Monitoring the Internet Computer

David Basin[1], Daniel Stefan Dietiker[2], Srđan Krstić[1(✉)],
Yvonne-Anne Pignolet[2], Martin Raszyk[2], Joshua Schneider[1(✉)],
and Arshavir Ter-Gabrielyan[2]

[1] Department of Computer Science, ETH Zürich, Zurich, Switzerland
{basin,srdan.krstic,joshua.schneider}@inf.ethz.ch
[2] DFINITY, Zurich, Switzerland
{danielstefan.dietiker,yvonneanne,martin.raszyk,
arshavir.ter.gabrielyan}@dfinity.org

Abstract. The Internet Computer (IC) is a distributed platform for Web3 applications, spanning over 1,200 nodes worldwide. We present results on applying runtime monitoring to the IC. We use the MonPoly monitor and its expressive policy language with quantifiers over infinite domains, aggregations, and past and future operators. We formalize complex policies that cover common kinds of production incidents and IC-specific protocol properties, including malicious behaviors and infrastructure outages. Using these policies, we evaluate MonPoly's performance in a large-scale case study that includes logs from both production and testing environments. We find, for example, that MonPoly performs well on testing logs, and that half of our policies applicable to production logs can be monitored in an online setting. Overall, our policies and IC traces constitute a new benchmark for first-order temporal logic monitors.

Keywords: Runtime monitoring · Temporal logic · Internet Computer

1 Introduction

In runtime monitoring, a monitor observes a system's execution, typically encoded as a sequence of *events*, checks whether the execution complies with a policy formalizing the system's correct behavior, and outputs detected violations. Online monitors incrementally process an *unbounded stream* of events produced by a running system, whereas offline monitors process a *finite log*. Good online monitors output timely violations, while good offline monitors process the log quickly, i.e., the former have low latency, whereas the latter have high throughput.

A real-world system's execution contains complex events, which include arbitrary data values. Such systems also require complex checks, for example based on aggregated values, dependencies between values, and possibly values coming from events spread over time. It is therefore important that monitors support

expressive policy languages and complex events. Furthermore, *distributed* systems pose additional monitoring challenges as policies may refer to (only partially ordered) events coming from different distributed components.

While many monitors support expressive policy languages [12,29,30] and there exist approaches for monitoring distributed systems [9,13,36,39] (Sect. 6), there is a substantial gap to bridge when applying them in the real world. With a notable exception [18], current literature has no answers to questions concerning policy engineering, measuring effectiveness, maintainability, as well as process organization, roles, and responsibilities in the context of runtime monitoring.

In this paper, we report on our experience in monitoring the *Internet Computer* (Sect. 2), a complex distributed system that facilitates the governance and execution of *Web3 applications*, i.e., applications processing data and financial assets with decentralized ownership and control of the applications' data, assets, and code. The Internet Computer is itself governed by a Web3 application, for example letting stakeholders vote on the Internet Computer's configuration and the addition and replacement of the machines that provide computing power to the system. The Internet Computer also possesses numerous other features that are challenging to monitor, both individually and when combined. These features include a long-lived execution with high event rates, a software architecture with multiple layers, dynamic configuration, and continuous evolution. Our case study is the outcome of a collaboration between Internet Computer developers at DFINITY and researchers in monitoring at ETH Zürich.

Assurance of the Internet Computer's correct behavior is critical for its stakeholders as it is a complex system managing financial assets. We show how runtime monitoring complements system testing and metric-based observability, two existing assurance techniques. In particular, our case study shows that *MonPoly* [11,12], a state-of-the-art monitor supporting an expressive policy language, is well-suited for monitoring logs obtained from system tests. Moreover, MonPoly can process the event stream from the production system in real time for some policies, but for other, more complex policies, it incurs a monitoring backlog. We identify several opportunities for future optimizations and report on lessons learned.

Overall, we make the following contributions: (1) We formalize a set of policies that express common symptoms of production incidents in the Internet Computer as well as domain-specific properties of its protocol, including malicious behaviors and infrastructure outages that the protocol must tolerate (Sect. 3). (2) We use these policies for a *quantitative* evaluation of MonPoly's performance (Sect. 4) and its applicability in both testing and production scenarios. (3) We obtain *qualitative* insights about the integration of runtime monitoring into a complex production system. In particular, we report on insights on policy engineering and monitoring maintainability (Sect. 5). (4) We publish the artifact [7] containing the logs, policies, and code used in this case study. It can be used to benchmark monitors for policy languages that support first-order temporal logic with aggregations.

We believe that our results are valuable to others applying runtime monitoring in practice (Sect. 7). Our policies formalizing infrastructure outages, although

specific to Internet Computer in their current form, generalize well to other systems. Moreover, our policies that formalize properties of the Internet Computer's protocol may be adapted to other distributed systems with replicated execution proceeding in rounds.

2   Background

Runtime Monitoring. A runtime monitor [4,25] verifies whether a running system satisfies a *policy* by observing the system's execution. We now briefly describe the MonPoly monitor [12], its policy language called metric first-order temporal logic (MFOTL) [10], and data-parallel monitoring [38].

We fix a set of event names $\mathbb{E}$, an infinite domain $\mathbb{D}$ of values, and an infinite set $\mathbb{V}$ of variables such that $\mathbb{E}$, $\mathbb{D}$, and $\mathbb{V}$ are pairwise disjoint. Let $\mathbb{T}$ be a set of terms over variables in $\mathbb{V}$. In the case of MonPoly, the domain $\mathbb{D}$ contains integers, floats, and strings, and the constant and function symbols available in terms provide basic arithmetic operations over integers and floats. For example, $x + 4$ is a well-formed term. Let Ω be a set of aggregation functions that map multisets over $\mathbb{D}$ to $\mathbb{D} \cup \{\bot\}$. For example, SUM $\in \Omega$ computes SUM($\{\!|1, 1, 3, 4, 4, 5|\!\}$) = 18, but SUM($\mathbb{N}$) = $\bot$ as the result is infinite. Each name $r \in \mathbb{E}$ has an arity $\iota(r) \in \mathbb{N}$. An *event* $r(d_1, \ldots, d_{\iota(r)})$ is an element of $\mathbb{E} \times \mathbb{D}^*$ and $d_i \in \mathbb{D}$ are its *parameters*. Let $\mathbb{I}$ be the set of nonempty intervals $[a, b) := \{x \in \mathbb{N} \mid a \le x < b\}$, where $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$. MFOTL formulas φ are defined inductively, where r, x, $\bar{x}$, t, $\bar{t}$, ω, and I range over $\mathbb{E}$, $\mathbb{V}$, $\mathbb{V}^*$, $\mathbb{T}$, $\mathbb{T}^*$, Ω, and $\mathbb{I}$, respectively:

$$\varphi ::= r(\bar{t}) \mid t = t \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists \bar{x}.\, \varphi \mid \bullet_I \varphi \mid \bigcirc_I \varphi \mid \varphi\, \mathsf{S}_I\, \varphi \mid \varphi\, \mathsf{U}_I\, \varphi$$
$$\mid x \leftarrow \omega\, t; \bar{x}\, \varphi \mid \mathbf{let}\ r(\bar{x}) := \varphi\ \mathbf{in}\ \varphi$$

The set $\mathsf{fv}(\varphi)$ contains φ's free variables. Formulas of the form $r(\bar{t})$ are called *predicates* and require $|\bar{t}| = \iota(r)$. The temporal operators $\bullet_I$ (previous), $\bigcirc_I$ (next), S_I (since), and U_I (until) may be nested arbitrarily. The aggregation operator $r \leftarrow \omega\, t; \bar{g}\, \varphi$ requires $\bar{g} \cup \mathsf{fv}(t) \subseteq \mathsf{fv}(\varphi)$ and $r \notin \mathsf{fv}(\varphi)$. The let operator $\mathbf{let}\ r(\bar{x}) := \varphi\ \mathbf{in}\ \psi$ requires $\bar{x} = \mathsf{fv}(\varphi)$ and it (re)defines $\iota(r) = |\bar{x}|$ in ψ. We distinguish the let predicates (defined by a let operator) from the input predicates. We derive other operators: truth $\top := \exists x.\, x = x$, inequality $t_1 \neq t_2 := \neg(t_1 = t_2)$, conjunction $\varphi \wedge \psi := \neg(\neg\varphi \vee \neg\psi)$, and once $\blacklozenge_I \varphi := \top\, \mathsf{S}_I\, \varphi$.

A valuation v is a mapping $\mathbb{V} \to \mathbb{D}$, assigning domain elements to variables. We write $v[\bar{x} \mapsto \bar{d}]$ for the function equal to v, except that the variables $\bar{x}$ are mapped to values $\bar{d}$, where $|\bar{x}| = |\bar{d}|$. Overloading notation, v is extended to the domain $\mathbb{T}$, evaluating the term t based on the valuations of $\mathsf{fv}(t)$. A *trace* is an infinite sequence $(\tau_i, D_i)_{i \in \mathbb{N}}$ of *timestamp* ($\tau_i \in \mathbb{N}$), *database* ($D_i \in 2^{\mathbb{E} \times \mathbb{D}^*}$) pairs. Timestamps in a trace are *monotone* ($\forall i.\ \tau_i \le \tau_{i+1}$) and *progressing* ($\forall \tau.\ \exists i.\ \tau < \tau_i$). Databases are finite. Given a trace $\rho = (\tau_i, D_i)_{i \in \mathbb{N}}$, we write $\rho[r(\bar{x}) \mapsto R]$ for the trace $\rho' = (\tau_i', D_i')_{i \in \mathbb{N}}$ with $\tau_i' = \tau_i$ and $D_i' = D_i - \{r(\bar{d}) \mid \bar{d} \in \mathbb{D}^{\iota(r)}\} \cup \{r(\mathsf{map}(v, \bar{x})) \mid v \in R(i)\}$ for all $i \in \mathbb{N}$, where R is a function from natural numbers to sets of valuations. The function $\mathsf{map}(f, [d_1, \ldots, d_n])$ returns

$$
\begin{array}{ll}
v, i \models_\rho r(\bar{t}) & \text{if } r(\mathsf{map}(v, \bar{t})) \in D_i \\
v, i \models_\rho \neg\varphi & \text{if } v, i \not\models_\rho \varphi \\
v, i \models_\rho \exists \bar{x}.\, \varphi & \text{if } v[\bar{x} \mapsto \bar{d}], i \models_\rho \varphi \text{ for some } \bar{d} \in \mathbb{D}^{|\bar{x}|} \\
v, i \models_\rho \bullet_I \varphi & \text{if } i > 0,\, \tau_i - \tau_{i-1} \in I, \text{ and } v, i-1 \models_\rho \varphi \\
v, i \models_\rho \bigcirc_I \varphi & \text{if } \tau_{i+1} - \tau_i \in I \text{ and } v, i+1 \models_\rho \varphi \\
v, i \models_\rho \varphi \, \mathsf{S}_I \, \psi & \text{if } v, j \models_\rho \psi \text{ for some } j \leq i,\, \tau_i - \tau_j \in I,\, v, k \models_\rho \varphi \text{ for all } k,\, j < k \leq i \\
v, i \models_\rho \varphi \, \mathsf{U}_I \, \psi & \text{if } v, j \models_\rho \psi \text{ for some } j \geq i,\, \tau_j - \tau_i \in I,\, v, k \models_\rho \varphi \text{ for all } k,\, i \leq k < j \\
v, i \models_\rho r \leftarrow \omega\, t; \bar{g}\, \varphi & \text{if } v(r) = \omega(M) \text{ and if } M = \emptyset \text{ then } \bar{g} = \emptyset,
\end{array}
$$

$$
\begin{array}{ll}
v, i \models_\rho t_1 = t_2 & \text{if } v(t_1) = v(t_2) \\
v, i \models_\rho \varphi \vee \psi & \text{if } v, i \models_\rho \varphi \text{ or } v, i \models_\rho \psi
\end{array}
$$

$$
\text{where } \bar{b} = \mathsf{fv}(\varphi) - \bar{g} \text{ and } M = \{ v[\bar{b} \mapsto \bar{d}](t) \mid v[\bar{b} \mapsto \bar{d}], i \models_\rho \varphi \text{ for some } \bar{d} \in \mathbb{D}^{|\bar{b}|} \}
$$

$$
v, i \models_\rho \mathbf{let}\ r(\bar{x}) := \varphi\ \mathbf{in}\ \psi \quad \text{if } v, i \models_{\rho[r(\bar{x}) \mapsto \lambda j.\{u \mid u, j \models_\rho \varphi\}]} \psi
$$

Fig. 1. Semantics of MFOTL

$[f(d_1), \ldots, f(d_n)]$. The relation $v, i \models_\rho \varphi$ (Fig. 1) defines the satisfaction of the formula φ for a valuation v at an index i with respect to the trace ρ.

A *runtime monitor* like MonPoly monitors an MFOTL *policy formula* φ by incrementally observing a finite prefix of some execution trace and computing a set of valuations and indices that satisfy φ given the observed prefix. The formula φ typically formalizes the negation of a *policy*, i.e., a desired system property, such that each valuation–index pair indicates a *violation* of the policy.

We distinguish between events and *log entries*, which are text strings reported by a running system. For monitoring, a log entry like "`[WARN] TLS handshake failed`" is mapped to zero or more events like $\mathsf{TLSError}()$ and $\mathsf{Log}(\ldots, \mathsf{WARN}, \ldots)$. A recent survey [25] overviews existing monitoring tools and their languages.

Target System. The Internet Computer (IC) [40] is a public, blockchain-based distributed platform for general-purpose Web3 applications (*apps*), also known as smart contracts. The IC's distributed nature and its replication are transparent to the app developers and users. Users submit their requests and the apps process them, possibly communicating with other apps, and reply back to the users.

The machines (*nodes*) running the IC's protocol are partitioned into *subnets* [40] (currently 13–40 nodes), each replicating and executing a set of apps. Thus, unlike most other blockchain-based platforms, the IC does not employ a global consensus protocol; instead, nodes participate in consensus only among their subnet peers. Each subnet maintains its own (small) blockchain instance, characterized by blocks each occurring at a *height* (the block's position in the chain). Besides the metadata (e.g., timestamps), blocks contain app requests from users and from apps on other subnets. Each subnet produces blocks at rates as high as ca. 0.5–1.0 blocks/s. To ensure that consensus is not just fast, but also trustworthy, each subnet's nodes are hosted on servers distributed among many stakeholders, e.g., data center providers from multiple countries and jurisdictions. A special app called *registry* maintains the IC configuration (e.g., active nodes and their assignment to subnets) and logs configuration changes.

The IC currently consists of more than 1,200 nodes, hosting ca. 150,000 apps [3]. The IC generates ca. 1,500 log entries per second, i.e., over 400 GB
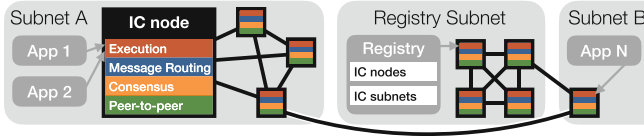
Fig. 2. Overview of the IC

of logs per day across all nodes. Each node has four layers (Fig. 2): (i) the *peer-to-peer* layer reliably disseminates information among nodes; (ii) the *consensus* layer validates and orders the requests to the apps; (iii) the *message routing* layer delivers those requests to the apps; and (iv) the *execution* layer runs the apps.

System Testing and Metrics. The development process of distributed systems such as the IC involves various kinds of testing. Here we focus on *system testing*, i.e., end-to-end testing of the complete system *in isolation* from the production environment. In system testing, a new software version, constituting the system under test (SUT), is deployed over a dedicated testing infrastructure. Requests are then sent to the SUT via its public interface. Optionally, the SUT is manipulated in a controlled way, modeling effects like network failures or configuration changes. Finally, the test checks if the SUT responded to all the requests correctly.

Unlike runtime monitoring, system tests do not check if the sequence of states that arise during the system execution is correct. Instead, they only check the system's final output. Moreover, scenarios covered by system testing are fixed *a priori*. These aspects limit the issues that can be potentially detected by system testing.

Even if the system is well-tested, detecting, e.g., unforeseen real-world attacks requires *observability*, i.e., the degree to which the internal state can be determined based on system's output [28]. Observability is crucial also for other requirements that are not covered by system testing: auditing, accounting, performance assessments, and design feedback [37]. For example, observability enables engineers to recognize failures and users to confirm whether the system does what is promised.

In practice, distributed systems typically output additional data, called *metrics*, into an external centralized metrics database [37]. IC's metrics enable humans to observe and visualize, e.g., the height of the blockchain or the number of requests submitted to a subnet. Programmatic rules running atop of the metrics database, called *alerts*, can send notifications, e.g., to the developers of the IC, whenever the block production rate drops below a threshold value. Metrics are a lossy representation of the system state as they are locally preprocessed before being sent. As they do not record the context that has lead to an alert, developers need other data sources, like logs, to find an alert's root cause. Furthermore, metrics are typically collected periodically (as defined by the metrics

database), which is not suitable for checking the precise temporal evolution of
the system's state.

3   Policies

In this section, we first describe *how* we devised new IC policies (Sect. 3.1) and
then present a selection of those policies that we formalized (Sect. 3.2).

3.1   Methodology

Operational concerns were the main driver for the policies we formulated. In
particular, we wanted to ensure that logs are produced consistently, abnormal
node behavior can be detected, and crucial properties of the IC protocol (like
agreement on requests, progress, and recovery from failure [40]) hold. We did
not aim to exhaustively cover all properties of the IC. We focused instead on
aspects that cannot be sufficiently covered by existing system tests and metric-
based alerts. For example, system tests cannot detect malicious behavior in the
production system, and metrics are ill-suited to observe a subnet's behavior
holistically.

   We started with high-level, natural-language specifications based on the exist-
ing logging instrumentation provided by the IC software engineers. In most cases,
however, the logged information was insufficient for monitoring. To bridge this
gap, we proceeded iteratively; each iteration started with a formalization attempt
for a high-level specification. Since this required precise knowledge about which
events are observable from which logs, we consulted with the engineers who
provided insights on the implementation of particular system components and
extended the log messages when necessary. In some cases, the developers con-
cluded that logging the requested events was infeasible, so the affected policies
had to be abandoned (see also Sect. 5).

   Next, we performed preliminary monitoring of the policies on sample logs and
analyzed the output. We then *triaged* each violation, classifying it as (1) a *true
bug* in the system, (2) an *imprecise policy* due to insufficient understanding
of the system, or (3) a *formalization error*, e.g., due to typos or an incorrect
understanding of MFOTL semantics. In some cases, we could not easily triage
the violation. We then contacted the IC software engineers who either provided
insights for improving the policy or, in case of true bugs, submitted bug reports
to IC's internal issue tracker. To date, more kinds of true bugs have been discov-
ered while *developing* the preliminary policies than while monitoring their final
version.

3.2   Policy Formulas

Our policies cover three broad categories, which differ in their scope and gen-
erality, and which demonstrate a variety of runtime monitoring use cases. We
present policy formulas for just a few selected policies. These policies showcase

Table 1. Summary of MFOTL-based IC policies

Policy	Past	Fut	Agg	Loc	Reg	Test	Prod	Ops1	Ops2
`clean-logs`	✓	–	–	✓	✓	✓	✓	13	11
`logging-behavior`	✓	✓	✓	–	✓	✓	✓	54	1,098
`finalized-height`	✓	–	–	–	✓	✓	–	56	89
`finalization-consistency`	✓	–	–	–	✓	✓	–	16	22
`replica-divergence`	✓	–	–	✓	✓	✓	–	16	13
`block-validation-latency`	✓	✓	✓	–	✓	✓	–	50	229
`unauthorized-connections`	✓	–	–	✓	✓	✓	✓	22	39
`reboot-count`	✓	–	✓	–	✓	✓	✓	25	21

the most challenging aspects of formalizing distributed system properties and justify the required features of MFOTL. The accompanying artifact [7] provides all formulas.

Table 1 summarizes the IC policies and the characteristics of the MFOTL policy formulas that formalize them. All formulas contain at least one past-temporal operator (column Past). There are two formulas with a future operator (Fut), and three formulas that use aggregations (Agg). Three policies can be monitored locally (Loc) on each node using only the node's log entries. All policies depend on the initial IC configuration obtained using the IC registry app (Reg) and they can be checked against the testing logs (Test). Finally, four of the policies can also be checked against the IC's production log (Prod), whereas the other policies require debug-level log entries, which are not available in production in order to decrease the load on the logging infrastructure. We estimated the complexity of the formulas by counting the numbers of their unary and binary operators before unfolding the **let** definitions (Ops1) and after (Ops2).

Common Fragments. Some aspects are shared by all policies, e.g., the policies restrict the behavior of *active* nodes only. A subset of policies additionally requires knowledge about which node belongs to which subnet at any point in time. As explained earlier, the IC's configuration can be changed by a voting-driven governance mechanism and hence we must observe configuration changes to correctly monitor these policies. We devised the following pattern to express both the set of currently active nodes n (predicate $\mathsf{InIC}(n)$) and the property that a node n belongs to a subnet s (predicate $\mathsf{InSubnet}(n, s)$):

$$\mathsf{In\underline{X}}(\bar{p}) := \big((\blacklozenge\, \mathsf{In\underline{X}_0}(\bar{p})) \wedge \neg\, \blacklozenge\, \mathsf{RegistryRemove\underline{X}}(\bar{p})\big) \vee$$
$$\big(\neg \mathsf{RegistryRemove\underline{X}}(\bar{p}) \math:S\: \mathsf{RegistryAdd\underline{X}}(\bar{p})\big)$$

With $\underline{\mathsf{X}} = \mathsf{IC}$ and $\bar{p} = [n]$, we define the predicate $\mathsf{InIC}(n)$ and, with $\underline{\mathsf{X}} = \mathsf{Subnet}$ and $\bar{p} = [n, s]$, we define the predicate $\mathsf{InSubnet}(n, s)$. The $\mathsf{InIC}(n)$ and $\mathsf{InIC}_0(n)$ predicates determine whether the node n belongs to the IC at the

```
clean-logs:
```
  let $\mathsf{InIC}(n) \coloneqq \cdots$ **in**
  let $\mathsf{ErrorLevel}(l) \coloneqq (l = \texttt{"CRITICAL"}) \vee (l = \texttt{"ERROR"})$ **in**
  $\mathsf{InIC}(n) \wedge \mathsf{Log}(h, n, s, c, l, m) \wedge \mathsf{ErrorLevel}(l)$

```
finalized-height:
```
  let $\mathsf{InSubnet}(n, s) \coloneqq \cdots$ **in**
  let $\mathsf{Growing}(s) \coloneqq \exists n_1, n_2.\, \mathsf{InSubnet}(n_1, s) \wedge \mathsf{InSubnet}(n_2, s) \wedge n_1 \neq n_2 \wedge$
                                $\neg(\neg \mathsf{p2pRemoveNode}(n_1, s, n_2)\ \mathsf{S}\ \mathsf{p2pAddNode}(n_1, s, n_2))$ **in**
  let $\mathsf{Shrinking}(s) \coloneqq \exists n_1, n_2.\, \mathsf{InSubnet}(n_1, s) \wedge$
        $\big((\neg \mathsf{p2pRemoveNode}(n_1, s, n_2)\ \mathsf{S}\ \mathsf{p2pAddNode}(n_1, s, n_2)) \vee$
          $\mathsf{InSubnet}(n_1, s) \wedge (\blacklozenge\, \mathsf{InSubnet}_0(n_2, s)) \wedge$
          $\neg(\blacklozenge\, \mathsf{p2pRemoveNode}(n_1, s, n_2)) \wedge \neg(\blacklozenge\, \exists s'.\, \mathsf{p2pAddNode}(n_1, s', n_2))\big) \wedge$
        $\neg \mathsf{InSubnet}(n_2, s)$ **in**
  let $\mathsf{Changing}(s) \coloneqq \mathsf{Growing}(s) \vee \mathsf{Shrinking}(s)$ **in**
  let $\mathsf{First}(n, s, h, b, v) \coloneqq \mathsf{Finalized}(n, s, h, b, v) \wedge \neg \bullet \blacklozenge \exists n'.\mathsf{Finalized}(n', s, h, b, v)$ **in**
  $(\neg \mathsf{Changing}(s)\ \mathsf{S}_{(80s, \infty)}\ \mathsf{First}(n_1, s, h_1, b_1, v)) \wedge \mathsf{First}(n_2, s, h_2, b_2, v) \wedge h_2 = h_1 + 1$

Fig. 3. Examples of policy formulas

current moment and when monitoring originally started, respectively. The predicates $\mathsf{InSubnet}(n, s)$ and $\mathsf{InSubnet}_0(n, s)$ are analogous. The input predicates prefixed with $\mathsf{Registry}$ directly correspond to log entries from the IC registry app; these events indicate the removal and addition of IC nodes (to a subnet or the IC). To maintain the predicates, we rely on the IC registry as opposed to relying on (potentially incorrect) node-local information. For each node n, the $\mathsf{InIC}_0(n)$ and $\mathsf{InSubnet}_0(n, s)$ events are prepended to the log by querying the registry before monitoring starts.

    We use MFOTL's **let** to define the InIC and $\mathsf{InSubnet}$ predicates. As their definitions are syntactically encapsulated, it is easy to keep them in sync across all policies in case input predicates change.

Generic Policies. Our goal here is to detect general signs of system malfunction.

clean-logs. The log entries produced by IC nodes have different priority levels. Our `clean-logs` policy asserts that only *warning-* and *info*-level log entries are allowed, whereas *critical-* or *error*-level entries are not. In the IC, these levels indicate logical errors, violation of assumptions, or similarly severe problems. The corresponding formula (Fig. 3, top) uses the $\mathsf{Log}(h, n, s, c, l, m)$ predicate, which is satisfied by every log message m emitted by component c running on node n in subnet s with host name h, where l is the log level. As previously noted, we ignore decommissioned nodes. We also formulate all policy formulas to be satisfied whenever the corresponding policy is violated.

logging-behavior. Although `clean-logs` can detect many problems, it only produces violations once a fault has already become a failure. In contrast, the

`logging-behavior` policy aims to detect faults before the failure occurs. We use the fact that operations are replicated on multiple nodes of a subnet: If the *frequency* of the log entries matching the replicated operations deviates on a relatively small group of nodes within a subnet, this indicates that the nodes are in an abnormal state that may lead to failure. For each subnet, the policy compares its nodes' logging frequencies computed over a *sliding window* [2] against the *median* logging frequency over all nodes in the subnet.

This policy formula uses multiple aggregations (count, sum, median, minimum, and maximum) and both past and future temporal operators. We also use *regular expression matching*, a recent addition to MonPoly, to select log entries that belong to a replicated operation. As the typical behavior may change over time depending on the workload, we incorporate smoothing to avoid false positives. Specifically, we estimate the typical behavior from multiple overlapping time intervals. Since log frequencies vary significantly between IC node layers (Sect. 2), we monitor this policy separately for each layer.

IC Protocol Policies. We summarize some properties of the IC consensus protocol [14] used in this group of policies. Given a subnet of n nodes, among which f are faulty (i.e., behaving in a *Byzantine way* [34]) and the remaining $n-f$ nodes adhere to the protocol, the condition $n \geq 3f+1$ must hold (otherwise, consensus is not possible [26]). Intuitively, this means that to achieve consensus, more than ⅔ of the subnet nodes *must not* be faulty, where the lowest tolerated number of non-faulty nodes is $2f+1$. The IC consensus protocol uses the concept of *rounds*; out of all the *block proposals* created by the nodes for round r, exactly one block is *finalized*, i.e., irreversibly added to the blockchain at *height r*.

Violations of the following IC protocol policies indicate software bugs or the presence of more than f faulty nodes in a subnet.

`finalized-height`. To ensure that a subnet's consensus makes progress, this policy checks that the block at height $h+1$ in a subnet is finalized by some node no later than 80 s after the *earliest* finalization of the block at height h. The time between finalizations depends on node failures and network conditions. In practice, the mean time elapsed between two finalized blocks is around 1 s. 80 s is thus a rather conservative upper bound that allows us to turn a probabilistic property into a safety property that we can monitor automatically.

The nodes changing their subnet membership require care, as the upper bound on the time between finalizations may be exceeded, specifically, when a new node is *catching up*, e.g., due to a temporary network outage. We therefore ignore violations that occur during subnet membership changes. To detect changing subnets, we over-approximate by comparing the registry's view of the subnet membership to the nodes' own view (as captured by the p2pAddNode and p2pRemoveNode events from the peer-to-peer communication layer).

The formula illustrates how **let** operators reduce formula duplication and improve its structure (Fig. 3, bottom). Specifically, we define the InSubnet predicate as explained above. The predicate Growing on subnets is satisfied if a node in

the subnet is not yet aware of another node in the same subnet, while the predicate Shrinking detects when a node still considers another node as part of the same subnet whereas the registry does not. In both cases, we over-approximate because the nodes' local view is not known before one of the two p2p events has been observed. A subnet is considered to be Changing if it is Growing or Shrinking.

The condition on the time between finalizations is expressed using a metric temporal operator in the policy's formula (Fig. 3, bottom), where the let predicate First(n, s, h, b, v) represents the first finalization (event Finalized) of block b at height h by some node (specifically node n) in subnet s, running IC software version v. The S operator asserts that there is such a finalization by node n_1 more than 80 s ago (the interval $(80\,\text{s}, \infty)$ is open), and its subnet must not have been changing in the meantime. To detect a violation, the policy must additionally observe a finalization *at the next height* by node n_2.

finalization-consistency. This policy represents the core correctness property of the IC consensus protocol: when a node finalizes a block at a given height, no other node in the same subnet finalizes a different block at the same height.

replica-divergence. This policy expresses a *liveness* property. Whenever the replicated state maintained by the nodes is not the same on all nodes in a subnet, the nodes must eventually detect and overcome this *divergence*. State divergence might occur even in absence of malicious behavior, e.g., due to software bugs or hardware problems. A subnet can overcome a divergence when at least $2f + 1$ of its nodes have the same replicated state. The protocol achieves this as nodes periodically emit *shares* based on their local replicated state; $2f + 1$ such shares are needed for *catch-up packages*—messages enabling the nodes to restore the correct state and contribute to the consensus protocol again. In particular, a catch-up package contains the hash of the correct replicated state, which allows nodes to detect that they have diverged and obtain the correct state. However, only shares from $2f + 1$ nodes with the same state can be used for a catch-up package. Hence, if a node's share contributes to a catch-up package after the node has diverged, this indicates that the node has since corrected its local state.

Note that system tests always produce finite logs; this enables us to phrase the policy as a *safety* property: End() $\wedge$ InSubnet$(a, \_, s)$ $\wedge$ ($\neg$CupShareProposed(a, s) S Diverged(a, s)). Here, CupShareProposed(a, s) holds when a catch-up package share is proposed by node a of subnet s. Diverged(a, s) indicates that a has reported a state divergence (recall that our formulas express the *negation* of the required properties). Lastly, End(), which is added by the preprocessor, is the final event in the stream. Intuitively, the nullary predicate End() binds the formula to the final time point of the test.

block-validation-latency. This policy formalizes network progress before finalization is reached. Recall that the IC consensus protocol proceeds in rounds. In each round, the nodes may create and propose new blocks to their peers via the P2P layer. When receiving these blocks, the peers declare them validated if a set of conditions is satisfied; these conditions concern the block's metadata
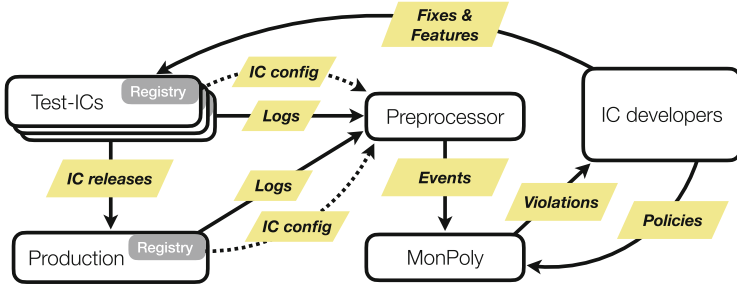
Fig. 4. Overview of IC's monitoring pipeline. Rounded boxes are parties involved in monitoring, arrows depict data flow, and dotted arrows show initial pipeline steps.

and the app requests, e.g., authentication. Upon validating the block, the node informs its peers. Progress to the next round is possible only if *more* than ⅔ of the nodes validate a block. This policy measures the time until a block proposal created at one node has been validated by more than ⅔ of the nodes in the same subnet; the policy then checks that this time does not exceed a threshold.

unauthorized-connections. IC nodes should receive peer-to-peer connections only from other nodes within the same subnet. As these connections are secured by TLS [41], any illicit connection attempt should cause a TLS handshake failure as the certificate is rejected. This policy states that such failures must not occur unless the illicitly connecting node and the receiver were members of the same subnet *in the recent past* (we set the threshold to 15 min), as the nodes may not have learned yet that they are no longer peers.

Infrastructure Outage. We also consider platform-level aspects of the IC.

reboot-count. Data center problems may be accompanied by frequent server reboots. This policy identifies problematic data centers, detecting when servers hosting IC nodes within a data center are rebooted too frequently. For each data center, the policy counts the number of unplanned (re-)boots within the past 30 min; for this purpose, we employ the count aggregation and the ◆ operator. A violation is emitted if the number of reboots exceeds two, i.e., up to two reboots are tolerated. Data centers are identified by prefixes of the node's IPv6 address, which are available as predicate arguments (Sect. 4).

4   Evaluation

In our evaluation of MonPoly's performance, we address the following questions: (Q1) How much time and memory does MonPoly require for monitoring complex policies *offline*? (Q2) Is MonPoly able to monitor the IC's production logs *online*? (Q3) What are the main performance and scalability factors?

Table 2. Evaluation results (median and maximum in parentheses) for offline monitoring

Measurement		$Test$ – 67 logs		$Prod$ – 1 log	
Raw log	entries	8,059 (860,164)		16,887,502	
	MiB	15.6 (3,250.3)		57,216.4	
Processed log	events	1,394 (634,790)		1,553,159	
	events/s	10.7 (168.1)		143.8	
	MiB	0.5 (207.2)		713.3	
Preprocessor time	ms/entry	0.12 (5.61)		0.07 (0.08)	
		ms/event	MiB	ms/event	MiB
`clean-logs`		3.20 (145.0)	10 (10)	3.93	11
`logging-behavior`		2.74 (144.4)	11 (1265)	TO	TO
`unauthorized-connections`[*]		3.09 (145.6)	10 (1109)	TO	TO
`reboot-count`		2.68 (151.1)	10 (11)	3.54	11
`finalized-height`[*]		3.93 (138.7)	10 (19)	–	–
`finalization-consistency`		2.57 (143.1)	10 (16)	–	–
`replica-divergence`		2.80 (145.2)	10 (10)	–	–
`block-validation-latency`[†]		5.04 (143.1)	13 (26)	–	–

[*] Timeout on 1 log each.   [†] Timeout on 3 logs.

Pipeline. We implemented a *monitoring pipeline* (Fig. 4) that downloads logs from the IC's log server (either from a *Test-IC* or from production), preprocesses them, and manages MonPoly's execution. The same pipeline was added to the IC's continuous development workflow, alerting IC software engineers of detected policy violations and providing them with the context required to reproduce and investigate the underlying problems. The pipeline's log *preprocessor* converts log entries into events encoded in MonPoly's input format. Most events require simple syntactic manipulations (e.g., extracting parameters with regular expressions), but some require information about the IC configuration, e.g., the mapping between node IDs and IP addresses. The preprocessor obtains this information, as well as the InIC_0 and $\mathsf{InSubnet}_0$ events (Sect. 3.2), from the registry.

The top half of Table 2 summarizes basic properties of logs used in our experiments, aggregating data across all logs and, where applicable, policies. The median is shown as well as the maximum in parentheses. We obtained logs from the IC's system tests (*Test*) as well as a three hour fragment of the production log (*Prod*). For repeatability, this step was performed separately from the experiments and the logs were stored as files. The *Test* logs were collected from 3 runs of every system test in the IC's hourly and nightly test suites, over a 3-day period. We only considered successful test runs, as a failed test already requires an engineer's attention and monitoring would not add much value. In both *Test*

and *Prod* logs, the pipeline's preprocessor discarded all log entries that cannot be assigned to an IC node, e.g., messages from *systemd.*

We approximated the time spent in preprocessing. Since the pipeline transforms log entries on the fly before sending the events to the monitor, we accumulated the time spent in the preprocessing step for each entry ("preprocessor time"). Due to the logs' diversity, we normalized this value by dividing it by the number of log entries; the result is the inverse of throughput.

We instrumented the pipeline to collect performance measurements for offline monitoring (Q1). Specifically, we obtained the wall-clock time for the combined execution of pipeline and monitor ("monitoring time") and the peak resident set size ("monitoring memory") of the MonPoly process. Monitoring time was normalized based on the number of *events* comprising the input to MonPoly. To address Q2, we simulated a real-time log stream based on the stored fragment of the production log, using a *replayer* [33] that writes the log entries at the appropriate time to MonPoly's input. We performed additional experiments to answer Q3.

We ran at most 13 experiments in parallel on a server with two 3 GHz 16-core AMD EPYC 7302 CPUs, 512 GiB RAM, and an SSD. We used Linux 5.4.0 as the operating system and the MonPoly Docker image 1.4.2 as the monitor. All the logs, policies, and code used in our experiments are publicly available [7].

Offline Monitoring. The bottom half of Table 2 shows the aggregated performance measurements for offline monitoring, i.e., processing the stored logs as quickly as the monitor allows. We instantiated the monitoring pipeline separately for every combination of policy and log (i.e., system test run or the production fragment). For *Test*, the table shows the median (and maximum) monitoring time and memory. Some of the policies are not applicable to production (see Table 1) and hence are marked with '–'. We set a timeout of 30 min for *Test* to limit the experiments' duration. It was reached in five runs, which are excluded from the results, as shown in the table. For *Prod*, we set a timeout of 4 h (the length of the *Prod* fragment plus a safety margin), marked 'TO' in the table.

The results for the *Test* scenario are similar across policies, with few exceptions. Both `logging-behavior` and `unauthorized-connections` require significantly more memory on certain inputs, since they store many snapshots of the InSubnet relation in proportion to the *index rate*, i.e., the number of indices in the corresponding trace per unit of real time. The relation's size depends on the number of nodes created in the test. The policies perform nontrivial computations for every event *and* node, resulting in the timeouts for *Prod*, which has ten times more nodes than *Test*. The timeouts for `block-validation-latency` are likely caused by the larger number of subnets (29 compared to maximal 3) in the corresponding logs; we plan to confirm this in the future. The other two *Test* timeouts occurred with the largest log file (3.3 times the size of the next largest).

Online Monitoring. Long-running systems like the IC are not expected to terminate and hence they produce logs with unbounded streams of events. Therefore,
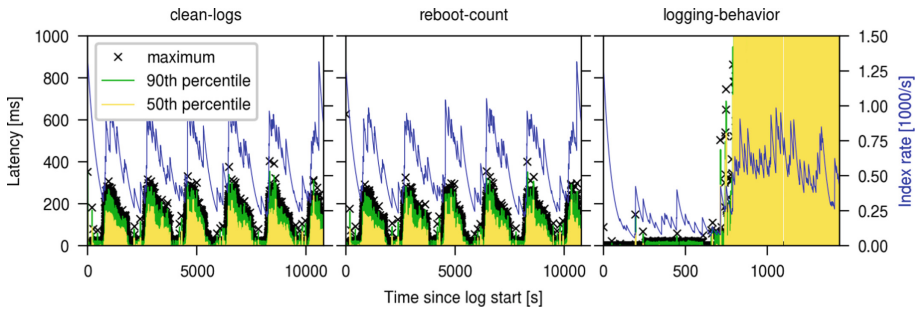
Fig. 5. Replayer latency for online monitoring

online monitoring with low (bounded) latency is a prerequisite for continuous monitoring. Logging activity may also be bursty, rendering the offline performance a bad predictor for the online case. We therefore conducted separate online monitoring experiments using the *Prod* data. Specifically, we measured the latency *at the replayer*, which was provided with an already processed log. While this measure is not equivalent to end-to-end latency, it is practically relevant as it indicates how much log data must be buffered by system components *before* the monitor.

Figure 5 shows the latency distribution over elapsed time, relative to the log entries' time-stamps. For the `clean-logs` and `reboot-count` policies, we observed regular bursts of increased latency. Since the maximum latency does not grow over time, it would be possible to monitor these policies online in a production deployment. The bursts are clearly correlated with the index rate as shown by the thin line drawn on top of the latency distribution.

In contrast, the latency increased steeply after approximately 13 min for `logging-behavior`, simultaneously with the first index rate burst. The experiment was terminated once a latency of 10 min was reached. We do not show results for the `unauthorized-connections` policy as it immediately reached the latency limit. The quickly increasing latency indicates that the time spent monitoring the events generated within an interval of real time is longer than the interval itself. This coincides with the timeouts observed in the offline experiments.

In addition to the above experiments, we parallelized online monitoring of the *Prod* fragment using an existing framework [38]. We observed improvements but were unable to achieve low-latency monitoring for `logging-behavior` and `unauthorized-connections`. We conjecture that the framework's inability to reduce the index rate observed by the parallel monitors prevents latency reduction.

Results. We found that offline monitoring of IC system test logs is possible using moderate resources: monitoring extends the tests' runtime by less than

23%,[1] while the peak memory usage of MonPoly was 5 GiB (Q1). Low-latency online monitoring was possible for two applicable policies (Q2). By analyzing this result, we identified three factors that significantly influence online monitoring performance, namely, repeating relational computations, future operators, and eager processing of **let** expressions (Q3). We believe that the insights from our case study are helpful to developers of other monitoring tools.

5   Lessons Learned

We now summarize our case study's qualitative findings on policy engineering and monitoring maintainability.

Policy Engineering. Introducing runtime monitoring into an existing system is challenging. Policy engineering is the process of *identifying* sources of policies, *selecting* useful policies, and making them *precise* and *formal*. The distinction between the last two characteristics is crucial: we argue that the former is difficult to achieve (even using natural language), whereas the latter is relatively straightforward for runtime monitoring experts, if the policy is already precise.

Colombo and Pace [18] claim that policies should not be defined by developers, but rather by a quality assurance (QA) team, as the policies address end users and concern high-level system properties. We agree with this assessment in part: IC policies were sourced from IC's formal method engineers who knew the system and its high-level properties well. However, additional software engineers and researchers were still needed to confirm the semantics of the existing log entries observed by the monitor and possibly augment logging, for example by adding new parameters or new events. Software engineers also had to evaluate the production impact of such modifications (e.g., due to an increase in log volume), and on the debugging processes (e.g., due to increased noise). Such developer insights crucially influenced the final policies. We decided to drop various drafted policies due to the lack of the required log entries.

Colombo and Pace argue that monitoring policies assured by other engineering techniques (e.g., unit testing) is wasteful. They identify cross-cutting properties [18] as the most useful policy class. We agree but additional selection criteria are also relevant. Namely, policies must be *effective* (i.e., capable of detecting relevant problems), *precise* (i.e., producing a low number of false violations), and *actionable* (i.e., given a true violation, a developer can debug it).

We found that an iterative process is needed to devise sufficiently precise policies. Even domain experts can be misguided by their intuition, suggesting policies that fail to account for corner cases and recent system changes. Natural language ambiguity is another source of imprecision. Moreover, typos and logical errors may occur in policy formalizations. In our case study, we experienced all these issues.

---

[1] Maximum monitoring time (80 min) divided by the longest test (362 min).

Finally, we mention some MFOTL policy formula patterns that commonly appeared in our formalizations. Such patterns implement policies that are intuitively and easily expressible in natural language, but cannot be encoded using a single operator of the policy language. For example, one could expect that valuations assigning 0 to c satisfy the policy $c \leftarrow \text{CNT } m; n \text{ Log}(n, m)$ when monitoring a trace without any Log events. However, this is not the case according to MFOTL's semantics as $\text{Log}(n, m)$ is not satisfiable for any n. Sometimes it is necessary to report such valuations (typically, for a finite set values of n). Our formalization of `logging-behavior` demonstrates a pattern that achieves this:

$$c \leftarrow \text{SUM } c; n \left(\left((c \leftarrow \text{CNT } m; n \ \blacklozenge_I \text{Log}(n, m)) \land \text{InIC}(n) \right) \lor \left(\text{InIC}(n) \land c = 0 \right) \right)$$

Here, we count the number of log messages c per node n in an interval I. The result is used to compute the sum of the counts for each node. It is important to include all known nodes (c.f. InIC), even if they did not log any message m in that interval. The above encoding achieves this by adding the actual count to the default of zero (the right disjunct), assigned to *all* nodes. Other common policy formula patterns we identified are *outer joins* [1] and *sliding windows* [2].

Monitoring Maintainability. As in many software projects, engineers assume that logs are inspected by humans, and often freely modify the logging statements [16]. We observed that such changes break the monitoring pipeline outright because the preprocessor fails to process log entries not matching expected patterns. A more challenging problem is that the *meaning* of a log entry may also subtly change, for example, when moved to a different location in the control flow.

To address this, we used system tests that exercise code paths containing policy-relevant logging statements. The test checks if the preprocessor correctly processes log entries. However, we believe that for an evolving system, a structured and type-safe logging interface is necessary to maintain runtime monitoring. Structured logging provides a way of introducing logging statements systematically at different levels of granularity. Type-safe logging can additionally detect a mismatch between the log entry format expected by the monitor and one produced by the logging statements at compile time. Detection of a change in the semantics of a log entry, however, remains an open problem.

6   Related Work

We first summarize approaches to monitoring distributed systems. Afterwards, we describe industrial case studies similar to ours that monitor distributed systems.

A classic result for predicate detection in distributed systems [15] states that exponentially many interleavings of components' traces must be checked in the worst case, which does not scale [27]. Efficient algorithms exist for predicate

classes [36] or under certain assumptions [35,39]. Basin et al. [9] monitor distributed systems with a centralized monitor by merging all the components' traces. As in our work, their merged trace has events with same time-stamps occur in an arbitrary order. They further restrict policies to a logical fragment where that order does not influence the monitor's output. Other approaches focus on distributing the monitor. Bauer and Falcone [13] orchestrate multiple distributed monitors based on the structure of the input LTL formula, such that they jointly monitor the input formula with minimal need to exchange knowledge.

Similar approaches hierarchically organize monitors [17], use regular expressions [24], or stream equations [20] as the policy language. None, however, support an expressive language like MFOTL, with the exception of Schneider et al. [8,38], whose framework we used in our attempts to reduce monitoring latency.

Basin et al. [9] monitored Nokia's data usage policies in three databases running on different distributed components; they also monitored Google's network security policies [6]. El-Hokayem and Falcone [23] monitored traces collected from 27 distributed smart apartment sensors. Colombo et al. [19] monitored policies for an online payment service with millions of credit cards. Kane et al. [31] monitor a controller-area automotive network. Unlike the languages used in these works, we use a more expressive first-order temporal policy language with aggregations.

We conducted a systematic literature review, following best practices [32], to identify and classify monitoring case studies. We collected papers from five conferences and two journals by matching keywords related to runtime verification and case studies. This yielded 54 papers that we manually analyzed to select those 33 papers that use temporal logic as policy languages. Our `finalized-height` policy is more complex than any policy we found: it has a greater number of operators (56) than the next most complex one (44) [21]. Note that without the **let** operator, the `logging-behavior` policy would have required more than 1,000 operators.

7 Conclusion

We have shown how to enrich system testing and metrics with runtime monitoring. In our case study, we formalize and monitor complex, non-local, metric first-order temporal policies of the Internet Computer (IC), a real-world distributed system. The monitoring pipeline we use is tailored to the IC, but we believe that its design can serve as blueprint for monitoring other distributed systems. Some of our policies, although IC-specific in their current form, generalize well to other systems, specifically, to replicated distributed systems that execute in rounds. Another contribution to the formal methods community is our data set, which we publish and which provides a challenging benchmark for monitors supporting metric first-order temporal policies with aggregations.

As future work, in addition to formalizing other IC policies, we plan to improve the feedback that monitors provide to engineers. The emerging research

area of *explanations* [5] for monitoring verdicts can aid the process of fault local-ization, e.g., by visualizing minimal parts of the trace causing a violation. Further monitor optimizations are required to achieve practical online monitoring of the IC production deployment by handling high index rates. We conjecture that this problem is solvable taking inspiration from the algorithms used in signal-based monitoring [22].

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Boston (1995)
2. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2), 121–142 (2006)
3. Internet Computer Association. Internet Computer dashboard (2022). https://dashboard.internetcomputer.org/
4. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification. LNCS, vol. 10457. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5
5. Basin, D., Bhatt, B.N., Traytel, D.: Optimal proofs for linear temporal logic on lasso words. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 37–55. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_3
6. Basin, D., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring of temporal specifications. Formal Methods Syst. Des. **49**(1), 75–108 (2016). https://doi.org/10.1007/s10703-016-0242-y
7. Basin, D., et al.: Monitoring the Internet Computer (artifact) (2022). https://doi.org/10.5281/zenodo.7340850
8. Basin, D., Gras, M., Krstić, S., Schneider, J.: Scalable online monitoring of distributed systems. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 197–220. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_11
9. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: Monitoring data usage in distributed systems. IEEE Trans. Softw. Eng. **39**(10), 1403–1426 (2013)
10. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal Methods Syst. Des. **46**(3), 262–285 (2015). https://doi.org/10.1007/s10703-015-0222-7
11. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015)
12. Basin, D., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES). Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017)
13. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Formal Methods Syst. Des. **48**(1–2), 46–93 (2016). https://doi.org/10.1007/s10703-016-0253-8
14. Camenisch, J., Drijvers, M., Hanke, T., Pignolet, Y.-A., Shoup, V., Williams, D.: Internet Computer consensus. In: Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing, PODC 2022, pp. 81–91. ACM, New York (2022)

15. Chase, C.M., Garg, V.K.: Detection of global predicates: techniques and their limitations. Distrib. Comput. **11**(4), 191–201 (1998)

16. Chen, B., Jiang, Z.M.: Characterizing logging practices in Java-based open source software projects – a replication study in apache software foundation. Empir. Softw. Eng. **22**(1), 330–374 (2017)

17. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Formal Methods Syst. Des. **49**(1), 109–158 (2016). https://doi.org/10.1007/s10703-016-0251-x

18. Colombo, C., Pace, G.J.: Industrial experiences with runtime verification of financial transaction systems: lessons learnt and standing challenges. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 211–232. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_7

19. Colombo, C., Pace, G.J., Abela, P.: Safer asynchronous runtime monitoring using compensations. Formal Methods Syst. Des. **41**(3), 269–294 (2012). https://doi.org/10.1007/s10703-012-0142-8

20. Danielsson, L.M., Sánchez, C.: Decentralized stream runtime verification. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 185–201. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_11

21. Desai, A., Dreossi, T., Seshia, S.A.: Combining model checking and runtime verification for safe robotics. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 172–189. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_11

22. Deshmukh, J.V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., Seshia, S.A.: Robust online monitoring of signal temporal logic. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 55–70. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_4

23. El-Hokayem, A., Falcone, Y.: Bringing runtime verification home. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 222–240. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_13

24. Falcone, Y., Cornebize, T., Fernandez, J.-C.: Efficient and generalized decentralized monitoring of regular languages. In: Ábrahám, E., Palamidessi, C. (eds.) FORTE 2014. LNCS, vol. 8461, pp. 66–83. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43613-4_5

25. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transfer **23**(2), 255–284 (2021). https://doi.org/10.1007/s10009-021-00609-z

26. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)

27. Ganguly, R., et al.: Distributed runtime verification of metric temporal properties for cross-chain protocols. CoRR, abs/2204.09796 (2022)

28. Gopal, M.: Modern Control System Theory. New Age International (1993)

29. Gorostiaga, F., Sánchez, C.: HLola: a very functional tool for extensible stream runtime verification. In: Groote, J.F., Larsen, K.G. (eds.) TACAS 2021. LNCS, vol. 12652, pp. 349–356. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_18

30. Havelund, K., Peled, D., Ulus, D.: DejaVu: a monitoring tool for first-order temporal logic. In: 3rd Workshop on Monitoring and Testing of Cyber-Physical Systems, MT@CPSWeek 2018, Porto, Portugal, 10 April 2018, pp. 12–13. IEEE (2018)

31. Kane, A., Chowdhury, O., Datta, A., Koopman, P.: A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In: Bartocci, E., Majum-

dar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 102–117. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_7

32. Kitchenham, B.A., Brereton, P., Budgen, D., Turner, M., Bailey, J., Linkman, S.G.: Systematic literature reviews in software engineering – a systematic literature review. Inf. Softw. Technol. **51**(1), 7–15 (2009)

33. Krstić, S., Schneider, J.: A benchmark generator for online first-order monitoring. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 482–494. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-60508-7_27

34. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)

35. Momtaz, A., Basnet, N., Abbas, H., Bonakdarpour, B.: Predicate monitoring in distributed cyber-physical systems. In: Feng, L., Fisman, D. (eds.) RV 2021. LNCS, vol. 12974, pp. 3–22. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88494-9_1

36. Ogale, V.A., Garg, V.K.: Detecting temporal logic predicates on distributed computations. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 420–434. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75142-7_32

37. Sacerdoti, F.D., Katz, M.J., Massie, M.L., Culler, D.E.: Wide area cluster monitoring with Ganglia. In: CLUSTER 2003, p. 289. IEEE Computer Society (2003)

38. Schneider, J., Basin, D., Brix, F., Krstić, S., Traytel, D.: Scalable online first-order monitoring. Int. J. Softw. Tools Technol. Transfer **23**(2), 185–208 (2021). https://doi.org/10.1007/s10009-021-00607-1

39. Stoller, S.D.: Detecting global predicates in distributed systems with clocks. Distrib. Comput. **13**(2), 85–98 (2000)

40. The DFINITY Team. The Internet Computer for geeks. Cryptology ePrint Archive, Paper 2022/087 (2022). https://eprint.iacr.org/2022/087

41. Turner, S.: Transport layer security. IEEE Internet Comput. **18**(6), 60–63 (2014)