous monitors is exactly the same algorithm described in Section 3.2 with one modification. Specifically, the predicate detection algorithm in this Section uses Hybrid Vector Clocks [12] while Garg and Chase's algorithm (like the algorithm in Section 3.2) uses Vector Clocks [7,15]. Hybrid Vector Clocks are Vector Clocks augmented with the information about the bound of clock synchronization error ϵ. Thus, when Vector Clocks are used, the monitor uses the criteria of *hb*-consistency (defined in Section 2.4) to determine whether a global snapshot is consistent; when Hybrid Vector Clocks are used, the monitor's decision is based instead on ϵ-consistency.

4.2 Analytical Model

In the analytical model for partially synchronous monitors, we use the same parameters that have been introduced in Section 3.1. As validated in Section 3, the value of δ is not important. Hence, we only focus on the relation between ϵ_{mon} and ϵ_{app}.

While asynchronous monitors assume an arbitrary clock skew ($\epsilon = \infty$), partially synchronous monitors assume that clocks do not differ more than a finite bound ϵ_{mon}. The monitor's assumption ϵ_{mon} is based on its best knowledge about the application's assumption ϵ_{app}. Nevertheless, the application may implicitly rely on the assumption that clocks are synchronized to be within ϵ_{app}, which is difficult to compute and is unavailable to the monitor. Such an application may use ϵ_{app} with the use of timeouts, or even more implicitly may rely on database update and cache invalidation schemes to ensure that no two events that are more than ϵ_{app} apart can be part of the same global state as observed by the clients [14].



(a) (b)

Fig. 9: (AWS) Interval-based asynchronous monitors. Figure 9a: Compatibility between analytical model prediction, simulation results, and AWS experiment results. Figure 9b: Comparison between the original derivative of raw experiment data and its smoothed version

If $\epsilon_{app} < \epsilon_{mon}$, then the situation is similar to that of the asynchronous monitors, where $\epsilon_{mon} = \infty$. The monitor's recall is always 1 (Observation 3). However, if ϵ_{mon} is finite then it will reduce the false positives as this monitor will avoid detecting some snapshot instances where the time difference between the local predicates being true is too large ($> \epsilon_{mon}$).

If $\epsilon_{app} > \epsilon_{mon}$, the situation is reversed, i.e., the precision will always be 1 (Observation 4). However, the recall of the monitor would be less than 1, as the monitor may fail to find some snapshots that are ϵ_{app}-consistent but not ϵ_{mon}-consistent.

The following theorem identifies the precision and recall of partially synchronous monitors. (We sometimes use $\phi(\epsilon)$ as a short form of $\phi(\epsilon, n, \beta, \ell)$ to simplify the notation in the rest of the paper.)

Theorem 3 *When a monitor designed for an $\langle \epsilon_{mon}, \delta \rangle$-program is used to monitor an $\langle \epsilon_{app}, \delta \rangle$-program with n processes where the local predicate $\mathcal{P}_i$ becomes true with probability β and remains true for duration ℓ, the precision and recall of the monitor are as follows:*

$$Precision = \frac{\phi(\min(\epsilon_{app}, \epsilon_{mon}))}{\phi(\epsilon_{mon})},$$

$$Recall = \frac{\phi(\min(\epsilon_{app}, \epsilon_{mon}))}{\phi(\epsilon_{app})}$$

$$where\ \phi(\epsilon) = (1 - (1 - \beta)^{\epsilon + \ell})^{n-1}$$

Proof We consider the case $\epsilon_{app} \leq \epsilon_{mon}$, then

$$Recall = \frac{\phi(min(\epsilon_{app}, \epsilon_{mon}))}{\phi(\epsilon_{app})} = \frac{\phi(\epsilon_{app})}{\phi(\epsilon_{app})} = 1$$

which complies with Observation 3.

We use $\langle g, \epsilon \rangle$ to denote that a snapshot g is ϵ-consistent. (Consequently, when $\epsilon = \infty$, $\langle g, \infty \rangle$ denotes that snapshot g is *hb*-consistent.) By the law of total probability, we have

$$P(\langle g, \epsilon \rangle) = P(\langle g, \epsilon \rangle \cap \langle g, \infty \rangle) + P(\langle g, \epsilon \rangle \cap \neg \langle g, \infty \rangle)$$

The probability $P(\langle g, \epsilon \rangle \cap \neg \langle g, \infty \rangle) = 0$ since if snapshot g is not *hb*-consistent, g is also not ϵ-consistent (Observation 2). By Theorem 1, the probability

$$P(\langle g, \epsilon \rangle \cap \langle g, \infty \rangle) = \phi(\epsilon, n, \beta, \ell) = (1 - (1 - \beta)^{\epsilon + \ell})^{n-1}$$

Thus

$$P(\langle g, \epsilon \rangle = P(\langle g, \epsilon \rangle \cap \langle g, \infty \rangle) = \phi(\epsilon, n, \beta, \ell) = \phi(\epsilon)$$

The precision of the monitor is the probability that a ϵ_{mon}-consistent snapshot found by the monitor is also ϵ_{app}-consistent, which is $P(\langle g, \epsilon_{app} \rangle | \langle g, \epsilon_{mon} \rangle)$. Note that every ϵ_{app}-consistent snapshot is also ϵ_{mon}-consistent (since $\epsilon_{app} \leq \epsilon_{mon}$), thus $\langle g, \epsilon_{app} \rangle \cap \langle g, \epsilon_{mon} \rangle$ is equivalent to $\langle g, \epsilon_{app} \rangle$. Using the formula of conditional probability, we have:

$$\begin{aligned} Precision &= P(\langle g, \epsilon_{app} \rangle | \langle g, \epsilon_{mon} \rangle) \\ &= \frac{P(\langle g, \epsilon_{app} \rangle \cap \langle g, \epsilon_{mon} \rangle)}{P(\langle g, \epsilon_{mon} \rangle)} = \frac{P(\langle g, \epsilon_{app} \rangle)}{P(\langle g, \epsilon_{mon} \rangle)} \\ &= \frac{\phi(\epsilon_{app})}{\phi(\epsilon_{mon})} = \frac{\phi(min(\epsilon_{app}, \epsilon_{mon}))}{\phi(\epsilon_{mon})} \end{aligned}$$

The proof for the case $\epsilon_{mon} \leq \epsilon_{app}$ is similar. In particular:

$$Precision = \frac{\phi(min(\epsilon_{app}, \epsilon_{mon}))}{\phi(\epsilon_{mon})} = \frac{\phi(\epsilon_{mon})}{\phi(\epsilon_{mon})} = 1$$

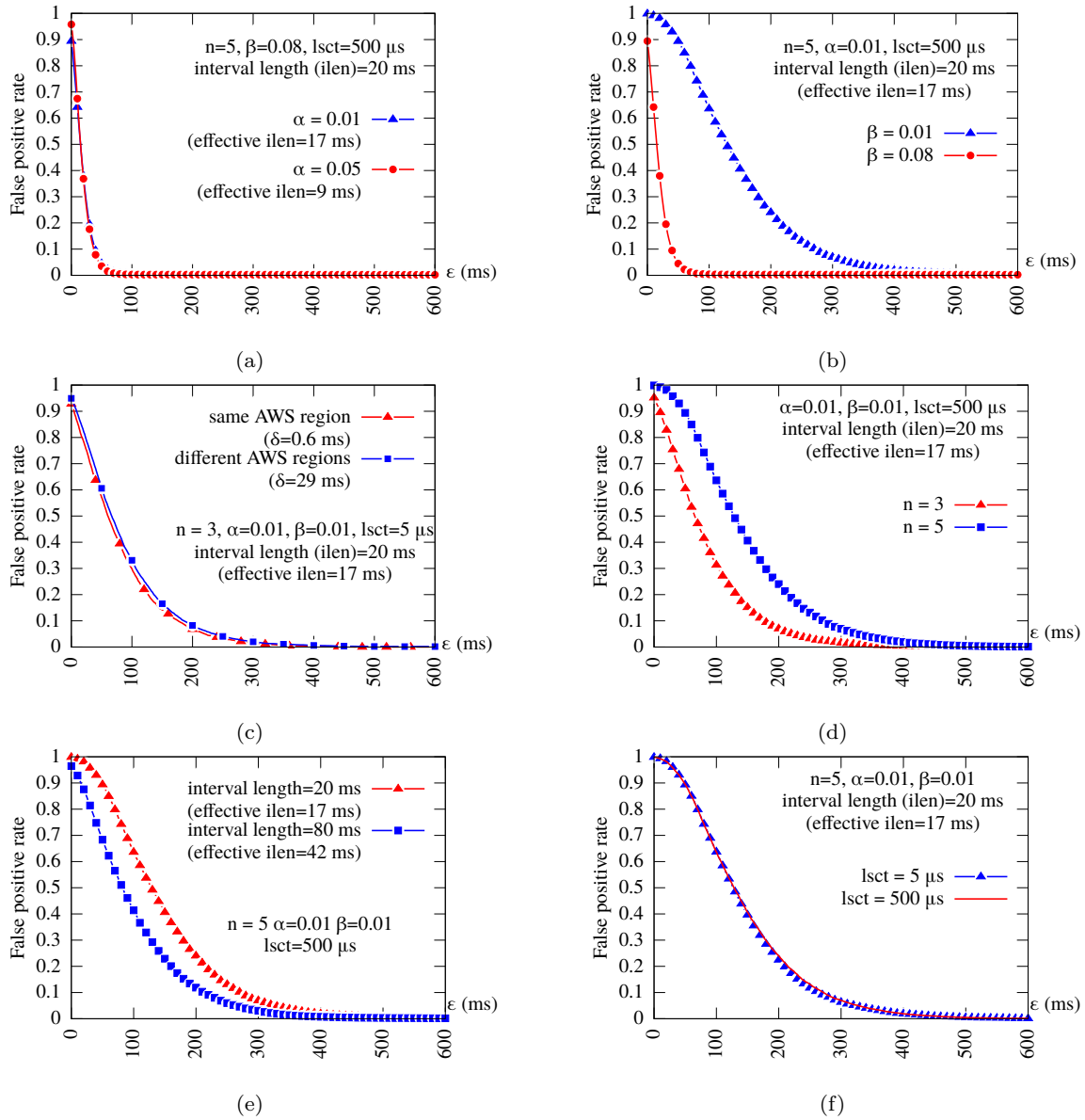


Fig. 10: (AWS) Interval-based asynchronous monitors: Impact of different factors α, β, δ, n, ℓ (interval length), and $lsct$ (local sleep computation time) on false positive rate. We note that the effective values of α and β are closely approximated. For example, in Figure 10d, the effective values of $\alpha = 0.01, \beta = 0.01$ are $\alpha = 0.01006, \beta = 0.01007$, respectively. The effective values of interval length (ilen, ℓ) are shown in the figures. [3]

The recall of the monitor is the probability that a ϵ_{app}-consistent snapshot is also ϵ_{mon}-consistent (i.e. the ϵ_{app}-consistent snapshot is also reported by the monitor), which is:

$$Recall = P(\langle g, \epsilon_{mon}\rangle | \langle g, \epsilon_{app}\rangle) = \frac{P(\langle g, \epsilon_{mon}\rangle \cap \langle g, \epsilon_{app}\rangle)}{P(\langle g, \epsilon_{app}\rangle)}$$
$$= \frac{P(\langle g, \epsilon_{mon}\rangle)}{P(\langle g, \epsilon_{app}\rangle)} = \frac{\phi(\epsilon_{mon})}{\phi(\epsilon_{app})} = \frac{\phi(min(\epsilon_{app}, \epsilon_{mon}))}{\phi(\epsilon_{app})}$$

□

Next, we examine the *sensitivity* –changes in the precision and recall of the monitor caused by the changes in $|\epsilon_{app} - \epsilon_{mon}|$– of partially synchronous monitors. We visualize the sensitivity of partially synchronous monitors by a diagram named PR-sensitivity diagram (Precision/Recall-sensitivity diagram). A PR-

[3] Recall that in AWS experiments, we cannot control the precise value of α, ℓ and β, as we cannot control what happens with the interaction with network and the operating system. For example, in Figure 10d, we attempted to keep $\alpha = 0.01$. However, the effective value of α computed by the number of messages sent in a given interval was 0.01006. Likewise, we attempted to keep interval length in Figure 10e to be $80ms$. However, the effective length was $42ms$, in part due to the fact that intervals are split by message send as required in [8].

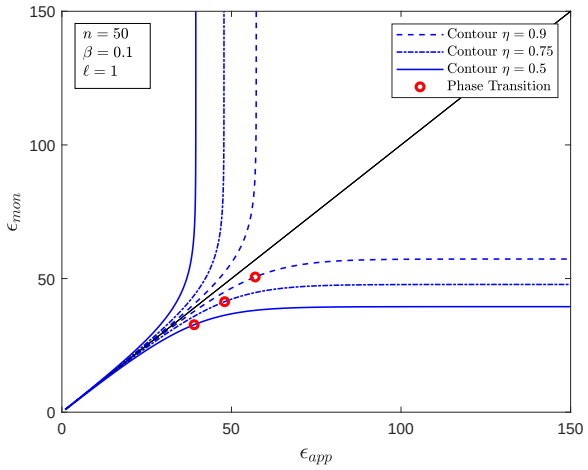


Fig. 11: (Analytical) An example of a PR-sensitivity diagram of a partially synchronous monitor.

sensitivity diagram is a contour map of a monitor's precision and recall where the two axes of the mapping are ϵ_{app} and ϵ_{mon}, and each curve is a contour line where the value of the monitor's precision/recall is a constant. The precision (respectively, recall) contours are at the upper (respectively, lower) half of the diagram where $\epsilon_{app} < \epsilon_{mon}$ (respectively, $\epsilon_{mon} < \epsilon_{app}$).

Figure 11 is a PR-sensitivity diagram of a partially synchronous monitor with $n = 50$, $\beta = 0.1$, $\ell = 1$. We observe that the contour lines (for example the lines where $\eta = 0.5$ and $\eta = 0.75$) are far apart when $\epsilon_{app}/\epsilon_{mon}$ are large and get closer when $\epsilon_{app}/\epsilon_{mon}$ are small. This means for small values of $\epsilon_{app}/\epsilon_{mon}$, a small uncertainty in ϵ_{app} (since the application's assumption is an unknown variable to the monitor) could change the precision/recall substantially, whereas for large values of $\epsilon_{app}/\epsilon_{mon}$, a small uncertainty in ϵ_{app} would leave the precision/recall fairly unaffected. Monitoring in such a region where $\epsilon_{app}/\epsilon_{mon}$ are small should be done carefully so that the monitor precision/recall is still within an acceptable level.

For the monitor to achieve a precision/recall that is not less than some desirable threshold η, the relation between ϵ_{app} and ϵ_{mon} needs to meet some conditions. The next theorem identifies those conditions so that the precision/recall of the monitor is within the useful range $[\eta, 1]$.

Theorem 4 *When an $\langle \epsilon_{app}, \delta \rangle$-program with n processes where local predicate $\mathcal{P}_i$ becomes true with probability β and remains true for duration ℓ, is monitored by a monitor designed for an $\langle \epsilon_{mon}, \delta \rangle$-program, the precision/recall of the monitor will be at least η if the following conditions hold:*

$$\begin{aligned} & \log_{1-\beta}(1 - \eta^{\frac{+1}{n-1}} h(\beta, \epsilon_{mon}, \ell)) \\ & \leq \epsilon_{app} + \ell \\ & \leq \log_{1-\beta}(1 - \eta^{\frac{-1}{n-1}} h(\beta, \epsilon_{mon}, \ell)) \\ \textit{where:} \quad & h(\beta, \epsilon_{mon}, \ell) = 1 - (1-\beta)^{\epsilon_{mon}+\ell} \end{aligned}$$

Proof First we consider the case of precision (when $\epsilon_{app} \leq \epsilon_{mon}$). For the precision at least η, by Theorem 3, we have:

$$\eta \leq \mathit{Precision} = \frac{\phi(\epsilon_{app})}{\phi(\epsilon_{mon})} = \frac{(1-(1-\beta)^{\epsilon_{app}+\ell})^{n-1}}{(1-(1-\beta)^{\epsilon_{mon}+\ell})^{n-1}}$$

$$\Leftrightarrow \eta^{\frac{1}{n-1}} \leq \frac{1-(1-\beta)^{\epsilon_{app}+\ell}}{1-(1-\beta)^{\epsilon_{mon}+\ell}}$$

$$\Leftrightarrow \eta^{\frac{1}{n-1}}\ h(\beta, \epsilon_{mon}, \ell) \leq 1-(1-\beta)^{\epsilon_{app}+\ell}$$

$$\Leftrightarrow (1-\beta)^{\epsilon_{app}+\ell} \leq 1 - \eta^{\frac{1}{n-1}}\ h(\beta, \epsilon_{mon}, \ell)$$

Since $0 < 1-\beta < 1$, we have

$$\epsilon_{app} + \ell \geq \log_{1-\beta}(1 - \eta^{\frac{1}{n-1}}\ h(\beta, \epsilon_{mon}, \ell))$$

So when

$$\log_{1-\beta}(1 - \eta^{\frac{1}{n-1}}\ h(\beta, \epsilon_{mon}, \ell)) - \ell \leq \epsilon_{app} \leq \epsilon_{mon}$$

the monitor's precision is in the range $[\eta, 1]$ (its recall is 1).

In case of recall ($\epsilon_{mon} \leq \epsilon_{app}$), we have

$$\eta \leq \mathit{Recall} = \frac{\phi(\epsilon_{mon})}{\phi(\epsilon_{app})} = \frac{(1-(1-\beta)^{\epsilon_{mon}+\ell})^{n-1}}{(1-(1-\beta)^{\epsilon_{app}+\ell})^{n-1}}$$

$$\Leftrightarrow \eta^{\frac{1}{n-1}} \leq \frac{1-(1-\beta)^{\epsilon_{mon}+\ell}}{1-(1-\beta)^{\epsilon_{app}+\ell}}$$

$$\Leftrightarrow 1-(1-\beta)^{\epsilon_{app}+\ell} \leq \eta^{\frac{-1}{n-1}}\ h(\beta, \epsilon_{mon}, \ell)$$

$$\Leftrightarrow 1 - \eta^{\frac{-1}{n-1}}\ h(\beta, \epsilon_{mon}, \ell) \leq (1-\beta)^{\epsilon_{app}+\ell}$$

$$\Leftrightarrow \log_{1-\beta}(1 - \eta^{\frac{-1}{n-1}}\ h(\beta, \epsilon_{mon}, \ell)) \geq \epsilon_{app} + \ell$$

So when

$$\epsilon_{mon} \leq \epsilon_{app} \leq \log_{1-\beta}(1 - \eta^{\frac{-1}{n-1}}\ h(\beta, \epsilon_{mon}, \ell)) - \ell$$

the monitor's recall is the range $[\eta, 1]$ (its precision is 1).

Combining both formulae, the monitor will have its precision/recall in the range $[\eta, 1]$ when

$$\begin{aligned} & \log_{1-\beta}(1 - \eta^{\frac{+1}{n-1}} h(\beta, \epsilon_{mon}, \ell)) \\ & \leq \epsilon_{app} + \ell \\ & \leq \log_{1-\beta}(1 - \eta^{\frac{-1}{n-1}} h(\beta, \epsilon_{mon}, \ell)) \end{aligned}$$

□

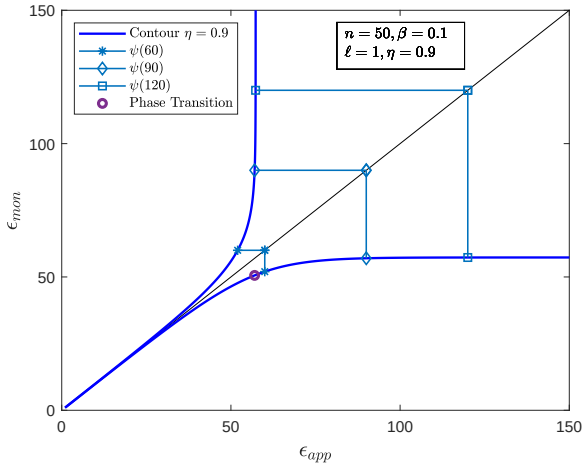


Fig. 12: (Analytical) An example of the width function $\psi(\epsilon_{app})$. The width changes drastically after the phase transition.

We note that since the role of ϵ_{app} and ϵ_{mon} are symmetrical in terms of precision and recall. Specifically

$$Precision = \frac{\phi(\epsilon_{app})}{\phi(\epsilon_{mon})},\ Recall = \frac{\phi(\epsilon_{mon})}{\phi(\epsilon_{app})}$$

Hence, with a calculation similar to the proof of Theorem 4, we can show that the precision/recall of the monitor will be at least η when:

$$\begin{aligned} &\log_{1-\beta}(1-\eta^{\frac{+1}{n-1}}h(\beta,\epsilon_{app},\ell)) \\ &\leq \epsilon_{mon}+\ell \\ &\leq \log_{1-\beta}(1-\eta^{\frac{-1}{n-1}}h(\beta,\epsilon_{app},\ell)) \\ \text{where:}\quad &h(\beta,\epsilon_{app},\ell) = 1-(1-\beta)^{\epsilon_{app}+\ell} \end{aligned}$$

In Figure 12, consider a vertical line for a given ϵ_{app} value. We define the width function $\psi(\epsilon_{app})$ to be the difference between where the vertical line intersects $\epsilon_{mon} = \epsilon_{app}$ and where it intersects the contour line for recall $= \eta$. On the contour where recall $= \eta$, the following condition holds:

$$\epsilon_{mon} = \log_{1-\beta}(1-\eta^{\frac{1}{n-1}}h(\beta,\epsilon_{app},\ell)) - \ell$$

For a given η, β, n, ℓ and the description of the width function above, we have

$$\psi(\epsilon_{app}) = \epsilon_{app} - contour(\epsilon_{app})$$

where

$$\begin{aligned} contour(\epsilon_{app}) &= \log_{1-\beta}(1-\eta^{\frac{1}{n-1}}h(\beta,\epsilon_{app},\ell)) - \ell \\ &= \log_{1-\beta}(1-\eta^{\frac{1}{n-1}}(1-(1-\beta)^{\epsilon_{app}+\ell})) - \ell \end{aligned}$$

We note that the width function $\psi(\epsilon_{app})$ identifies the margin of error between ϵ_{app} and ϵ_{mon} where both precision and recall is at least η. Observe that $\psi(\epsilon_{app})$ decreases when ϵ_{app} decreases. We are interested in how fast $\psi(\epsilon_{app})$ will change when ϵ_{app} changes. The rate of change of ψ as ϵ_{app} changes is measured by ψ', the first derivative of ψ with respect to ϵ_{app}. In the next theorem, we identify the inflection point (phase transition point) of ψ'.

Theorem 5 *Suppose an $\langle \epsilon_{app}, \delta \rangle$-program with n processes where local predicate $\mathcal{P}_i$ becomes true with probability β and remains true for duration ℓ, is monitored by a monitor designed for an $\langle \epsilon_{mon}, \delta \rangle$-program. Let η be the recall of the monitor $(0 < \eta < 1)$. Let the width function $\psi(\epsilon_{app})$ be*

$$\psi(\epsilon_{app}) = \epsilon_{app} - \log_{1-\beta}(1-\eta^{\frac{1}{n-1}}(1-(1-\beta)^{\epsilon_{app}+\ell})) + \ell$$

Let ψ' denote the first derivative of $\psi(\epsilon_{app})$ with respect to ϵ_{app}. Then the inflection point (phase transition) of ψ' is given by the following:

$$\epsilon_{app} = \log_{1-\beta}(\eta^{\frac{-1}{n-1}} - 1) - \ell$$

Proof Similar to the proof of Lemma 1, the inflection point of ψ' is where the third derivative of ψ with respect to ϵ_{app} equals 0.

Let $B = 1-\beta$, $C = \eta^{\frac{1}{n-1}}$. We note that these values are independent from ϵ_{app}. Denote $x = \epsilon_{app} + \ell$. Since the derivative of x with respect to ϵ_{app} is $x' = 1$, the derivative (or any order) of ψ with respect to x is the same as the derivative of ψ with respect to ϵ_{app}. So we will use x as the variable to make the formulas below easier to follow.

We obtain the first, second, and third derivatives of ψ with respect to x (as well as ϵ_{app}) as below:

$$\begin{aligned} \psi =& \epsilon_{app} - \log_{1-\beta}(1-\eta^{\frac{1}{n-1}}(1-(1-\beta)^{\epsilon_{app}+\ell})) + \ell \\ =& x - \log_B(1-C(1-B^x)) \\ \psi' =& 1 - \frac{1}{\ln B} \times \frac{(1-C(1-B^x))'}{1-C(1-B^x)} \\ =& 1 - C \times \frac{B^x}{1-C+CB^x} \\ \psi'' =& -C \times \frac{\ln B B^x(1-C+CB^x) - B^x C \ln B B^x}{(1-C+CB^x)^2} \\ =& -C \times \frac{\ln B B^x(1-C+CB^x-CB^x)}{(1-C+CB^x)^2} \\ =& -C(1-C)\ln B \times \frac{B^x}{(1-C+CB^x)^2} \end{aligned}$$

$$\begin{aligned}\psi''' =& -C(1-C)\ln B\times \\ & \frac{\ln BB^x(1-C+CB^x)^2 - B^x 2(1-C+CB^x)C\ln BB^x}{(1-C+CB^x)^4} \\ =& -C(1-C)\ln B\times \\ & \ln BB^x(1-C+CB^x)\times \\ & \frac{(1-C+CB^x-B^x 2C)}{(1-C+CB^x)^4} \\ =& -C(1-C)(\ln B)^2 B^x \times \frac{(1-C-CB^x)}{(1-C+CB^x)^3}\end{aligned}$$

Note that $0 < B = 1-\beta < 1$. Since $0 < \eta < 1$, $0 < C = \eta^{\frac{1}{n-1}} < 1$. As a result:

$$\begin{aligned}&\psi''' = 0 \Leftrightarrow 1-C-CB^x = 0 \\ &\Rightarrow B^x = \frac{1}{C} - 1 \Rightarrow x = \epsilon_{app} + \ell = \log_B(\frac{1}{C} - 1) \\ &\Rightarrow \epsilon_{app} = \log_B(\frac{1}{C} - 1) - \ell = \log_{1-\beta}(\eta^{\frac{-1}{n-1}} - 1) - \ell\end{aligned}$$

□

4.3 Point-based Predicate Simulations

Figure 13a shows the precision/recall diagram in point-based simulation. We observe that the contour map is denser in the area where $\epsilon_{app}/\epsilon_{mon}$ are small. This confirms the analytical prediction that the precision and recall of partially synchronous monitors for point-based predicates are highly sensitive when ϵ_{app} is small and not sensitive when ϵ_{app} is large.

From Figure 13a, we observe that when n increases, the precision/recall decreases. To see how this relation is illustrated, in Figure 13a we choose any point on the precision contour line (below the identity line where $\epsilon_{mon} = \epsilon_{app}$) of $n = 20$, $P = 0.9$, and let that point's coordinates be ϵ^*_{mon} and ϵ^*_{app}. For the same point location ($\epsilon_{mon} = \epsilon^*_{mon}$, $\epsilon_{app} = \epsilon^*_{app}$) but $n = 5$, we check the new precision of the monitor. We observe that the contour line $P = 0.9$, $n = 20$ is between the contour line $P = 0.9$, $n = 5$ and the contour line $P = 1.0$, $n = 5$ (the identity line). Therefore, a point on contour line $P = 0.9$, $n = 20$ such as $(\epsilon^*_{mon}, \epsilon^*_{app})$ belongs to some contour line with $n = 5$ and precision P somewhere between 0.9 and 1.0. In other words, when n is decreased from 20 to 5, the precision increases from 0.9 to some value between 0.9 and 1.0. A similar observation also applies for recall.

When processes' local predicates are correlated, we observe similar effects of n on precision/recall as shown in Figure 13b.

4.4 Interval-based Predicate AWS Experiments

For scenarios in which we consider intervals where local predicates are true, the results are presented in Figure 14. We observe that, similar to point-based scenario simulations, for interval-based predicates the precision and recall of partially synchronous monitors in the experiments are highly sensitive when ϵ_{app} is small and not sensitive when ϵ_{app} is large. The interval-based precision/recall increases when n decreases (Figure 14a), ℓ increases (Figure 14b) as expected.

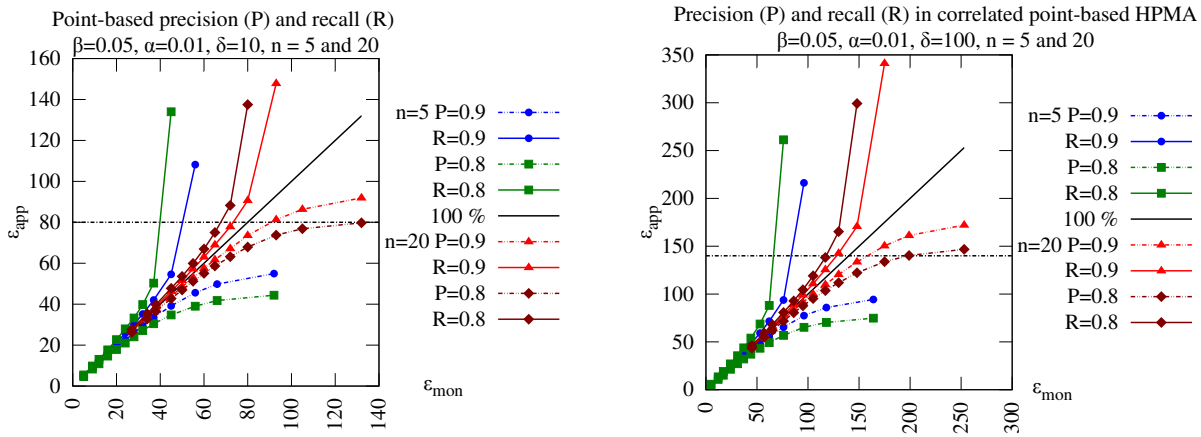
4.5 Interval-based Predicate Simulations

Figure 15 shows interval-based simulation results for partially synchronous monitors. Similar to experimental results in Figure 14 , we observe in the simulation results that the monitor precision/recall is highly sensitive when ϵ_{app} is small. Furthermore, the monitor's precision/recall increases when n decreases (Figure 15a), or ℓ increases (Figure 15b). These simulation results are compatible with the analytical model and the experimental results for partially synchronous monitors.

5 Effectiveness of Quasi-Synchronous Monitors with O(1) Timestamps

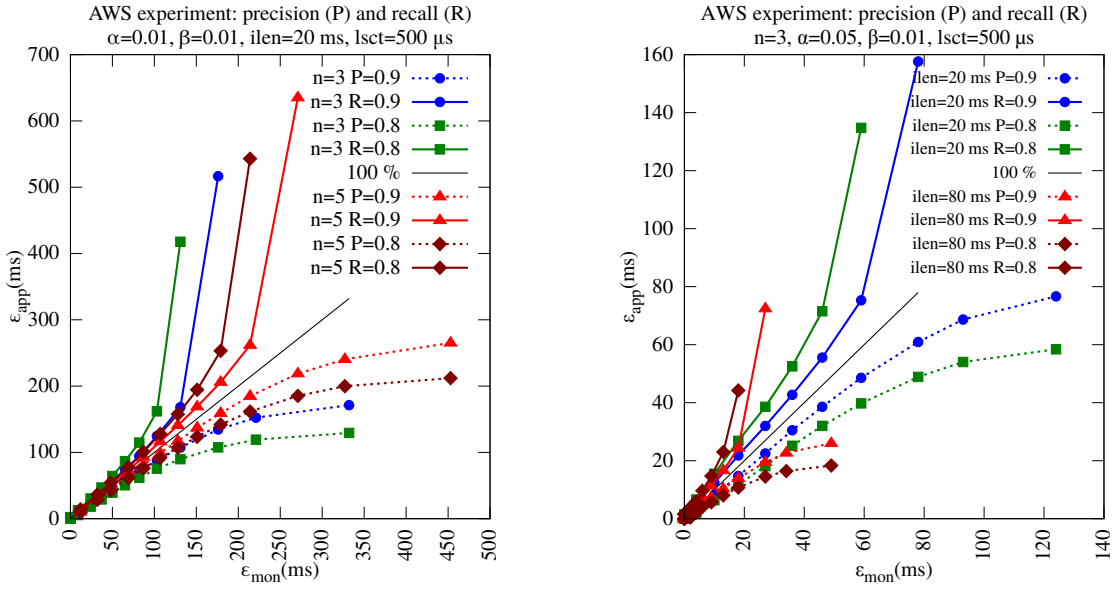
Asynchronous monitors considered in Section 3 had a recall of 1 while their precision was less than 1. Thus, the natural question is: can we have monitors whose precision is 1 but recall is less than 1? We consider one such monitor in this section. While this monitor does not follow the same algorithm as in [8], we find that the analytical model for the recall of this monitor is very similar to the precision and recall in Sections 3 and 4. We also find that the analytical model is validated by the simulations and experiments.

To provide a brief motivation of quasi-synchronous monitors, observe that the analysis in Section 4 can be instantiated for the case where the program assumes fully synchronized clocks, i.e., where clock skew is 0. Although achieving fully synchronized clocks is difficult/impossible in a distributed program, they offer an inherent advantage. Specifically, for programs in asynchronous or partially synchronous systems, to identify whether two events could have happened at the same time, the monitors need to use techniques such as vector clocks [7,15] that require $O(n)$ space, where n is the number of processes. Even though there are attempts to reduce the size [24,1], the worst-case size is still $O(n)$. By contrast, for programs in fully synchronous systems, if two events at two different processes have identical



(a) (b)

Fig. 13: (Simulation) Precision and Recall Diagram in point-based predicate detection when processes' local predicates become true independently (Figure 13a) or in a correlated manner (Figure 13b)



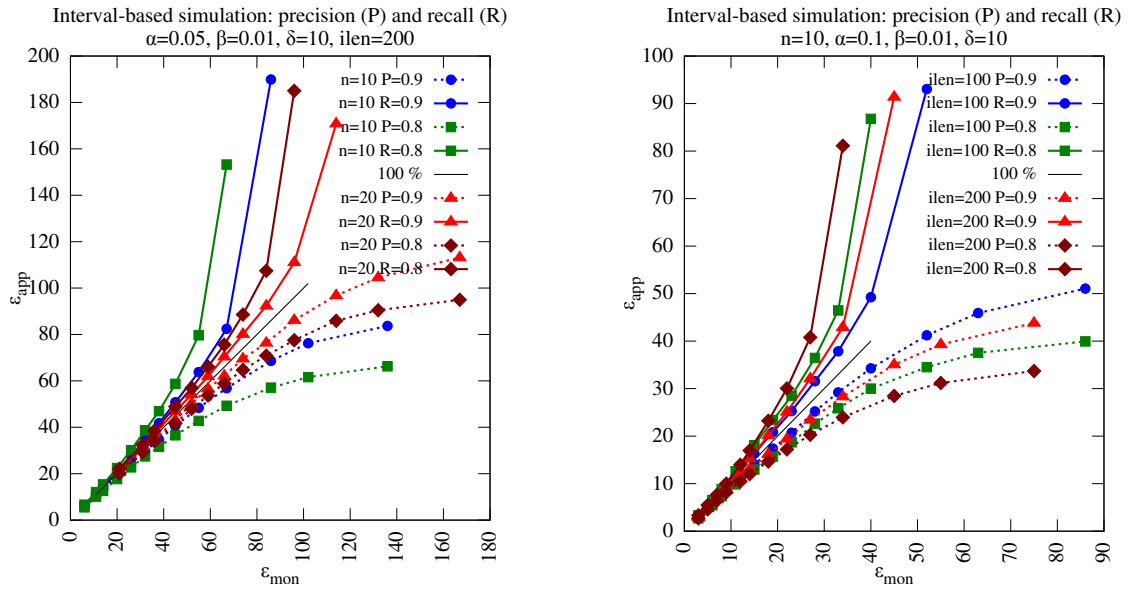
(a) (b)

Fig. 14: (AWS) Interval-based Partially Synchronous Monitors: PR diagram and its sensitivity to n (Figure 14a) and ℓ (denoted as *ilen* in the Figure 14b)

clock values, the monitors can conclude that they happened at the same time. In other words, $O(1)$ information suffices with fully synchronous clocks.

Although fully synchronous *physical* clocks are hard to achieve, we can get *simulated* clocks (e.g., Hybrid Logical Clocks (HLC)[12]) that achieve the same property, i.e. the ability to conclude that events with the same timestamp value happened at the same time. Our goal in this section is to evaluate the effectiveness of monitors that use HLC in monitoring programs in partially synchronous systems. We denote the monitors that use such simulated clocks as **quasi-synchronous monitors**.

For an application that assumes perfectly synchronized clocks, we can implement the monitoring algorithm as follows: if all local predicates are true at the same time t, then the conjunction of the local predi-



(a) (b)

Fig. 15: (Simulation) Interval-based Partially Synchronous Monitors: Precision and Recall Diagram and the impact of n (Figure 15a) and ℓ (Figure 15b)

cates is true at t. [4] However, if the application assumed a clock skew of $\epsilon_{app} > 0$ then the above approach will suffer from false negatives. In other words, the monitor may miss instances where the conjunctive predicate is true. This may happen if events on two processes do not happen exactly at the same time but are within the clock skew bound (i.e. within ϵ_{app} of each other in physical time).

With this motivation, we focus on the following problem: Given a quasi-synchronous monitor that relies on a simulated clock (which guarantees that two events with equal clock value are concurrent, i.e. the two events do not depend upon each other) to determine the causality among events. If that quasi-synchronous monitor is used to monitor an $\langle \epsilon_{app}, \delta \rangle$-program (in which concurrent events could have their physical timestamps differ by at most ϵ_{app}), what is the rate of false negatives (or the recall) of the monitor? Since this analysis depends upon how the simulated clock is implemented (although not on how the monitoring algorithm itself is implemented given the simulated clock), we describe one such simulated clock and identify its effectiveness in the next section.

[4] Note that our analysis is based on the property of the monitor and, hence, we do not consider how the monitoring algorithm can be evaluated/implemented most efficiently.

5.1 Simulated Clocks: Hybrid Logical Clocks

Hybrid Logical Clocks (HLC) [12] are one such instance of simulated clocks. HLC combines both physical clocks and logical clocks [13]. In HLC, each event e is timestamped with $hlc.e = \langle pt.e, l.e, c.e \rangle$, where $pt.e$ is the physical time, $l.e$ is the logical time and $c.e$ is a counter. HLC ensures that the logical clock is always *close* to the physical clock. Specifically, for any event e, $pt.e \leq l.e \leq pt.e + \epsilon$, where ϵ is the clock skew. HLC also preserves the property of logical clocks, i.e. for events e and f, $e \; hb \; f \Rightarrow hlc.e < hlc.f$, where $hlc.e < hlc.f$ iff $(l.e < l.f) \vee ((l.e = l.f) \wedge (c.e < c.f)))$.

From the above discussion, if $(l.e = l.f) \wedge (c.e = c.f)$ then this implies that events e and f are concurrent. Observe that this is exactly the property required of simulated clocks [5]. In other words, HLC can be used to monitor conjunction of $\mathcal{P}_i$ $(0 \leq i < n)$ by finding HLC timestamp t, such that $\mathcal{P}_i$ is true at t for every process i.

Problem Statement 3: If we use an HLC-based monitor for monitoring an $\langle \epsilon_{app}, \delta \rangle$-program, what is the recall of that monitor?

[5] Note that this property is not guaranteed even with physical clocks, because a message send event and the corresponding message receive event can have equal physical timestamps due to clock skew, so events with equal physical timestamps may not be concurrent events.

5.2 Analytical Model for Detecting Predicates using Simulated Clocks (HLC)

In essence, a quasi-synchronous monitor that uses a simulated clock (HLC) detects a snapshot if and only if there is a point (HLC timestamp) common to the local intervals associated with all processes. As a result, any snapshot discovered by a quasi-synchronous monitor is always an ϵ_{app}-consistent snapshot, for any value of ϵ_{app}. In other words, the precision of detection using a simulated clock is always equal to one. So, we focus only on recall.

Recall is the probability that an ϵ_{app}-consistent snapshot will be reported by the quasi-synchronous monitor. The quasi-synchronous monitor will detect a snapshot if the clock values of all the processes in the snapshot are identical. Thus, recall is the same as the probability that a common clock value is present for every process in an ϵ_{app}-consistent snapshot. So we can compute recall as the probability that a snapshot has a common clock value for all processes, given the probability that it is an ϵ_{app}-snapshot. To compute the probability that a snapshot contains a common clock value, we fix the first event in the first interval (first duration for which the local predicate is true) that happened at process 0 at time 0. Then we compute the latest starting point among the intervals at processes $0 < i < n$ (i.e. we pick the starting point of the interval that is farthest from the interval at process 0). The intervals have a common clock value if and only if the distance of this latest starting point from time 0 is shorter than the length of interval ℓ. With this notion and using the same analysis as in the proof of Theorem 1, we present the following theorem.

Theorem 6 *When a quasi-synchronous monitor is used to monitor an $\langle \epsilon_{app}, \delta \rangle$-program with n processes and their local predicates become true with probability β and remain true for duration ℓ, its recall is:*

$$Recall = \frac{\chi(\ell)}{\chi(\epsilon_{app} + \ell)}$$

Where

$$\chi(x) = (1 - (1 - \beta)^x)^{n-1}$$

Note that the formula for Recall can be rewritten using the formula ϕ from Theorem 3 where $Recall = \frac{\phi(0)}{\phi(\epsilon_{app})}$, where $\phi(\epsilon) = (1 - (1 - \beta)^{\epsilon+\ell})^{n-1}$ and values of β, n and ℓ are implicitly provided. In other words, the recall of the quasi-synchronous monitor is essentially same as a partially synchronous monitor that assumed perfect clock synchronization (i.e., $\epsilon_{mon} = 0$).

By taking the derivatives of the recall with respect to β, ℓ, and n, we observe that the value of recall is improved when β or ℓ increases, or n decreases.

Given that we can compute the recall of quasi-synchronous monitoring, we also want to know when the majority of true snapshots are found by a quasi-synchronous monitor. That is, given an application configuration, we want to compute the necessary condition such that the recall is at least 0.5.

$$\frac{\chi(\ell)}{\chi(\epsilon_{app} + \ell)} \geq 0.5$$

Solving the above inequality by simple algebraic manipulation we have the following corollary.

Corollary 3 *With a quasi-synchronous monitor, the recall is at least 0.5 if and only if the following inequality holds:*

$$\ell \geq \log_{1-\beta}\left(\frac{2^{1/(n-1)} - 1}{2^{1/(n-1)} - (1-\beta)^{\epsilon_{app}}}\right)$$

$$\Leftrightarrow \epsilon_{app} \leq \log_{1-\beta}(1 - 2^{n-1}(1 - (1-\beta)^{\ell}) - \ell$$

5.3 Simulation Results for Detecting Predicates with Simulated Clocks

Effect of interval length. Figure 16 shows our results when a quasi-synchronous monitor is used to monitor an $\langle \epsilon_{app}, \delta \rangle$-program with $\epsilon_{app} = 10, \delta = 5, n = 3, \alpha = 0.03, \beta = 0.01$, and the interval length ℓ is varied from 1 to 150. From this figure, we observe that quasi-synchronous monitors are not effective when the interval length is small. For example, when $\ell = 1$ (point-based predicate), about 9% of all ϵ_{app}-consistent snapshots (actual violations) are detected. However, as the interval length increases, the quasi-synchronous monitor is able to detect more ϵ_{app}-consistent snapshots. This is expected, because as the interval length increases, so does the chance that a common HLC timestamp is found in all the local intervals of an ϵ_{app}-consistent snapshot. Consequently, the chance that the ϵ_{app}-consistent snapshot is detected by the quasi-synchronous monitor increases. When the interval length is at least 20, the recall of the quasi-synchronous monitor is greater than 0.5, i.e. the monitor is able to report at least half of the application's violations. Moreover, the simulation results (red line) are very close to the analytical prediction (blue dotted line), which validates our analytical model.

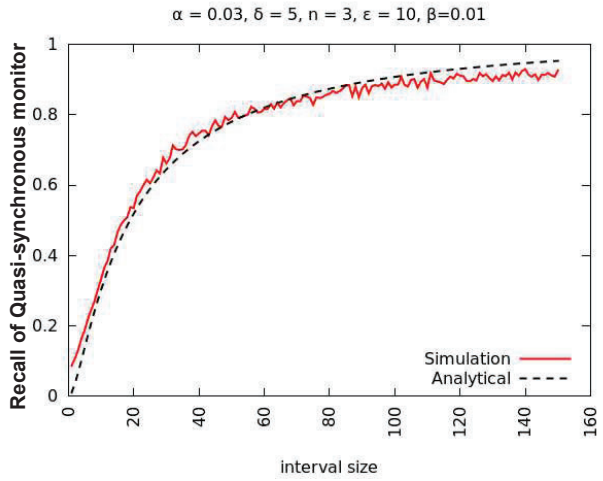
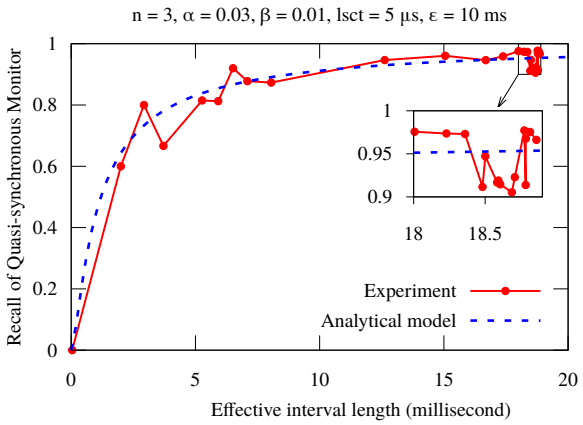
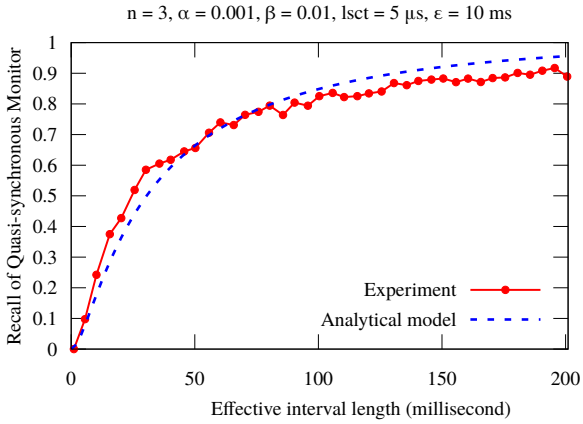


Fig. 16: (Simulation) Analytical Model vs. Simulation Results: The impact of interval size on the recall of a quasi-synchronous monitor. We note that the measurement on the y-axis is also equal to the ratio between the number of snapshots detected by the quasi-synchronous monitor and the number of snapshots detected by the partially synchronous monitor that assumes $\epsilon_{mon} = \epsilon_{app}$



(a)



(b)

Fig. 17: (AWS) Analytical Model vs. AWS-based Experimental Results: The impact of interval size on the recall of a quasi-synchronous monitor. The value of α in Figures 17a and 17b is 0.03 and 0.001, respectively.

5.4 Experimental Results for Detecting Predicates with Simulated Clocks

We experimentally validated the analytical model for quasi-synchronous monitors (see [19] for source code and results). We used the same application and experimental setup of Section 3.4. The experiment parameters $(\epsilon, \delta, \alpha, \beta, \ell)$ are measured as described in Section 3.6.

In these experiments, the effective interval lengths cannot be increased arbitrarily because when a message is sent or received within an interval, the interval is split, as required by the algorithm in [8], thus reducing the effective interval length. (Note that this is not an issue with simulations, as the interval length is just a parameter.) For this reason, for $\alpha = 0.03$ (the same parameter used in simulation), we were able to run the experiments with effective interval length of approximately 20 ms. To achieve a higher effective interval length, we have conducted experiments for smaller values of α. Figure 17 shows that the results obtained on AWS are compatible with the prediction of the analytical model. The inset in Figure 17a magnifies the results of our attempts to increase the interval length by keeping local predicates true for longer than 20 ms. However, because of the interval split that occurs whenever a message is sent or received, the effective interval length we achieved in these experiments was still approximately 20 ms. Nevertheless, the recall values in these cases were close to the analytical prediction. We also observe that (as in Section 5.3 and Figure 16) when the interval length increases, the recall of quasi-synchronous monitors improves.

5.5 Detecting Partial Conjunctive Predicates With Quasi-Synchronous Monitors

In the earlier discussion of Section 5, we considered the case where the predicate being monitored is a conjunctive predicate involving all n processes. Now, we consider the case where the predicate being detected involves only a subset of p processes, $p \leq n$. Instances of using such partial conjunctive predicates include sce-

p	Fraction of snapshots detected by quasi-synchronous monitor
2	0.79
3	0.68
4	0.60
5	0.42

Table 3: Partial conjunctive predicate detection by quasi-synchronous monitor in a partially synchronous program with 5 processes, $\beta = 0.01, \epsilon = 10, \delta = 5, \alpha = 0.03$, and p is the number of processes involved in the partial conjunctive predicate.

narios where the monitor needs to check if a token is possessed by more than one process at the same time.

We analyze the effectiveness of quasi-synchronous monitors in detecting such predicates. We expect quasi-synchronous monitors to perform better when the predicate involves a smaller number of processes. We examine the effectiveness of quasi-synchronous monitors by simulations where an $\langle \epsilon_{app}, \delta \rangle$-program with n processes is monitored simultaneously by a quasi-synchronous monitor and a partially synchronous monitor to detect (snapshot) instances where a partial predicate involving p ($p \leq n$) processes is true. The simulation results are presented in Table 3. When $p = n$ the quasi-synchronous monitor detected about half of the number of snapshots (where the global predicate is true) that were identified by the partially synchronous monitor. As p decreased, the number of snapshots (where the partial predicate is satisfied) detected by the quasi-synchronous monitor started to approach the number of snapshots detected by the partially synchronous monitor.

5.6 Quasi-Synchronous vs. Partially Synchronous Monitoring

The formula of recall in Theorem 6 can also be interpreted as the ratio between the detection capacity of a quasi-synchronous monitor and the detection capacity of a partially synchronous monitor that assumes $\epsilon_{mon} = \epsilon_{app}$. This ratio is also present in Figure 16. Figure 16 and the results in Section 5.5 suggest that quasi-synchronous monitors are able to achieve half of (or even close to) the recall of partially synchronous monitors in some scenarios where the interval length is large or the predicate of interest involves a small subset of processes. These results are notable when we know that partially synchronous monitors require vector clocks (size $O(n)$ in worst cases) whereas quasi-synchronous monitors use scalar clocks (size $O(1)$). Deploying a quasi-synchronous monitor is expected to be simpler than a partially synchronous monitor. However, we can observe that even with such scalar clocks, quasi-synchronous monitors are able to provide high detection coverage. By utilizing the analytical model, one can estimate the recall of quasi-synchronous monitors and determine whether such a monitor should be deployed.

6 Related Work

In distributed programs, processes execute with limited information about other processes. This further implies that the program developers/operators also have limited visibility and information about the program. Monitoring/tracing and predicate detection tools are important components of large-scale distributed programs as they provide valuable information to the developers/operators about their programs under execution.

Monitoring large-scale web-services and cloud computing systems. Dapper [21] is Google's production distributed programs tracing infrastructure. The primary application for Dapper is performance monitoring to identify the sources of tail latency at scale. Making the program scalable and reducing performance overhead was facilitated by the use of adaptive sampling. The Dapper team found that a sample of just one out of thousands of requests provides sufficient information for many common uses of the tracing data.

Facebook's Mystery Machine [3] has goals similar to Google's Dapper. Both use similar methods, however, Mystery Machine tries to accomplish the task relying on less instrumentation than Google Dapper. The novelty of Mystery Machine is that it tries to infer the component call graph implicitly via mining the logs, whereas Google Dapper instruments each call in a meticulous manner and explicitly obtains the entire call graph.

Predicate detection with vector clocks. Lot of work has been done on predicate detection (e.g., Marzullo & Neiger [4] WDAG 1991, Verissimo [23] 1993), using vector clock (VC) timestamped events sorted via happened-before (hb) relationship. The work in [4] not only defined Definitely and Possibly detection modalities, but also provided algorithms for predicate detection using VC for these modalities. The authors in [4] also showed that information about clock synchronization (i.e., ϵ) can be translated into additional happened-before constraints and fed into the predicate detection algorithm to take into account clock synchronization behavior and avoiding false positives in VC-only-based predicate detection. However, they did not investigate the rates of false positives with respect to mismatch in clock synchronization assumptions and event occurrence rates.

Predicate detection with physical clocks and NTP synchronization. For programs in partially synchronized systems, Stoller [22] investigated global predicate detection using NTP clocks, showing that using NTP synchronized physical clocks provide some benefits over using VC in terms of the complexity of predicate detection. The worst-case complexity for predicate detection using hb captured by VC is $\Omega(E^n)$, where E is the maximum number of events executed by each process, and n is the number of processes. With some assumptions on the inter-event spacing being larger than clock synchronization uncertainty, it is possible to have worst-case time complexity for physical clock based predicate detection to be $O(3^n En^2)$ — linear in E. Our work is similar to [22], in that we too monitor programs in partially synchronous systems. Further, we use Hybrid Vector Clock and Hybrid Logical Clock, similar to Dapper [21] and Mystery Machine [3]. The main goal of our work, however, is different from that of these previous efforts. In [22], the objective is to improve the performance of existing predicate detection algorithms by utilizing the information on clock synchronization to reduce complexity and overhead. The goal in [3,21] is also performance: they aim to reduce overhead and achieve scalable monitoring by using different sampling or modeling methods. By contrast, our goal is to study the effect of clock synchronization (and other factors as well) on the accuracy (precision, recall) of the monitors, independent of the choice of predicate detection algorithm used by the monitors.

Predicate detection in programs in partially synchronous systems. The duality of the literature on monitoring predicates forces one to make a binary choice beforehand: To go with either VC-based or physical clock-based timestamping and detection [12,6]. Hybrid Vector Clocks (HVC) obviate this duality. While VC is of $\Theta(N)$ [2], thanks to loosely-synchronized clock assumption, it is possible with HVC to keep the sizes of HVC small, to be a couple of entries at each process [24]. HVC captures the communications in the timestamps and provides the best of VC and physical clock worlds.

All existing work discussed in this section rely on specific clock synchronization assumptions when monitoring or while performing predicate detection. They do not consider scenarios where the clock synchronization uncertainty of the application is unknown or where there are mismatches in the clock synchronization assumptions made by the application and the monitoring algorithm. In this paper, we focused on such scenarios and analyzed the effect of mismatches in the clock synchronization assumptions on the effectiveness (precision/recall) of the monitor.

7 Summary of the Main Results and Their Implications

In this section, we briefly provide an overview of our results. We also identify the implications of this work in monitoring distributed programs.

Precision, recall, and sensitivity of asynchronous monitors. We presented an analytical model that characterizes the precision of a monitor when the application assumes a specific clock skew ϵ_{app} but the monitor assumes that clock skew can grow unbounded. In this scenario, the monitor can only suffer from false positives: The monitor has perfect recall (i.e., there are no false negatives) but suffers from a lack of precision.

Our analytical results identified two parameters ϵ_{p_1} and ϵ_{p_2} that help classify the precision of an asynchronous monitor into three regions based on the value of the application's clock synchronization assumption ϵ_{app}. The monitor's precision will be low in the first region, i.e., if $0 < \epsilon_{app} < \epsilon_{p_1}$. In the third region which corresponds to $\epsilon_{p_2} < \epsilon_{app} < \infty$, the precision is high. Furthermore, in these two regions the monitor's precision is not sensitive, i.e. changes in ϵ_{app} do not significantly affect the precision. By contrast, in the second region the range of $\epsilon_{p_1} \leq \epsilon_{app} \leq \epsilon_{p_2}$ is mapped to a broad range of precision, and a small change in ϵ_{app} has a substantial impact on the monitor's precision. If we know that ϵ_{app} is in the first or in the third region (although we do not know the exact value of ϵ_{app}), we can estimate the precision of the asynchronous monitor well. On the other hand, if we know that ϵ_{app} is in the second region but do not know its exact value, the precision of the monitor is highly uncertain. Another interesting observation in this context was that the relative width of the high sensitivity range $\frac{\epsilon_{p_2}-\epsilon_{p_1}}{\epsilon_{p_1}}$ approaches 0 when the number of processes $n \to \infty$.

We also examined the impact of other factors such as the number of processes n, communication frequency α, communication delay δ, and the length ℓ and rate β of local predicate truthification. We find that asynchronous monitors will deliver a lower precision when β decreases, or n increases, or ℓ decreases. On the other hand, the effect of α and δ on the precision of the monitor is neutral or small.

Precision, recall, and sensitivity of partially synchronous monitors. We also considered the case of partially synchronous monitors, where the monitor assumes that the clocks are synchronized to be within ϵ_{mon}, which is different from ϵ_{app} since the precise clock synchronization assumption of the application is unknown to the monitor. We found that for small $\epsilon_{app}/\epsilon_{mon}$ there is a trade-off between precision, recall,

and sensitivity. If the monitor tries to achieve very high precision and recall (say at 95%) at the same time, the precision/recall is also highly sensitive to changes in ϵ_{app}. For large $\epsilon_{app}/\epsilon_{mon}$, the trade-off dilutes, i.e. the monitor is able to provide high precision and recall while being relatively insensitive to the uncertainty of ϵ_{app}.

A partially synchronous monitor has similar characteristics as the asynchronous counterpart in terms of the impact of system parameters $n, \alpha, \beta, \ell, \delta$.

We validated the analytical results for asynchronous and partially synchronous monitors with simulations and experiments on Amazon Web Services (AWS) platform. Our simulation and experimental results agree with the analytical prediction.

Relating precision and the mismatch in clock synchronization assumptions. To further illustrate the implication of the previous results for partially synchronous monitors, we analyze concrete examples. Table 4 contains examples of $(\epsilon_{app}, \epsilon_{mon})$ pairs for different precision requirements. In these examples, there are 5 application processes where the local predicate becomes true roughly every 20 ms and remains true for approximately $1ms$. If the application relied on the assumption that clocks are synchronized to be within 20 ms then the precision will be at least 50% if the monitor assumes that the clock synchronization is within $20 - 28.66ms$. Thus, if the mismatch between the assumption made by application and monitor is within 43.3% margin then at least half of the errors identified by the monitor are actual errors.

Furthermore, if the monitor relies on the assumption that clocks can differ by at most 49.34 ms then precision is 25%. In other words, if the mismatch between application and monitor assumptions is about 146.7% then at least a quarter of the errors identified by the monitor are actual errors.

For the same configurations, if the application relied on 10 ms clock synchronization assumption, in order to achieve the precision of 50% (or 25%), the monitor assumption must be within $10 - 12.99ms$ (or $10 - 17.24ms$), i.e. within 29.9% (or 72.4%) margin.

The reduced relative margin as the application assumes a smaller clock synchronization illustrates the higher sensitivity of precision (as well as recall) when ϵ_{app} is small.

Next, we also consider the case where the application assumption is not precisely known. As an illustration, consider the same application in Table 4. Suppose the monitor knows that the application is assuming that clocks are synchronized to be within 15 – 25 ms. However, the precise assumption is not known. If the monitor assumes that clocks are synchronized to be within $15ms$, there is a potential false negative rate of 67%. On the other hand, if the monitor assumes that clocks are synchronized to be within $25ms$, there is a potential false positive rate of 67%. Furthermore, if the monitor assumes that clocks are synchronized to be within $20ms$ then there is a possibility of 49% false positives or and 36% false negatives. Depending upon the specific application needs, the monitor can choose the right trade-off for the application.

ϵ_{app}	Precision = 50%		Precision = 25%	
	ϵ_{mon}	margin	ϵ_{mon}	margin
20 ms	28.66 ms	43.3%	49.34 ms	146.7%
10 ms	12.99 ms	29.9%	17.24 ms	72.4%

Table 4: Illustration for the mismatch margin between ϵ_{mon} and ϵ_{app} at different precision requirements. The number of processes (n) is 5, frequency of local predicate to become true (β) is every 20 ms, each local predicate remains true (ℓ) for 1 ms, margin $= \frac{|\epsilon_{mon} - \epsilon_{app}|}{\epsilon_{app}}$. The higher precision sensitivity as ϵ_{app} small is illustrated by the reduced margin value.

Precision, recall, and sensitivity of quasi-synchronous monitors. Quasi-synchronous monitors are partially synchronous monitors with the additional condition that if two events have the same physical clock value then they could have possibly happened at the same time. Adopting a quasi-synchronous model allows us to obviate the need for using $O(n)$ sized vector clocks [7,15] and instead use inexpensive $O(1)$ sized clocks (e.g. hybrid logical clocks [12]) for predicate detection/monitoring. When the monitors designed for programs in quasi-synchronous systems ($\epsilon_{mon} = 0$) are used for programs in partially synchronous systems (ϵ_{app} is finite), they have perfect precision but suffer from false negatives (they miss some valid instances of predicate satisfaction).

We presented an analytical model as well as simulation/experimental results to characterize the effectiveness of such quasi-synchronous monitors. The effectiveness of quasi-synchronous monitors depends on several factors. For example, their recall is improved when β increases or ℓ increases or n decreases. We also observed that when monitoring partial conjunctive predicates (i.e. predicates involving a subset of processes), quasi-synchronous monitors achieve better recall when the predicates involve a smaller number of processes.

Finally, for a convenient overall view of the paper, Table 5 summarizes the main contents discussed in the paper and their corresponding sections.

Table 5: Summary of the results in the paper

			Interval-based	Point-based
Asynchronous Monitors ($\epsilon_{mon} = \infty$, $\epsilon_{app} < \infty$). Section 3	Analytical	Precision	Section 3.3	Section 3.3
		Recall	$= 1.0$	$= 1.0$
		Sensitivity	Section 3.3	Section 3.3
	Simulation	Precision	Section 3.5.3	Section 3.5.1
		Recall	$= 1.0$	$= 1.0$
		Sensitivity	Section 3.5.3	Section 3.5.1
		Impact of other factors, e.g $\alpha, \beta, n, \ell, \delta$	Section 3.5.3	Section 3.5.1
	Experiments	Precision	Section 3.6	See table note [1]
		Recall	$= 1.0$	$= 1.0$
		Sensitivity	Section 3.6	See table note [1]
		Impact of $\alpha, \beta, n, \ell, \delta$	Section 3.6	See table note [1]
Partially Synchronous Monitors ($\epsilon_{mon} < \infty$, $\epsilon_{app} < \infty$). Section 4	Analytical	Precision	Section 4.2	Section 4.2
		Recall	Section 4.2	Section 4.2
		Sensitivity	Section 4.2	Section 4.2
	Simulation	Precision	Section 4.5	Section 4.3
		Recall	Section 4.5	Section 4.3
		Sensitivity	Section 4.5	Section 4.3
		Impact of n, ℓ	Section 4.5	Section 4.3
	Experiments	Precision	Section 4.4	See table note [1]
		Recall	Section 4.4	See table note [1]
		Sensitivity	Section 4.4	See table note [1]
		Impact of n, ℓ	Section 4.4	See table note [1]
Quasi Synchronous Monitors (Using HLC). Section 5	Analytical	Precision	$= 1.0$	$= 1.0$
		Recall	Section 5.2	Section 5.2
		Sensitivity	Section 5.2	Section 5.2
	Simulation	Precision	$= 1.0$	$= 1.0$
		Recall	Section 5.3	Section 5.3
		Sensitivity	Section 5.3	Section 5.3
	Experiments	Precision	$= 1.0$	$= 1.0$
		Recall	Section 5.4	Section 5.4
		Sensitivity	Section 5.4	Section 5.4

[1] Table note 1: Since the predicates are inherently true for some duration of time in AWS experiment, experiments for point-based are not feasible. However, in the experiment, we also make interval length as small as possible to approximate point-based scenarios. The results are in Section 3.6. We also note that the AWS experiment results are similar to the simulation results in Section 3.5.1

8 Conclusion

Due to their inherent nature of concurrency and lack of total order among events, designing distributed programs is error-prone. For this reason, in addition to static analysis techniques, it is desirable to monitor them so that one can report violations that may occur due to race conditions even if those race conditions resolved favorably in the given run. For this reason, monitoring them at runtime is crucial.

An issue that complicates monitoring is that distributed programs are often designed with assumptions related to clock synchronization among relevant nodes. These assumptions allow one to design efficient programs by reducing the need for explicit communication. If these assumptions are not precisely known to the monitor, it would result in the monitor reporting false positives (phantom errors) or false negatives (miss errors). Especially when applications are designed with implicit assumptions or where the exact assumptions are unknown to the monitor, these errors are inevitable. This paper focuses on quantitative analysis of these errors based on the potential mismatch between assumptions made by the application and the assumptions made by the monitor.

We considered three types of monitors (1) asynchronous monitors, (2) partially synchronous monitors and (3) quasi-synchronous monitors. The first category

of monitors provides perfect recall (no false negatives) but suffers from imperfect precision (existence of false positives). The third category of monitors provides perfect precision and imperfect recall. And, the second category allows one to have a trade-off between different precision and recall. Specifically, by changing the assumptions made by the monitor we can increase precision at the cost of recall and vice versa.

The monitors considered in this paper focused on detecting satisfaction of a global predicate $\mathcal{P}$ which is of the form $\bigwedge \mathcal{P}_i$, where $\mathcal{P}_i$ is a local predicate at process i. If $\mathcal{P}$ denotes violation of a safety requirement in a safety-critical system, one would generally require the monitor to have a perfect recall (no false negatives) but would be willing to tolerate imperfect precision. Asynchronous monitors or partially synchronous monitors with a high value of clock skew assumption would be suitable in this context. The work from this paper will allow users to identify the expected level of false positives in this context so that the designer can select the right monitor based on the uncertainties in the clock skew assumptions made by the application.

On the other hand, if $\mathcal{P}$ denotes a relatively stable predicate related to application performance then one would generally require the monitor to have perfect precision (no false positives) and imperfect recall. For example, if $\mathcal{P}$ denotes unbalanced workloads on different machines then it is expected that if the mismatch occurs, it will be present for a certain duration. In that case, having false negatives would be acceptable, as the mismatch in the load will be eventually detected and reported. Moreover, in the event that it resolves on its own, that would be acceptable as well. However, having (significant) false positives would mean that the system is taking unnecessary actions to correct the problem that may not actually exist. Based on the acceptable level of false negatives, the designer can select either quasi-synchronous monitors or partially synchronous monitors that assume a small clock skew.

Our analysis also identifies the highly sensitive region for the monitor. Here, a small change in the application assumption can cause substantial change in precision and/or recall. In terms of using the monitor, if possible, this region should be avoided. In other words, if the designer finds that the monitor is in the highly sensitive region then the designer needs to take more efforts to ensure that application assumptions are known precisely. On the other hand, in other (not highly sensitive) regions, uncertainty in the application assumptions is not as harmful. In the context of asynchronous monitors, we find that this region of high sensitivity is relatively small when the number of processes is large.

There are several future extensions of these results. One extension is to evaluate the error probability for more complex predicates in terms of conjunctive predicate detection. Here, if the predicate was $\phi_1 \vee \phi_2$ there is a possibility that even if ϕ_1 is detected incorrectly, ϕ_2 may still be true, causing detection of $\phi_1 \vee \phi_2$. Apart from conjunctive predicates, it is an open question if similar error probabilities hold for distributed runtime verification for linear temporal logic (LTL) such as in [17]. Another future extension is to consider the case for specific instances of monitors which have potential inbuilt errors introduced for the sake of efficiency during monitoring. We are also interested in developing analytical models for the scenarios where the truthification of local predicates is correlated.

Acknowledgements This work is supported in part by NSF CNS-1329807, NSF CNS-1318678, NSF XPS-1533870, and NSF XPS-1533802.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Almeida, J.B., Almeida, P.S., Baquero, C.: Bounded version vectors. In: R. Guerraoui (ed.) Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings, *Lecture Notes in Computer Science*, vol. 3274, pp. 102–116. Springer (2004)
2. Charron-Bost, B.: Concerning the size of logical clocks in distributed systems. Inf. Process. Lett. **39**(1), 11–16 (1991)
3. Chow, M., Meisner, D., Flinn, J., Peek, D., Wenisch, T.: The mystery machine: End-to-end performance analysis of large-scale internet services. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pp. 217–231 (2014)
4. Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, California, USA, May 20-21, 1991, pp. 167–174 (1991)
5. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pp. 251–264. USENIX Association, Berkeley, CA, USA (2012). URL `http://dl.acm.org/citation.cfm?id=2387880.2387905`
6. Demirbas, M., Kulkarni, S.: Beyond truetime: Using augmentedtime for improving google spanner. LADIS '13:

7th Workshop on Large-Scale Distributed Systems and Middleware (2013)
7. Fidge, J.: Timestamps in message-passing systems that preserve the partial ordering. Proceedings of the 11th Australian Computer Science Conference **10**(1), 56–66 (1988)
8. Garg, V.K., Chase, C.: Distributed algorithms for detecting conjunctive predicates. International Conference on Distributed Computing Systems pp. 423–430 (1995)
9. Heinzelman, W.B., Chandrakasan, A.P., Balakrishnan, H.: An application-specific protocol architecture for wireless microsensor networks. IEEE Transactions on Wireless Communications **1**(4), 660–670 (2002)
10. Kandris, D., Tsioumas, P., Tzes, A., Nikolakopoulos, G., Vergados, D.D.: Power conservation through energy efficient routing in wireless sensor networks. Sensors **9**(9), 7320–7342 (2009)
11. Kulkarni, S.S., Arumugam, M.: Infuse: A TDMA based data dissemination protocol for sensor networks. IJDSN **2**(1), 55–78 (2006)
12. Kulkarni, S.S., Demirbas, M., Madappa, D., Avva, B., Leone, M.: Logical physical clocks. In: 18th International Conference on Principles of Distributed Systems OPODIS 2014, vol. 8878, pp. 17–32 (2014)
13. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978)
14. Lu, H., Veeraraghavan, K., Ajoux, P., Hunt, J., Song, Y.J., Tobagus, W., Kumar, S., Lloyd, W.: Existential consistency: measuring and understanding consistency at facebook. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 295–310. ACM (2015)
15. Mattern, F.: Virtual time and global states of distributed systems. Parallel and Distributed Algorithms pp. 215–226 (1989)
16. Mills, D.: A brief history of ntp time: Memoirs of an internet timekeeper. ACM SIGCOMM Computer Communication Review **33**(2), 9–21 (2003)
17. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015, pp. 494–503 (2015). DOI 10.1109/IPDPS.2015.95. URL `https://doi.org/10.1109/IPDPS.2015.95`
18. Nguyen, D.: Supplementary materials (source code and raw experimental results) for the paper Precision, Recall, and Sensitivity of Monitoring Partially Synchronous Distributed Programs (2020). DOI 10.5281/zenodo.3778190. URL `https://doi.org/10.5281/zenodo.3778190`
19. Nguyen, D.: Quasi-asynchronous Monitors: Supplementary materials (source code and raw experimental results) for the paper Precision, Recall, and Sensitivity of Monitoring Partially Synchronous Distributed Programs (2021). DOI 10.5281/zenodo.4557924. URL `https://doi.org/10.5281/zenodo.4557924`
20. Nguyen, D.N., Charapko, A., Kulkarni, S.S., Demirbas, M.: Using weaker consistency models with monitoring and recovery for improving performance of key-value stores. J. Braz. Comp. Soc. **25**(1), 10:1–10:25 (2019)
21. Sigelman, B., Barroso, L., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc. (2010). URL `http://research.google.com/archive/papers/dapper-2010-1.pdf`
22. Stoller, S.: Detecting global predicates in distributed systems with clocks. Distributed Computing **13**(2), 85–98 (2000)
23. Verissimo, P.: Real-time communication. Distributed Systems **2** (1993)
24. Yingchareonthawornchai, S., Kulkarni, S.S., Demirbas, M.: Analysis of bounds on hybrid vector clocks. In: OPODIS 2015, December 14-17, 2015, Rennes, France, pp. 34:1–34:17 (2015). DOI 10.4230/LIPIcs.OPODIS.2015.34. URL `http://dx.doi.org/10.4230/LIPIcs.OPODIS.2015.34`