

¹ **Distributed Runtime Verification under Partial** ² **Synchrony**

³ **Ritam Ganguly**

⁴ Michigan State University, East Lansing, U.S.A.

⁵ gangulyr@msu.edu

⁶ **Anik Momtaz** 

⁷ Michigan State University, East Lansing, U.S.A.

⁸ momtazan@msu.edu

⁹ **Borzoo Bonakdarpour** 

¹⁰ Michigan State University, East Lansing, U.S.A

¹¹ borzoo@msu.edu

¹² **Abstract**

¹³ In this paper, we study the problem of runtime verification of distributed applications that
¹⁴ do *not* share a global clock with respect to specifications in the linear temporal logics (LTL). Our
¹⁵ proposed method distinguishes from the existing work in three novel ways. First, we make a practical
¹⁶ assumption that the distributed system under scrutiny is augmented with a clock synchronization
¹⁷ algorithm that guarantees bounded clock skew among all processes. Second, we do not make any
¹⁸ assumption about the structure of predicates that form LTL formulas. This relaxation allows us
¹⁹ to monitor a wide range of applications that was not possible before. Subsequently, we propose
²⁰ a distributed monitoring algorithm by employing SMT solving techniques. Third, given the fact
²¹ that distributed applications nowadays run on massive cloud services, we extend our solution to
²² a parallel monitoring algorithm to utilize the available computing infrastructure. We report on
²³ rigorous synthetic as well as real-world case studies and demonstrate that scalable online monitoring
²⁴ of distributed applications is within our reach.

²⁵ **2012 ACM Subject Classification** Theory of computation → Logic and verification; Theory of
²⁶ computation → Distributed computing models

²⁷ **Keywords and phrases** Runtime monitoring, Distributed systems, Formal methods, Cassandra

²⁸ **Digital Object Identifier** [10.4230/LIPIcs.OPODIS.2020.20](https://doi.org/10.4230/LIPIcs.OPODIS.2020.20)

²⁹ **Funding** *Borzoo Bonakdarpour*: This work is partially sponsored by the NSF FMitF Award 1917979.

³⁰ **1 Introduction**

³¹ A *distributed system* consists of a collection of (possibly) geographically separated processes
³² that attempt to solve a problem by means of communication and local computation. Applications
³³ of distributed systems range over small-scale networks of deeply embedded systems
³⁴ to monitoring a collection of sensors in smart buildings to large-scale cluster of servers
³⁵ in cloud services. However, design and analysis of such systems has always been a grand
³⁶ challenge due to their inherent complex structure, amplified by nondeterminism and the
³⁷ occurrence of faults. Reasoning about the correctness of distributed systems is particularly
³⁸ a tedious task, as nondeterministic choice of actions results in combinatorial explosion of
³⁹ possible executions. This makes exhaustive model checking techniques not scalable and
⁴⁰ under-approximate techniques such as testing not so effective.

⁴¹ In this paper, we advocate for a *runtime verification* (RV) approach, where a monitor
⁴² observes the behavior of a distributed system at run time and verifies its correctness with
⁴³ respect to a temporal logic formula. Distributed RV has to overcome a significant challenge.
⁴⁴ Although RV deals with finite executions, due to lack of a global clock, there may potentially



© R. Ganguly and A. Momtaz and B. Bonakdarpour;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:1–20:18



Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

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

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

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



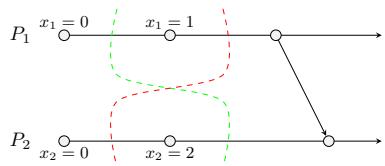
© R. Ganguly and A. Momtaz and B. Bonakdarpour;
 licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:2–20:18



Leibniz International Proceedings in Informatics
 Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Distributed computation.

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 $\Sigma = 2^{AP}$ be the set of all possible *states*. A *trace* is a sequence $s_0s_1\cdots$, 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 $\alpha = s_0s_1\cdots s_k$, $|\alpha|$ denotes its *length*, $k + 1$. Also, for $\alpha = s_0s_1\cdots s_k$, by α^i , we mean trace $s_is_{i+1}\cdots s_k$ 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 $p \in AP$, 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, $\text{true} = p \vee \neg p$, $\text{false} = \neg\text{true}$, $\varphi \rightarrow \psi = \neg\varphi \vee \psi$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\diamond\varphi = \text{true} \mathcal{U} \varphi$ (*eventually* φ), and $\square\varphi = \neg\diamond\neg\varphi$ (*always* φ).

The infinite-trace semantics of LTL is defined as follows. Let $\sigma = s_0s_1s_2\cdots \in \Sigma^\omega$, $i \geq 0$, and let \models denote the *satisfaction* relation:

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

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 $\mathbb{B}_3 = \{\top, \perp, ?\}$, where \top (resp., \perp) 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:

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

For example, consider formula $\varphi = \square p$, and a finite trace $\alpha = s_0s_1\cdots s_n$. If $p \notin s_i$ for some $i \in [0, n]$, then $[\alpha \models_3 \varphi] = \perp$, 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.

► **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, $\delta : Q \times \Sigma \rightarrow$

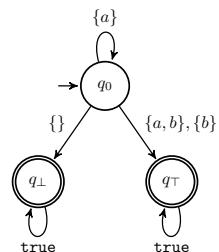


Figure 2 LTL_3 monitor for $\varphi = a \mathcal{U} b$.



© R. Ganguly and A. Momtaz and B. Bonakdarpour;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:3–20:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

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

130 2.2 Distributed Computations

131 We assume a loosely coupled asynchronous message passing system, consisting of n reliable
132 processes (that do not fail), denoted by $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$, without any shared memory or
133 global clock. Channels are assumed to be FIFO, and lossless. In our model, each local state
134 change is considered an event, and every message activity (send or receive) is also represented
135 by a new event. Message transmission does not change the local state of processes and the
136 content of a message is immaterial to our purposes. We will need to refer to some global
137 clock which acts as a ‘real’ timekeeper. It is to be understood, however, that this global clock
138 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 $c_i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$, 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:

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

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

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

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

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

$$\forall \tau, \tau' \in \mathbb{R}_+. \forall \sigma, \sigma' \in \mathbb{Z}_{\geq 0}. (\sigma < \sigma') \rightarrow (e_{\tau, \sigma}^i \rightsquigarrow e_{\tau', \sigma'}^i).$$

149 ■ If e is a message send event in a process, and f is the corresponding receive event by
150 another process, then we have $e \rightsquigarrow f$.
151 ■ For any two processes P_i and P_j , and any two events $e_{\tau, \sigma}^i, e_{\tau', \sigma'}^j \in \mathcal{E}$, if $\tau + \epsilon < \tau'$, then
152 $e_{\tau, \sigma}^i \rightsquigarrow e_{\tau', \sigma'}^j$, where ϵ is the maximum clock skew.
153 ■ If $e \rightsquigarrow f$ and $f \rightsquigarrow g$, then $e \rightsquigarrow g$. ■

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

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



© R. Ganguly and A. Momtaz and B. Bonakdarpour;
159 licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:4–20:18



Leibniz International Proceedings in Informatics
159 Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

160 **2.3 Problem Statement**

Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, a *valid* sequence of consistent cuts is of the form $C_0 C_1 C_2 \dots$, 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:

$$\text{Tr}(\mathcal{E}, \rightsquigarrow) = \left\{ \text{front}(C_0) \text{front}(C_1) \dots \mid C_0 C_1 C_2 \dots \in \mathcal{C} \right\}.$$

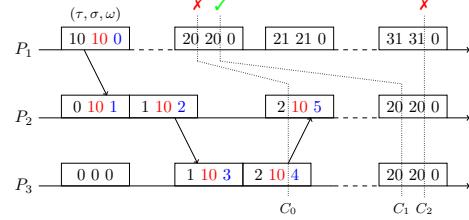
Now, the evaluation of an LTL formula φ with respect to $(\mathcal{E}, \rightsquigarrow)$ in the 3-valued semantics is the following:

$$[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] = \left\{ (\alpha, \rightsquigarrow) \models_3 \varphi \mid \alpha \in \text{Tr}(\mathcal{E}, \rightsquigarrow) \right\}$$

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

163 **2.4 Hybrid Logical Clocks**

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



171 **Figure 3** HLC example.

172 ■ τ is the local clock value of events, where for any process P_i and two events $e_{\tau, \sigma, \omega}^i, e_{\tau', \sigma', \omega'}^i \in \mathcal{E}$, we have $\tau < \tau'$ iff $e_{\tau, \sigma, \omega}^i \rightsquigarrow e_{\tau', \sigma', \omega'}^i$.

173 ■ σ stipulates the logical time, where:

- 174 ■ For any process P_i and any event $e_{\tau, \sigma, \omega}^i \in \mathcal{E}$, τ never exceeds σ , and their difference is bounded by ϵ (i.e., $\sigma - \tau \leq \epsilon$).
- 175 ■ For any two processes P_i and P_j , and any two events $e_{\tau, \sigma, \omega}^i, e_{\tau', \sigma', \omega'}^j \in \mathcal{E}$, where event $e_{\tau, \sigma, \omega}^i$ receiving a message sent by event $e_{\tau', \sigma', \omega'}^j$, 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.
- 176 ■ $\omega : \mathcal{E} \rightarrow \mathbb{Z}_{\geq 0}$ is a function that maps each event in \mathcal{E} to the causality updates, where:
 - 177 ■ For any process P_i and a send or local event $e_{\tau, \sigma, \omega}^i \in \mathcal{E}$, if $\tau < \sigma$, then ω is incremented. Otherwise, ω is reset to 0.
 - 178 ■ For any two processes P_i and P_j and any two events $e_{\tau, \sigma, \omega}^i, e_{\tau', \sigma', \omega'}^j \in \mathcal{E}$, where event $e_{\tau, \sigma, \omega}^i$ receiving a message sent by event $e_{\tau', \sigma', \omega'}^j$, $\omega(e_{\tau, \sigma, \omega}^i)$ is updated based on $\max\{\sigma, \sigma', \tau\}$.
 - 179 ■ For any two processes P_i and P_j , and any two events $e_{\tau, \sigma, \omega}^i, e_{\tau', \sigma', \omega'}^j \in \mathcal{E}$, $(\tau = \tau') \wedge (\omega < \omega') \rightarrow e_{\tau, \sigma, \omega}^i \rightsquigarrow e_{\tau', \sigma', \omega'}^j$.

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



184 © R. Ganguly and A. Momtaz and B. Bonakdarpour;

185 licensed under Creative Commons License CC-BY

186 24th International Conference on Principles of Distributed Systems (OPODIS 2020).

187 Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:5–20:18



188 Leibniz International Proceedings in Informatics

189 Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

193 **3 SMT-based Solution**

194 **3.1 Overall Idea**

195 Recall from Section 1 (Fig. 1) that monitoring a distributed computation may result in
 196 multiple verdicts depending upon different ordering of events. In other words, given a
 197 distributed computation $(\mathcal{E}, \rightsquigarrow)$ and an LTL formula φ , different ordering of events may reach
 198 different states in the monitor automaton $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ (as defined in Definition 1).
 199 In order to ensure that all possible verdicts are explored, we generate an SMT instance
 200 for (1) the distributed computation $(\mathcal{E}, \rightsquigarrow)$, and (2) each possible path in the LTL_3 monitor.
 201 Thus, the corresponding decision problem is the following: given $(\mathcal{E}, \rightsquigarrow)$ and a monitor path
 202 $q_0 q_1 \cdots q_m$ in an LTL_3 monitor, can $(\mathcal{E}, \rightsquigarrow)$ reach q_m ? If the SMT instance is satisfiable, then
 203 $\lambda(q_m)$ is a possible verdict. For example, for the monitor in Fig. 2, we consider two paths
 204 $q_0^* q_\perp$ and $q_0^* q_\top$ (and, hence, two SMT instances). Thus, if both instances turn out to be
 205 unsatisfiable, then the resulting monitor state is q_0 , where $\lambda(q_0) = ?$.

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

211 **3.2 SMT Entities**

212 We now introduce the entities that represent a path in an LTL_3 monitor
 213 $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ for LTL formula φ and distributed computation
 214 $(\mathcal{E}, \rightsquigarrow)$.

215 **Monitor automaton.** Let $q_0 \xrightarrow{s_0} q_1 \xrightarrow{s_1} \cdots (q_j \xrightarrow{s_j} q_j)^* \cdots \xrightarrow{s_{m-1}} q_m$
 216 be a path of monitor \mathcal{M}_φ , which may or may not include a self-loop.
 217 We include a non-negative integer variable k_i for each transition
 218 $q_i \xrightarrow{s_i} q_{i+1}$, where $i \in [0, m-1]$ and $s_i \in \Sigma$. Observe that we include
 219 only one non-negative integer variable k_j for the self-loop $q_j \xrightarrow{s_j} q_j$.

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

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

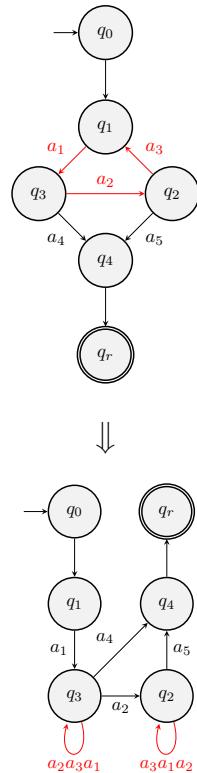


Figure 4 LTL_3 Monitor cycle.



© R. Ganguly and A. Momtaz and B. Bonakdarpour;
 licensed under Creative Commons License CC-BY

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
 Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

240 **3.3 SMT Constraints**

241 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:

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

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:

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

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:

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

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:

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

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, \dots, C_n be the corresponding SMT constraints for these n paths. We include the following constraint:

$$C_1 \vee C_2 \vee C_3 \vee \dots \vee C_n$$

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

244 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* $(seg_1, \rightsquigarrow)(seg_2, \rightsquigarrow) \dots (seg_g, \rightsquigarrow)$ 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:

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

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

249 We now show that the verification of a sequence of segments of a distributed computation
 250 results in the same set of verdict as verification of the computation in one shot. This can be
 251 formally proved by construction as follows. Given $(\mathcal{E}, \rightsquigarrow)$ and φ , where $(\mathcal{E}, \rightsquigarrow)$ is chopped
 252 into two segments $(seg_1, \rightsquigarrow)$ and $(seg_2, \rightsquigarrow)$, we have: $[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] = [(seg_1 \& seg_2, \rightsquigarrow) \models_3 \varphi]$.
 253 Let Q_1 be the set of all reachable monitor states at the end of verifying $(seg_1, \rightsquigarrow)$. This set
 254 represents the valuation of $(seg_1, \rightsquigarrow)$ with respect to φ . Since in our algorithm verification of
 255 $(seg_2, \rightsquigarrow)$ starts with states in Q_1 as initial states of the monitor, we do not lose the temporal
 256 order of events. In other words, Q_1 encodes all the important observations in $(seg_1, \rightsquigarrow)$.
 257 This implies that by construction, the set Q_2 of reachable monitor states after verification of
 258 $(seg_2, \rightsquigarrow)$ starting from Q_1 is the set of all reachable monitor states when verifying $(\mathcal{E}, \rightsquigarrow)$.
 259 By induction, the same can be proved for g segments.

260 3.5 Parallelized Monitoring

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

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



© R. Ganguly and A. Momtaz and B. Bonakdarpour;
 licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).
 Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:8–20:18



Leibniz International Proceedings in Informatics
 Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 $RM[i][j][\top] = \text{true}$, then $RM[i'][\top][\top] = \text{true}$ for all $i' \in [i, |Q| - 1]$. Likewise, if $RM[i][j][\perp] = \text{true}$, then $RM[i'][\perp][\perp] = \text{true}$ for all $i' \in [i, |Q| - 1]$.

288 The second step is to generate a verdict reachability tree from RM . The goal of the tree
 289 is to check if a monitor state $q_m \in Q$ can be reached from the initial monitor state q_0 . This
 290 is achieved by setting q_0 as the root and generating all possible paths from q_0 using RM .
 291 That is, if $RM[i][k][j] = \text{true}$, then we create a tree node with label q_j and add it as a child
 292 of the node with the label q_k . Once the tree is generated, if q_m is one of the leaves, only then
 293 we can say q_m is reachable from q_0 . In general, all leaves of the tree are possible monitoring
 294 verdicts. Note that creation of the tree is achieved using a sequential algorithm. For example,
 295 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_0	q_{\top}	q_{\perp}	q_0	q_{\top}	q_{\perp}	q_0	q_{\top}	q_{\perp}
q_0	T	F	F	T	T	F	T	T	T	T	T
	F	F	F	F	T	F	F	T	F	F	T
q_{\top}	q_0	q_{\top}	q_{\perp}	q_0	q_{\top}	q_{\perp}	q_0	q_{\top}	q_{\perp}	q_0	q_{\top}
	F	F	F	F	T	F	F	T	F	F	T
q_{\perp}	q_0	q_{\top}	q_{\perp}	q_0	q_{\top}	q_{\perp}	q_0	q_{\top}	q_{\perp}	q_0	q_{\top}
	F	F	F	F	F	T	F	F	T	F	F

Figure 5 Reachability Matrix for $a \cup b$

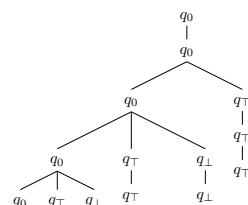


Figure 6 Reachability Tree for $a \cup b$

296 **4 Case Studies and Evaluation**

297 In this section, we evaluate our technique using synthetic experiments and a case study
298 involving Cassandra, a distributed database ¹. We emphasize although RV involves many
299 dimensions such as instrumentation, data collection, data transfer to the monitor, etc., our
300 goal in this section is to evaluate our SMT-based technique, as in a distributed setting, the
301 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/191F-jfUVX-l8ssxuRl1sixw2vctmofA/view?usp=sharing>

302 **4.1 Implementation and Experimental Setup**

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

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

322 **4.2 Analysis of Results – Synthetic Experiments**

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

331 **Impact of assuming partial synchrony (single core)**

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

338 **Impact of predicate structure (single core)**

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

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



© R. Ganguly and A. Momtaz and B. Bonakdarpour;

licensed under Creative Commons License CC-BY

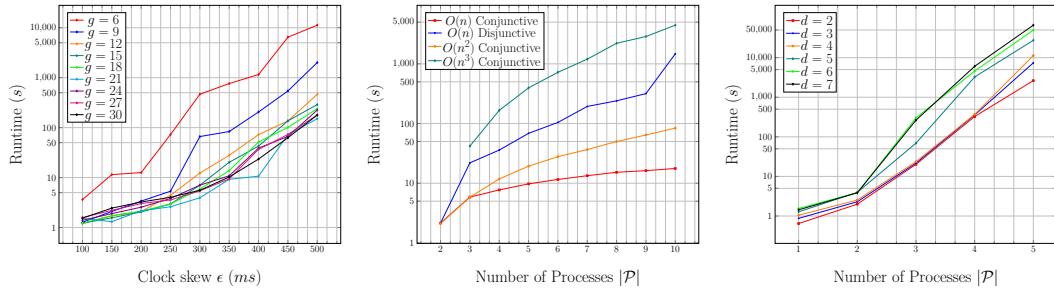
24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:10–20:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) Different values ϵ , LTL formula $\square p$ ($d = 2$), 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 $m = 1/s$.

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

348 **$O(n)$ Disjunctive:** Similar to $O(n)$ conjunctive predicates, here, we have a disjunction of n
 349 atomic propositions. Compared to its conjunctive counterpart, disjunction of propositions
 350 requires more time to verify. This follows the theoretical result that monitoring linear
 351 predicates is more complex than monitoring regular predicates [9].

352 **$O(n^2)$ Conjunctive:** Here, φ is a conjunction of atomic propositions, where each pro-
 353 position depends on the state of 2 processes, thereby having a total of $\binom{n}{2}$ predicates.
 354 Monitoring such predicates clearly require more time than $O(n)$ conjunctive predicates,
 355 but surprisingly less than $O(n)$ disjunctive predicates.

356 **$O(n^3)$ Conjunctive:** Here, we consider a conjunction of $\binom{n}{3}$ predicates chosen symbolizing
 357 a situation where each predicate is dependent on the state of 3 processes. This case is
 358 the most time-consuming structure to monitor.

359 Impact of LTL formula (single core)

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

$$\begin{array}{ll}
 \varphi_1 = \square(\neg p) & d = 2 \\
 \varphi_2 = \diamond r \rightarrow (\neg p \cup r) & d = 3 \\
 \varphi_3 = \square((q \wedge \neg r \wedge \diamond r) \rightarrow (\neg p \cup r)) & d = 4 \\
 \varphi_4 = \square((q \wedge \diamond r) \rightarrow (\neg p \cup (r \vee (s \wedge \neg p \wedge \square(\neg p \cup t))))) & d = 5 \\
 \varphi_5 = \diamond r \rightarrow (s \wedge \square(\neg r \cup t) \rightarrow \square(\neg r \cup (t \wedge \diamond p))) \cup r & d = 6 \\
 \varphi_6 = \square((q \wedge \diamond r) \rightarrow (p \rightarrow (\neg r \cup (s \wedge \neg r \wedge \square(\neg r \cup t)))) \cup r) & d = 7
 \end{array}$$

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



© R. Ganguly and A. Momtaz and B. Bonakdarpour;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:11–20:18



LIPICS Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

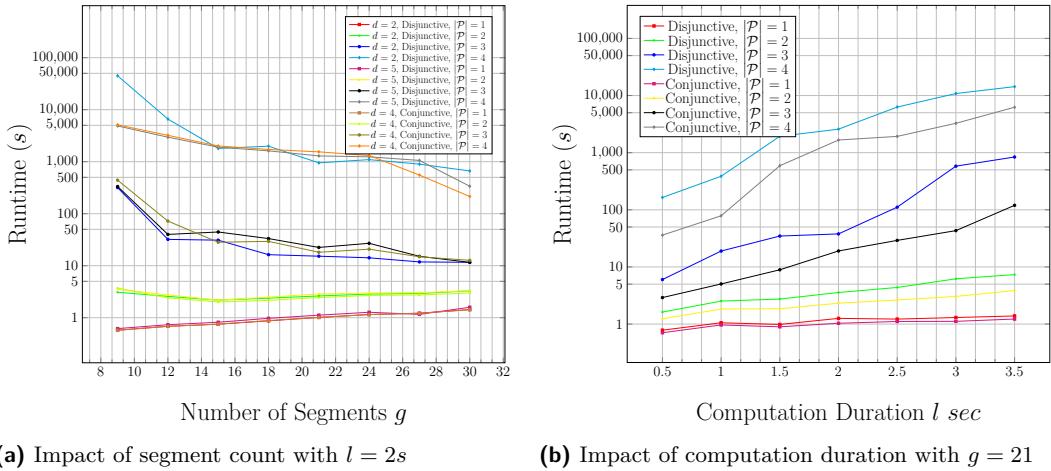


Figure 8 Impact of segment count and computation duration with $\epsilon = 250ms$, $O(1)$ predicates, $r = 10$, $m = 1$, and formula $\square p$.

369 Impact of segment count (single core)

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

381 Impact of computation duration (single core)

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

387 Impact of the event rate (single core)

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



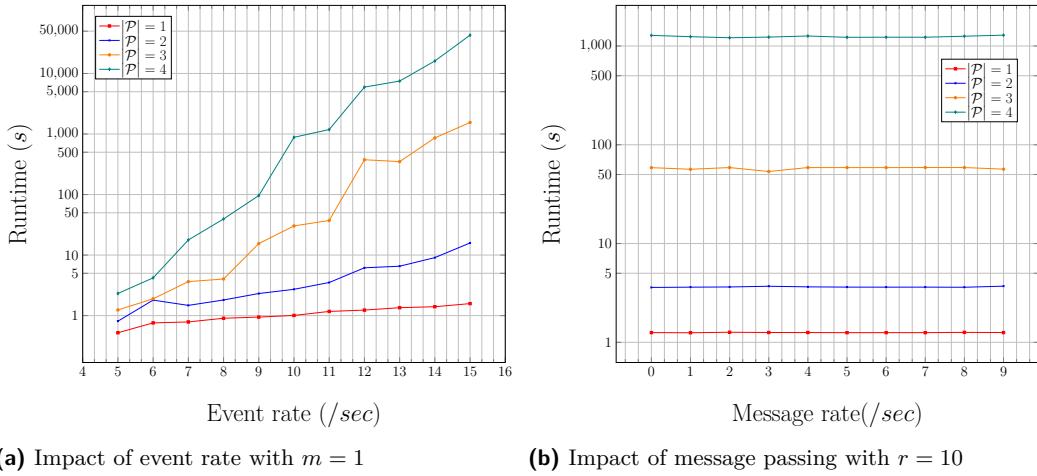
© R. Ganguly and A. Momtaz and B. Bonakdarpour;
 licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:12–20:18



Leibniz International Proceedings in Informatics
 Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(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 $\square p$ with $O(1)$ predicate structure.

394 Impact of the message rate (single core)

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

404 Impact of parallelization

405 To demonstrate the drastic increase in performance due to parallelization, we evaluate formula
 406 $\square p$ on a distributed computation with $l = 20s$, $r = 10$, $g = 20$, and $\epsilon = 150ms$, while varying
 407 the number of cores from 1 to 100, as shown in Fig. 10a. Observe that beyond 40 cores, there
 408 is no significant relative change in runtime regardless of the number of processes, as the time
 409 required to build the SMT formulation starts dominating the total run time. This graph
 410 also shows that parallelization can result in orders of magnitude speedup.

411

412

413 4.3 Case Study: Cassandra

414 Facebook developed Cassandra [15] as an
 415 open-source, distributed, No-SQL database
 416 management system. It is capable of han-
 417 dling large amounts of data across many ser-
 418 vers (nodes), spanning over multiple data-
 419 centers, providing high availability with no
 420 single point of failure, and asynchronous mas-

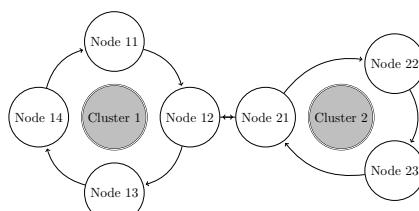


Figure 11 A network with two Cassandra clusters, Node-12 and Node-21 are the seed nodes of the respective clusters



© R. Ganguly and A. Momtaz and B. Bonakdarpour;

licensed under Creative Commons License CC-BY

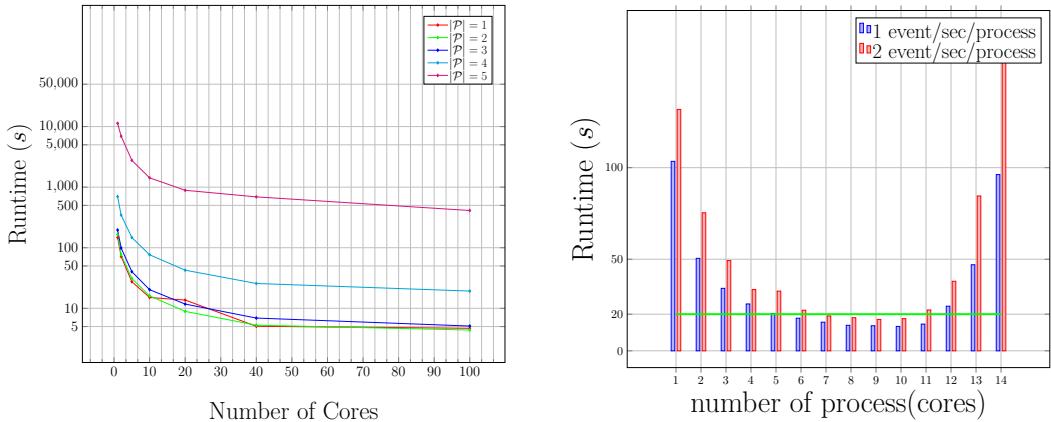
24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:13–20:18

LIPICS

Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



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

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

Figure 10 Impact of parallelization.

421 terless replication of data. Cassandra is designed such that it has no master or slave nodes.
 422 All the nodes are part of a ring-type architecture, distributing data equally over all the nodes
 423 in the cluster. We simulate a system of multiple processes. Each process is responsible for
 424 inter-process communication apart from basic database operations (read, write and update).
 425 We deployed a system with two data centers (see Fig. 11), where Cluster 1, contains 4 nodes
 426 (Nodes 11 – 14) and Cluster 2 contains 3 nodes (Node 21 – 23). Node 12 and Node 21 are the
 427 seed nodes of the respective clusters. Data is replicated in all the nodes in both the clusters.
 428 Each of the nodes is part of the *Red Hat OpenStack Platform* with the following configuration:
 429 4 VCPUs, 4GB RAM, Ubuntu 18.04, Cassandra 3.11.6, Java 1.8.0_252, and Python 3.6.9.

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

436 Processes are capable of reading, writing, and updating all entries of the database. The
 437 exact type of the event is selected by a uniform distribution (0, 2). Each process selects the
 438 available node at run time. In order to prevent deadlocks, no two processes are allowed to
 439 connect to the same node at the same time. If there exists no free node at any point of
 440 time, it waits for a node to be released and then it continues with the task. Once there is a
 441 write or update, the process responsible for the change sends a message to each of the other
 442 processes notifying about the change. We assume that a message is read by the receiving
 443 process immediately upon receiving. All database operations (i.e. send and receive events)
 444 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



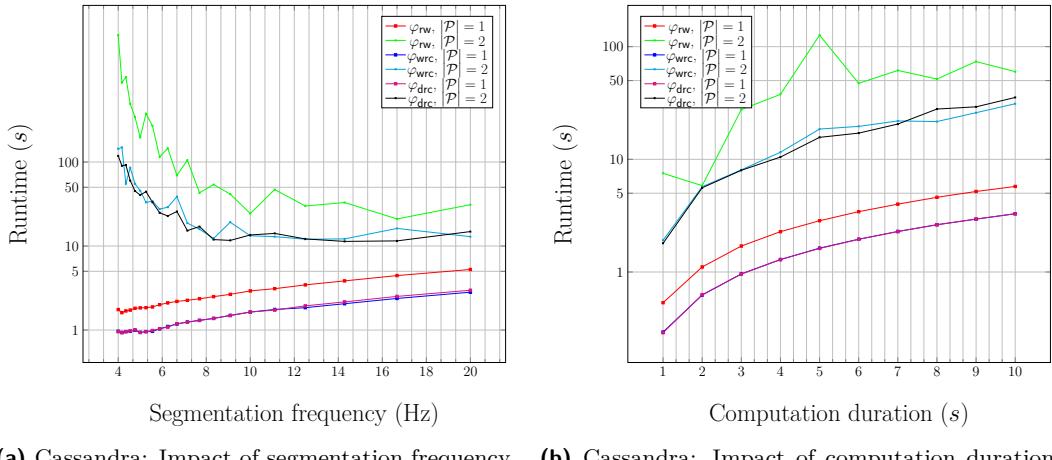
© R. Ganguly and A. Momtaz and B. Bonakdarpour;
 licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:14–20:18



Leibniz International Proceedings in Informatics
 Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(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:

$$\varphi_{rw} = \bigwedge_{i=0}^n \left(\text{write}(i) \rightarrow \Diamond \text{read}(i) \right)$$

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:

$$\text{Student}(id, name) \quad \text{Enrollment}(id, course).$$

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:

$$\begin{aligned} \varphi_{wrc} &= \neg \left(\neg \text{write}(\text{Student}.id) \cup \text{write}(\text{Enrollment}.id) \right) \\ \varphi_{drc} &= \neg \left(\neg \text{delete}(\text{Enrollment}.id) \cup \text{delete}(\text{Student}.id) \right) \end{aligned}$$

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.



© R. Ganguly and A. Momtaz and B. Bonakdarpour;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:15–20:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

463 **Moderate load scenario**

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

474 **4.4 Discussion**

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

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

489 **5 Related Work**

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

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



© R. Ganguly and A. Momtaz and B. Bonakdarpour;
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:16–20:18



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

512 **6 Conclusion and Future Work**

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

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

528 **References**

- 529 1 A. Bauer and Y. Falcone. Decentralised LTL monitoring. *Formal Methods in System Design*,
530 48(1-2):46–93, 2016.
- 531 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.
- 532 3 B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, D. A. Rosenblueth, and C. Travers. Decentralized
533 asynchronous crash-resilient runtime verification. In *Proceedings of the 27th International
Conference on Concurrency Theory (CONCUR)*, pages 16:1–16:15, 2016.
- 534 4 H. Chauhan, V. K. Garg, A. Natarajan, and N. Mittal. A distributed abstraction algorithm for
535 online predicate detection. In *Proceedings of the 32nd IEEE Symposium on Reliable Distributed
Systems (SRDS)*, pages 101–110, 2013.
- 536 5 C. Colombo and Y. Falcone. Organising LTL monitors over distributed systems with a global
537 clock. *Formal Methods in System Design*, 49(1-2):109–158, 2016.
- 538 6 L. M. Danielsson and C. Sánchez. Decentralized stream runtime verification. In *Proceedings
539 of the 19th International Conference on Runtime Verification (RV)*, pages 185–201, 2019.
- 540 7 L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the
541 Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.
- 542 8 A. El-Hokayem and Y. Falcone. On the monitoring of decentralized specifications: Semantics,
543 properties, analysis, and simulation. *ACM Transactions on Software Engineering Methodologies*,
544 29(1):1:1–1:57, 2020.
- 545 9 V. K. Garg. *Elements of distributed computing*. Wiley, 2002.
- 546 10 V. K. Garg. Predicate detection to solve combinatorial optimization problems. In *Proceedings
547 of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages
548 235–245. ACM, 2020.
- 549 11 V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates.
550 *International Conference on Distributed Computing Systems*, pages 423–430, June 1995.



551 © R. Ganguly and A. Momtaz and B. Bonakdarpour;

552 licensed under Creative Commons License CC-BY

553 24th International Conference on Principles of Distributed Systems (OPODIS 2020).

554 Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:17–20:18



555 Leibniz International Proceedings in Informatics

556 Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

554 12 Google. Usage limits, sheets api, google developer. <https://developers.google.com/sheets/api/limits>. Accessed: 2020-09-09.

556 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.

559 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.

562 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](https://doi.org/10.1145/1773912.1773922).

564 16 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

566 17 D. Mills. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, RFC Editor, June 2010.

568 18 N. Mittal and V. K. Garg. Techniques and applications of computation slicing. *Distributed Computing*, 17(3):251–277, 2005.

570 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.

573 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.

576 21 A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

578 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.

581 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.

584 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.



© R. Ganguly and A. Momtaz and B. Bonakdarpour;
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:18–20:18



LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany