Distributed Runtime Verification under Partial Synchrony

Ritam Ganguly
Michigan State University, East Lansing, U.S.A.
gangulyr@msu.edu

Anik Momtaz
Michigan State University, East Lansing, U.S.A.
momtazan@msu.edu

Borzoo Bonakdarpour
Michigan State University, East Lansing, U.S.A
borzoo@msu.edu

Abstract

In this paper, we study the problem of runtime verification of distributed applications that do *not* share a global clock with respect to specifications in the linear temporal logics (LTL). Our proposed method distinguishes from the existing work in three novel ways. First, we make a practical assumption that the distributed system under scrutiny is augmented with a clock synchronization algorithm that guarantees bounded clock skew among all processes. Second, we do not make any assumption about the structure of predicates that form LTL formulas. This relaxation allows us to monitor a wide range of applications that was not possible before. Subsequently, we propose a distributed monitoring algorithm by employing SMT solving techniques. Third, given the fact that distributed applications nowadays run on massive cloud services, we extend our solution to a parallel monitoring algorithm to utilize the available computing infrastructure. We report on rigorous synthetic as well as real-world case studies and demonstrate that scalable online monitoring of distributed applications is within our reach.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Distributed computing models

Keywords and phrases Runtime monitoring, Distributed systems, Formal methods, Cassandra

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.20

Funding *Borzoo Bonakdarpour*: This work is partially sponsored by the NSF FMitF Award 1917979.

1 Introduction

A *distributed system* consists of a collection of (possibly) geographically separated processes that attempt to solve a problem by means of communication and local computation. Applications of distributed systems range over small-scale networks of deeply embedded systems to monitoring a collection of sensors in smart buildings to large-scale cluster of servers in cloud services. However, design and analysis of such systems has always been a grand challenge due to their inherent complex structure, amplified by nondeterminism and the occurrence of faults. Reasoning about the correctness of distributed systems is particularly a tedious task, as nondeterministic choice of actions results in combinatorial explosion of possible executions. This makes exhaustive model checking techniques not scalable and under-approximate techniques such as testing not so effective.

In this paper, we advocate for a *runtime verification* (RV) approach, where a monitor observes the behavior of a distributed system at run time and verifies its correctness with respect to a temporal logic formula. Distributed RV has to overcome a significant challenge. Although RV deals with finite executions, due to lack of a global clock, there may potentially

exist events whose order of occurrence cannot be determined by a runtime monitor. Additionally, different orders of events may result in different verification verdicts. Enumerating all possible orders at run time often incurs an exponential blow up, making it impractical. This is of course, on top of the usual monitor overhead to evaluate an execution. For example, consider the distributed computation in Fig. 1, where processes P_1 and P_2 host discrete variables x_1 and x_2 , respectively. Let us also consider LTL formula $\varphi = \bigcirc(x_1 + x_2 \leq 1)$. Since events $x_1 = 1$ and $x_2 = 2$ are *concurrent* (i.e., it is not possible to determine which happened before or after which in the absence of a global clock), the formula can be evaluated to both true and false, depending upon different order of occurrences of these events. Handling concurrent events generally results in combinatorial enumeration of all possibilities and, hence, intractability of distributed RV. Existing distributed RV techniques operate in two extremes: they either assume a global clock [1], which is unrealistic for large-scale distributed settings or assume complete asynchrony [20, 19], which do not scale well.

We propose a sound and complete solution to the problem of distributed RV with respect to LTL formulas by incorporating a middle-ground approach. Our solution uses a fault-proof central monitor, and may be summarized as follows. In order to remedy the explosion of different interleavings, we make a practical assumption, that is, a *bounded skew* ϵ between local clocks of every pair of processes, guaranteed by a fault-proof clock synchronization algorithm (e.g., NTP [17]). This means time instants from different clocks within ϵ are considered concurrent, i.e., it is not possible to determine their order of occurrence. This setting constitutes *partial synchrony*, which does not assume a global clock but limits the impact of asynchrony within clock drifts. Following the work in [14], we augment the classic *happened-before* relation [16] with the bounded skew assumption. This way, concurrent events are limited to those that happen within the ϵ time window, and those cannot be ordered according to communication. We transform our monitoring decision problem into an SMT solving problem. The SMT instance includes constraints that encode (1) our monitoring algorithm based on the 3-valued semantics of LTL [2], (2) behavior of communicating processes and their local state changes in terms of a distributed computation, and (3) the happened-before relation subject to the ϵ clock skew assumption. Then, it attempts to concretize an uninterpreted function whose evaluation provides the possible verdicts of the monitor with respect to the given computation. Furthermore, given the fact that distributed applications nowadays run on massive cloud services, we extend our solution to a parallel monitoring algorithm to utilize the available computing infrastructure and achieve better scalability.

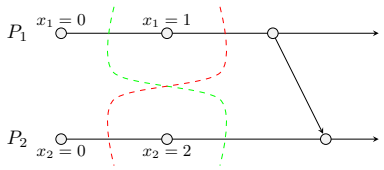


Figure 1 Distributed computation.

We have fully implemented our techniques and report results of rigorous experiments on monitoring synthetic data, as well as monitoring consistency conditions in data centers that run Cassandra [15] as their distributed database management system. We make the following observations. First, although our approach is based on SMT solving, it can be employed for offline monitoring (e.g., log analysis) as well as online monitoring for less intensive applications such as consistency checking in Google Drive. Secondly, we show how the structure of global predicates (e.g., conjunctive vs. disjunctive) and LTL formulas affect the performance of monitoring. Third, we illustrate how monitoring overhead is *in*dependent of the clock skews when practical clock synchronization protocols are applied, making the drift sufficiently small. Finally, we demonstrate how our parallel monitoring algorithm achieves scalability, especially for predicate detection.

Organization. Section 2 presents the background concepts. Our SMT-based solution is

described in Section 3, while experimental results are analyzed in Section 4. Related work is discussed in Section 5. Finally, we make concluding remarks in Section 6.

2 Preliminaries

2.1 Linear Temporal Logic (LTL) for RV

Let AP be a set of *atomic propositions* and $\Sigma = 2^{\mathsf{AP}}$ be the set of all possible *states*. A *trace* is a sequence $s_0 s_1 \cdots$, where $s_i \in \Sigma$ for every $i \geq 0$. We denote by Σ^* (resp., Σ^ω) the set of all finite (resp., infinite) traces. For a finite trace $\alpha = s_0 s_1 \cdots s_k$, $|\alpha|$ denotes its *length*, $k+1$. Also, for $\alpha = s_0 s_1 \cdots s_k$, by α^i, we mean trace $s_i s_{i+1} \cdots s_k$ of α.

The syntax and semantics of the *linear temporal logic* (LTL) [21] are defined for infinite traces. The syntax is defined by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \varphi$$

where $p \in \mathsf{AP}$, and where $\bigcirc$ and $\mathcal{U}$ are the 'next' and 'until' temporal operators respectively. We view other propositional and temporal operators as abbreviations, that is, $\texttt{true} = p \vee \neg p$, $\texttt{false} = \neg\texttt{true}$, $\varphi \rightarrow \psi = \neg\varphi \vee \psi$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\Diamond\varphi = \texttt{true}\,\mathcal{U}\,\varphi$ (*eventually* φ), and $\Box\varphi = \neg\Diamond\neg\varphi$ (*always* φ).

The infinite-trace semantics of LTL is defined as follows. Let $\sigma = s_0 s_1 s_2 \cdots \in \Sigma^\omega$, $i \geq 0$, and let $\models$ denote the *satisfaction* relation:

$$\begin{array}{lll}
\sigma, i \models p & \text{iff} & p \in s_i \\
\sigma, i \models \neg\varphi & \text{iff} & \sigma, i \not\models \varphi \\
\sigma, i \models \varphi_1 \vee \varphi_2 & \text{iff} & \sigma, i \models \varphi_1 \text{ or } \sigma, i \models \varphi_2 \\
\sigma, i \models \bigcirc\varphi & \text{iff} & \sigma, i+1 \models \varphi \\
\sigma, i \models \varphi_1 \,\mathcal{U}\, \varphi_2 & \text{iff} & \exists k \geq i . \sigma, k \models \varphi_2 \text{ and } \forall j \in [i,k) : \sigma, j \models \varphi_1
\end{array}$$

Also, $\sigma \models \varphi$ holds if and only if $\sigma, 0 \models \varphi$ holds.

In the context of RV, the 3-valued LTL (LTL_3 for short) [2] evaluates LTL formulas for *finite* traces, but with an eye on possible future extensions. In LTL_3, the set of truth values is $\mathbb{B}_3 = \{\top, \bot, ?\}$, where $\top$ (resp., $\bot$) denotes that the formula is *permanently* satisfied (resp., violated), no matter how the current finite trace extends, and '?' denotes an *unknown* verdict, i.e., there exists an extension that can violate the formula, and another extension that can satisfy the formula. Let $\alpha \in \Sigma^*$ be a non-empty finite trace. The truth value of an LTL_3 formula φ with respect to α, denoted by $[\alpha \models_3 \varphi]$, is defined as follows:

$$[\alpha \models_3 \varphi] = \begin{cases} \top & \text{if} \quad \forall \sigma \in \Sigma^\omega : \alpha\sigma \models \varphi \\ \bot & \text{if} \quad \forall \sigma \in \Sigma^\omega : \alpha\sigma \not\models \varphi \\ ? & \text{otherwise.} \end{cases}$$

For example, consider formula $\varphi = \Box p$, and a finite trace $\alpha = s_0 s_1 \cdots s_n$. If $p \notin s_i$ for some $i \in [0,n]$, then $[\alpha \models_3 \varphi] = \bot$, that is, the formula is permanently violated. Now, consider formula $\varphi = \Diamond p$. If $p \notin s_i$ for all $i \in [0,n]$, then $[\alpha \models_3 \varphi] = ?$. This is because there exist infinite extensions to α that can satisfy or violate φ in the infinite semantics of LTL.

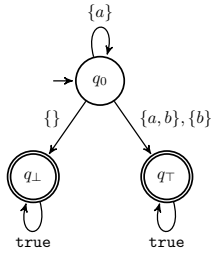


Figure 2 LTL_3 monitor for $\varphi = a \,\mathcal{U}\, b$.

▶ **Definition 1.** *The LTL_3 monitor for a formula φ is the unique deterministic finite state machine $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$, where Q is the set of states, q_0 is the initial state, $\delta : Q \times \Sigma \rightarrow$*

Q *is the transition function, and* $\lambda : Q \to \mathbb{B}_3$ *is a function such that* $\lambda\big(\delta(q_0, \alpha)\big) = [\alpha \models_3 \varphi]$, *for every finite trace* $\alpha \in \Sigma^*$. ■

For example, Fig. 2, shows the monitor automaton for formula $\varphi = a \,\mathcal{U}\, b$.

2.2 Distributed Computations

We assume a loosely coupled asynchronous message passing system, consisting of n reliable processes (that do not fail), denoted by $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, without any shared memory or global clock. Channels are assumed to be FIFO, and lossless. In our model, each local state change is considered an event, and every message activity (send or receive) is also represented by a new event. Message transmission does not change the local state of processes and the content of a message is immaterial to our purposes. We will need to refer to some global clock which acts as a '*real*' timekeeper. It is to be understood, however, that this global clock is a theoretical object used in definitions, and is *not* available to the processes.

We make a practical assumption, known as *partial synchrony*. The *local clock* (or time) of a process P_i, where $i \in [1, n]$, can be represented as an increasing function $c_i : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, where $c_i(\chi)$ is the value of the local clock at global time χ. Then, for any two processes P_i and P_j, we have:

$$\forall \chi \in \mathbb{R}_{\geq 0}.|c_i(\chi) - c_j(\chi)| < \epsilon$$

with $\epsilon > 0$ being the maximum *clock skew*. The value ϵ is assumed to be fixed and known by the monitor in the rest of this paper. In the sequel, we make it explicit when we refer to 'local' or 'global' time. This assumption is met by using a clock synchronization algorithm, like NTP [17], to ensure bounded clock skew among all processes.

An *event* in process P_i is of the form $e^i_{\tau,\sigma}$, where σ is *logical time* (i.e., a natural number) and τ is the local time at global time χ, that is, $\tau = c_i(\chi)$. We assume that for every two events $e^i_{\tau,\sigma}$ and $e^i_{\tau',\sigma'}$, we have $(\tau < \tau') \Leftrightarrow (\sigma < \sigma')$.

▶ **Definition 2.** *A* distributed computation *on N processes is a tuple $(\mathcal{E}, \rightsquigarrow)$, where $\mathcal{E}$ is a set of events partially ordered by Lamport's* happened-before *($\rightsquigarrow$) relation [16], subject to the partial synchrony assumption:*

- *In every process P_i, $1 \leq i \leq N$, all events are totally ordered, that is,*

$$\forall \tau, \tau' \in \mathbb{R}_+.\forall \sigma, \sigma' \in \mathbb{Z}_{\geq 0}.(\sigma < \sigma') \to (e^i_{\tau,\sigma} \rightsquigarrow e^i_{\tau',\sigma'}).$$

- *If e is a message send event in a process, and f is the corresponding receive event by another process, then we have $e \rightsquigarrow f$.*
- *For any two processes P_i and P_j, and any two events $e^i_{\tau,\sigma}, e^j_{\tau',\sigma'} \in \mathcal{E}$, if $\tau + \epsilon < \tau'$, then $e^i_{\tau,\sigma} \rightsquigarrow e^j_{\tau',\sigma'}$, where ϵ is the maximum clock skew.*
- *If $e \rightsquigarrow f$ and $f \rightsquigarrow g$, then $e \rightsquigarrow g$.* ■

▶ **Definition 3.** *Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, a subset of events $C \subseteq \mathcal{E}$ is said to form a* consistent cut *iff when C contains an event e, then it contains all events that happened-before e. Formally, $\forall e \in \mathcal{E}.(e \in C) \wedge (f \rightsquigarrow e) \to f \in C$.* ■

The *frontier* of a consistent cut C, denoted $\mathsf{front}(C)$ is the set of events that happen last in the cut. $\mathsf{front}(C)$ is a set of e^i_{last} for each $i \in [1, |\mathcal{P}|]$ and $e^i_{last} \in C$. We denote e^i_{last} as the last event in P_i such that $\forall e^i_{\tau,\sigma} \in \mathcal{E}.(e^i_{\tau,\sigma} \neq e^i_{last}) \to (e^i_{\tau,\sigma} \rightsquigarrow e^i_{last})$.

2.3 Problem Statement

Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, a *valid* sequence of consistent cuts is of the form $C_0C_1C_2\cdots$, where for all $i \geq 0$, we have (1) $C_i \subset C_{i+1}$, and (2) $|C_i| + 1 = |C_{i+1}|$. Let $\mathcal{C}$ denote the set of all valid sequences of consistent cuts. We define the set of all traces of $(\mathcal{E}, \rightsquigarrow)$ as follows:

$$\mathsf{Tr}(\mathcal{E}, \rightsquigarrow) = \Big\{\mathsf{front}(C_0)\mathsf{front}(C_1)\cdots \mid C_0C_1C_2\cdots \in \mathcal{C}\Big\}.$$

Now, the evaluation of an LTL formula φ with respect to $(\mathcal{E}, \rightsquigarrow)$ in the 3-valued semantics is the following:

$$[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] = \Big\{(\alpha, \rightsquigarrow) \models_3 \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow)\Big\}$$

This means evaluating a distributed computation with respect to a formula results in a *set* of verdicts, as a computation may involve several traces.

2.4 Hybrid Logical Clocks

A *hybrid logical clock* (HLC) [14] is a tuple (τ, σ, ω) for detecting one-way causality, where τ is the local time, σ ensures the order of send and receive events between two processes, and ω indicates causality between events. Thus, in the sequel, we denote an event by $e^i_{\tau,\sigma,\omega}$. More specifically, for a set $\mathcal{E}$ of events:

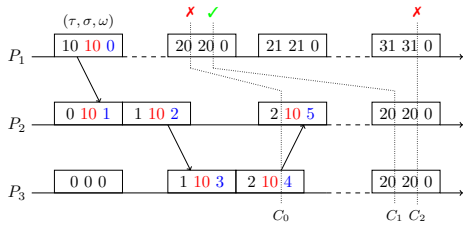


Figure 3 HLC example.

- τ is the local clock value of events, where for any process P_i and two events $e^i_{\tau,\sigma,\omega}, e^i_{\tau',\sigma',\omega'} \in \mathcal{E}$, we have $\tau < \tau'$ iff $e^i_{\tau,\sigma,\omega} \rightsquigarrow e^i_{\tau',\sigma',\omega'}$.
- σ stipulates the logical time, where:
  - For any process P_i and any event $e^i_{\tau,\sigma,\omega} \in \mathcal{E}$, τ never exceeds σ, and their difference is bounded by ϵ (i.e, $\sigma - \tau \leq \epsilon$).
  - For any two processes P_i and P_j, and any two events $e^i_{\tau,\sigma,\omega}, e^j_{\tau',\sigma',\omega'} \in \mathcal{E}$, where event $e^i_{\tau,\sigma,\omega}$ receiving a message sent by event $e^j_{\tau',\sigma',\omega'}$, σ is updated to $\max\{\sigma, \sigma', \tau\}$. The maximum of the three values are chosen to ensure that σ remains updated with the largest τ observed so far. Observe that σ has similar behavior as τ, except the communication between processes has no impact on the value of τ for an event.
- $\omega : \mathcal{E} \to \mathbb{Z}_{\geq 0}$ is a function that maps each event in $\mathcal{E}$ to the causality updates, where:
  - For any process P_i and a send or local event $e^i_{\tau,\sigma,\omega} \in \mathcal{E}$, if $\tau < \sigma$, then ω is incremented. Otherwise, ω is reset to 0.
  - For any two processes P_i and P_j and any two events $e^i_{\tau,\sigma,\omega}, e^j_{\tau',\sigma',\omega'} \in \mathcal{E}$, where event $e^i_{\tau,\sigma,\omega}$ receiving a message sent by event $e^j_{\tau',\sigma',\omega'}$, $\omega(e^i_{\tau,\sigma,\omega})$ is updated based on $\max\{\sigma, \sigma', \tau\}$.
  - For any two processes P_i and P_j, and any two events $e^i_{\tau,\sigma,\omega}, e^j_{\tau',\sigma',\omega'} \in \mathcal{E}$, $(\tau = \tau') \wedge (\omega < \omega') \to e^i_{\tau,\sigma,\omega} \rightsquigarrow e^j_{\tau',\sigma',\omega'}$.

In our implementation of HLC, we assume that it is fault-proof. Fig. 3 shows an HLC incorporated partially synchronous concurrent timelines of three processes with $\varepsilon = 10$. Observe that the local times of all events in $\mathsf{front}(C_1)$ are bounded by ε. Therefore, C_1 is a consistent cut, but C_0 and C_2 are not.

3 SMT-based Solution

3.1 Overall Idea

Recall from Section 1 (Fig. 1) that monitoring a distributed computation may result in multiple verdicts depending upon different ordering of events. In other words, given a distributed computation $(\mathcal{E}, \rightsquigarrow)$ and an LTL formula φ, different ordering of events may reach different states in the monitor automaton $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ (as defined in Definition 1). In order to ensure that all possible verdicts are explored, we generate an SMT instance for (1) the distributed computation $(\mathcal{E}, \rightsquigarrow)$, and (2) each possible path in the LTL_3 monitor. Thus, the corresponding decision problem is the following: given $(\mathcal{E}, \rightsquigarrow)$ and a monitor path $q_0 q_1 \cdots q_m$ in an LTL_3 monitor, can $(\mathcal{E}, \rightsquigarrow)$ reach q_m? If the SMT instance is satisfiable, then $\lambda(q_m)$ is a possible verdict. For example, for the monitor in Fig. 2, we consider two paths $q_0^* q_\perp$ and $q_0^* q_\top$ (and, hence, two SMT instances). Thus, if both instances turn out to be unsatisfiable, then the resulting monitor state is q_0, where $\lambda(q_0) = ?$.

We note that since LTL_3 monitors may contain cycles, we first transform the monitor into an acyclic monitor. To this end, we collapse each cycle into one state with a self-loop labeled by the sequence of events on the cycle (see Fig. 4 for an example). In the next two subsections, we present the SMT entities and constraints with respect to *one* monitor path and a distributed computation.

3.2 SMT Entities

We now introduce the entities that represent a path in an LTL_3 monitor $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ for LTL formula φ and distributed computation $(\mathcal{E}, \rightsquigarrow)$.

Monitor automaton. Let $q_0 \xrightarrow{s_0} q_1 \xrightarrow{s_1} \cdots (q_j \xrightarrow{s_j} q_j)^* \cdots \xrightarrow{s_{m-1}} q_m$ be a path of monitor $\mathcal{M}_\varphi$, which may or may not include a self-loop. We include a non-negative integer variable k_i for each transition $q_i \xrightarrow{s_i} q_{i+1}$, where $i \in [0, m-1]$ and $s_i \in \Sigma$. Observe that we include only one non-negative integer variable k_j for the self-loop $q_j \xrightarrow{s_j} q_j$.

Distributed computation. In our SMT encoding, we represent the set $\mathcal{E}$ by a bit-vector for efficiency. However, for simplicity, we keep referring to the events in a distributed computation by the set $\mathcal{E}$. In order to express the happened-before relation in our SMT encoding, we conduct a pre-processing phase, where we create an $|\mathcal{E}| \times |\mathcal{E}|$ matrix E, such that $E[i,j] = 1$, if $E[i] \rightsquigarrow E[j]$, else $E[i,j] = 0$. This pre-processing phase incorporates the HLC algorithm, described in Section 2.4, to construct the matrix. In the sequel, for simplicity, we keep using the $\rightsquigarrow$ relation between events when needed.

In order to establish the connection between events and atomic propositions in AP based on which the LTL formula φ is constructed, we introduce a Boolean function $\mu : \mathcal{E} \times \Sigma \to \{\texttt{true}, \texttt{false}\}$. We note that if processes have non-Boolean variables and more complex relational predicates (e.g., $x_1 + x_2 \geq 2$), then function μ can be defined accordingly. Finally, in order to identify the sequence of consistent cuts whose run on the monitor starts from q_0 and ends in q_m, we introduce an *uninterpreted* function $\rho : \mathbb{Z}_{\geq 0} \to 2^{\mathcal{E}}$. That is, if the SMT

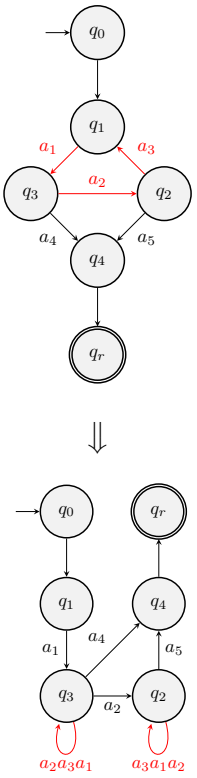


Figure 4 LTL_3 Monitor cycle.

24th International Conference on Principles of Distributed Systems (OPODIS 2020).
Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:6–20:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

instance is satisfiable, then the interpretation of ρ is the sequence of consistent cuts that ends in monitor state q_m. Otherwise, no ordering of concurrent events results in the verdict given by state q_m.

3.3 SMT Constraints

Once we define the necessary SMT entities, we move onto the SMT constraints.

Consistent cut constraints over ρ. We first identify the constraints over uninterpreted function ρ, whose interpretation is a sequence of consistent cuts that starts and ends in the given monitor automaton path. Thus, we first require that each element in the range of ρ must be a consistent cut:

$$\forall i \in [0,m].\forall e, e' \in \mathcal{E}.\Big((e' \rightsquigarrow e) \wedge (e \in \rho(i))\Big) \rightarrow \Big(e' \in \rho(i)\Big)$$

Next, we require the sequence of consistent cuts that ρ identifies to start from an empty set of events and in each consistent cut of the sequence, there is one more event in the successor cut:

$$\forall i \in [0,m].\ |\rho(i+1)| \ = \ |\rho(i)| + 1$$

Finally, the progression of consistent cuts should yield a subset relation. Otherwise, the successor of a consistent cut is not an immediately reachable cut in $(\mathcal{E}, \rightsquigarrow)$:

$$\forall i \in [0,m].\ \rho(i) \subseteq \rho(i+1)$$

Monitoring constraints over ρ. These constraints are responsible for generating a valid sequence of consistent cuts given a distributed computation $(\mathcal{E}, \rightsquigarrow)$ that runs on monitor path $q_1 \xrightarrow{s_1} q_2 \cdots q_j^* \cdots \xrightarrow{s_{m-1}} q_m$. We begin with interpreting $\rho(k_m)$ by requiring that running $(\mathcal{E}, \rightsquigarrow)$ ends in monitor state q_m. The corresponding SMT constraint is:

$$\mu(\mathsf{front}(\rho(k_m)), s_{m-1})$$

For every monitor state q_i, where $i \in [0, m-1]$, if q_i does not have a self-loop, the corresponding SMT constraint is:

$$\mu(\mathsf{front}(\rho(k_{i+1}-1)), s_i) \ \wedge \ (k_i = k_{i+1} - 1)$$

For every monitor state q_j, where $j \in [0, m-1]$, suppose q_j has a self-loop (recall that a cycle of r transitions in the monitor automaton is collapsed into a self-loop labeled by a sequence of r letters). Let us imagine that this self-loop executed z number of times for some $z \geq 0$. Furthermore, we denote the sequence of letters in the self-loop as $s_{j_1} s_{j_2} \cdots s_{j_r}$. The corresponding SMT constraint is:

$$\bigwedge_{i=1}^{z} \bigwedge_{n=1}^{r} \mu\Big(\mathsf{front}\big(\rho(k_j + r(i-1) + n)\big), s_{j_n}\Big)$$

Again, since z is a free variable in the above constraint, the solver will identify some value $z \geq 0$ which is exactly what we need. To ensure that the domain of ρ starts from the empty consistent cut (i.e., $\rho(0) = \emptyset$), we add:

$$k_0 = 0.$$

Finally, let C denote the conjunction of all the above constraints. Recall that this conjunction is with respect to only one monitor path from q_0 to q_m. Since there may be

multiple paths in the monitor automaton that can reach q_m from q_0, we replicate the above constraints for each such path. Suppose there are n such paths and let $C_1, C_2, \ldots, C_n$ be the corresponding SMT constraints for these n paths. We include the following constraint:

$$C_1 \vee C_2 \vee C_3 \vee \cdots \vee C_n$$

This means that if the SMT instance is satisfiable, then computation $(\mathcal{E}, \rightsquigarrow)$ can reach monitor state q_m from q_0.

3.4 Segmentation of Distributed Computation

Since the RV problem is known to be NP-complete even for detecting predicates [9] (i.e., LTL formula of the form $\Box p$) in the sizes of the processes, we are inherently dealing with a computationally difficult problem. This complexity can grow to higher classes in the presence of nested temporal operators. In order to cope with this complexity, our strategy is to chop a computation $(\mathcal{E}, \rightsquigarrow)$ into a sequence of small *segments* $(seg_1, \rightsquigarrow)(seg_2, \rightsquigarrow)\cdots(seg_g, \rightsquigarrow)$ to create more but smaller-size SMT problems. This is likely to improve the overall performance dramatically. More specifically, in a computation whose duration is l, for g number of segments (i.e., segment duration $\frac{l}{g} \pm \epsilon$), the set of events in segment j, where $j \in [1, g]$, is the following:

$$seg_j = \left\{ e^n_{\tau,\sigma,\omega} \mid \sigma \in [\max\{0, \frac{(j-1)l}{g} - \epsilon\}, \frac{jl}{g}] \wedge n \in [1, |\mathcal{P}|] \right\}$$

Observe that monitoring a segment has to be conducted from ϵ time units before the segment actually starts. Also, when monitoring segment j is concluded, monitoring segment $j+1$ should start from all possible monitor states that can be reached by segment j. In Section 4, we show the impact of segmentation on the overall performance of monitoring.

We now show that the verification of a sequence of segments of a distributed computation results in the same set of verdict as verification of the computation in one shot. This can be formally proved by construction as follows. Given $(\mathcal{E}, \rightsquigarrow)$ and φ, where $(\mathcal{E}, \rightsquigarrow)$ is chopped into two segments $(seg_1, \rightsquigarrow)$ and $(seg_2, \rightsquigarrow)$, we have: $[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] = [(seg_1 seg_2, \rightsquigarrow) \models_3 \varphi]$. Let Q_1 be the set of all reachable monitor states at the end of verifying $(seg_1, \rightsquigarrow)$. This set represents the valuation of $(seg_1, \rightsquigarrow)$ with respect to φ. Since in our algorithm verification of $(seg_2, \rightsquigarrow)$ starts with states in Q_1 as initial states of the monitor, we do not lose the temporal order of events. In other words, Q_1 encodes all the important observations in $(seg_1, \rightsquigarrow)$. This implies that by construction, the set Q_2 of reachable monitor states after verification of $(seg_2, \rightsquigarrow)$ starting from Q_1 is the set of all reachable monitor states when verifying $(\mathcal{E}, \rightsquigarrow)$. By induction, the same can be proved for g segments.

3.5 Parallelized Monitoring

Many cloud services use clusters of computers equipped with multiple processors and computing cores. This allows them to deal with high data rates and implement high-performance parallel/distributed applications. Monitoring such applications should also be able to exploit the massive infrastructure. To this end, we now discuss parallelization of our SMT-based monitoring technique.

Let G be a sequence of g segments $G = seg_1 seg_2 \cdots seg_g$. Our idea is to create a job queue for each available computing core, and then distributing the segments evenly across all the queues to be monitored by their respective cores independently. However, simply distributing all the segments across cores is not enough for obtaining a correct result. For

example, consider formula $\varphi = a\,\mathcal{U}\,b$ and two segments, seg_1 and seg_2 across two cores, Cr_1 and Cr_2, respectively. In order for the monitor running on Cr_2 to give the correct verdict, it must know the result of the monitor running on Cr_1. In a scenario, where Cr_1 observes one or more $\neg a$ in seg_1, a violation must be reported even if Cr_2 does not observe b and no $\neg a$. Generally speaking, the temporal order of events makes independent evaluation of segments impossible for LTL formulas. Of course, some formulas such as *safety* (e.g., $\Box p$) and *co-safety* (e.g., $\Diamond q$) properties are exceptions.

We address this problem in two steps. Let $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ be an LTL_3 monitor. Our first step is to create a 3-dimensional reachability matrix RM by solving the following SMT decision problem: given a current monitor state $q_j \in Q$ and segment seg_i, can this segment reach monitor state $q_k \in Q$, for all $i \in [1, g]$, and $j, k \in [0, |Q| - 1]$. If the answer to the problem is affirmative, then we mark $RM[i][j][k]$ with `true`, otherwise with `false`. This is illustrated in Fig. 5 for the monitor shown in Fig. 2, where the grey cells are filled arbitrarily with the answer to the SMT problem. This step can be made embarrassingly parallel, where each element of RM can be computed independently by a different computing core. One can optimize the construction of RM by omitting redundant SMT executions. For example, if $RM[i][j][\top] = \texttt{true}$, then $RM[i'][\top][\top] = \texttt{true}$ for all $i' \in [i, |Q| - 1]$. Likewise, if $RM[i][j][\bot] = \texttt{true}$, then $RM[i'][\bot][\bot] = \texttt{true}$ for all $i' \in [i, |Q| - 1]$.

The second step is to generate a verdict reachability tree from RM. The goal of the tree is to check if a monitor state $q_m \in Q$ can be reached from the initial monitor state q_0. This is achieved by setting q_0 as the root and generating all possible paths from q_0 using RM. That is, if $RM[i][k][j] = \texttt{true}$, then we create a tree node with label q_j and add it as a child of the node with the label q_k. Once the tree is generated, if q_m is one of the leaves, only then we can say q_m is reachable from q_0. In general, all leaves of the tree are possible monitoring verdicts. Note that creation of the tree is achieved using a sequential algorithm. For example, Fig.6 shows the verdict reachability tree generated from the matrix in Fig. 5.

	seg_1			seg_2			seg_3			seg_4		
q_0	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$
	T	F	F	T	T	F	T	T	T	T	T	T
$q_\top$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$
	F	F	F	F	T	F	F	T	F	F	T	F
$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$
	F	F	F	F	F	T	F	F	T	F	F	T

Figure 5 Reachability Matrix for $a\,\mathcal{U}\,b$

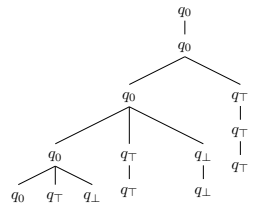


Figure 6 Reachability Tree for $a\,\mathcal{U}\,b$

4 Case Studies and Evaluation

In this section, we evaluate our technique using synthetic experiments and a case study involving Cassandra, a distributed database [1]. We emphasize although RV involves many dimensions such as instrumentation, data collection, data transfer to the monitor, etc., our goal in this section is to evaluate our SMT-based technique, as in a distributed setting, the analysis time is the dominant factor over other types of overhead.

[1] All experimental code and data is available at https://drive.google.com/file/d/19lF-jfUXV-l8ssxuRli1sixw2vctmofA/view?usp=sharing

4.1 Implementation and Experimental Setup

Each experiment in this section consists of two phases: (1) data collection, and (2) verification. We developed a program that randomly generates a distributed computation (i.e., the behavior of a set of processes in terms of their local events and communication). We use a uniform distribution $(0, 2)$ to define the type of the event (computation, send, receive). Then another program observes the execution of these processes and generates a trace log. Then, the monitor attempts to verify the trace log with respect to a given LTL specification using our monitoring algorithm.

We use the *Red Hat OpenStack Platform* servers to generate data. We consider the following *parameters*: (1) number of processes $|\mathcal{P}|$, (2) computation duration l, (3) number of segments g, (4) event rate per process per second r, (5) maximum clock skew ϵ, (6) number of messages sent per second m, and (7) LTL formulas under monitoring, in particular, depth of the monitor automaton d. Our main *metric* to measure is the SMT solving time for each configuration of parameters. Note that in all the plots presented in this section, the time axis is shown in log-scale. When we analyze the effects of one parameter, all the other parameters are held at a relevant constant value. We use a MacBook Pro with Intel i7-7567U(3.5Ghz) processor, 16GB RAM, 512 SSD and Python 3.6.9 interface to the Z3 SMT solver [7]. To evaluate our parallel algorithm, we also use a server with 2x Intel Xeon Platinum 8180 (2.5Ghz) processor, 768GB RAM 112 vcores and python 3.6.9 interface to the Z3 SMT solver [7].

4.2 Analysis of Results – Synthetic Experiments

In this set of experiments we attempt to exhaust all the available parameters and metrics discussed earlier. We aim to put all the parameters to test, and examine how they affect the runtime of the verifier. Since the data generated in this case is synthetic and does not depend on any external factors apart from the system configuration, we induce delay after every event in order to uniformly distribute these events throughout the execution of each process, and to achieve different event rates. That is, the events were generated such that they were evenly spread out over the entire simulation. The value of each of the computation events were selected from a uniform distribution over the set Σ.

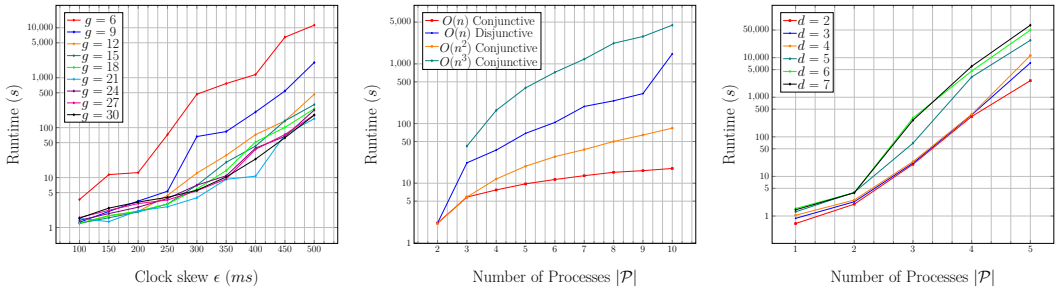
Impact of assuming partial synchrony (single core)

Figure 7a shows that with increase in the value of ϵ, the runtime increases significantly. This is true for different number of segments. This observation demonstrates that employing HLC and assuming bounded clock skew helps in ordering events and as ϵ increases so does the number of concurrent events, and in turn the complexity of verification. Figure 7a also shows that on breaking the computation into smaller segments, the runtime keeps on decreasing for each value of ϵ. We will study the impact of segment duration in other experiments as well.

Impact of predicate structure (single core)

In this experiment (see Fig. 7b), we consider formula $\Box\varphi$, and ensure that it remains true throughout the computation duration. This is to ensure that the monitor does not reach a terminal state in the middle of the computation. We consider the following four different predicate structures for φ:

- $O(n)$ *Conjunctive:* In a system of n processes, φ is a conjunction of n atomic propositions, each depending on the local state of only one process. Over a set of increasing total



(a) Different values ϵ, LTL formula $\Box p$ $(d = 2)$, where p is a disjunctive predicate and $|\mathcal{P}| = 2$.

(b) Different predicate structures with $g = 15$, $d = 2$ and $\epsilon = 250ms$

(c) Different formula with $g = 21$, predicate structure: $O(1)$ conjunctive and $\epsilon = 250ms$

Figure 7 Comparison of how the clock skew ϵ, structure of predicates and different LTL formulas play a role in monitoring with $l = 2s$, $r = 10$, and $m = 1/s$.

number of processes, we observe a linear increase in the runtime. This is somewhat expected, as it is known that monitoring conjunctive predicate is not computationally complex [11].

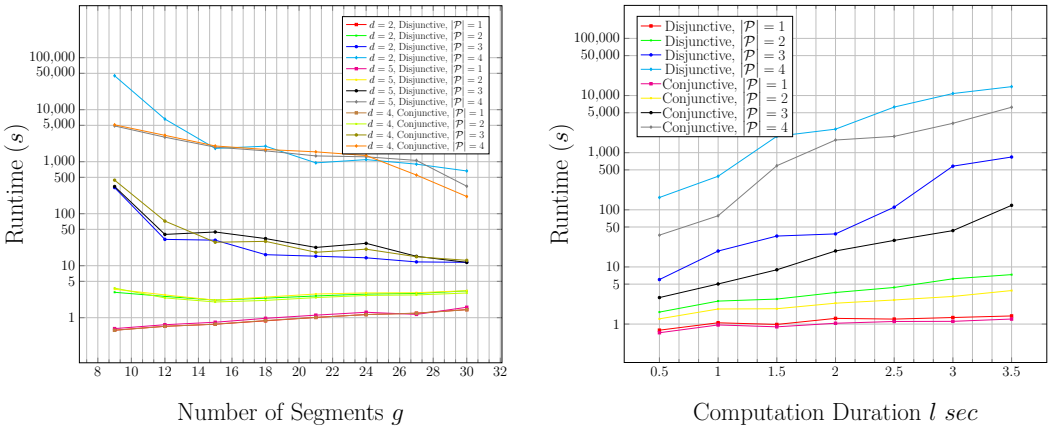
- $O(n)$ *Disjunctive:* Similar to $O(n)$ conjunctive predicates, here, we have a disjunction of n atomic propositions. Compared to its conjunctive counterpart, disjunction of propositions requires more time to verify. This follows the theoretical result that monitoring linear predicates is more complex than monitoring regular predicates [9].
- $O(n^2)$ *Conjunctive:* Here, φ is a conjunction of atomic propositions, where each proposition depends on the state of 2 processes, thereby having a total of $\binom{n}{2}$ predicates. Monitoring such predicates clearly require more time than $O(n)$ conjunctive predicates, but surprisingly less than $O(n)$ disjunctive predicates.
- $O(n^3)$ *Conjunctive:* Here, we consider a conjunction of $\binom{n}{3}$ predicates chosen symbolizing a situation where each predicate is dependent on the state of 3 processes. This case is the most time-consuming structure to monitor.

Impact of LTL formula (single core)

Given an LTL formula, the depth of the monitor automaton d is the length of the longest path from the initial to the accept/reject state. In Fig. 7c, we experimented with the following LTL formulas:

$$
\begin{aligned}
\varphi_1 &= \Box(\neg p) && d = 2\\
\varphi_2 &= \Diamond r \rightarrow (\neg p \,\mathcal{U}\, r) && d = 3\\
\varphi_3 &= \Box((q \wedge \neg r \wedge \Diamond r) \rightarrow (\neg p \,\mathcal{U}\, r)) && d = 4\\
\varphi_4 &= \Box((q \wedge \Diamond r) \rightarrow (\neg p \,\mathcal{U}\, (r \vee (s \wedge \neg p \wedge \bigcirc(\neg p \,\mathcal{U}\, t))))) && d = 5\\
\varphi_5 &= \Diamond r \rightarrow (s \wedge \bigcirc(\neg r \,\mathcal{U}\, t) \rightarrow \bigcirc(\neg r \,\mathcal{U}\, (t \wedge \Diamond p))) \,\mathcal{U}\, r && d = 6\\
\varphi_6 &= \Box((q \wedge \Diamond r) \rightarrow (p \rightarrow (\neg r \,\mathcal{U}\, (s \wedge \neg r \wedge \bigcirc(\neg r \,\mathcal{U}\, t)))) \,\mathcal{U}\, r) && d = 7
\end{aligned}
$$

Clearly, deeper monitors incur greater overhead. The predicate structure used is $O(1)$, meaning that the predicates are in terms of the state of all processes. Runtime for smaller values of d are comparable since the overall runtime is dominated by the evaluation of the uninterpreted function ρ (defined in Section 3). As d increases, it starts to influence the overall runtime of the verification algorithm.



(a) Impact of segment count with $l = 2s$

(b) Impact of computation duration with $g = 21$

Figure 8 Impact of segment count and computation duration with $\epsilon = 250ms$, $O(1)$ predicates, $r = 10$, $m = 1$, and formula $\Box p$.

Impact of segment count (single core)

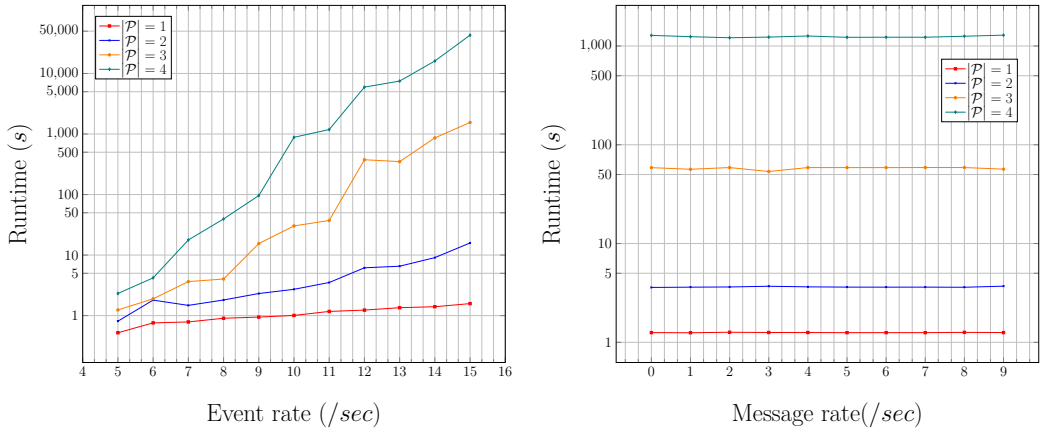
As mentioned in Section 3, we anticipate that chopping a distributed computation into smaller segments tackles the intractability of distributed RV, as it may reduce the number of concurrent events. In Fig. 8a, we observe that the runtime keeps on decreasing with increase in the number of segments per computation duration, until it hits a certain level, after which it does not improve any further. This is due to the fact that the total runtime also contains the time required to set up the SMT solver. With increase in the number of segments, the total time required to setup the SMT solver also increases and dominates the speedup. Also, decreasing the segment duration beyond a certain point does not have any effect on the runtime. This is due to the clock skew ϵ, which makes each segment start from ϵ before. Observe that in Fig. 8a, this result holds for different number of processes, LTL formulas, and conjunctive/disjunctive predicates.

Impact of computation duration (single core)

The computation duration has a direct effect on the size of $\mathcal{E}$, and thus, the number of events in a segment. With a unit increase in the number of events in the SMT formulation, the size of $2^{\mathcal{E}}$ doubles, increasing the SMT solver search space for ρ. This makes the runtime in Fig. 8b increase significantly. Observe that in Fig. 8b, this result holds for different number of processes, and conjunctive/disjunctive predicates.

Impact of the event rate (single core)

Until now, the event rate was fixed at 10 events/sec per process, following the latency time obtained in a real network of replicated database (Cassandra), discussed in detail in Section 4.3. Here, we change the event rate and study its effect on the verification runtime. In Fig. 9a, we see increasing the event rate causes the runtime to increase significantly. This result is valid for different number of processes, though for more processes the increase is more dramatic.



(a) Impact of event rate with $m = 1$

(b) Impact of message passing with $r = 10$

Figure 9 Impact of event and message rates with $l = 2s$, $\epsilon = 250ms$, and $g = 21$, formula $\Box p$ with $O(1)$ predicate structure.

Impact of the message rate (single core)

Consider a send message event $e^i_{\tau,\sigma,\omega}$ and its corresponding receive event $e^j_{\tau',\sigma',\omega'}$. This results in a $\rightsquigarrow$-relation between these two events. Such events are expected to reduce the number of concurrent events and consequently the monitor overhead. However, Fig.9b shows no effect on the monitor run time. This is due to the relatively short $\epsilon = 250ms$, which is actually much larger than the maximum clock skew of off-the-shelf protocols such as NTP. In other words, when ϵ dominates the impact of event ordering that message passing can achieve. This is another reason to believe that partial synchrony is an effective way to deal with distributed RV. We vary messages sent for inter-process communication, from 0 to 9 with 10events/sec.

Impact of parallelization

To demonstrate the drastic increase in performance due to parallelization, we evaluate formula $\Box p$ on a distributed computation with $l = 20s$, $r = 10$, $g = 20$, and $\epsilon = 150ms$, while varying the number of cores from 1 to 100, as shown in Fig. 10a. Observe that beyond 40 cores, there is no significant relative change in runtime regardless of the number of processes, as the time required to build the SMT formulation starts dominating the total run time. This graph also shows that parallelization can result in orders of magnitude speedup.

4.3 Case Study: Cassandra

Facebook developed Cassandra [15] as an open-source, distributed, No-SQL database management system. It is capable of handling large amounts of data across many servers (nodes), spanning over multiple datacenters, providing high availability with no single point of failure, and asynchronous mas-

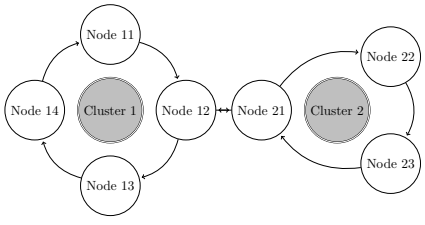
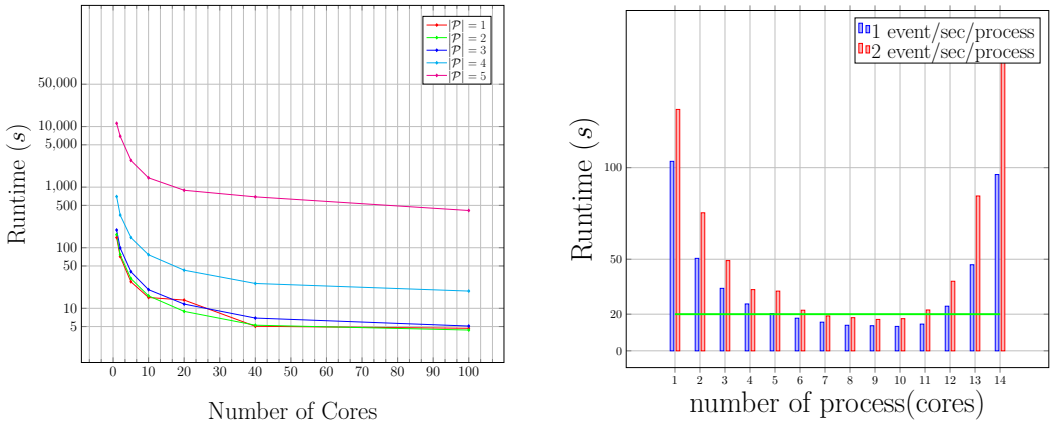


Figure 11 A network with two Cassandra clusters, Node-12 and Node-21 are the seed nodes of the respective clusters



(a) Impact of parallelization with $l = 20s$, $r = 10$, $\epsilon = 150ms$, $g = 200$, formula $\Box p$ with $O(1)$ predicate structure.

(b) Impact of varying event rate on parallelization for Cassandra.

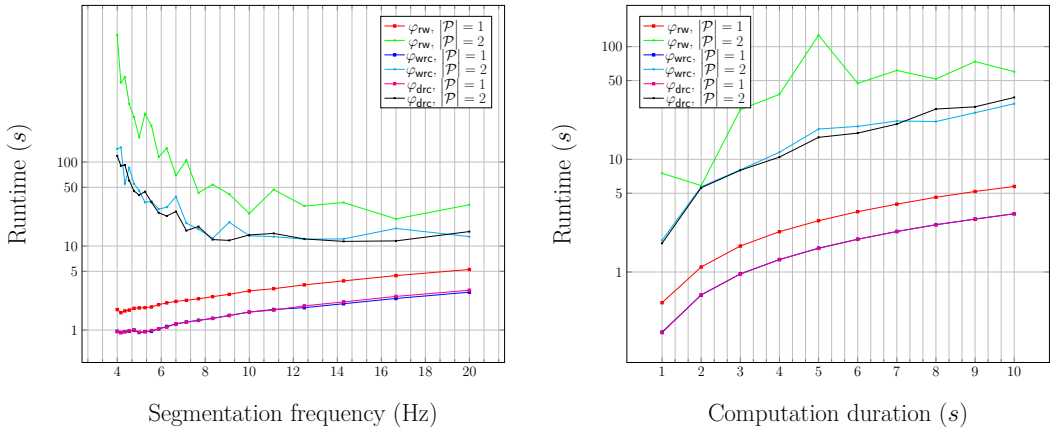
Figure 10 Impact of parallelization.

terless replication of data. Cassandra is designed such that it has no master or slave nodes. All the nodes are part of a ring-type architecture, distributing data equally over all the nodes in the cluster. We simulate a system of multiple processes. Each process is responsible for inter-process communication apart from basic database operations (read, write and update). We deployed a system with two data centers (see Fig. 11), where Cluster 1, contains 4 nodes (Nodes $11-14$) and Cluster 2 contains 3 nodes (Node $21-23$). Node 12 and Node 21 are the seed nodes of the respective clusters. Data is replicated in all the nodes in both the clusters. Each of the nodes is art of the *Red Hat OpenStack Platform* with the following configuration: 4 VCPUs, 4GB RAM, Ubuntu 1804, Cassandra 3.11.6, Java 1.8.0_252, and Python 3.6.9.

We have tested ping time of servers on Google Cloud Platform, Microsoft Azure and Amazon Web Service. The fastest ping was received at $41ms$. In a real-life datacenter, networks used to communicate within the nodes usually have a speed on the scale of few Gigabytes per second. Here, we use a private broadband that offers a speed of 100 Megabytes per second. We measure the latency time of our system to be around $100ms$. We consider this to be our standard and setup all our experiments based on this assumption.

Processes are capable of reading, writing, and updating all entries of the database. The exact type of the event is selected by a uniform distribution $(0, 2)$. Each process selects the available node at run time. In order to prevent deadlocks, no two processes are allowed to connect to the same node at the same time. If there exists no free node at any point of time, it waits for a node to be released and then it continues with the task. Once there is a write or update, the process responsible for the change sends a message to each of the other processes notifying about the change. We assume that a message is read by the receiving process immediately upon receiving. All database operations (i.e. send and receive events) are considered to be separate.

Consistency level in a database dictates the minimum number of replications that needs to perform on an operation in order to consider the operation to be successfully executed. Cassandra recommends that the sum of the read consistency and the write consistency be more than the replication factor for no read or write anomaly in the database. By default, the read and the write consistency level is set to one. For a database with replication factor



(a) Cassandra: Impact of segmentation frequency for $l = 5s$

(b) Cassandra: Impact of computation duration with segmentation frequency 10Hz

Figure 12 Experimental results for Cassandra

3, our goal is to monitor and identify *read/write anomalies* in the database:

$$\varphi_{\mathsf{rw}} = \bigwedge_{i=0}^{n} \Big(write(i) \rightarrow \Diamond\, read(i) \Big)$$

where n is the number of read/write requests.

Since Cassandra does not allow normalization of database, the other two properties we aim to monitor are *write reference check* and *delete reference check*. To give a sense of database normalization, we use a database with two tables:

$$\mathsf{Student}(id, name) \qquad \mathsf{Enrollment}(id, course).$$

We enforce that if there is a write in the Enrollment table, it should be led by a write in the Student table with the same *id*. The *id* and *name* to be written are a random string of length 8. Likewise, in the case of deletion of some entry from Student table, it should be led by deletion of all entries with the same *id* from Enrollment table. These enforce that there is no insertion and delete anomaly, and thereby gives a sense of normalization in Cassandra:

$$\varphi_{\mathsf{wrc}} = \neg\Big(\neg write(\mathsf{Student}.id)\ \mathcal{U}\ write(\mathsf{Enrollment}.id) \Big)$$

$$\varphi_{\mathsf{drc}} = \neg\Big(\neg delete(\mathsf{Enrollment}.id)\ \mathcal{U}\ delete(\mathsf{Student}.id) \Big)$$

Extreme load scenario

Figures 12a and 12b, plot runtime vs segmentation frequency and runtime vs computation duration, respectively for the case where the processes experience full read/write load that network latency allows. Compared to the results plotted for the synthetic experiments, we see a bit of noise in the result. This owes to the fact that in synthetic experiments, the events are uniformly distributed over the entire computation duration, however, in case of Cassandra, the events are not uniform. Database operations like read, write and update take about 100ms of time but sending and receiving of message is relatively faster taking about 20-30ms making the overall event distribution quite non-uniform.

Moderate load scenario

In Figure 12a, with event rate $r = 10$, we are just about making it even for number of processes as 2 and with a computation duration of $20s$. Now, consider Google Sheets API, which allows maximum 500 requests per 100 seconds per project and 100 requests per seconds per user, i.e., 5 events/sec per project and a user can generate 1 event/sec [12] on an average. To see how our algorithm performs in such a scenario, we increase the number of processes and so as the number of monitoring cores and analyze the time taken to verify such a trace log. We plot our findings in Figure 10b. We see that for processes $8, 9$ and 10, we get the best results for event rate of both 1 and 2 event(s)/sec/process. We emphasize that the 2 event(s)/sec/process is twice more than what Google Sheets allow to happen. This makes us confident that our algorithm can pave the path for implementation in a real-life setting.

4.4 Discussion

One may argue that our results can at best monitor a distributed system with only a few processes in an online fashion. We note that in industrial scale data centers, clusters of high performance servers make it possible to achieve up to one million writes per second (e.g., for Netflix). First, online monitoring of fine-grained read/write consistency is highly unlikely to be useful while the system is operational. Having said that, in a testing scenario, such a cluster of servers should also be monitored with a cluster of servers (typically a machine with 64GB RAM, 8 vCPUs, Intel Xeon processor, with a high network bandwidth is used, which when compared to our machine outperforms when executed in a multi-threaded environment).

Furthermore, observe that for two processes reading/writing or updating tables, with the right segmentation frequency the monitoring overhead competes with the computation duration. Our experiments are encouraging in the sense that by employing acceptable computation power, and incorporating parallel monitors, monitoring highly frequent activities of a data center such as read/write consistency is within our reach.

5 Related Work

Lattice-theoretic centralized and decentralized online predicate detection in asynchronous distributed systems has been extensively studied in [4, 18]. Extensions of this work to include temporal operators appear in [20, 19]. The line of work in [4, 18, 20, 19, 22] operates in a fully asynchronous setting. On the contrary in this paper, we leverage a practical assumption and employ an off-the-shelf clock synchronization algorithm to limit the time window of asynchrony. Predicate detection has been shown to be a powerful tool in solving combinatorial optimization problems [10] and our results show that our approach is pretty effective in handling predicate detection (e.g., Fig. 10b). In [24], the authors study the predicate detection problem using SMT solving. Also, knowledge-based monitoring of distributed processes was first studied in [22]. Here, the authors design a method for monitoring safety properties in distributed systems using the past-time linear temporal logic. This approach, however, suffers from producing false negatives.

Runtime monitoring of LTL formulas for synchronous distributed systems has been studied in [8, 6, 5, 1]. This approach has the shortcoming of assuming a global clock across all distributed processes. Predicate detection for asynchronous system has been studied in [23] but the assumption needed to evaluate happen-before relationship is too strong. We utilize HLC which not only is more realistic but also decreases the level of concurrency. Finally, fault-

tolerant monitoring, where monitors can crash, has been investigated in [3] for asynchronous and in [13] for synchronized global clock with no clock shew across all distributed processes. In this paper, we use a clock synchronization algorithm which guarantees bounded clock shews. Our solution is also SMT based and to our knowledge this is the first SMT based distributed monitoring algorithm for LTL, which results in better scalability.

6 Conclusion and Future Work

In this paper, we focused on runtime verification (RV) of distributed systems. Our SMT-based technique takes as input an LTL formula and a distributed computation (i.e., a collection of communicating processes along with their local events). We employed a partially synchronous model, where a clock synchronization algorithm ensures bounded clock skew among all processes. Such an algorithm significantly limits the impact of full asynchrony and remedies combinatorial explosion of interleavings in a distributed setting. We conducted detailed and rigorous synthetic experiments, as well as a case study on monitoring consistency conditions on Cassandra, a non-SQL replicated database management system used in data centers. Our experiments demonstrate the potential of scalability of our technique to large applications.

As for future work, there are several interesting research directions. Our first step will be to scale up our technique to monitor cloud services with big data. This can be achieved by studying the tradeoff between accuracy and scalability. Another important extension of our work is distributed RV for timed temporal logics. Such expressiveness will allow us to monitor distributed applications that are sensitive to explicit timing constraints. A prominent example of such a setting is in blockchain and cross-chain protocols.

References

1. A. Bauer and Y. Falcone. Decentralised LTL monitoring. *Formal Methods in System Design*, 48(1-2):46–93, 2016.
2. A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14:1–14:64, 2011.
3. B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, D. A. Rosenblueth, and C. Travers. Decentralized asynchronous crash-resilient runtime verification. In *Proceedings of the 27th International Conference on Concurrency Theory (CONCUR)*, pages 16:1–16:15, 2016.
4. H. Chauhan, V. K. Garg, A. Natarajan, and N. Mittal. A distributed abstraction algorithm for online predicate detection. In *Proceedings of the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 101–110, 2013.
5. C. Colombo and Y. Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1-2):109–158, 2016.
6. L. M. Danielsson and C. Sánchez. Decentralized stream runtime verification. In *Proceedings of the 19th International Conference on Runtime Verification (RV)*, pages 185–201, 2019.
7. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.
8. A. El-Hokayem and Y. Falcone. On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. *ACM Transactions on Software Engineering Methodologies*, 29(1):1:1–1:57, 2020.
9. V. K. Garg. *Elements of distributed computing.* Wiley, 2002.
10. V. K. Garg. Predicate detection to solve combinatorial optimization problems. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 235–245. ACM, 2020.
11. V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. *International Conference on Distributed Computing Systems*, pages 423–430, June 1995.

12 Google. Usage limits, sheets api, google developer. https://developers.google.com/sheets/api/limits. Accessed: 2020-09-09.

13 S. Kazemloo and B. Bonakdarpour. Crash-resilient decentralized synchronous runtime verification. In *Proceedings of the 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 207–212, 2018.

14 S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *Proceedings of the 18th International Conference on Principles of Distributed Systems*, pages 17–32, 2014.

15 Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. doi:10.1145/1773912.1773922.

16 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

17 D. Mills. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, RFC Editor, June 2010.

18 N. Mittal and V. K. Garg. Techniques and applications of computation slicing. *Distributed Computing*, 17(3):251–277, 2005.

19 M. Mostafa and B. Bonakdarpour. Decentralized runtime verification of LTL specifications in distributed systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 494–503, 2015.

20 V. A. Ogale and V. K. Garg. Detecting temporal logic predicates on distributed computations. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*, pages 420–434, 2007.

21 A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

22 K. Sen, A. Vardhan, G. Agha, and G.Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 418–427, 2004.

23 Scott D. Stoller. Detecting global predicates in distributed systems with clocks. In Marios Mavronicolas and Philippas Tsigas, editors, *Distributed Algorithms*, pages 185–199, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

24 V. T. Valapil, S. Yingchareonthawornchai, S. S. Kulkarni, E. Torng, and M. Demirbas. Monitoring partially synchronous distributed systems using SMT solvers. In *Proceedings of the 17th International Conference on Runtime Verification (RV)*, pages 277–293, 2017.