Self-Adaptive Distributed Runtime Monitoring for Autonomous Systems

Thomas Wright, Morten Haahr Kristensen, Mirgita Frasheri, Peter Gorm Larsen, Lukas Esterle, Cláudio Gomes
Department of Electrical and Computer Engineering, DIGIT
Aarhus University
Aarhus, Denmark
{thomas.wright, mhk, mirgita.frasheri, pgl, lukas.esterle, claudio.gomes}@ece.au.dk

***Abstract*—This paper introduces a framework for self-adaptive distributed runtime monitoring of autonomous systems, where centralised monitoring is often impractical due to communication overhead, latency, and data privacy concerns. We extend stream-based runtime verification with distribution constraints, which specify how monitoring should be distributed across a network, and are continually verified as the system runs. To maintain valid monitor placements under changing conditions (such as node mobility, failures, or topology changes), we introduce a MAPE-K-based scheduler that detects distribution constraint violations and triggers monitor reconfiguration. We evaluate the approach in a ROS 2/Gazebo multi-robot patrolling scenario with four robots and machine-monitoring tasks.**

***Index Terms*—runtime verification, distributed systems, self-adaptation**

I. Introduction

Physical systems operating in the real world, usually governed by autonomous software agents, are required to work not only correctly but, more importantly, safely. Various techniques in runtime monitoring and runtime enforcement have been developed and employed [1], [2]. These methods allow operators to define properties of the system, that are then monitored at runtime. This allows for monitoring of phenomena, the detection of anomalies, and the enforcement of safety properties in real-time. If a property is violated, the system can either respond appropriately or the operator can be notified. While this is widely applied in cyber-physical systems, it is also applicable for cloud computing and the entire compute continuum, where software components are distributed across different nodes, and the behaviour of the system as a whole needs to be monitored and controlled. In this context, runtime monitoring can be used to ensure that the system is operating correctly and to detect and respond to anomalies in real-time, e.g., when a compute node is overloaded or when a component is behaving unexpectedly.

Typically, such monitors are implemented on a single, centralised node, receiving information for the individual subcomponents, and monitoring the system as a whole. This of course introduces an overhead in communication, puts the computational load on a single node, and introduces a single point of failure. In the context of distributed systems, this is not ideal, and therefore, distributed runtime monitoring has been proposed as an alternative [3]. In distributed runtime monitoring, the specification is decomposed and assigned to monitors distributed across the system. Many approaches exploit locality in observations to partially evaluate the specification at the source, and when a complete system overview is required, local monitors communicate their verdicts to a central component, reducing communication overhead and improving monitoring efficiency. However, this decomposition and distribution of monitors is performed at design time, and monitors are statically assigned to nodes.

When dynamic conditions arise, whether due to dynamics in the environment or due to changes in the monitoring system, this approach will meet its limits. Dynamics in the environment can, for example, be migrating processes in the cloud [4], data traffic on a computational network [5], smart grids with prosumers [6], or wildlife monitored with static sensors [7]. Here, the dynamics in the environment require the monitors to adapt their specifications accordingly. In contrast, dynamics in the monitoring systems can, for example, be mobile robots for inspection tasks or satellites for monitoring the environment [8]. In this case, only the specification based on their current location is required. In self-adaptive and self-organising systems, we often face both dynamics, where mobile systems monitor mobile phenomena, requiring repeated adaptations to their specifications [9]. With either of these dynamic situations, simply decomposing specifications and distributing monitors accordingly at design time is not sufficient as changing conditions may require the monitors to consider new or different specifications.

In this paper we tackle precisely this problem of dynamically updating monitors based on distribution constraints in dynamic environments. First, we provide a monitoring framework that allows for the verification of distribution constraints at runtime using our novel stream-based approach. This allows us to continuously monitor the distribution of monitors alongside the underlying system they are monitoring. Second, we propose an approach allowing to distribute specifications to monitors based on distribution constraints, and to adapt the distribution of monitors at runtime based on changes in the environment and the monitoring system. This approach introduces a self-adaptive scheduler which takes into account

The work presented here is supported by the RoboSAPIENS project funded by the European Commission's Horizon Europe programme under grant agreement number 101133807. We also acknowledge Andreas Kaag Thomsen for useful discussions and input.

the distribution constraints as well as the current state of the system and its individual nodes. Finally, we present a case study to demonstrate the applicability and feasibility of our approach in a multi-robot application focusing on navigation tasks.

The remainder of this paper is structured as follows: In Section II, some background information and the problem formulation is provided. In Section III, we discuss related work in the field of distributed runtime monitoring and self-adaptive systems. In Section IV, we present our case study system and specification. In Section V, we introduce our dynamic distributed runtime verification approach. We carry out the case study and present our results in Section VI. Finally, in Section VII, we conclude the paper and discuss future work.

II. Background and Problem Formulation

In this section we first introduce the Stream-Based Runtime Verification language DSRV [10] (1), before formulating the main problems of distributed runtime verification (2) and of monitor placement within a distributed runtime verification system (3).

1) Stream-Based Runtime Verification in DSRV: Runtime Verification seeks to ensure the correct behaviour of systems by coupling a system with a runtime monitor, an additional program which monitors the behaviour of a system (as observed by a number of input variables) and produces verdicts reporting its adherence to or violation of its specification (reported via a number of Boolean output variables). In Stream-based Runtime Verification (SRV) [11] approaches, monitors are viewed as stream-transformers — reactive programs transforming input streams into output streams — and are specified via a list of equations relating stream variables.

For this paper we use DSRV [10], a recently introduced SRV language with support for dynamic reconfiguration of monitors (originally published under the name DynSRV). DSRV is capable of handling a full range of input data types (Bools, Floats, Strings, Maps, etc., as well as domain-specific object types), and supports a range of messaging protocols for receiving inputs and producing outputs (structured input files, ROS 2 topics [12], MQTT [13], and Redis pub-sub channels [14]). It also supports asynchronous evaluation of streams, so that new verdicts may be produced whenever new values are provided on any input channel.

An example specification is

```
in x: Float
in y: Float
out close: Bool
out closeRecently: Bool
close = (x - y) <= 5 && (y - x) <= 5
closeRecently =    close
                && default(close[1], true)
                && default(close[2], true)
```

which includes two Floating point input streams, `x` and `y`, and produces two Boolean output streams `close` and `closeRecently` respectively.

The floating-point inputs streams can be supplied with sequences of inputs,

$$\begin{aligned} \mathtt{x} &= (\mathtt{x}_1, \mathtt{x}_2, \ldots) \\ \mathtt{y} &= (\mathtt{y}_1, \mathtt{y}_2, \ldots) \end{aligned}$$

whilst the output streams are computed by combining existing streams using a number of operators, such as pointwise data and logical operations so that e.g.,

$$\begin{aligned} \mathtt{close} = \big(&(\mathtt{x}_1 - \mathtt{y}_1) \leq 5 \wedge (\mathtt{y}_1 - \mathtt{x}_1) \leq 5, \\ &(\mathtt{x}_2 - \mathtt{y}_2) \leq 5 \wedge (\mathtt{y}_2 - \mathtt{x}_2) \leq 5, \\ &\ldots \big) \end{aligned}$$

whilst streams can index previous stream values with temporal indexing operators `x[n]` where `n` is a non-negative integer time index, and `default(x, d)` is a builtin function which replaces the value of `x` with d when undefined. Thus the stream `closeRecently` is true whenever the values of the input streams `x` and `y` have been within 5 of each other for all of the last 3 timesteps. In the asynchronous semantics, timestep/time indexes are local to individual streams and are merged using last-known-value semantics, similar to asynchronous stream-based languages such as TeSSLa [15]. This means that there is no assumed shared global clock, and any synchronisation between local times must be handled explicitly by the monitors.

2) Distributed Runtime Verification: In distributed runtime verification, we wish to monitor a distributed system via a network of communicating monitors which run on individual nodes of the network to monitor a system component locally, rather than a single centralised monitor for the whole system. Consider a distributed network, which we can represent as a graph $G = (V, E)$, with a set of nodes V and edges E. Given G and a DSRV specification spec with a set N of input/output stream variables, distributed runtime verification of `spec` involves finding a family of specifications spec_v such that concurrently monitoring spec_v on each node $v \in V$ is equivalent to monitoring the original specification (assuming that the local monitors have access to each other's outputs as inputs). In other words, we want to split up the original monitor to a collection of local monitors running on individual network nodes, while providing equivalent verdicts compared to a specification centrally monitoring the entire system.

To control how this distribution happens, we introduce a *distribution function* $\mathrm{res}(v) \subseteq N$. That is, for each node $v \in V$, we have a set of variables $N_v = \mathrm{res}(v)$ which are the output variables, locally used by the monitor of that node. Then a distribution of the specification spec over the network G with respect to distribution function res is a family of specifications spec_v such that for all $v \in V$, we have that spec_v is the *localisation of* spec *to the set of variables* $N_v = \mathrm{res}(v)$; that is, output streams from the original specification spec are present in spec_v exactly if they occur in N_v.

This localised spec $\mathtt{spec}_v$ can be obtained by a source-to-source transformation which retains from `spec` just a given set of streams $\{x_1, x_2, \ldots, x_n\}$. Then the output stream

variables of $\texttt{spec}_v$ are given by the expressions for $N_v = \{x_1, x_2, \ldots, x_n\}$ in $\texttt{spec}$ whilst the input variables for $\texttt{spec}_v$ consist of any input variables on which these depend, alongside a new input variable for any removed stream variable upon which the remaining stream variables depend. Thus, when a specification contains multiple interdependent streams, it is split into multiple different interdependent localised specifications, each of whose outputs supply inputs to the others.

3) The Monitor Placement Problem: Finally, if we wish to achieve distributed runtime verification in an automated way, we need to decide how to split up the variables of the original specification $\texttt{spec}$ across a distributed network G. Whilst this could be done arbitrarily, we often have specific requirements or constraints for how this distribution is carried out.

The *monitor placement problem* is the problem of finding a valid distribution function res for a given distributed network G and specification spec which meets a given distribution requirement property R which may depend on both the distribution function res, the topology of the distributed network G, and the current state of the system.

III. Related Work

Formalisms to describe *synchronous* systems, such as SIGNAL [16], and formalisms to specify properties to be monitored of *synchronous* systems, such as Linear Temporal Logic (LTL) [17] and LOLA [11] have evolved in tandem. In the same fashion, distributed systems, and their techniques, have driven the need and development of distributed monitoring. To meet these needs, dedicated scheduling strategies and supporting tools have been proposed that support the distributed implementation of SIGNAL [18].

In the remainder of this Section, we discuss the state of the art along three axes, defined in the following subsections: Monitoring Distributed Systems; Distributed Monitoring & Distribution Methods; and Types of Properties. It is important to emphasise that we distinguish works that focus on the monitoring of a system under study that is distributed, from works that decompose the specification and distribute it as sub-monitors that are deployed across the distributed system under study.

A. Monitoring Distributed Systems

The system that is being monitored is typically defined as the system under study. It can be a distributed system, a distributed software system, or a specific distributed system such as blockchain smart contracts, or adaptive kinematic robot swarms. A distributed system is a system that consists of multiple nodes or components that are located on different networked computers, which communicate and coordinate their actions by passing messages to one another. In addition, note that we use the adjective *adaptive* to refer to systems that can change their structure and/or behaviour at runtime.

Bauer et al. [19] present one of the first algorithms to distribute a Linear Temporal Logic (LTL) formula into subformulas monitored on different components without a central observer. The local monitors collaboratively decide satisfaction or violation of a global LTL property. Mostafa et al. [20] employ a three-valued semantics for LTL (true/false/unknown) to handle partial knowledge at each monitor until enough information arrives. The approach ensures that even without a global clock, monitors can collaboratively detect property satisfaction or violation.

Other works consider the spatial location of the components in the system. One example in the category of *spatially distributed systems* is the work of Bartocci et al. [21], which introduces STREL (Spatio-Temporal Reach and Escape Logic), a novel logic designed to specify and monitor spatio-temporal properties of mobile and distributed cyber-physical systems (CPS).

Ganguly et al. [22] leverage Metric Temporal Logic (MTL) in partially synchronous distributed systems. The authors introduce a progression-based formula rewriting technique: each MTL formula is progressively evaluated over time and reduced to simpler formulas, with checks delegated to an SMT solver. This approach yields a generalised distributed RV algorithm for time-sensitive properties.

B. Distributed Monitoring & Distribution Method

Most of the works in the state of the art focus on decentralised monitoring, where the monitors are distributed along the components of the system under study. For instance, Audrito et al. [23] used past-CTL, a past-time variant of the branching-time logic Computational Tree Logic (CTL) [24], to reason about the knowable history of events in a distributed network. Monitors for past-CTL formulas are automatically derived and deployed on each device in a network of IoT-like distributed nodes. The collective behaviour of these monitors, coordinated through a novel use of the field calculus (an aggregate computing paradigm), ensures the property is checked across space and time without any central coordinator. Yan et al. [25] define a Swarm Signal Temporal Logic (SwarmSTL) for properties of robot swarms, and develop a decentralised monitoring framework. Each robot runs a local monitor that evaluates the SwarmSTL formula relative to its own observations and those received via consensus from neighbours.

Within the distributed monitoring category, we can distinguish manual distribution from automatic distribution. For instance, the works of Audrito et al. [23], Colombo and Falcone [26], and Yan and Julius [25] fall in the automated distribution category.

C. Types of Properties

Another axis of our classification is the type of properties that are monitored. The most common properties are temporal properties, such as LTL (as implemented in [26]), STL (as implemented in Bonakdarpour et al. [27]), and MTL (implemented in Ganguly et al. [22]), which are used to specify the behaviour of systems over time. However, there are also other types of properties that are subsets of temporal languages such as PT-DTL (subset of LTL focused on safety), introduced in Sen et al. [28], and languages that give more control over what to monitor and the distribution of the monitors, such as LOLA.

Danielsson and Sánchez [29] introduced a formal model for a decentralised extension of LOLA including routing, lazy vs. eager communication strategies, and a novel monitorability condition called decentralised efficient monitorability, ensuring bounded memory use regardless of input trace length. The authors also provide correctness proofs, complexity bounds, and a prototype implementation, dLola, which they evaluate empirically.

Instead of implementing a new language to specify properties to be monitored, Henry et al. [30] leverage existing automata formalisms which have accepting states for properties that are violated or accepted. They focus on real-time systems and consider properties specified as reachability conditions on timed automata in a distributed setting. The algorithm dynamically updates the set of possible states of the timed automaton as new events arrive, and computes verdicts as soon as they become conclusive.

D. Comparison with our Work

Our work, driven by the need to monitor distributed adaptive systems, is unique in its focus on adaptive distributed runtime verification, where monitors can migrate and adapt the distribution at runtime. We have found no works that perform adaptive distribution as described. This is a key feature of our approach, and we believe it is a promising direction for future research. In Table I the state of the art in distributed runtime verification is summarised, highlighting the system under study, types of properties, monitoring method, and distribution method, as introduced above.

IV. Running Example

We consider a scenario involving a team of $N = 4$ autonomous mobile robots operating in a static, enclosed environment, such as a floor plan in a warehouse. Each robot is initialised in a specific location in the environment, and is assigned a periodic patrolling task that consists of continuously following a given path from start to finish (depicted in Figure 1).

The floor plan is divided into four areas, each containing one machine. In addition to their patrolling task, robots are responsible for monitoring the operational state of the machine in their current area and reporting any anomalies. Since each robot operates solely on information from its onboard sensors, such as cameras or LiDAR, and a preloaded map, a machine can only be observed when a robot is physically present within its area. Once a robot moves on, monitoring of that machine is interrupted until another robot enters. Crucially, no robot has access to the global state of the entire system; acquiring and maintaining such information would be both bandwidth-intensive and computationally impractical in a distributed setting. As a result, the monitoring approach must allow for decentralised monitoring, relying only on local observations and partial, spatially bounded knowledge of the environment. The system-level goal is to maintain continuous, or near-continuous, coverage of all machines simultaneously, ensuring that each machine has a robot assigned to it at all times, or with minimal gaps.

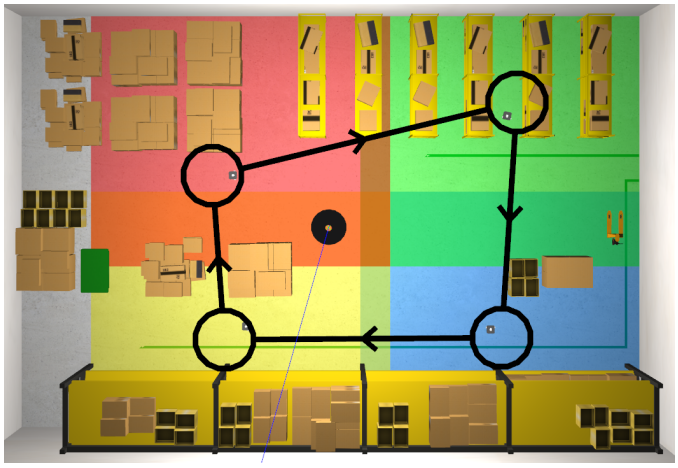


Fig. 1. Path in the floor plan in black tracked by each robot, with circles surrounding each waypoint and arrows highlighting the direction of travel. The floor plan is divided into the four bounding areas for the machines, highlighted in red, green, yellow, and blue.

Each of the four machines presents a distinct monitoring task. A conveyor belt requires tracking the number of items currently loaded, which must be non-negative and remain within a defined upper threshold to prevent overloading. A stacker crane must be monitored for its rail position, which should stay within a designated operational range to avoid collisions or mechanical faults. A vertical lift requires continuous weight monitoring, ensuring the load remains within the lift's rated capacity for safe operation. Finally, a horizontal carousel is subject to stochastic failure, i.e., it operates normally under most conditions but may malfunction at random, requiring a robot to report these unexpected failures.

DSRV Properties: The basic properties for each of the machines are modelled as

```
CPred = ConveyorSystem < 0
SPred =    StackerCrane > 0
        && StackerCrane < 30
VPred = VerticalLift < 100
HPred = HorizontalCarousel == 0
```

each of which takes in a numerical input representing the robot's observations of the corresponding machine and imposes constraints on its values.

We also want to track whether the robots are close enough to the original machines to monitor them, in order to monitor how well the robots cover the space. For each robot `Ri` we define the machine proximity streams,

```
baRiC =    abs(RiX - CX) <= D
        && abs(RiY - CY) <= D
baRiS =    abs(RiX - SX) <= D
        && abs(RiY - SY) <= D
baRiV =    abs(RiX - VX) <= D
        && abs(RiY - VY) <= D
```

TABLE I
OVERVIEW OF MONITORING APPROACHES IN DISTRIBUTED SYSTEMS. THE ASTERISK (*) INDICATES THAT THE SPECIFICATION LANGUAGE IS ORIGINALLY INTRODUCED IN THE REFERENCE. THE LAST ROW SHOWS OUR WORK.

Paper	System Under Study	Types of Properties	Monitoring Method	Distribution Method
[28]	Distributed Software System	PT-DTL*	Decentralised	Manual
[31]	Distributed Software System	PT-DTL	Decentralised	Manual
[20]	Distributed Software System	LTL	Decentralised	Manual
[19]	Distributed System	LTL	Decentralised	Manual
[26]	Distributed System	LTL	Decentralised	Automated
[29]	Distributed Systems (Synchronous)	Lola	Decentralised	Manual
[21]	Spatially Distributed Systems	STREL*	Centralised	N/A
[22]	Blockchain smart contracts	MTL	Centralised	N/A
[23]	Distributed Systems (Homogeneous)	past-CTL*	Decentralised	Automated
[25]	Adaptive Kinematic Robot Swarms	SwarmSTL*	Decentralised	Automated
[32]	Distributed System	STL (3-valued)	Centralised	N/A
[27]	Distributed System	STL	Centralised	N/A
[30]	Distributed System	Reachability timed automata	Decentralised	Manual
Our Work	Adaptive Distributed System	DSRV + distribution	Decentralised, Dynamic	Automated, Adaptive

```
baRiH =     abs(RiX - HX) <= D
        && abs(RiY - HY) <= D
```

where the constant `D` $= 5$ is the coordinate-wise distance threshold for the monitorable region around each monitor and the constants `CX`, `CY`, `SX`, `SY`, `VX`, `VY`, `HX`, and `HY` are the X and Y coordinates of each of the respective machines. Each constraint checks whether the robot is within a square bounding area around the machine.

These can be used to compute conditional alert messages

```
CSiAlert = if baR1C then
           if CPred then 1
           else 0 else -1
SCiAlert = if baR1S then
           if SPred then 1
           else 0 else -1
VLiAlert = if baR1V then
           if VPred then 1
           else 0 else -1
HCiAlert = if baR1H then
           if HPred then 1
           else 0 else -1
```

for each machine and for each robot `Ri` which report 1 if robot `Ri` is in range and the verdict is `true`, 0 if robot `Ri` is in range and the verdict is `false`, or -1 if the robot is out of range of the machine.

V. DYNAMIC DISTRIBUTION OF RUNTIME MONITORS

In this section we outline our approach to dynamic distribution of runtime monitors.

A. Distribution Constraints

A key component of our approach to dynamic distribution is distribution constraints (or dist constraints for short), which specify when and how a distributed runtime verification system should be reconfigured. Distribution constraints extend traditional runtime verification properties by imposing limitations on how monitors may be spatially deployed within a distributed system. That is, distribution constraints are a form of *meta-property* focused on where other properties are being monitored by querying the distributed network and the current distribution function, rather than just the underlying system state. These constraints are important in scenarios where monitor placement is automated but not all deployment configurations are allowed.

Distribution constraints follow the same structure as stream-based monitors in the DSRV language, except they allow for one additional primitive, `monitored_at(x, v)` which returns the Boolean verdict of whether the stream `x` is being monitored on the node `v`. For example, this can be used to specify that a given property `prop1` is monitored at robot A with the property

```
monitored_at(prop1, A).
```

Distribution constraints can be combined with other DSRV expressions to specify state from the underlying system, logical relationships, temporal behaviour, and deployment-related requirements. For example the following logical distribution constraint,

```
   monitored_at(prop1, A)
|| monitored_at(prop1, B)
```

combines two `monitored_at` assertions with a logical or to non-deterministically allow any of the work assignments with `prop1` at either of the nodes `A` or `B`.

And finally, we allow temporal distribution constraints relating historical states such as,

```
   !monitored_at(prop1, A)[2]
|| !monitored_at(prop1, A)[1]
|| !monitored_at(prop1, A)
```

which states that the property `prop1` can be monitored at `A` for at most 2 timesteps in a row, after which it must switch to being monitored at a different robot.

Constraints for Running Example: For the running example introduced in Section IV, we wish to ensure the monitor for a machine is always run by the robot which is currently in the

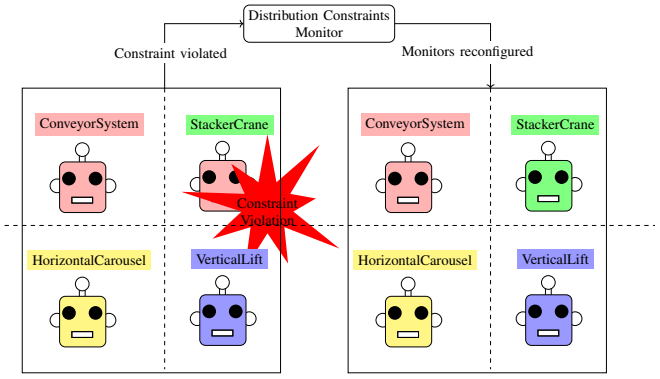


Fig. 2. Example of a distribution-constraint violation and subsequent monitor reconfiguration. Each coloured machine should be monitored by a robot in the corresponding area.

same area as the machine. This is achieved by adding one distribution constraint for each machine, so for example, for the Vertical Lift (VL) we have a distribution constraint, `distVL` which asserts that for the closest robot A, both `VL1Alert` and `baR1V` must be monitored at A, whilst all other `baRjV` and `VLj` streams must be monitored nowhere; that is, we assert, for example, `monitored_at(VL2Alert, None)` where `None` is a distinguished network node meaning "monitored nowhere".

B. Distribution approach

There are several possible approaches to distributing monitors across a network. The first is a centralised approach in which all properties are monitored on a single node (either a particular local node within a distributed system, or a separate central monitor node). The second statically distributes the properties to the individual nodes of the system. This can be done either redundantly and symmetrically, so each node attempts to monitor as much of the full specification as possible, or asymmetrically so that each node is given a particular part of the specification achieving a distribution pattern satisfying the distribution constraints. Finally, the main approach of this paper dynamically splits up the specification and distributes the parts in a manner which tries to satisfy the distribution constraints throughout the runtime of the system.

Dynamic scheduling and distribution of monitors relies upon the following three aspects:

1) *Distribution constraint monitoring* to detect if the distribution constraints hold at a given point in the execution of the system and the accompanying local runtime monitors.
2) *Planning/Scheduling* to decide how to split up the streams of the specification across a distributed network $G = (V, E)$ by solving a monitor placement problem to find an appropriate distribution function res.
3) *Reconfiguration* of a node monitoring a given (localised) specification $\mathtt{spec}_1$ to monitor a new (localised) specification $\mathtt{spec}_2$.

The planning/scheduling stage decides which of the streams of the original specification should run on each node of the network, whilst localisation is used to transform the original global specification into an equivalent network of local specifications. Distribution constraints monitoring allows us to detect when the distribution constraints are broken in order to trigger replanning and a subsequent relocalisation and reconfiguration of the specification being executed by the local monitor at each node (as shown for our running example in Fig. 2).

Each of these aspects is implemented as follows:

Distribution constraint monitoring: Since distribution constraints extend the syntax of DSRV monitors, they can be monitored in a similar way to regular DSRV properties, except that we track the chosen distribution function res_i at each time step, so that we can evaluate `monitored_at` constraints such that `monitored_at(prop1, A)` is `True` at timestep i exactly if $\mathtt{prop1} \in \mathrm{res}_i(A)$. Here, i is the local time index of the distribution-constraint monitor, induced by updates to relevant input variables rather than a globally shared clock tick. Then each of the individual distribution constraints d_j are combined into a single constraint $d = d_1 \wedge \ldots \wedge d_m$, which can be monitored to confirm whether all distribution constraints are met at a moment of time. In static monitoring, this can be checked once given the initial state of the system, whilst in dynamic monitoring, we can continually monitor it as the system runs to trigger reconfiguration on violations.

Planning/scheduling: For the planning/scheduling phase, we attempt to find a distribution function res which is compatible with each of the distribution constraints given the state from the system. That is, we want to solve the monitor placement problem with the distribution requirements property R defined by applying the distribution constraints to the current monitored system state and configuration. This is obtained by evaluating the combined distribution constraints $d \equiv d_1 \wedge \ldots \wedge d_n$ for the system given the current system input signals and network G.

The resulting monitor placement problem can either be solved in a brute force manner, by testing the requirements against every possible distribution function res until a compatible one is found, or by using a SAT solver, by encoding the distribution constraints as a propositional logic formula and using the solution to determine the placement of monitors. We favour the latter approach, and argue that it scales sufficiently to validate our core approach for reasonably complex distribution constraints and distribution graphs G. Although SAT-solving is NP-complete in general, modern SAT solvers scale well for many practical instances, including those arising from our distribution constraint encoding, using algorithms such as DPLL [33] and Conflict-Driven Clause Learning (CDCL) [34]. SAT solving may still become limiting for larger systems or more complex constraints, in which case heuristic or approximate placement strategies could be used. The current implementation treats placement as a feasibility problem: any satisfying assignment returned by the SAT solver may be used. Preferences such as minimising reconfiguration, balancing monitor load, or reducing communication cost are

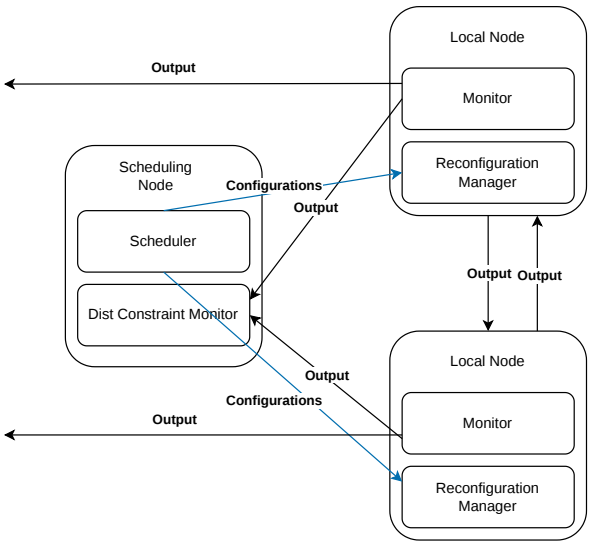


Fig. 3. Distributed architecture with a scheduler and reconfigurable local monitor nodes. Monitor verdicts are shown in black and scheduler reconfiguration messages in blue.

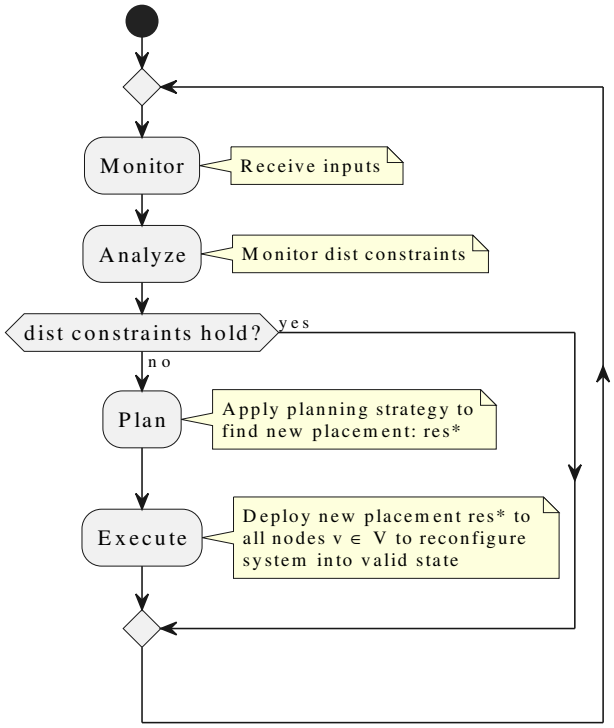


Fig. 4. Scheduler MAPE-K control loop.

not optimised directly, but could be encoded as additional constraints or addressed in a future extension to the planner.

It is possible that there is no valid solution to the monitor placement problem. In this case we are unable to reconfigure the system, and must retain the existing configuration until a valid solution becomes possible. The current implementation does not rank properties during an unsatisfiable state; priority or fallback policies must be expressed explicitly in the distribution constraints.

Reconfiguration: In order to be reconfigurable, each node must listen for requests to change the specification it is currently monitoring to a new one. This involves changing what local stream variables are being monitored locally, as well as reconfiguring its place in the topology of the distributed monitoring system, by subscribing to additional new inputs and by publishing a new set of output streams. This is done via an extension of the DSRV runtime to enable full specification reconfiguration, building on the approach to reconfigurable runtime verification introduced in [10].

C. Distributed architecture

In order to support dynamic, distributed runtime verification, our overall architecture combines reconfigurable local monitors which run on individual network nodes (which correspond to the nodes of the underlying distributed system we are monitoring), as well as an additional scheduling node which monitors the distribution constraints and coordinates reconfiguration. The overall architecture is shown in Fig. 3.

Within this, the local nodes correspond to the nodes of the underlying distributed network G, whilst the Monitor nodes represent instances of a DSRV runtime monitor with the Dist Constraint Monitor being a specialised DSRV instance monitoring the distribution constraints.

Communication between local nodes and the scheduler can use asynchronous message passing, such as ROS topics or MQTT, assuming that the underlying transport provides the reliability guarantees required by the deployment, e.g. through ROS 2 QoS settings. Since local monitors, the monitored system, and the scheduler do not share a clock, delayed or out-of-sync updates can give the scheduler a stale or inconsistent snapshot. This may temporarily result in an invalid placement, or cause the scheduler to retain the existing placement when the observed constraints are unsatisfiable. Local monitor verdicts remain sound for the streams they observe, so asynchrony affects placement timeliness and monitoring coverage rather than local verdict correctness.

The scheduler is centralised in the control plane, but not in the monitoring data plane: local monitors execute localised specifications, while the scheduler evaluates distribution constraints and issues reconfiguration commands. This split is most useful when monitored data streams are larger or more frequent than reconfiguration traffic, since local monitors process high-volume data near its source while the scheduler handles lower-frequency placement changes. Centralisation gives a consistent global placement view and simplifies SAT-based placement, but makes the scheduler a single point of failure for adaptation: existing monitors can continue running after scheduler failure, but new violations require recovery or replacement. Decentralised scheduling, leader election, or hierarchical schedulers are natural extensions for stronger fault

tolerance.

D. Overall Dynamic Monitoring Loop

The overall dynamic reconfiguration loop for the system (shown in Fig. 4) is as an autonomic loop following the MAPE-K architecture [35].

This proceeds through the following stages:

- **Monitor:** The Scheduler continuously monitors two information streams: the current physical network topology, represented by the weighted graph $G = (V, E)$ and the input variables required to evaluate the truth of the distribution constraints (from either the monitored systems or from local monitors).
- **Analyse:** In this phase, the Scheduler decides whether a new monitor placement is required. This uses a version of the DSRV runtime to evaluate the truth of each distribution constraint d_i and thus evaluate the overall conjoined constraint $d = d_1 \wedge \ldots \wedge d_n$. If this is false, we trigger a (re)planning phase.
- **Plan:** If a new placement is needed, the Scheduler invokes a pluggable *planning strategy* to pick a new distribution res solving the monitor placement problem defined by distribution constraints. If no satisfying placement exists, no execution step is triggered and the scheduler goes back to the monitor phase to wait for future state updates.
- **Execute:** Once a valid plan, res^*, is found, the Scheduler enacts it. It communicates with each node $v \in V$, instructing it to reconfigure to monitor a new localised specification spec_v for the variable set $\mathrm{res}^*(v)$, thereby reconfiguring the whole system of local monitors to a new configuration.

These stages depend on the *Knowledge* provided by the historical monitor distributions res_i and the historical state of the distributed system as tracked by the DSRV runtime monitor.

VI. Evaluation

A. Dynamic Monitoring During Navigation

Our first scenario is a full version of the multi-robot dynamic monitoring scenario described in Section IV, with a multi-robot simulation built on the Robot Operating System (ROS 2) [12] and the off-the-shelf navigation stack (Nav2), with Gazebo as the underlying physics simulator [36]. ROS 2 uses the Distributed Data Service (DDS) implementation for the middleware [37], and is designed to support the development of robotics applications, from simple drivers to complex algorithms, e.g., for navigation, vision, etc. The main concepts in ROS include: nodes, topics, services, and action servers. Nodes are executables, and are typically single purpose. They exchange information either through a publish/subscribe mechanism (via topics), or through calls similar to remote procedure calls (via services and action servers). On the other hand, Nav2 provides localisation[1] and mapping capabilities and takes care of the low level control (motion and path planning) for the robots in the simulation.

[1]Used here in the sense of robot navigation, not in the sense of localisation of specifications.

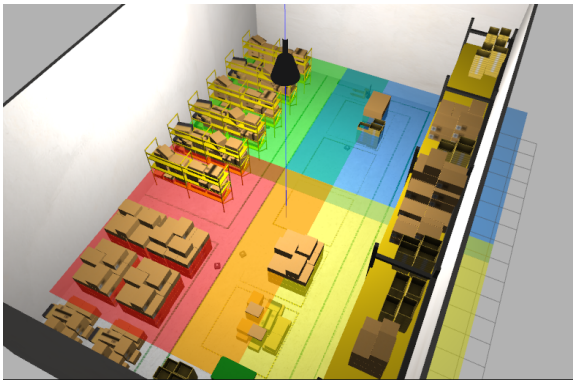


Fig. 5. Gazebo simulation with 4 TurtleBots. Coloured squares show the bounding area of each monitored machine.

a) Simulation Setup: In the environment, we spawn four TurtleBot3 robots (Fig. 5), each equipped with an individual Nav2 stack operating in localisation mode. Each robot navigates independently within the shared environment using only local sensor data and the reference map, providing a realistic testbed for evaluating distributed monitoring in decentralised settings.

In order to emulate the monitoring of the properties described in Section IV, a ROS 2 node has been implemented that generates and publishes test values to appropriate topics for the necessary input variables `ConveyorSystem`, `StackerCrane`, `VerticalLift`, and `HorizontalCarousel`. With this approach we aim to mimic the ability of the robots to use cameras, LiDAR or other sensors to retrieve and analyse information from the environment, while avoiding the complexity of their actual implementation, which is outside the scope of this paper. Meanwhile, the appropriate robot Odometry values come from the Gazebo simulation as structured data on the inputs `R1Odom`, `R2Odom`, `R3Odom`, and `R4Odom`, and are processed into the corresponding cartesian inputs `RiX` and `RiY` by appropriate auxiliary streams.

b) Monitoring Scenarios: To evaluate our approach we compare three monitoring setups for the distributed TurtleBot case study: *centralised monitoring*, where one node monitors the whole specification; *static distributed monitoring*, where localised specifications are assigned to the TurtleBots at the start of the simulation; and *dynamic distributed monitoring*, where the scheduler redistributes specifications at runtime.

c) Measurement Criteria: Firstly, we compare the total number of messages required to measure the system using each of these approaches. Secondly, we measure *monitoring coverage*: the proportion of time the system is able to monitor the system provided that the distribution constraints are actually satisfiable. Coverage also captures transient effects of asynchrony: delayed or out-of-sync updates can temporarily leave the deployed monitors inconsistent with the currently satisfiable distribution constraints. This allows us to measure

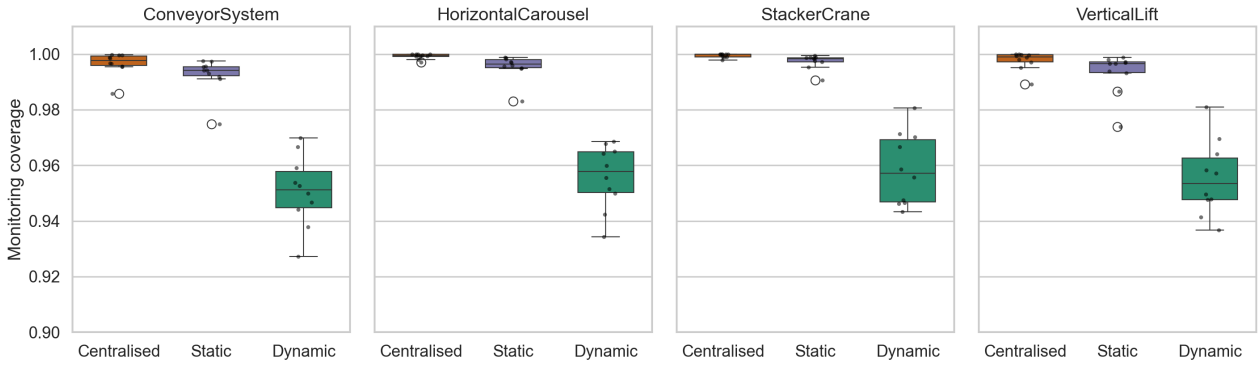


Fig. 6. Distribution of monitoring coverage for the three monitoring approaches.

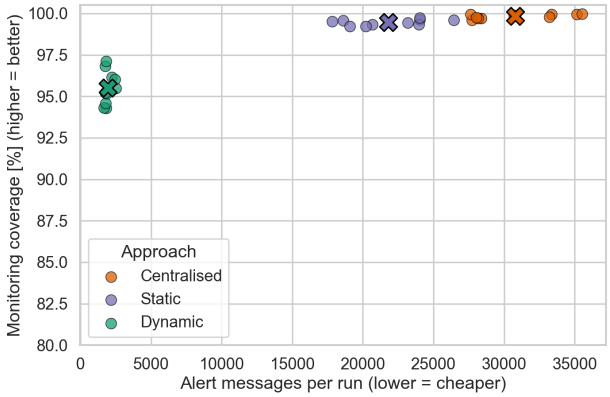


Fig. 7. Pareto front of monitoring quality vs. communication cost. The X marker shows the mean of each approach across 10 runs.

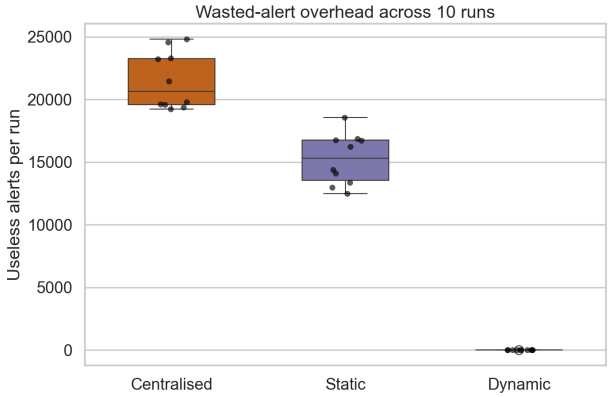


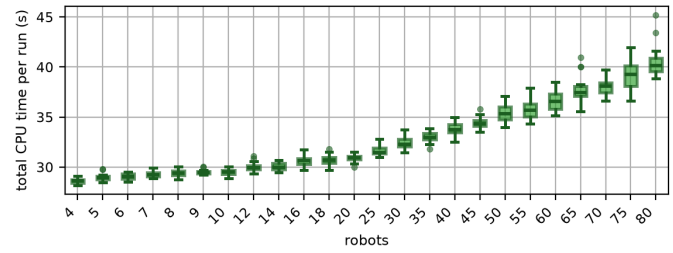
Fig. 8. Useless alerts for the three monitoring approaches.

how successfully our self-adaptive scheduler is able to maintain the distribution constraints throughout the execution of the system, taking into account the time necessary for scheduling and for individual nodes to be reconfigured.

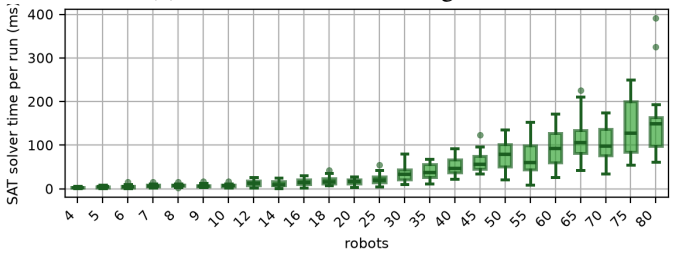
d) Results: Figure 7 highlights the main strength of the dynamically distributed approach by drastically reducing the communication overhead compared to the centralised approach, while maintaining a high monitoring coverage (quality). The static distributed approach also reduces communication overhead, but only by about 10000 messages per run compared to the centralised approach. In contrast, the dynamic distributed approach only requires about 2000 messages in total compared to over 20000 and 30000 messages for statically distributed and centralised approaches, respectively. At the same time, we see only a small decrease in monitoring quality. The magnitude of this reduction is scenario-dependent, since it depends on how much irrelevant data or verdict traffic can be avoided by localising monitors. Figure 6 provides a more detailed overview of the monitoring coverage for the dynamically distributed approach, showing a coverage of around 95%, while the centralised and static distributed approaches achieve close to 100% coverage.

A huge overhead for static distributed monitoring and centralised monitoring in this example is the number of useless alert messages (for the `CSiAlert`, `SCiAlert`, `VLiAlert`, and `HCiAlert` streams). We can clearly observe that dynamic distributed monitoring does not have any useless alerts, while the other two approaches have a mean around 21000 and 15000 useless alerts per run, respectively. This is due to these approaches receiving alerts from monitors that are not actually monitoring the relevant properties.

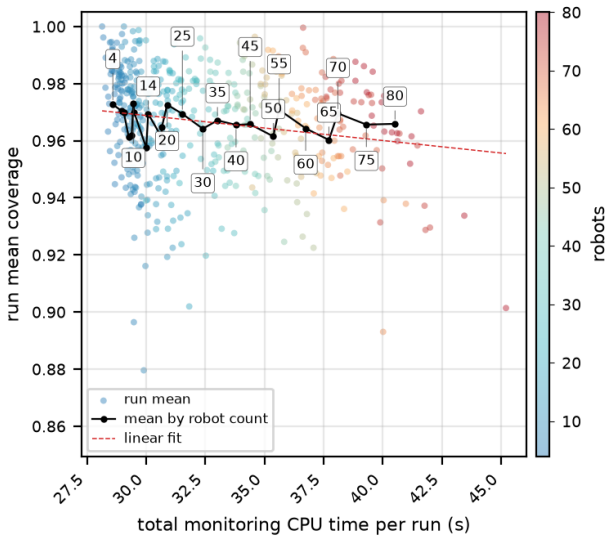
The dynamic distributed approach adds a small reconfiguration overhead, with a mean cost of around 127 messages per run, but this is negligible compared to the reduction in communication cost. In this case study, the distributed monitoring system responds to dynamic environmental changes caused by robot motion, as robots entering or leaving machine regions change which monitor placements satisfy the distribution constraints. These results also provide evidence of adaptivity: robot motion repeatedly invalidates previous placements; the dynamic setup detects these violations, replans, and reconfigures local monitors while maintaining around 95% coverage.



(a) Total monitor CPU usage.



(b) Total SAT solver time.



(c) Coverage versus monitor CPU usage.

Fig. 9. Scalability results from the lightweight Brownian-motion simulator.

B. Scalability Benchmark

To assess scalability beyond what is practical in the Gazebo/Nav2 setup, we use a lightweight ROS 2 Brownian-motion simulator implemented in Rust using the Bevy game engine, which removes physics and navigation-stack overhead while retaining the same monitoring structure. This isolates the cost of monitor placement, constraint checking, and reconfiguration as the number of candidate monitor locations increases.

1) Simulation Setup: The simulator represents each robot as a point following an independent random walk with boundary avoidance in a bounded arena containing the four fixed machine regions from the Gazebo case study. The benchmark uses the same property emulator; the scheduler observes robot positions, evaluates distribution constraints, invokes the SAT-based planner when needed, and sends reconfiguration commands to local monitors.

We vary the number of robots while keeping the monitored machine regions fixed, repeat each configuration across 20 random seeds, which determine the starting positions and motion of the robots, and measure total monitor CPU time, SAT-solver time, and monitoring coverage. As before, coverage denotes the proportion of satisfiable time for which the relevant properties are monitored.

2) Results: Figure 9a–b show that total monitor CPU time increases with the number of robots, since larger deployments provide more candidate placements and therefore larger placement problems. However, SAT-solving remains a small component of the overall monitor cost, indicating that the encoding remains practical and that the centralised scheduler is not a significant computational bottleneck in these workloads, although it remains a control-plane single point of failure as discussed in Section V. Figure 9c shows the cost–coverage relationship: a linear regression over all runs finds a weak negative association between total monitor CPU time and coverage, with slope -9×10^{-4} coverage per CPU-second ($p = 1.37 \times 10^{-4}$, $R^2 = 0.03$), consistent with CPU pressure increasing reconfiguration time in harder runs. The scheduler triggered a mean of 29.3 worker reconfiguration events per run, with median per-run reconfiguration latency of 24.0 ms and median p95 latency of 43.1 ms, providing direct evidence of repeated low-latency runtime adaptation.

VII. Conclusion

In this paper, we proposed a framework for distributed monitoring of self-adaptive systems using distribution constraints and a self-adaptive scheduler to dynamically reconfigure local monitors; the multi-robot case study demonstrates the feasibility of this approach. The results show that distribution constraints can be maintained under mobility-induced changes while substantially reducing communication compared with centralised monitoring. By separating the specification of where monitors may run from the runtime mechanism that chooses placements, the framework supports repeated reconfiguration without requiring properties to be rewritten for each deployment state. Although evaluated in robotic scenarios, the scheduler operates over abstract nodes, streams, and distribution constraints under asynchronous message passing with input-induced timesteps. Future work includes stronger control-plane fault tolerance through replication, leader election, or hierarchical scheduling, as well as quantitative constraints and optimisation-based placement to choose among multiple valid assignments or find best-effort placements for larger networks and network-level properties such as latency and congestion.

References

[1] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to runtime verification," *Lectures on Runtime Verification: Introductory and Advanced Topics*, pp. 1–33, 2018.

[2] S. Pinisetty, Y. Falcone, T. Jéron, and H. Marchand, "Runtime enforcement of regular timed properties," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1279–1286.

[3] A. Francalanza, J. A. Pérez, and C. s. Sánchez, "Runtime verification for decentralised and distributed systems," *Lectures on Runtime Verification: Introductory and Advanced Topics*, pp. 176–210, 2018.

[4] M. F. Gholami, F. Daneshgar, G. Low, and G. Beydoun, "Cloud migration process—a survey, evaluation framework, and open challenges," *Journal of Systems and Software*, vol. 120, pp. 31–69, 2016.

[5] A. D'Alconzo, I. Drago, A. Morichetta, M. Mellia, and P. Casas, "A survey on big data for network traffic monitoring and analysis," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 800–813, 2019.

[6] H. Chen, Z. Zhang, S. Roy, E. Bartocci, S. A. Smolka, S. Stoller, and S. Lin, "Cumulative-time signal temporal logic," *ACM Trans. Embed. Comput. Syst.*, vol. 24, no. 5s, Sep. 2025.

[7] T. Bruce, Z. Amir, B. L. Allen, B. F. Alting, M. Amos, J. Augusteyn, G.-A. Ballard, L. M. Behrendorff, K. Bell, A. J. Bengsen *et al.*, "Large-scale and long-term wildlife research and monitoring using camera traps: a continental synthesis," *Biological Reviews*, vol. 100, no. 2, pp. 530–555, 2025.

[8] A. J. Lee, W. Song, B. Yu, D. Choi, C. Tirtawardhana, and H. Myung, "Survey of robotics technologies for civil infrastructure inspection," *Journal of Infrastructure Intelligence and Resilience*, vol. 2, no. 1, p. 100018, 2023.

[9] J. Kline, A. Zhong, K. Irizarry, C. V. Stewart, C. Stewart, D. I. Rubenstein, and T. Berger-Wolf, "Wildwing: An open-source, autonomous and affordable uas for animal behaviour video monitoring," *Methods in Ecology and Evolution*, vol. 17, no. 2, pp. 291–300, 2026.

[10] M. H. Kristensen, T. Wright, C. Gomes, L. Esterle, and P. G. Larsen, "DynSRV: Dynamically updated properties for stream runtime verification," in *Runtime Verification*, B. Könighofer and H. Torfah, Eds. Cham: Springer Nature Switzerland, 2026, pp. 101–119.

[11] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. Sipma, S. Mehrotra, and Z. Manna, "LOLA: Runtime monitoring of synchronous systems," in *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, Jun. 2005, pp. 166–174.

[12] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[13] A. Banks and R. Gupta, "Mqtt version 3.1.1," OASIS Open, Oct. 2014, oASIS Standard. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[14] Redis Ltd., "Redis," 2026. [Online]. Available: https://redis.io

[15] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, and A. Schramm, "TeSSLa: Runtime verification of non-synchronized real-time streams," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18. Association for Computing Machinery, 2018, pp. 1925–1933.

[16] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: The SIGNAL language and its semantics," *Science of Computer Programming*, vol. 16, no. 2, pp. 103–149, Sep. 1991.

[17] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, 1977, pp. 46–57.

[18] O. Maffeïs and P. Le Guernic, "Distributed implementation of SIGNAL: Scheduling & graph clustering," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, H. Langmaack, W.-P. de Roever, and J. Vytopil, Eds. Berlin, Heidelberg: Springer, 1994, pp. 547–566.

[19] A. Bauer and Y. Falcone, "Decentralised LTL monitoring," *Formal Methods in System Design*, vol. 48, no. 1, pp. 46–93, Apr. 2016.

[20] M. Mostafa and B. Bonakdarpour, "Decentralized Runtime Verification of LTL Specifications in Distributed Systems," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 494–503.

[21] E. Bartocci, L. Bortolussi, M. Loreti, and L. Nenzi, "Monitoring Mobile and Spatially Distributed Cyber-Physical Systems," in *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pp. 146–155, rV specifically for spatially distributed systems.

[22] R. Ganguly, Y. Xue, A. Jonckheere, P. Ljung, B. Schornstein, B. Bonakdarpour, and M. Herlihy, "Distributed Runtime Verification of Metric Temporal Properties for Cross-Chain Protocols," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, Jul. 2022, pp. 23–33.

[23] G. Audrito, F. Damiani, V. Stolz, G. Torta, and M. Viroli, "Distributed runtime verification by past-CTL and the field calculus," *Journal of Systems and Software*, vol. 187, p. 111251, May 2022.

[24] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[25] R. Yan and A. Julius, "Distributed Consensus-Based Online Monitoring of Robot Swarms With Temporal Logic Specifications," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 9413–9420, Oct. 2022.

[26] C. Colombo and Y. Falcone, "Organising LTL monitors over distributed systems with a global clock," *Formal Methods in System Design*, vol. 49, no. 1, pp. 109–158, Oct. 2016.

[27] B. Bonakdarpour, A. Momtaz, D. Ničković, and N. E. Saraç, "Approximate Distributed Monitoring Under Partial Synchrony: Balancing Speed & Accuracy," in *Runtime Verification*, E. Ábrahám and H. Abbas, Eds. Cham: Springer Nature Switzerland, 2025, pp. 282–301.

[28] K. Sen, A. Vardhan, G. Agha, and G. Rosu, "Efficient decentralized monitoring of safety in distributed systems," in *Proceedings. 26th International Conference on Software Engineering*, May 2004, pp. 418–427.

[29] L. M. Danielsson and C. Sánchez, "Decentralized Stream Runtime Verification," in *Runtime Verification*, B. Finkbeiner and L. Mariani, Eds. Cham: Springer International Publishing, 2019, pp. 185–201.

[30] L. Henry, T. Jéron, N. Markey, and V. Roussanaly, "Distributed Monitoring of Timed Properties," in *Runtime Verification*, E. Ábrahám and H. Abbas, Eds. Cham: Springer Nature Switzerland, 2025, pp. 243–261.

[31] T. Scheffel and M. Schmitz, "Three-valued asynchronous distributed runtime verification," in *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Oct. 2014, pp. 52–61.

[32] A. Momtaz, H. Abbas, and B. Bonakdarpour, "Monitoring Signal Temporal Logic in Distributed Cyber-physical Systems," in *Proceedings of the ACM/IEEE 14th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2023)*, ser. ICCPS '23. New York, NY, USA: Association for Computing Machinery, May 2023, pp. 154–165.

[33] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.

[34] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, May 1999.

[35] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[36] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. Ieee, 2004, pp. 2149–2154.

[37] "Object Management Group: A Data Distribution Service for Real-time Systems," https://www.omg.org/omg-dds-portal/, accessed: 2025-04-30.