IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 4 July 2023, accepted 14 July 2023, date of publication 24 July 2023, date of current version 14 August 2023.
Digital Object Identifier 10.1109/ACCESS.2023.3298329

RESEARCH ARTICLE

Decentralized Stream Runtime Verification for Timed Asynchronous Networks

LUIS MIGUEL DANIELSSON[1,2] **AND CÉSAR SÁNCHEZ**[2], **(Senior Member, IEEE)**
[1]DLSIIS Departamento de Lenguajes y Sistemas Informaticos e Ingenieria del Software, Universidad Politécnica de Madrid, 28040 Madrid, Spain
[2]IMDEA Software Institute, 28223 Madrid, Spain

Corresponding author: Luis Miguel Danielsson (lm.danielsson@imdea.org)

This work was supported in part by FPU18/04362 Grant, in part by PRODIGY Project under Grant TED2021-132464B-I00 and Grant MCIN/AEI/10.13039/501100011033, in part by the European Union NextGenerationEU/PRTR, and in part by the Research grant from Nomadic Labs and the Tezos Foundation.

ABSTRACT **Problem:** We study the problem of monitoring distributed systems such as smart buildings, ambient living, wide area networks and other distributed systems that get monitored periodically in human scale times. In these systems computers communicate using message passing and share an almost synchronized clock. This is a realistic scenario for networks where the speed of the monitoring is sufficiently slow (like seconds or tens of seconds) to permit efficient clock synchronization, where clock deviations are small compared to the time precision and frequency required by the monitoring. **Solution:** More concretely, we propose a solution to monitor decentralized systems where monitors are expressed as stream runtime verification specifications. We solve the problem for ''timed asynchronous networks'', where computational nodes where the monitors run have a synchronized clock with a small bounded maximum drift. These nodes communicate using a network, where messages can take arbitrarily long but cannot be duplicated or lost. This setting is common in many cyber-physical systems like smart buildings and ambient living. This assumption generalizes the synchronous monitoring case. Previous approaches to decentralized monitoring were limited to synchronous networks, which are not easily implemented in practice because of network failures. Even when network failures are unusual, they can require several monitoring cycles to be repaired. **Methodology:** We describe formally the monitoring problem for timed-asynchronous networks, we describe a decentralized algorithm and provide proofs of its correctness. Afterwards, we formally analyze the complexity of our solutions and provide two analysis techniques to approximate the memory requirements. Finally, we implement the algorithm and perform an empirical evaluation with real data extracted from four different datasets. **Contributions:** We propose a solution to the timed asynchronous decentralized monitoring problem. We study the specifications and conditions on the network behavior that allow the monitoring to take place with bounded resources, independently of the trace length. Finally, we report the results of an empirical evaluation of an implementation and verify the theoretical results in terms of effectiveness and efficiency.

INDEX TERMS Decentralized monitoring, distributed, runtime verification, stream runtime verification, time asynchronous networks.

I. INTRODUCTION

In many scenarios, the data collected for a given task is intrinsically geographically distributed. In turn, the distribution of data sensed leads to the use of distributed systems.

Examples include smart buildings and ambient living systems that use wide-area networks to capture and perform computations over the data. Furthermore, in order to preserve privacy and improve reliability and fault tolerance, it is often preferable to perform the computations as close to the data collection locations as possible. This is the concept of edge computing, as opposed to centralized computing (including cloud solutions) where all data collected is sent to a central server for computation.

In this paper we propose a solution to observe a large class of distributed systems based on runtime verification,

The associate editor coordinating the review of this manuscript and approving it for publication was Asad Waqar Malik.

a lightweight formal method approach. Sometimes the system under analysis requires a decentralized or distributed monitoring process for various reasons: geographical distribution of data, performance, privacy or fault-tolerance among others. In this case a network of interconnected monitors will cooperate to achieve the monitoring task by consuming a trace of input observations collected at geographically distributed locations, computing partial results and combining them to get to the final verdict by exchanging messages. In decentralized monitoring a specification is decomposed into a network of monitors that communicate by exchanging messages and share a global clock that allow to synchronize the computation. This synchronization is feasible when the clock skews are small compared to the duration of the cycles, like when the monitoring cycle is in human scale in seconds or tens of seconds and the clock skew is, at most, in the order of milliseconds. Decentralized solutions typically also assume that the network guarantees that all messages arrive in a bounded time (for example one monitoring cycle). In contrast, in distributed monitoring processes do not share a global clock and the network is purely asynchronous.

The solution that we propose in this paper is based on runtime verification, a lightweight formal method approach that originates from static formal verification. Static Verification is concerned with ensuring before deployment that all executions of a software artifact has certain desirable properties formally specified (termination, lack of runtime errors, liveness, privacy, efficiency, etc). One difficulty of static verification is scalability as all traces of the system have to be considered. Static verification is even more difficult for reactive systems such as cyber-physical systems where both software and hardware interact with the environment. Verifying reactive systems is even harder when the environment is not of a computational nature, as human-in-the-loop systems, systems that depend on the physical environment (like drones), and distributed systems. In these systems it is very difficult to model precisely the environment. For this reason, static verification techniques are often limited or even not practically viable at all for these kinds of systems in practice. Runtime verification attempts to alleviate these issues by (1) avoiding modelling the environment, and (2) avoiding scalability problems by observing one trace and not all traces of the system. Runtime verification studies how to generate a monitor from a formal specification, and how a monitor will process a trace at runtime to check whether the trace satisfies or violates the specification (this process is called monitoring).

Early approaches to RV specification languages were based on temporal logics [8], [18], [31], regular expressions [46], timed regular expressions [2], rules [3], or rewriting [42]. Another approach to monitor specifications is Stream Runtime Verification (SRV)—pioneered by Lola [16]—which defines monitors by declaring equations that describe the dependencies between output streams of results and input streams of observations. SRV is a richer formalism than most RV solutions that goes beyond Boolean verdicts (like in logical techniques) by allowing specifications that compute richer verdicts as output. Examples include counting events and other statistics, computation of robustness values or generating explanations of the errors. See [16], [17], [23], [29], and [30] for examples illustrating the expressivity of SRV languages. In this work we study and present a solution to the problem of decentralized runtime verification using stream runtime verification (SRV) specifications under the timed asynchronous model of computation.

Another important aspect of runtime verification is the operational execution of monitors: how to collect information and how to perform the monitoring task. We focus in this paper in *online* monitoring where the monitoring happens incrementally as the input trace is being observed. In [9], [17], and [19] the authors consider a centralized specification which gets deployed as network of distributed monitors connected via a synchronous network, where the global synchronous clock is used both for communication and periodic sampling. Monitors exchange messages and cooperate to perform the global monitoring task. This problem is called *decentralized monitoring* (see [25]). This synchronous execution model is realistic in many scenarios, for example in smart buildings or smart cities—where clocks can be synchronized using a time network protocol—that is sufficiently precise for round cycles of tens of seconds. A degenerated case of this setting is a centralized solution: nodes with mapped observations send their sensed values to a fixed central node that is responsible of computing the whole specification. Consider for example an *if-then-else* specification with a slow computation needed to obtain the value of both the *then* and the *else* parts. Consider a decentralized deployment with three monitors connected as a tree: the leaf monitors compute the *then* and the *else* parts, while the root monitor computes the specification using a Boolean input stream for *if* part. Assume that the condition is true 90% of the time, so most of the time the *then* value is used and the *else* value is discarded. Now, also consider that the network link between the root monitor and the leaf monitor that computes the *else* part is slow, the throughput of the root of the specification remains unaffected for that 90% of times and the result can be produced without waiting for the long computation and the network delay of the link that affect the *else* part.

We study *time asynchronous* networks [14], where nodes share a global clock (built upon bounding the network synchronicity delays and hardware clock drifts) but monitoring messages can take arbitrarily long. Time asynchronous networks [14] *"…allow practically needed services such as clock synchronization, membership, consensus, election, and atomic broadcast to be implemented"*. Synchronous networks are a special case where, additionally, messages take a known bounded time to arrive. We study here the timed asynchronous networks of communication together with periodic sampling of inputs, that is a synchronous computation over an asynchronous network. Our solution subsumes our previously available SRV solution for synchronous computation and a synchronous reliable network [17].

We call the more general problem studied in this paper the *timed asynchronous decentralized monitoring problem*. Our goal is to generate local monitors at each node that collaborate to monitor the specification, distributing the computational load while minimizing the network bandwidth and the latency of the computation of verdicts. Apart from more efficient evaluation, decentralized monitoring can provide fault-tolerance as the processes can partially evaluate a specification using the information provided by the part of the network that does not fail. In the same spirit, if part of the network of cooperating monitors is clogged—in the sense that it is working slower for some reason—the other part can keep its normal throughput.

Also, we consider inputs to always be available so that the input traces are complete and we have a value for each time instant. In this paper we provide a solution to the decentralized monitoring problem for Lola [16] specifications for arbitrary network topologies and placement of the local monitors. We also assume in this paper a reliable system: nodes do not crash, and messages are not lost or duplicated. The only possible fault that we consider is network delays. These assumptions are realistic in a practical distributed setting using the following standard techniques. For the reliability of nodes we could use persistent memory in the sense that at the end of each computing round all the internal memory of each node is dumped to persistent storage. For the message loss and duplication we could employ a standard technique: either TCP/IP protocol or sliding windows. Both of them guarantee that messages are delivered in order to the destination.

Our assumptions on nodes not crashing, and messages not getting lost or duplicated are common in the decentralized RV community [1], [9], [17], [19], [20], [21], [22], [43], [47] and in classical distributed algorithms [35]. Most previous works are centralized and of those that are decentralized the majority are based on synchronous systems where messages take a fixed amount of time to arrive, or even a single time unit.

Other typical solutions to attack this problem may be to use a cloud solution or a centralized server that would collect all data and process it. Some limitations of this approach are a single point of failure, extra latency compared to processing near the data gathering locations, loss of privacy: since all data must be sent to a central node that gets to observe all information. Consequently, a core idea of decentralized RV is to be able to monitor in the same infrastructure as the observed system and, ideally, consume near to zero resources per node in order to avoid affecting the observed system performance. To this end, decentralizing the monitoring task is the most straightforward way of reducing the impact of the monitoring task(by distributing it) on all nodes of the system. In the RV community decentralized monitoring is desirable in order to improve on the privacy, latency and fault-tolerance of the system [1], [9], [17], [19], [20], [21], [22], [43], [47]. We build upon this tradition and extend it in our work.

A. OUR SOLUTION

We use the fact that a global clock is available to use a model of computation for monitoring that proceeds in rounds, where each round consists on input readings and process incoming messages, followed by an update the internal state of the local monitors and finally producing output messages. These messages may take arbitrarily long to arrive. In our solution, different parts of the specification are deployed into different network nodes as a local monitors. Each monitor models a different part of the specification modeled as a streams, including input readings. Local monitors will communicate with other monitors when necessary to resolve the streams assigned to them, trying to minimize the communication overhead.

Intuitively, data will be read from sensors, and then each layer of intermediate monitors will compute sub-expressions and communicate partial results to remote monitors in charge of super-expressions, ultimately computing the stream of values of the root expression. The SRV language that we consider to write the specifications is Lola [16], [45]. We will identify those specifications and conditions on the network behavior that allow the monitoring to take place with bounded resources, independently of the trace length.

B. MOTIVATING EXAMPLE

Example 1: We use as a running example a smart building with rooms equipped with sensors and a central node. The aim is to generate alarms when there is a fire risk. There are two ways of generating the alarm, either by manual activation or when the increment in a certain room in both temperature and CO_2 is higher than a threshold. The following specification captures this risk by obtaining the increment in temperature and CO_2 in a given room. We place the computations needed to obtain the increment of the inputs to those nodes where the sensor readings take place. In this way, the central node only needs to compute which rooms present an increment higher than the threshold in both temperature and CO_2 while considering if there is a manual activation of the alarm. We omit the fact that there would be several rooms that are identical, and the only difference in the building monitor is that it should consider all the increments of all the rooms.

```
@Room{
  input num t
  input num co2
  define num tinc = (t - t[-1|1]) / t[-1|1]
  define num co2inc = (co2 - co2[-1|1]) / co2[-1|1]
}
@Building{
 input bool manual
 output bool fire_risk =
 manual_alarm or (tinc > 0.3~and co2inc > 0.3)
}
```

C. CONTRIBUTIONS

The main contribution of this paper is a solution to the timed asynchronous decentralized stream runtime verification problem. We provide a proof of correctness of the algorithms and show that our solution subsumes a synchronous decentralized problem without overhead. A second contribution is the description of those specifications and conditions on the network behavior that allow the monitoring to be carried out with bounded resources, independently of the trace length. Bounding resources is of uttermost importance in cyber-physical systems where memory, bandwidth and computational power are limited, and still the system must react properly and timely to the changing environment. If a monitor is trace-length independent it can run for indefinitely long even if the resources are physically constrained. A third contribution is a prototype implementation and an empirical evaluation. A fourth contribution is a modified algorithm that allows nodes to save bandwidth by only communicating stream values when requested.

D. STRUCTURE

Section IV contains the solution to to the timed asynchronous decentralized stream runtime verification problem as well as its proofs and the algorithm. Section V contains the analysis of resource usage based on the different conditions that affect it. The empirical evaluation is in Section VI. In Section VII, the modified version of the algorithm is presented. Section II contains the related work and Section III contains the preliminaries. Section VIII concludes.

II. RELATED WORK

The term *decentralized monitoring* is used in the survey [25] to distinguish the term from distributed monitoring where processes do not share a global clock. In distributed monitoring a complete asynchronous network is assumed, while typically decentralized monitoring assumes a completely synchronous network where all samples and communication occur in lockstep. In this paper we explore the middle ground: network nodes share a sufficiently synchronized global clock (like in synchronous distributed systems) but communication can take arbitrarily long (like in asynchronous distributed systems). Also, in [25] they present other concepts such as policy checking that are called decentralized monitoring that do not correspond to the monitoring presented in this paper, because they are concerned only about global safety properties that can be used for asynchronous networks with asynchronous computations.

In [27] they also study timed asynchronous networks of cooperating monitors but use an SMT-solver for simplifying LTL formulas. In [28] they use MTL for time asynchronous networks (bounded-skew clocks) to monitor blockchain executions using a formula rewriting scheme based on an SMT-solver.

Distributed stream processing has been largely studied. In [41] they use the concept of streams in Complex Event Processing, where events may be structured datatypes and where computation may be complex in the sense that several operations are needed for each event, for example in sliding window operations to make aggregate calculations on the arriving events. The aim of [41] is merging privacy and approximation techniques obtaining zero-knowledge privacy and low-latency and efficient analytics. In [11] Apache Flink is introduced where stream dataflows processing is used to handle continuous streams and batch processing.

Distributed and decentralized monitoring has been studied in the context of runtime verification. The work in [25] uses slices to support node crashes and message errors when monitoring distributed message passing systems with a global clock. Bauer et al. [7] introduce a first-order temporal logic and trace-length independent spawning automaton, and in [9] show a decentralized solution to monitor synchronous systems using formula rewriting. The language used is a three-valued variant of LTL with a central value that captures when an expression has an unknown value so far and we need to process more of the input trace to determine its truth value.

This is improved in [19], and [20] using a datastructure that stores the partially evaluated expressions by different monitors with their partial information that allow decentralized monitors to infer the state in which the monitoring automaton is in. In [21] the authors extend this data structure with distributed and multi-threaded support along with guaranteeing the determinism of the data structure by construction. Then they analyze the compatibility and monitorability of decentralized specifications in this setting. However, the verdicts and data are still Boolean and the network is assumed to be full synchronous. In [22] the authors use the same datastructure for a case study on the orange4Home dataset where they exploit hierarchies and modularity in decentralized specifications to reduce computation and bandwidth. In [32] global choreographies (as a kind of master-based protocol) are synthesized (including control flows, synchronization, notification, acknowledgment, computations embedding) to distributed systems. Also, they provide a transformation to Promela which allows to verify the implementation using LTL specifications. Some schemes that they showcase are a variant producer-consumer or two-phase commit and apply it to building micro-services such as a buying system. This work focuses on synthesizing the flow of monitors, but again (unlike in our solution) the observations and verdicts are Boolean. In [43] authors define a decentralized RV technique in an asynchronous computation model, based on timed regular expressions as the specification language. They consider only Boolean inputs and verdicts and a wandering specification gets updated as it travels through the monitors with the new information from inputs that the monitors store in memory. In [1] they use probabilistic traces to monitor in distributed systems where the input trace may lack some information, and obtain a probability that the monitor reaches certain states, because they cannot compute it precisely, though they need clock synchronization for informing of the presence of a lack of information. Monitors need to output without waiting for other monitors in order to avoid blocking the system.

In [34] a synchronous network of LTL-monitors cooperate to achieve a verdict on the system under test while they may suffer crashes. In this scenario an SMT-based algorithm for synthesizing the automata for the LTL-monitors is presented that achieves fault tolerance providing soundness even though crashed monitors never recover. Even though this work considers failures (which is out of the scope of our paper) they assume synchronous communication. In [10] decentralized monitoring in an asynchronous environment (network and computation) with crash-resilient monitors that may crash but then they do not recover for the duration of the execution. Sen et al. [47] introduced a variant of LTL logic for monitoring distributed systems, but they consider a complete asynchronous distributed system and they are limited to Boolean verdicts. In [12] totally asynchronous and fault-tolerant network and computing is considered for the problem of runtime verifying the linearization of wait-free concurrent algorithm implementations. In [36] monitoring signal temporal logic in a timed asynchronous network they encode the specification as an SMT problem using a central monitor for Boolean inputs. The tool Monpoly is introduced in [6] which implements a centralized monitoring algorithm for verifying policies expressed in metric first-order temporal logic. In [4] monitoring the Internet Computer with Monpoly they construct a log using a pre-processor that enables them to verify policies of this giant distributed systems using a centralized monitor. In [26] a benchmark to compare decentralized enforcement algorithms is presented. It implements different strategies (globally optimal, locally optimal) and effectively compares them by measuring bandwidth and computation used. In [44] runtime enforcement of message sequences is introduced, where nodes have a monitor that prevents sending messages that might produce unwanted message sequences. This is a limitation compared to asynchronous networks, where there is no possible way of ordering messages. Again, all these approaches consider only Boolean verdicts. In comparison, SRV can generate verdicts from arbitrary data domains.

All previous SRV efforts, from Lola [16], Lola2.0 [23], Copilot [38], [39], [40] and extensions to timed event streams, like TeSSLa [13], RTLola [24] or Striver [30] assume a centralized monitoring setting. In [29] the relationship between time-based (soft real time) and event-based models of computation and their effects on SRV are explored, but again in the centralized setting. The work in [5] shows how to monitor Metric Temporal Logic (MITL) specifications of distributed systems (including failures and message reordering) where the nodes communicate in a tree fashion and the root emits the final verdict. We extend our previous work in [17] to timed asynchronous networks.

The table in Fig. 1 classifies the related work according to whether the specifications emit Boolean verdicts (Logics vs Streams), according the whether the system is centralized or decentralized and the assumption on the network.

	centralized		decentralized		
	discrete	event	sync	t-async	async
Logics	[2], [3] [7], [8] [18], [31] [42], [46]	[4], [6] [36]	[9], [19] [20]–[22] [25], [34]	[1], [10] [27], [28]	[5], [12] [43], [47]
Streams	[16], [23] [38]–[40] [45]	[13], [24] [29], [30]	[17]	**This paper**	**Open problem**

FIGURE 1. RV works by specification language and computation-network model of the monitor(s).

III. PRELIMINARIES

We recall now SRV briefly. For a more detailed description see [16] and the tutorial [45]. The fundamental idea of SRV, pioneered by Lola [16] is to describe monitors declaratively via a set of equations that describe the dependencies between output streams of values and input streams of values. Different monitors can be generated to perform online monitoring (where the observations are events received incrementally) or offline (where there is a log produced during a past execution that can be traversed back and forth). We focus here on online monitoring.

A monitor is generated from a specification, which at runtime computes a sequence of values for the output streams as soon as possible after observing each value from input streams. Input values are typically extracted from some sensor or read from a log file.

A Lola specification declares output streams in relation to the input streams, including both future and past temporal dependencies. The Lola language cleanly separates the temporal dependencies from the individual operations to be performed at each step to compute the internal and output values of the monitors. For example, integers and Booleans verdicts require different operations but the temporal dependencies may be the same. This leads to a generalization of monitoring algorithms from temporal logics to the computation of richer values such as numbers, strings or richer data-types.

A. LOLA SYNTAX

A Lola specification consist of declaring the relation between output streams and input streams of events. Stream expressions are terms built using a collection of (interpreted) constructor symbols. Symbols are interpreted in the sense that each constructor is not only used to build terms, but it is also associated with an evaluation function, that given values of arguments produces a value of the return type. For example $+$ and $\vee$ are constructor symbols to build integer and Boolean expressions (resp.) and the interpretation is the usual: addition and disjunction.

Given a set Z of typed stream variables the set of *stream expressions* consists of (1) variables from Z, (2) offsets $v[k, d]$ where v is a stream variable of type D, k is a natural number and d a value from D, and (3) terms $f(t_1, \ldots, t_n)$ using constructor symbols f from the theories to previously defined terms. Stream variables represent sequences of values (streams) in the specification.

The intended meaning of an offset expression $v[-1, \mathit{false}]$ is the value of stream v in the previous position of the trace (or *false* if there is no such previous position, that is, at the beginning). We use $\mathit{Term}_D(Z)$ for the set of stream expressions of type D constructed from variables from Z (and drop Z if clear from the context). Given a term t, $\mathit{sub}(t)$ represents the set of sub-terms of t.

Definition 1 (Specification): A Lola specification $\varphi(I, O)$ consists of a set $I = \{r_1, \dots, r_m\}$ of input stream variables, a set $O = \{s_1, \dots, s_n\}$ of output stream variables, and a set of defining equations, $s_i = e_i(r_1, \dots, r_m, s_1, \dots, s_n)$ one per output variable $s_i \in O$. The term e_i is from $\mathit{Term}_D(I \cup O)$, where D is the type of s_i.

We will use $r, r_i, \dots$ to refer to input stream variables,; $s, s_i, \dots$ to refer to output stream variables; and $u, v, \dots$ for an arbitrary input or output stream variable. Given $\varphi(I, O)$ we use $\mathit{appears}(u)$ for the set of output streams s_i such that u appears in the defining equation of s_i, that is $\mathit{appears}(u) = \{s_i \mid u[-k, d] \in \mathit{sub}(e_i) \text{ or } u \in \mathit{sub}(e_i)\}$. Also, $\mathit{ground}(t)$ indicates whether expression t is a ground expression (contains no variables or offsets) and therefore can be evaluated into a value using the interpretations of constants and function symbols.

Example 2: The property "*sum the previous values in input stream* `y`*, but if the* `reset` *stream is true, reset the count*", can be expressed as follows, where stream variable `root` uses the accumulator `acc` and the input `reset` to compute the desired sum. The stream `acc` is defined with the keyword `define` to emphasize that it is an intermediate stream.

```
input bool reset
input num y
define int acc  = y + root[-1|0]
output int root = if reset then 0~else acc
```

B. LOLA SEMANTICS

We introduce now the formal semantics of Lola, that guarantees that there is a unique correct output stream for each input stream. This semantics is denotational and allows to prove that a monitoring algorithm is correct by showing that the algorithm produces the desired output. At runtime, input stream variables are associated incrementally with input streams of values.

Given an input streams σ_I (one sequence per input stream variable) and given an output candidate σ_O (one sequence per output stream) the formal semantics captures whether the pair (σ_I, σ_O) matches the specification, which we write $(\sigma_I, \sigma_O) \models \varphi$. We use σ_r for the stream in σ_I corresponding to input variable r and $\sigma_r(k)$ for the value of stream σ_r at position k. For $(\sigma_I, \sigma_O) \models \varphi$ to hold, all streams must be sequences of the same length.

A *valuation* of a specification φ is a pair $\sigma : (\sigma_I, \sigma_O)$ that contains one stream (of values of the appropriate type) and of the same length for each input and output stream variable in φ.

Given a term t, the *evaluation* $[\![t]\!]_\sigma$ is a sequence of values of the type of t defined as follows:

- If t is a stream variable u, then $[\![u]\!]_\sigma(j) = \sigma_u(j)$.
- If $f = f(t_1, \dots, t_k)$ then
$$[\![f(t_1, \dots, t_k)]\!]_\sigma(j) = f([\![t_1]\!]_\sigma(j), \dots, [\![t_k]\!]_\sigma(j)).$$
- Finally, if $t = v[i, c]$ is an offset then
  - $[\![v[i, c]]\!]_\sigma(j) = [\![v]\!]_\sigma(j + i)$ if $0 \leq j + i$, and
  - $[\![v[i, c]]\!]_\sigma(j) = c$ otherwise.

A valuation (σ_I, σ_O) satisfies a Lola specification φ whenever for every output variable s_i, $[\![s_i]\!]_{(\sigma_I,\sigma_O)} = [\![e_i]\!]_{(\sigma_I,\sigma_O)}$. In this case we say that σ is an evaluation model of φ and write $(\sigma_I, \sigma_O) \models \varphi$.

The intention of a specification φ is to describe a unique output from a given input, which is guaranteed if φ has no cycles in the following sense. A *dependency graph* D_φ of a specification $\varphi(I \cup O)$ is a weighted multi-graph (V, E) whose vertices are the stream variables $V = I \cup O$, and where E contains a directed weighted edge $u \xrightarrow{w} v$ whenever $v[w, d]$ is a sub-term in the defining equation of u. A specification φ is *well-formed* if D_φ contains no zero-weight cycles, which guarantees that no stream depends on itself at the current position.

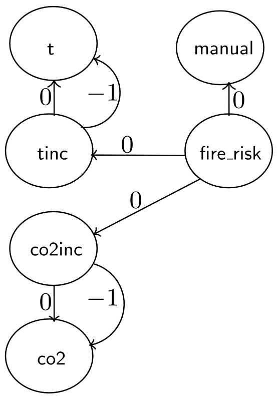


FIGURE 2. Dependency graph for Example 1.

Considering Example 1 its dependency graph is shown in Fig. 2. Given a stream variable u and position $i \geq 0$ an *instant stream variable* (or simply instant variable) is defined as the pair $u\langle i\rangle$, which is a fresh variable of the same type as u. Instant variable $u\langle i\rangle$ represents the value of the stream associated with u at time i. Note there is one different instant variable $u\langle i\rangle$ for each instant i. The *evaluation graph EG* is the unrolling expansion of the dependency graph for all instants. Given $\varphi(I, O)$ and a trace length M (or $M = \omega$ for infinite traces) the evaluation graph $G_{\varphi,M}$ has as vertices the set of instant variables $\{u\langle k\rangle\}$ for $u \in I \cup O$ and $0 \leq k < M$, and has edges $u\langle k\rangle \rightarrow v\langle k'\rangle$ if the dependency graph contains an edge $u \xrightarrow{j} v$ and $k + j = k'$

Consider again Example 1. The corresponding evaluation graph for trace length $M = 3$ is shown in Fig. 3. Note that $\mathit{tinc}\langle 2\rangle$ points to $t\langle 1\rangle$ in all evaluation graphs

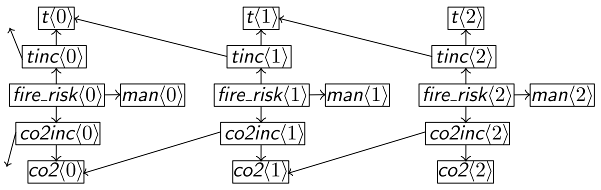


FIGURE 3. **Evaluation graph for Example 1.**

with $M \geq 2$. We denote by $e_s\langle k\rangle$ the term (whose leafs are instant variables) that results from e_s at k, by replacing the offset terms with the corresponding instant variables corrected with the appropriated shift. Consider again Example 1. The instant stream expression e_{tinc} for *tinc* at instant 2 is $tinc\langle 2\rangle = (t\langle 2\rangle - t\langle 1\rangle)/t\langle 1\rangle$.

Nodes of the dependency graph form a DAG of Maximal Strongly Connected Components (MSCCs). Specifications whose dependency graph has no positive cycles are called *efficiently monitorable specifications* [16]. There are no cycles in the evaluation graph of an efficiently monitorable specification, which enables us to reason by induction on evaluation graphs, as we will do later. Note that these specifications can have positive edges (corresponding to future dependencies) as long as they do not form a positive cycle. As it can be shown [45] these specifications can be evaluated online (incrementally) with finite memory with a central monitor.

Example 3: The following code snippet shows a non-efficiently monitorable, an efficiently monitorable specification and a very efficiently monitorable. The first snippet is a non-efficiently monitorable specification because the stream b depends on itself in the future, in the Evaluation Graph (EG) all instant variables will depend on the next instant unboundedly to the future. This will make all instant streams b to never be resolved in an infinite trace.

```
input int a
output int b = b[1|0]
```

Next specification is an efficiently monitorable specification because there are only bounded references to the future: each instant variable $b\langle k\rangle$ only depends on the instant variable a two positions ahead, so every instant variable $b\langle k\rangle$ it will be resolved at time $k + 2$.

```
input int a
output int b = a[2|0] + b[-1|0]
```

The following is a very efficiently monitorable specification because there are no reference to the future, all offsets are either negative or zero.

```
input int a
output int b = a + b[-1|0]
```

C. DECENTRALIZED SYNCHRONOUS ONLINE MONITORING

An online decentralized algorithm to monitor Lola specifications in a synchronous network is presented in [17]. The main idea is to use a network of cooperating nodes to monitor a Lola specification. The specification is sliced according to its syntax tree and each subexpression, including inputs, is mapped to a node (fixed at deployment time). Monitors share to other monitors their partial results (values of the instant variables of the subexpressions they control) via messages. At each time instant each monitor will read inputs, update its internal expressions and communicate results with the appropriate nodes so that the specification ends being computed by means of those partial results.

Therefore, given a well-formed Lola specification, the decentralized online algorithm presented in [17] incrementally computes the value for each output instant variable assuming a synchronous network where messages are not lost or duplicated. The algorithms presented here extend this solution to the more general setting of timed asynchronous networks.

IV. DECENTRALIZED STREAM RUNTIME VERIFICATION FOR TIMED ASYNCHRONOUS NETWORKS

In this section we describe our solution to decentralized monitoring of stream runtime verification for Timed Asynchronous Networks. Our algorithm computes the unique values of the output instant variables based on the values of the input readings. We prove the termination of the algorithm in Theorem 1 and its correctness in Theorem 2, verifying that the operational semantics are equivalent to the denotational semantics from Section III.

We start from a well-formed Lola specification and a static mapping between streams and the network nodes where they are computed. Each network node will host a local monitor that is responsible for computing some of the streams of the specification. We denote $\mu(s)$ for stream variable s is the network node whose local monitor is responsible for resolving the values of stream s. Local monitors exchange messages containing partial results whenever needed in order to compute the global monitoring task (the stream of values of the root stream of the specification) Our decentralized algorithm may compute some output values at different time instants than a centralized version, due to the different location of the inputs and the communication delays.

We study this effect both theoretically in Section V, and empirically in Section VI. A centralized monitor corresponds with the operational semantics in [16], and [45] which is equivalent with a network mapping that assigns all input and output streams to a single node and therefore avoids communication.

A. PROBLEM DESCRIPTION

1) NETWORK

We model the network by a set of nodes N that can communicate with each other sending messages. We assume reliable unicast communication (no message loss or duplication) over a timed asynchronous network, so a given message can take an arbitrary amount of time to arrive. Since network nodes share a global clock, the computation proceeds in cycles. In every cycle, all nodes in the network execute—in parallel and to completion—the following actions: (1) read input messages, (2) perform a terminating local computation, (3) generate output messages. We use the following type of message: $(s\langle k\rangle, c, n_s, n_d)$ where $s\langle k\rangle$ is an instant variable, c is a value of the type of s, n_s is the source node and n_d is the destination node. The interpretation of this message is that node n_s informs node n_d that the value of $s\langle k\rangle$ is c. We use the following abbreviations $msg.src = n_s$, $msg.dst = n_d$, $msg.stream = s\langle k\rangle$ and $msg.val = c$.

2) STREAM ASSIGNMENT AND COMMUNICATION STRATEGY

Given a specification $\varphi(I, O)$ and a network with nodes N, a *stream assignment* is a map $\mu : I \cup O \rightarrow N$ that assigns a network node to each stream variable. The node $\mu(r)$ for an input stream variable r is the location in the network where r is sensed in every clock tick. At runtime, the value of $r\langle k\rangle$ is read at node $\mu(r)$ at instant k. On the other hand, the node $\mu(s)$ for an output stream variable s is the network node responsible for resolving the values of s.

An instant value $v\langle k\rangle$ is automatically communicated to all potentially interested nodes whenever the value of $v\langle k\rangle$ is resolved. Let v and u be two stream variables such that v appears in the equation of u and let $n_v = \mu(v)$ and $n_u = \mu(u)$. Then, n_v informs n_u of every value $v\langle k\rangle = c$ that n_v resolves by sending a message $(v\langle k\rangle, c, n_v, n_u)$. We are finally ready to define the decentralized SRV problem.

Definition 2: A decentralized SRV problem $\langle \varphi, N, \mu\rangle$ is characterized by a specification φ, a network with notes N and a stream assignment μ for every stream variable.

We use DSRV to refer to the decentralized SRV problem. Solving a DSRV instance consists of computing the values of the instant variables corresponding to the output streams (based on the values of the instant variables of the input streams).

B. MODEL OF COMMUNICATION

We now describe in detail the timed asynchronous model of computation. Every message inserted in the network arrive at its destination according to the following conditions:

- *Always later*: every message m inserted at t will arrive at t' with $t' > t$;
- *Arbitrary delay*: there is no a-priori bound on the amount of time that any message will take to arrive.
- *FIFO between each pair of nodes*: let m_1 and m_2 be two messages with the same origin and destination, $m_1.src = m_2.src$ and $m_1.dst = m_2.dst$. Let m_1 is inserted at t_1 and arrive at t_1' and let m_2 be inserted at t_2 and arrive at t_2'. If $t_1 < t_1$, then $t_1' \leq t_2'$. That is, m_1 cannot arrive later than m_2.

The synchronous model is a particular case of the timed asynchronous model in which all messages inserted in the network will always take the same amount of time between each pair of network nodes. In this case the delay will always be a constant. Formally, to analyze the behavior of our algorithms we model the message delays as a family of functions $arr_{u\rightarrow v}$ (one for each pair of nodes u and v, which provides at every moment t the instant t' at which a message sent at t from u will arrive at v.

C. DSRV FOR TIMED ASYNCHRONOUS NETWORKS: MONITOR AND ALGORITHM

Our solution consists of a collection of local monitors, one for each network node n. A local monitor $\langle Q_n, U_n, R_n\rangle$ for n maintains an input queue Q_n and two storages:

- **Resolved** storage R_n, where n stores resolved instant variables $(v\langle k\rangle, c)$.
- **Unresolved** storage U_n, where n stores unresolved equations $v\langle k\rangle = e$ where e is not a value, but an expression that contains other instant variables.

When n receives a message from a remote node, the information is added to R_n, so future local requests for the information can be resolved locally and immediately. At the beginning of the cycle of computation at instant k, node n reads the values for input streams assigned to using local sensors and instantiates for k all output stream variables that n is responsible for. After that, the equations obtained are simplified using the knowledge acquired so far by n, which is stored in R_n. Finally, new messages are generated and inserted in the queues of the corresponding neighbors.

More concretely, every node n will execute the procedure Monitor shown in Algorithm 1, which invokes Step in every clock tick. The procedure Finalize is used to resolve the pending values at the end of the trace to their default. Note that this procedure is never invoked if the monitor trace never terminates (the monitor will be continuously observing and producing outputs). The procedure Step executes the following steps:

1) **Process Messages**: Lines 7 invokes ProcessMessages procedure in lines 23-25 that deals with the processing of incoming messages, adding them to R_n
2) **Read Inputs and Instantiate Outputs:** Line 8 reads new inputs for current time k, and line 9 instantiates the equation of every output stream that n is responsible for.
3) **Evaluate:** Line 10 invokes the procedure Evaluate, in lines $14 - 22$ which evaluates the unresolved equations.
4) **Send Responses:** Line 12 invokes SendResponses, in lines 26-28, sending messages for all newly resolved variables.
5) **Prune:** Line 29-31 prunes the set R from information that is no longer needed. See section V-A5.

D. FORMAL CORRECTNESS

We now show that our solution is correct by proving that the output computed is the same as in the denotational semantics, and that every output is eventually computed.

Theorem 1: All of the following hold for every instant variable $u\langle k\rangle$:

Algorithm 1 Local monitor at node n with $\langle Q_n, U_n, R_n\rangle$

```
1:  procedure Monitor
2:      $Q_n \leftarrow \emptyset$; $U_n \leftarrow \emptyset$; $R_n \leftarrow \emptyset$; $k \leftarrow$ Now()
3:      while not END do Step($k$)
4:      $M \leftarrow k$; Finalize($M$)
5:  procedure Step($k$)
6:      $R_{old} \leftarrow MS_n.R_n$
7:      ProcessMessages($MS_n$)
8:      $R_n.add(\{r\langle k\rangle \mapsto read(r,k) \mid r \in ins_n\})$
9:      $U_n.add(\{s\langle k\rangle \mapsto e_s\langle k\rangle \quad \mid s \in outs_n\})$
10:     Evaluate($MS_n$)
11:     $R_{new} \leftarrow MS.R_n \setminus R_{old}$
12:     SendResponses($MS_n$)
13:     Prune($MS_n$)
14: procedure Evaluate($MS_n$)
15:     $done \leftarrow false$
16:     while $not\ \ done$ do
17:         $done \leftarrow true$
18:         for all $s\langle k\rangle \mapsto e \in U_n$ do
19:             $e' \leftarrow$ Subst($e, R_n$)
20:             if $ground(e')$ then $done \leftarrow false$
21:                 $U_n.del(s\langle k\rangle \mapsto e)$; $R_n.add(s\langle k\rangle \mapsto e')$
22:             else $U_n.del(s\langle k\rangle \mapsto e)$; $U_n.add(s\langle k\rangle \mapsto e')$
23: procedure ProcessMessages($MS_n$)
24:     for all $msg = \langle$resp$, s\langle k\rangle, c\rangle \leftarrow Q_n.pop()$ do
25:         $R_n.add(s\langle k\rangle \mapsto c)$
26: procedure SendResponses($MS_n, R_{new}$)
27:     for all $u\langle k\rangle \mapsto c \in R_{new}$ do
28:         $send($resp$, s\langle k\rangle, c, n, n_r)$
29: procedure Prune($MS_n, R_{new}$)
30:     for all $u\langle j\rangle \mapsto c$ s.t. $now \geq MTR(u\langle j\rangle)$ do
31:         $R_n.del(u\langle j_i\rangle \mapsto c_i)\}$                ▷ Remove
```

(1) The value of $u\langle k\rangle$ is eventually resolved.
(2) The value of $u\langle k\rangle$ is c if and only if $(u\langle k\rangle, c) \in R$ at some instant.
(3) A response message for $u\langle k\rangle$ is eventually sent to all interested network nodes (all nodes responsible for streams v where $u \in appears(v)$).

Proof: The proof proceeds by induction on the evaluation graph, showing simultaneously in the induction step (1)-(3) as these depend on each other in the previous inductive steps. Let M be a length of a computation (which can be infinite, that is $M = \omega$) and σ_I be an input of length M. Note that (1) to (3) above are all statements about instant variables $u\langle k\rangle$, which are the nodes of the evaluation graph $G_{\varphi,M}$. We proceed by induction on $G_{\varphi,M}$ (which is acyclic because D_φ is well-formed, by assumption).

- **Base case**: The base case are vertices of the evaluation graph that have no outgoing edges, which are either
  - instant variables that correspond to inputs read from local sensors
  - defined variables whose instant equation does not contain other instant variables; This is the case when either the equation is a constant or the time instant is such that the resulting offset falls off the trace; the default value is used.

  Statement (1) follows immediately for inputs because at instant k, $u\langle k\rangle$ is read at node $\mu(u)$. For output equations that do not have variables, or whose variables have offsets that once instantiated become negative or greater than M, the value of its leafs is determined either immediately or at M when the offset is calculated. At this point, the value computed is inserted in R, so (2) also holds at $\mu(u)$. Note that (2) also holds for other nodes because the response message contains $u\langle k\rangle = c$ if and only if $(u\langle k\rangle, c) \in R_n$, where $\mu(u) = n$. Then the response message is inserted exactly at the point it is resolved, so (1) implies (3).
- **Inductive case**: Consider an arbitrary $u\langle k\rangle$ in the evaluation graph $G_{\varphi,M}$ and let $u_1\langle k_1\rangle, \ldots, u_l\langle k_l\rangle$ be the instant variables that $u\langle k\rangle$ depends on. These are nodes in $G_{\varphi,M}$ that are lower than $u\langle k\rangle$ so the inductive hypothesis applies, and (1)-(3) hold for these. Let $n = \mu(u)$. At instant k, $u\langle k\rangle$ is instantiated and inserted in U_n. The values of instant variables are calculated and sent as well (by (1) and (3)). At the latest time of arrival, the equation for $u\langle k\rangle$ has no more variables and it is evaluated to a value, so (1) holds and (2) holds at n. At this point, the response message is sent (so (1) holds for $u\langle k\rangle$) and so (1) also holds.

This finishes the proof.

Theorem 1 implies that every value of every instant is eventually resolved by our network of cooperating monitors. Therefore, given input streams σ_I, the algorithm computes (by (2)) the unique output streams σ_i one for each s_i. The element $\sigma_i(k)$ is the value resolved for $s_i\langle k\rangle$ by the local monitor for $\mu(s_i)$. The following theorem captures that Algorithm 1 computes the right values (according to the denotational semantics of Lola), Theorem 1 that all values are eventually computed.

We use $out(\sigma_I)$ as the function from input streams to output streams that the cooperating monitors compute. We use $[s]$ for the stream of values corresponding to stream variable s in $out(\sigma_I)$. We now show that the sequence of values computed corresponds to the semantics of the specification.

Theorem 2: Let φ be a specification, $S = \langle \varphi, \mathcal{T}, \mu\rangle$ be a decentralized SRV problem, and σ_I an input stream of values. Then $(\sigma_I, out(\sigma_I)) \models \varphi$.

Proof: Let σ_O be the unique evaluation model such that $(\sigma_I, \sigma_O) \models \varphi$ (we use $\sigma_O(s)$ for the output stream for stream variable s and $\sigma_O(s)(k)$ for its value in the k-th position). We need to show that for every s and k, $[s](k) = \sigma_O(s)(k)$. We again proceed by induction on the evaluation graph $G_{\varphi,M}$.

- **Base case:** For inputs the value follows immediately. The other basic case corresponds to output variables s at instants at which these that do not depend on other variables (because all occurrences of offsets, if any, fall off the trace). The evaluation of the value is performed by network node $\mu(s)$, and it satisfies the equation e_s of s, not depending on any value of any other stream. Therefore, it satisfies that $[s](k) = [\![e_s[k]]\!] = \sigma_O(s)(k)$, as desired

- **Inductive case:** Let s be an arbitrary stream variable and k an arbitrary instant within 0 and $M-1$ and assume that all instant variables $u\langle k'\rangle$ that $s\langle k\rangle$ can reach in the evaluation graph satisfy the inductive hypothesis. Let n be the node in charge of computing s. By Theorem 1, all the values are eventually received by n and in R_n, and by IH, these values are the same as in the denotational semantics, that is $[u](k') = \sigma_O(u)(k')$. The evaluation of $s\langle k\rangle$ corresponds to computing $[\![e_s]\!]$, which uses the semantics of the expression (according to Section III). A simple structural induction on the expression e_s shows that the result of the evaluation, that is the value assigned to $s\langle k\rangle$, is $[\![e_s]\!]_\sigma(k) = \sigma_O(s)(k)$, as desired.

This finishes the proof.

E. SIMPLIFIERS

The evaluation of expressions in Algorithm 1 assumes that all instant variables in an expression e are known (i.e., e is ground), so the interpreted functions in the data theory can evaluate e. Sometimes, expressions can be partially evaluated (or even the value fully determined) knowing only some but not all of the instant variables involved in the expression. As simplifier is a function $f : Term_D \rightarrow Term_D$ such that (1) the variables in $f(t)$ are a subset of the variables in t, and (2) every substitution of values for the variables of t produces the same value as the substitution of $f(t)$. For example, the following are typical simplifiers:

$$\begin{aligned}
&\mathit{if}\ \ \mathit{true}\ \ \mathit{then}\ \ t_1\ \ \mathit{else}\ \ t_2 \mapsto t_1\\
&\mathit{if}\ \ \mathit{false}\ \ \mathit{then}\ \ t_1\ \ \mathit{else}\ \ t_2 \mapsto t_2\\
&\mathit{true}\ \vee x \mapsto \mathit{true}\\
&\mathit{true}\ \wedge x \mapsto x\\
&0 \cdot x \mapsto 0
\end{aligned}$$

In practice, simplifiers can dramatically affect the performance in terms of the instant at which an instant variable is resolved and, in the case of decentralized monitoring, the delays and number of messages exchanged. Essentially, a simplifier is a function from terms to terms such that, for every possible valuation of the variables in the original term it does not change the final value obtained. It is easy to see that for every term t obtained by instantiating a defining equation and for every simplifier f, $[\![t]\!]_{\sigma_I,\sigma_O} = [\![f(t)]\!]_{(\sigma_I,\sigma_O)}$, because the values of the variables in t and in $f(t)$ are filled with the same values (taken from σ_I and σ_O).

Consider arbitrary simplifiers *simp* used in line 19 of Algorithm 1 to simplify expressions. Let U_n be the unresolved storage for node n and let $u\langle k\rangle$ be an instant variable with $\mu(u) = n$. By Algorithm 1 the sequence of terms $(u\langle k\rangle, t_0), (u\langle k\rangle, t_1), \ldots (u\langle k\rangle, t_k)$ that U_n will store are such that each t_i will have the simplifier applied. It follows that the value computed using simplifiers is the same as without simplifiers. It is also easy to show that the algorithm using simplifiers obtains the value of every instant variable no later than the algorithm that uses no simplifier. This is because in the worst case every instant variable is resolved when all the instant variables it depends on are known, and all response messages are sent at the moment they are resolved.

V. COMPLEXITY ANALYSIS

In this section we analyze the resource complexity of our algorithm and define conditions under which local monitors only need bounded resources to compute every output value. We first analyze memory complexity and find that it affects computational time complexity. Thus, the bounds on memory complexity(under certain conditions) allow time complexity to be also bounded. Finally, we also analyze message complexity.

The first thing to consider is that the specification must be *decentralized efficiently monitorable* [17], which essentially states that every strongly connected component in G_φ must be mapped to the same network node. That is, if u appears, transitively, in the declaration of v and v appears in the declaration of u (with some offsets), then $\mu(u) = \mu(v)$.

A. MEMORY COMPLEXITY ANALYSIS

In order to guarantee that a given storage in a local monitor for node n is bounded, we must provide an upper-bound for how long it takes to resolve an instant variable for a stream that is assigned to n. We use Time to Resolve (*TTR*) to refer to the amount of time that a given instant variable $u\langle k\rangle$ takes to get resolved. This is the number of time instants between the instantiation of the instant variable at time k and the instant at which $u\langle k\rangle$ gets resolved, leaving U_n and being stored in R_n. This happens in line 21 in Algorithm 1.

1) GENERAL EQUATIONS FOR THE TIME TO RESOLVE

We introduce now a general definition of recursive equations that capture when an instant variable $s\langle k\rangle$ is resolved. In order to bound the memory used by the monitor at network node n, we need to bound storages U_n and R_n:

- **Bound on R_n:** Resolved values that are needed remotely are sent immediately to the remote nodes, so R_n only contains resolved values that are needed in the future locally at n. Since efficiently monitorable specifications only contain (future) bounded paths there is a maximum future reference b used in the specification. This upper-bound limits for how long a resolved value $v\langle k\rangle$ can remain in R_n, because after at most b steps the instant variables $u\langle k'\rangle$ that need the value of $v\langle k\rangle$ stored in R_n will be instantiated (note that $k' - k \leq b$).
  That is $u\langle k\rangle$ is not needed after $t = max(k + b, k + TTR(u\langle k\rangle))$. At t, the value of $u\langle k\rangle$ can be removed from R_n. This guarantees that the size of R_n is always upped-bounded by a constant in every node n.

- **Bound on U_n**: The size of the memory required for storage U_n at the node n responsible to resolve s (that is $n = \mu(s)$) is proportional to the number of instantiated but unresolved instant variables. Therefore, to bound U_n we need to compute the bound on the time it takes to resolve instant variables of streams assigned to n.

The general equations that we present below depend on the delay of messages in the network. We will later instantiate these general equations for the following particular cases of network behavior:

- a synchronous network;
- a timed-asynchronous network with an upper-bound on message delays during the whole trace (we call this the **aeternal** case);
- timed-asynchronous network with an upper-bound for message delays in a given time-horizon (we call this the **temporary** case).

Note that the correctness of the algorithm (Theorem 2) establishes that the output streams σ_O only depend on the input streams σ_I but does not state bounds on the time at which each element of σ_O is resolved or on the delays of messages.

In this section we study how the delay of messages affects the time at which instant variables are resolved, which in turn affects the memory usage at the computations nodes. We use $d(t, a, b)$ for the time it takes for a message sent from a to b at time t to arrive. In other words $arr_{a\to b}(t) = t + d(t, a, b)$. Recall that we assume that messages are causal and queues are FIFO as we described in IV-B. Causality means that messages arrive after they are sent (that is, for every n, m and t, $arr_{a\to b}(t) > t$) and FIFO that for every n and m, if $t < t'$ then $arr_{a\to b}(t) \leq arr_{a\to b}(t')$.

We now capture the *Moment to Resolve* for a given instant variable $s\langle t\rangle$, represented as $MTR(s\langle t\rangle)$, which captures the instant of time at which $s\langle t\rangle$ is guaranteed to be resolved by the monitor at network node $\mu(n)$ responsible to compute s. Our definition considers two components, the delay in resolving all local instant variables that $s\langle t\rangle$ may depend on and the resolution of remote instant variables, which also involve message delays. We use the concept of *remote moment to resolve*, denoted $MTR_{rem}(s\langle t\rangle)$, as the instant at which all remote values that $s\langle t\rangle$ directly require have arrived (which is t if all values arrive before t).

$$\begin{aligned}
MTR(s\langle t\rangle) &\overset{\text{def}}{=} \max(\\
&\quad MTR_{rem}(s\langle t\rangle), \{MTR(r\langle t+w\rangle) \mid s \xrightarrow[loc]{w} r\}) \\
MTR_{rem}(s\langle t\rangle) &\overset{\text{def}}{=} \max(t, \\
&\quad \{arr_{r\to s}(MTR(r\langle t+w\rangle)) \mid s \xrightarrow[rem]{w} r \text{ and } t+w \geq 0\})
\end{aligned}$$

Note that this is well-defined for every well-formed specification because the evaluation graph is acyclic, and the equation for $s\langle t\rangle$ only depends on those variables lower in the evaluation graph, which is acyclic.

Example 4: Consider Example 2 with streams i and acc at network node 1 and streams $reset$ and $root$ computed at network node 2. Then, we can substitute in the equations to obtain the $MTR(root\langle 1\rangle)$.

$$\begin{aligned}
&MTR(root\langle 1\rangle) \\
&\ = \max(MTR(reset\langle 1\rangle), MTR_{rem}(acc\langle 1\rangle)) \\
&\ = \max(1, \max(1, arr_{acc\to root}(MTR(acc\langle 1\rangle)))) \\
&\ = \max(1, arr_{acc\to root}(\max(1, \\
&\qquad MTR(i\langle 1\rangle), MTR_{rem}(root\langle 0\rangle)))) \\
&\ = \max(1, arr_{acc\to root}(\max(1, \\
&\qquad \max(MTR(reset\langle 0\rangle) MTR_{rem}(acc\langle 0\rangle))))) \\
&\ = \max(1, arr_{acc\to root}(\max(1, \max(0, \\
&\qquad \max(0, arr_{acc\to root}(MTR(acc\langle 0\rangle))))))) \\
&\ = \max(1, arr_{acc\to root}(\max(1, arr_{acc\to root}(\max(0, \\
&\qquad MTR(i\langle 0\rangle), MTR_{rem}(root\langle -1\rangle)))))) \\
&\ = \max(1, arr_{acc\to root}(\max(1, arr_{acc\to root}(0))))
\end{aligned}$$

The instant variable $root\langle 1\rangle$ is guaranteed to be resolved when the response from the instant variable $acc\langle 1\rangle$ arrives—that is, the $\max(1, arr_{acc\to root}(\ldots))$ part. And this response can only be produced when the response for $acc\langle 0\rangle$ arrives, which is the innermost part: $\cdots \max(1, arr_{acc\to root}(0))$ Note that we do not need to account for $MTR_{rem}(root\langle -1\rangle)$ since it is resolved instantaneously to its default value. Likewise, the inputs are also resolved instantaneously and do not add any delay when obtaining the value of the MTR.

For $MTR_{rem}(s\langle t\rangle)$, we only consider those remote instant variables for which $t + w \geq 0$ because otherwise the default value will be used at the moment of instantiating $s\langle t\rangle$. In the equation for $MTR(s\langle t\rangle)$ we assume the base case $MTR(s\langle t\rangle) = 0$ when $t < 0$, because again, the default value in the offset expression is used instead, which is known immediately. It is easy to see that the first equation is equivalent to:

$$MTR(s\langle t\rangle) \overset{\text{def}}{=} \max(\{MTR_{rem}(r\langle t+w\rangle) \mid s \xrightarrow[loc]{w}{}^{*} r\})$$

We are now ready to prove that these definitions indeed capture the time at which $s\langle t\rangle$ is resolved.

Theorem 3: Let φ be a specification and μ a network placement, let σ_I be the input trace and arr a network behavior. Every $s\langle t\rangle$ is resolved at $MTR(s\langle t\rangle)$ or before.

Proof: The proof proceeds by induction on the evaluation graph $G_{\varphi,M}$ induced by φ and the length of σ_I.

- **Base case**: inputs and instant variables $s\langle t\rangle$ that do not depend on any other instant variables. These are the nodes of EG that do not have any outgoing edge. Since $s\langle t\rangle$ is instantiated at t, then the value is resolved exactly at t either by reading a sensor or instancing to a default value. Also, $MTR(s\langle t\rangle) = MTR_{rem}(s\langle t\rangle) = t$.
- **General case**. Let $s\langle t\rangle$ be an arbitrary instant variable and assume, by inductive hypothesis, that the theorem holds for all instant variables lower in the EG than $s\langle t\rangle$. At time $MTR_{rem}(s\langle t\rangle)$ all instant variables $r\langle t+w\rangle$ from remote nodes that $s\langle t\rangle$ depends on have arrived because $r\langle t + w\rangle$ will be resolved at $MTR(r\langle t\rangle)$ by

induction hypothesis. Similarly, all local elements that $s\langle t\rangle$ depends on are also below in the dependency graph, so the induction hypothesis also applies. Therefore, at time

$$\max(MTR_{rem}(s\langle t\rangle), \{MTR(r\langle t+w\rangle) \mid s \xrightarrow[loc]{w} r\}\}$$

or before all elements that $s\langle t\rangle$ depends on will be known and $s\langle t\rangle$ will be resolved.

This finishes the proof.

The following corollary follows from the fact, after an instant variable has been resolved, nothing can affect the value computed. Therefore, the value and time at which $s\langle t\rangle$ is computed does not depend on the future after $MTR(s\langle t\rangle)$.

Corollary 1: For all $s\langle t\rangle$ there is a t' such that $s\langle t\rangle$ only depends on σ_I and arr up to t'.

The MTR for an instant variable depends on the delay of messages $arr_{u\to v}$ between the network nodes that cooperate in order to compute that instant variable. Therefore we cannot guarantee a bound on MTR if those delays can be arbitrarily long, so we cannot bound the memory usage. Consequently, monitoring is not trace-length independent in a general Time Asynchronous Network.

Next, we study how different conditions on the network behavior (concerning the delays between links) affect the MTR establishing memory bounds and regain trace-length independent monitoring under those conditions.

2) INSTANTIATION TO SYNCHRONOUS TIME

In this sub-section we assume the synchronous model of computation, which is a particular case of the timed-asynchronous model where all message delays between two monitors take exactly the same amount of time throughout the trace. We use the predicate $dist_{r\_s}$ to represent the delay that every message will take from $\mu(r)$ to $\mu(s)$, independently of the time instant at which the message is sent. Therefore $arr_{r\to s}(t) = t + dist_{r\_s}$. This delay allows us to simplify MTR_{rem} for synchronous networks as follows:

$$MTR^{sync}_{rem}(s\langle t\rangle) \overset{\text{def}}{=} \max(t, M(s\langle t\rangle))$$

where

$$M(s\langle t\rangle) = \{MTR^{sync}(r\langle t+w\rangle) + dist_{r\_s} \mid s \xrightarrow[rem]{w} r, t+w \geq 0\})$$

Recall that the *time to resolve* is the time interval between the moment at which a variable is instantiated and the instant at which it is resolved (that is $TTR(s\langle t\rangle) = MTR(s\langle t\rangle) - t$) In the synchronous case we obtain:

$$\begin{aligned}
&TTR^{sync}(s\langle t\rangle)\\
&\quad = MTR^{sync}(s\langle t\rangle) - t\\
&\quad = \{MTR^{sync}_{rem}(r\langle t+w\rangle) \mid s \xrightarrow[loc]{w} r\} - t =\\
&\quad = \max(t, M(s\langle t\rangle) - t\\
&\quad = \max(0, \{TTR^{sync}(r\langle t+w\rangle) + dist_{r\_s} \mid s \xrightarrow[rem]{w} r\})
\end{aligned}$$

Note that the value that determines the result is the $TTR^{sync}(s\langle t\rangle)$ of the slowest remote dependency, which includes both the resolve time and the time the message needs to traverse through the network. Additionally, we can easily show by induction on the dependency graph that for every stream variable s there is a constant k such that $TTR^{sync}(s\langle t\rangle) \leq k$, that is, s always takes less than k instants to be resolved. It follows that all decentralized efficiently monitorable specifications can be monitored in constant space in every local monitor, that is, synchronous decentralized monitoring of decentralized efficient monitorable specifications is trace length independent.

3) TIMED ASYNCHRONOUS WITH **AETERNALLY** BOUNDED DELAYS

We now assume that there is a global upper bound on the delay time for every message, which we call **aeternally** bounded delays. Formally, this assumption states that if there is a d such that for every pair of streams r, s and for every time t, $arr_{r\to s}(t) \leq t + d$. Substituting the upper-bound value d in the equations for MTR, we obtain an constant upper-bound on the MTR:

$$MTR^{\mathbf{g}}_{rem}(s\langle t\rangle) \leq \max(t, M(s\langle t\rangle))$$

where

$$M(s\langle t\rangle) = MTR^{\mathbf{g}}(r\langle t+w\rangle) + d \mid s \xrightarrow[rem]{w} r, t+w \geq 0\}$$

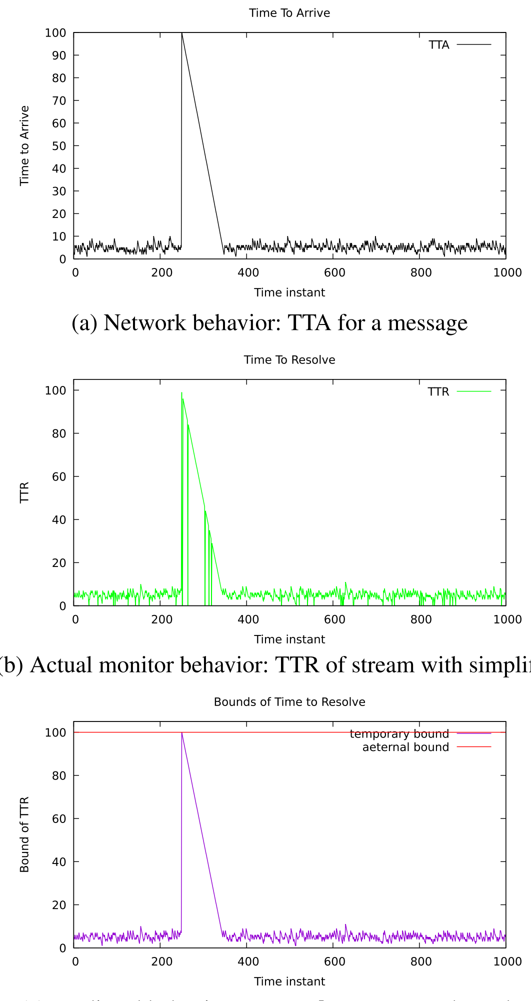
Note that in some cases $s\langle t\rangle$ can be resolved before $MTR^{\mathbf{g}}(s\langle t\rangle)$ because d is an upper bound. In this case we can also bound the memory necessary to store in every node to perform the monitoring process, but most of the time less memory will be necessary. We can see an example of a **aeternal** bound in Figure 4.

4) TIMED ASYNCHRONOUS WITH **TEMPORARILY** BOUNDED DELAYS

We now take a closer look at the equations to obtain a better bound on the time to resolve a given instant variable $s\langle t\rangle$, without assuming an upper-bound of all messages in the history of the computation, but only the necessary messages that can influence $s\langle t\rangle$. The main idea to bound $MTR(s\langle t\rangle)$ is to consider the time interval at which the messages that are relevant to compute $s\langle t\rangle$ are sent. We first define an auxiliary notion. We say that a stream variable r is a direct remote influence on s with delay w, and we write $s \xrightarrow[drem]{w} r$, whenever there is a path $s \xrightarrow[loc]{w_1} s_1 \xrightarrow[loc]{w_2} s_2 \ldots \xrightarrow[loc]{w_k} s_k \xrightarrow[rem]{w_{k+1}} r$ such that:

- no two nodes s_i and s_j are repeated (if $i \neq j$ then $s_i \neq s_j$), and
- $w = w_1 + \ldots + w_k + w_{k+1}$.

Note that $s \xrightarrow[drem]{w} r$ means that $s\langle t\rangle$ may be influenced by remote variable $r\langle t+w\rangle$. We define the window of interest



(a) Network behavior: TTA for a message

(b) Actual monitor behavior: TTR of stream with simplifiers

(c) Predicted behavior: **aeternal**, **temporary** bounds

FIGURE 4. TTA, aeternal and temporary bounds for a normalPeak network behavior.

for $s\langle t\rangle$ as:

$win(s\langle t\rangle) = [\min S, \max S]$ where S is defined as

$$S = \{t, MTR_{rem}(r\langle t+w\rangle) \mid s \xrightarrow[drem]{w} r \text{ and } t+w > 0\}$$

Note that S is the set of instants at which remote instant variables that influence $s\langle t\rangle$ are sent.

Example 5: Considering the specification in Example 2. Taking a look at the evaluation graph in Figure 3 we observe that the window of interest of all instant variables at any time includes those of its dependencies in the evaluation graph. Therefore, their window of interest will include the minimum time for the earliest dependency to be resolved and the maximum time for the last dependency to be resolved. In this example, the window for $root\langle 1\rangle$ will include the windows for $acc\langle 1\rangle$, $root\langle 0\rangle$ and $acc\langle 0\rangle$ and the time required for the response messages to travel from source to destination. Note that inputs do not affect the *MTR*.

Therefore $win(s\langle t\rangle)$ contains those instants at which the remote information relevant to $s\langle t\rangle$ is sent. This window always ends at most at $MTR(s\langle t\rangle)$. We then define the worst message sent to s for the computation of $s\langle t\rangle$ as:

$$d_{worst}(s\langle t\rangle) = \max\{t' - t \mid t' = arr_{r\to s}(t) \text{ for } s \xrightarrow[drem]{w} r \text{ and } t \in win(s\langle t\rangle)\}.$$

Note that d_{worst} is still an over-approximation of the messages sent in order to compute $s\langle t\rangle$ but in this case the bound considers all those messages and only looks at a bounded interval of time. Since all the values that influence $s\langle t\rangle$ are sent within $win(s\langle t\rangle)$ we can bound $MTR(s\langle t\rangle)$ as follows:

$$MTR^{temp}_{rem}(s\langle t\rangle) \leq \max(t, M(s\langle t\rangle))$$

where

$$M(s\langle t\rangle) = \{MTR(r\langle t+w\rangle) + d_{worst}(win(s\langle t\rangle)) \mid s \xrightarrow[drem]{w} r\}.$$

We have finally arrived at the desired outcome: a finite window of time that contains the sending and receiving of the relevant messages for the computation of a given instant variable. This implies that only a finite number of network delays affect the resolution of any instant variable $s\langle t\rangle$. As we can always find the maximum delay in the window, we can upper bound the time that it will take for any instant variable to be resolved, and we are able to know how much time these instant variables are stored in U_n and R_n. In turn, this allows to determine when certain instant variables are no longer needed and when they can be pruned releasing the used memory.

Fig. 4 shows the required time to resolve the stream (in Fig. 4(b)) and for the bounds(in Fig. 4(c)) to adapt to the network behavior (in Fig. 4(a)) that sets a baseline. This is extracted from Example 1 execution over a normalPeak network behavior. In detail:

- (a) the normalPeak network behavior defined as follows: message delays follow a normal distribution until time instant 250, where a failure of 100 time units happen, we see this as the peak in *TTR* at 250. Then, as the FIFO network condition states, all messages will be delayed until this one is delivered, so the *MTR* of the subsequent messages will be at the same time point (350) so all messages sent from 250 to 350 will arrive at 350. This is the reason that we see the *TTR* dropping linearly from 250 to 350, when the failure in the network has been corrected and the delays get back to follow the normal distribution.
- (b) the *TTR* of the stream adapts accordingly but can resolve faster when a simplifier can be used, this is when the manual alarm is used. As this stream is an input in the building monitor, it does not need to wait for the network communication and thus can resolve the fire_risk stream instantly. These are the cases when the curve drops to 0. In those cases the network is not needed and the stream can be resolved without any delay.
- (c) We can observe the difference between the **temporary** and **aeternal** bounds, where the **aeternal** bound is high and constant throughout the execution and the **temporary** is tighter to the actual *TTR* giving a much better approximation.

5) PRUNING THE RESOLVED STORAGE R_n

We are finally ready to prune R_n because we know now when every instant variable will be resolved.

Corollary 2: Every unresolved instant variable $s\langle t\rangle$ in U_n is resolved at most at $MTR(s\langle t\rangle)$.

As soon as $MTR(s\langle t\rangle$ is reached (or before), the value of $s\langle t\rangle$ will be known in the local monitor of $\mu(s)$ and its value will be sent to those remote monitors where it is needed. After this moment $s\langle t\rangle$ can be pruned from U_n. With this mechanism, we can assure that every instant variable will be in memory (U_n or R_n) for a bounded amount of time.

Corollary 2 implies that decentralized efficiently monitorable specifications in timed asynchronous networks can be monitored with bounded resources when there is a certain bound on the network behavior, be it synchronous, **aeternal** or a **temporary** bound. This memory bound depends only linearly on the size of the specification and the delays between the nodes of the network. This results can be interpreted from the opposite perspective: given a fixed amount of memory available, we could calculate the maximum delays in the network that would allow the monitoring to be performed correctly.

B. TIME COMPLEXITY ANALYSIS

The Time complexity is dependent on the memory complexity, namely on the size of U.

In Algorithm 1, we observe that all procedures in the algorithm perform actions based on the number of elements in U, and we know that it is bounded. In procedure Evaluate, the worst case is when only one instant variable is resolved at each iteration of the inner loop, resulting in another loop in the while. This results in a linear complexity of $O(|U|)$. The rest of the procedures present a single loop over a bounded datastructure, we observe that in the worst case they perform in constant time $O(1)$.

C. MESSAGE COMPLEXITY ANALYSIS

Here we analyze the total number of messages amortized. For each newly resolved instant variable, we will have a new message traversing the network.

Proposition 1: Let φ be a specification with s streams and N be the trace length, then there will be at most $s{*}N$ messages during the monitoring.

Proof: As we have s streams and a tracelength of N, then we have $s * N$ instant variables by instancing the streams at each time point. By procedure SendResponses in line 12 of Algorithm 1 we observe that line 28 will be invoked once per instant variable. Also, the worst case(when most messages are needed) arises when each stream is mapped to a different node. In that case we have up to $s * N$ messages.

VI. EMPIRICAL EVALUATION

We have implemented our solution using the Go programming language in a concurrent prototype tool tadLola (available at http://github.com/imdea-software/dLola). We describe now:

- (1) an empirical study of the capabilities of tadLola in different scenarios with real data extracted from four different realistic public datasets.
- (2) the effect of the network behavior—in terms of delays–into memory and time to resolve outputs.

Our experimental setup intends to empirically determine the behavior of the asynchronous network and how failures affect the time to resolve of the streams.

A. DATASETS AND NETWORK FAILURES

It is common in the RV community to have theoretical works (without empirical evaluation) [5], [7], [9], [10], [12] or to use synthetic data and specifications for the empirical evaluation of solutions [1], [17], [19], [20], [21], [30], [43], [44]. In contrast, there are some that use realistic data [4], [6], [16], [22]. In our case, we have used four different datasets of real recorded data for this empirical evaluation, namely: SmartPolitech [37], Tomsk Heating [48], Orange4Home [15] and Context [33]. All datasets are related to smart buildings except for Context that is about Industry 4.0.

The first two datasets are concerned about building climate control and use sensors in different rooms or buildings respectively. The Orange4Home dataset focuses on activity recognition where a tenant can move freely in an apartment, and the goal is to infer the activity performed. Lastly, Context is a dataset in a smart factory where a new class of failures, namely contextual failures arise when there is no specific sensor or data collected that signals directly the error but the presence of the error and its underlying cause need to be inferred from contextual knowledge.

For each dataset we created a synthetic specification that could showcase the functionality of our tool. We also injected synthetic delays to model network congestion and failures.

- *constant* behavior is modeled as a global constant delay between each pair of monitors, so every message takes exactly a fixed amount of time to reach the destination network node. This corresponds to the network behavior assumed in synchronous monitoring.
- *constantPeak* consists of a constant delay, but perturbed with a single high delay, modeling a network failure and recovery. Hence, all messages get delayed until the network disruption is solved, and then the network starts to recover gradually until normal operation is reached again.
- *Normal* behavior follows a normal distribution of the delays given an average delay.
- *normalPeak* is similar to the constantPeak but with a baseline of the normal behavior.

Note that all these behaviors are both **aeternal** and **temporary** bounded since for all of them we can find an upper bound for the whole trace as well as a bound by window of interest of each instant variable.

	const			peak		
	min	med	max	min	med	max
smartP	4	4	4	4	4	102
smartPShort	4	4	4	4	4	102
tomsk	2	2	2	2	2	100
contextAct	2	2	2	2	2	100
orange4H	2	2	2	2	2	100

(a) Const and Peak network behaviors

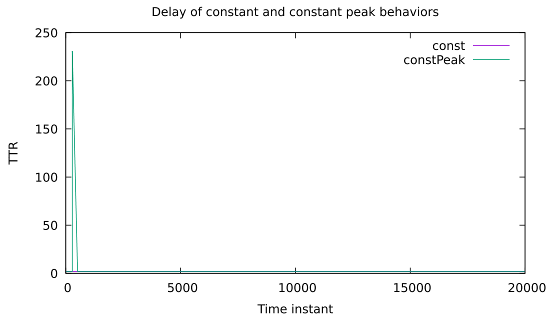
	normal			normalPeak		
	min	med	max	min	med	max
smartP	3	10	18	4	10	105
smartPShort	6	11	18	7	12	105
tomsk	1	5	10	1	5	100
contextAct	1	5	9	1	5	100
orange4H	1	5	10	1	5	100

(b) Normal and NormalPeak network behaviors

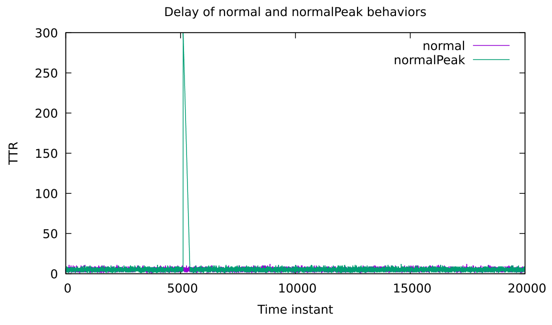
FIGURE 5. **TTR analysis of Tadlola for different network behaviors.**

Figure 5 shows the minimum, median and maximum TTR to resolve streams under these network behaviors.

We can observe an example of the delays observed under these behaviors in Figure 6.



(a) Const and const Peak



(b) Normal and NormalPeak

FIGURE 6. **Examples of network behaviors.**

The system under observation is sampled periodically, obtaining the input traces for each of the variables measured. Thus, having the length of the trace and the sampling period we can obtain the system time that gets monitored throughout the experiment. For example, a trace of length 200*k* with a sample period of 30 seconds, corresponds to monitoring a system during approximately 2.31 months.

For some of the experiments the traces of real data available in the datasets were not sufficiently long, so we extended those traces by repeating the samples as much as needed to reach the desired trace length. Also, some of those traces required interpolation in order to use a common clock tick for all events, since some of those traces were based on events instead of sensing periodically a variable. We did this interpolation whenever needed.

B. HYPOTHESIS

For the empirical evaluation of this paper we evaluated the following hypothesis:

- (H1) Our time Asynchronous algorithm behaves no worse than the synchronous algorithm from [17] when the network presents a synchronous behavior.
- (H2) Synchronous SRV can simulate the monitoring of a time asynchronous network with a software layer that provides the illusion of synchronicity, but at a very high cost in delays and memory usage.
- (H3) Our theoretical results of Section V hold for the execution of the experiments.
- (H4) The local memory of the root monitor is bounded, resulting in a trace length independent monitor our theoretical results predict.
- (H5) Our algorithm scales in terms of the number of monitors and network usage. We expect that memory will increase linearly with ''network usage'' but will remain constant when increasing the number of local monitors. Here we refer with local monitor to a non-empty set of streams that are computed at the same network node.
- (H6) The algorithm benefits from using redundant specifications and redundant topologies (exploiting simplifiers) to reduce *TTRs* by avoiding delays of slow or faulty links.

C. EMPIRICAL RESULTS

In order to validate hypothesis (H1) we built the following experiments:

- SmartPolitechDistr: we detect fire hazards by analyzing the levels of temperature, CO_2 and humidity in the air in different rooms in university buildings. We use a quantitative robust specification.
- tomskHeating: we check that the heating system is behaving as expected (extracted from the data). Again, this is a quantitative robust specification.
- orange4Home: we detect fire hazards by analyzing the activities performed by the tenant in the apartment.
- contextAct: we detect fire hazards by analyzing the levels of temperature, CO_2 and humidity in the air in different rooms in an smart apartment. This is also a quantitative specification.

We use TADSRV to refer to the solution in this paper, and DSRV for the solution in [17] which only works for synchronous networks. Figure 5 shows metrics of the delay of the root of the specification for the different datasets analyzed with different network behaviors. This proves empirically that TADSRV subsumes DSRV with no additional loss of performance, as expected by our theoretical proofs. Therefore (H1) holds. All these different network behaviors show that TADSRV is more general than DSRV, as we expected.

For the validation of (H2) we built an experiment with the specification of obtaining both the maximum and sum of the inputs. We placed this in the topology shown in Figure 7.

$$\begin{tikzcd}
 & & 0 \arrow[dl, dash] \arrow[dr, dash] & & \\
 & 1 \arrow[dl, dash] \arrow[dr, dash] & & 2 \arrow[dl, dash] \arrow[dr, dash] & \\
3 & & 4 & & 5
\end{tikzcd}$$

FIGURE 7. Monitor topology of the experiment for (H2).

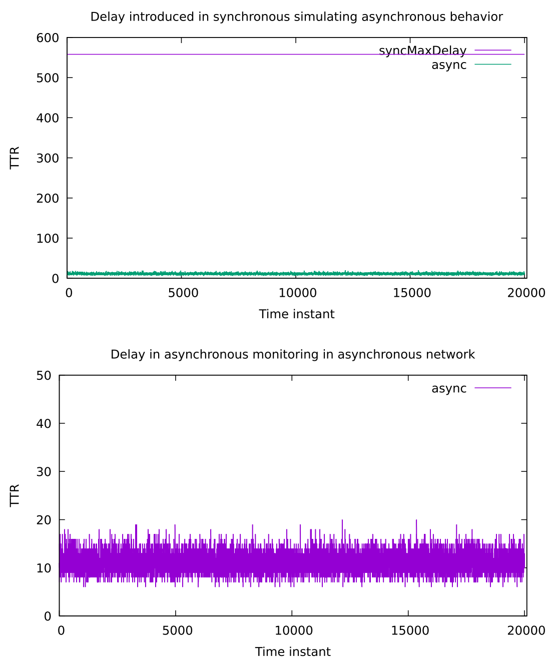
We looked for the maximum delay present in the normal-Peak traces that we have and used that duration as the global delay between each pair of monitors in the synchronous scenario. We measured both settings: simulating synchronicity and the execution of the timed asynchronous algorithm. The results are shown in Figure 8. The figure shows that we can emulate TADSRV with DSRV but with a high cost in memory usage (+200% than the worst instant) and incurring in delays of *worst delay* $*$ *depth of topology*, which in this case is 558 instants. This corresponds to an increase of around 30 times the delay when compared to the timed asynchronous solution. Therefore, (H2) holds as well. This results makes it clear that it is not feasible in practice to use DSRV in a time asynchronous scenario (even with the layer that simulates synchrony), where the contribution of this work applies naturally with much better performance.

Also, we can see that the TTRs obtained empirically are lower than or equal to our estimated bounds calculated a-priori with the equations described in Section V. Hence, (H3) holds.

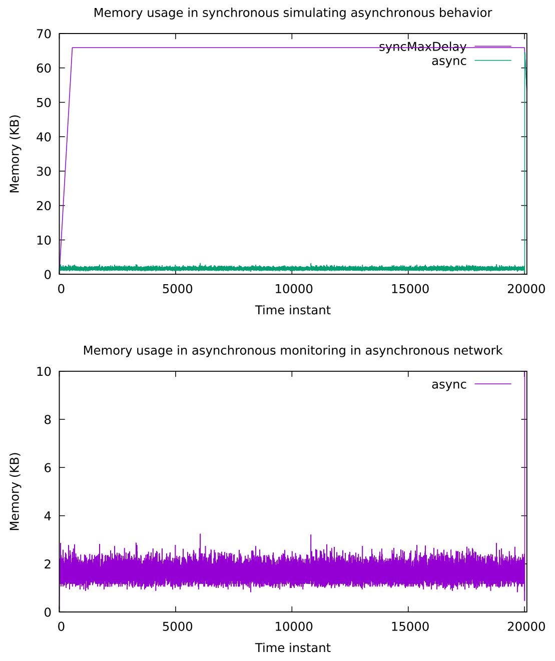
For the validation of (H4)—studying the scalability in terms of trace length—we used the smartPolitechDistr dataset and run it with a trace of 200k instants with the normalPeak behavior. In the extract shown below we compute both a Boolean and a quantitative stream to look for temperature uprisings.

```
define bool temp_up eval =
  temp > 1.1 * tempini and temp <= 1.6 * tempini
define num temp_up_q eval =
  if temp <= 1.1*tempini then 0~else
  if temp > 1.6*tempini then 1~else
  (temp - 1.1*tempini)/(1.6*tempini-1.1*tempini)
define bool temp_spike eval =
  temp > 1.6 *tempini
define num temp_spike_q eval =
  if temp <= 1.6*tempini then 0~else
  if temp > 2 * tempini then 1~else
  (temp - 1.6*tempini)/(2*tempini-1.6*tempini)
```

Figure 9 shows that the memory used in the root monitor of this experiment remains bounded. The pikes in memory correspond to higher delays in the network links among nodes. This forces monitors to keep records in their memory until the messages that they need arrive, allowing the monitor to resolve streams and prune their memories. This result suggests that the algorithm with a decentralized efficiently monitorable specification can behave in a trace-length independent fashion, validating hypothesis (H4).



(a) TTR of Synchronous and Asynchronous



(b) Root monitor memory of Synchronous and Asynchronous

FIGURE 8. Synchronous and asynchronous in an asynchronous network with details of asynchronous.

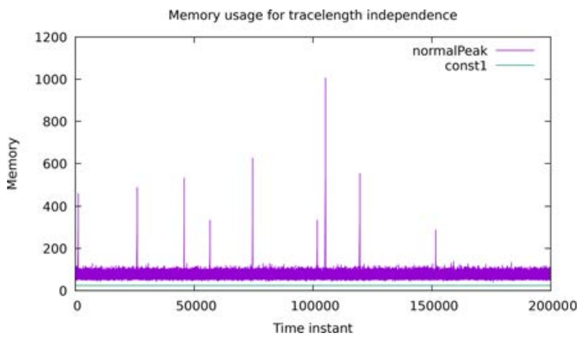


FIGURE 9. Memory of root monitor for a trace of 200000 instants.

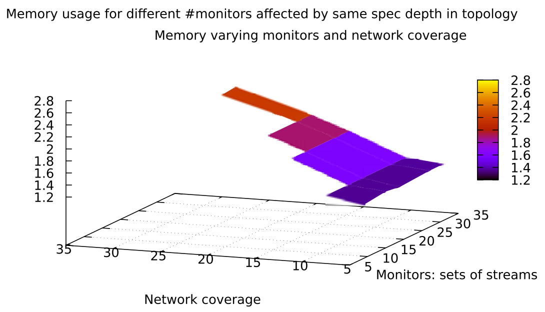


FIGURE 10. Average memory usage of the monitor that uses most memory of the last 1000 instants, tlen 20k, with different network coverages.

Figure 10 shows that the memory usage of a single monitor does not depend on the number of other monitors in the network but it depends on the maximum depth of its specification that travels the network. In this experiment the depth of the specification deployed in the network was kept constant (5) while we changed the number of monitors in a binary tree topology (preserving the depth in one branch). The intuition is that the variable that affects memory usage is not how many monitors we have but the number of network nodes and links among them that affect the monitoring performance. This is because adding more links increases the probability that there is a failure in the network (modelled as a delay) that affects the run. These results prove that hypothesis (H5) holds.

1) REDUNDANCY AND DELAYS

In this subsection we take a closer look at hypothesis (H6), so we build the topology and the specification to minimize the *TTR* of the instant variables. We seek to benefit from using simplifiers to minimize the effect network delays of messages required to compute the instant variables. Thus, we intend to exploit the messages that go through the fastest path in the network from the nodes that read the inputs to the nodes that compute the root of the specification. Intermediate results are generated faster in the least congested deployment and messages will travel through the least weight path (in terms of accumulated delays) between the inputs and the root of the specification yielding a minimum TTR for the instant variables.

This improvement can be achieved because intermediate results from slower monitors will not be needed due to the use of simplifiers, and therefore the engine will not wait to achieved a final result of the root monitor. We build the following fragment of the specification for the data in smart-Politech, where we make the streams `C3_fire_risk_q` and `C3_fire_risk_q_red` redundant of each other and we deploy them in different monitors so that they are affected by different delays. We use a normal delay for the whole network but introduce a failure in the form of a peak in the delays between the monitors connected to monitor 3. This will make the path through monitor 2 faster.

We can observe in Figure 11 how the delay of obtaining the value for the root of the specification takes the best delay possible. Since we use an OR to take advantage of the simplifiers, in the best case verdict (outcome true) there is a gain, but in the worst case verdict (false) the redundant solution gains no speed as the engine needs to wait for all the values to calculate the OR.

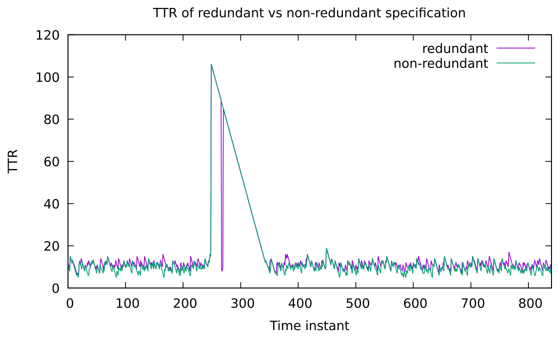


FIGURE 11. Benefits of using Redundancy in terms of accumulated delays.

```
@0{
define bool C3_alarm eval =
 (C3_fire_risk or C3_fire_risk_red) and
 (C3_fire_risk_q > 0.5~or C3_fire_risk_q_red >0.5)
}
@2{
define bool C3_fire_risk_red eval =
 AND(C3_temp_spike,C3_co2_spike,C3_humid_down)
define num C3_fire_risk_q_red eval =
 AVG(C3_temp_spike_q,C3_co2_spike_q,C3_humid_down_q)
}
@3{
define bool C3_fire_risk eval =
 AND(C3_temp_spike,C3_co2_spike,C3_humid_down)
define num C3_fire_risk_q eval =
 AVG(C3_temp_spike_q,C3_co2_spike_q,C3_humid_down_q)
}
```

Figure 11 shows the difference between using the redundant specification with redundant topology and not using any redundancy. Even though a general study of exploiting redundant paths in the network is out of the scope of this paper, this case study illustrates how redundant deployments can improve decentralized monitoring.

VII. LAZY ALGORITHM

We introduce now a variant of Algorithm 2 where some of the streams are not sent unless their values are requested by the

Algorithm 2 Local monitoring algorithm at node n with $MS_n = \langle Q_n, U_n, R_n, P_n, W_n \rangle$

```
 1: procedure Monitor
 2:     $MS_n \leftarrow \emptyset$; $k \leftarrow$ Now()
 3:     while not END do Step($k$)
 4:     $M \leftarrow k$; Finalize($M$)
 5: procedure Step($k$)
 6:     $R_{old} \leftarrow MS_n.R_n$
 7:     ProcessMessages($MS_n$)
 8:     $R_n.add(\{r\langle k\rangle \mapsto read(r, k) \mid r \in ins_n\})$
 9:     $U_n.add(\{s\langle k\rangle \mapsto e_s\langle k\rangle \quad \mid s \in outs_n\})$
10:     Evaluate($MS_n$)
11:     SendResponses($MS_n$)
12:     SendRequests($MS_n$)
13:     SendConfirmations($MS_n$)
14:     Prune($MS_n$)
15: procedure Evaluate($MS_n$)
16:     $done \leftarrow false$
17:     while $not \ \ done$ do
18:         $done \leftarrow true$
19:         for all $s\langle k\rangle \mapsto e \in U_n$ do
20:             $e' \leftarrow$ Subst($e$, $R_n$)
21:             if $ground(e')$ then $done \leftarrow false$
22:                 $U_n.del(s\langle k\rangle \mapsto e)$; $R_n.add(s\langle k\rangle \mapsto e')$
23:             else $U_n.del(s\langle k\rangle \mapsto e)$; $U_n.add(s\langle k\rangle \mapsto e')$
24: procedure ProcessMessages($MS_n$)
25:     for all $msg \leftarrow Q_n.pop()$ do
26:         switch $msg$ do
27:             case $\langle$req, $s\langle k\rangle\rangle$     $P_n.add(s\langle k\rangle)$
28:             case $\langle$resp, $s\langle k\rangle, c\rangle$
29:                 $R_n.add(s\langle k\rangle \mapsto c)$; $W_n.del(s\langle k\rangle)$
30: procedure SendResponses($MS_n$, $R_{old}$)
31:     $R_{new} \leftarrow MS.R_n \ R_{old}$
32:     for all $u\langle k\rangle \mapsto c \in R_{new}$ do   ▷ Eager new knowledge
33:         $send$(resp, $s\langle k\rangle, c, n, n_r$)
34:     for all $\langle$req, $s\langle k\rangle, n_r, n\rangle \in P_n$ do      ▷ Lazy requests
35:         if $s\langle k\rangle \mapsto c \in R_n$ then
36:             $send$(resp, $s\langle k\rangle, c, n, n_r$)
37:             $P_n.del(\langle$req, $s\langle k\rangle, n_r, n\rangle)$
38: procedure SendRequests($MS_n$)
39:     for all $(\_, e) \in U_n$ do
40:         for all $u\langle k'\rangle \in sub(e)$ do
41:             if $u\langle k'\rangle \notin W_n \land \mu(u) \neq n$ then
42:                 $send$(req, $u\langle k'\rangle, n, \mu(u)$); $W_n.add(u\langle k'\rangle)$
43: procedure SendConfirmations($MS_n$)
44:     for all $u\langle k\rangle \mapsto c \in R_{new}$ do          ▷ new knowledge
45:         $send$(confirm, $s\langle k\rangle, n, n_r$)
46: procedure Prune($MS_n$)        ▷ If $u\langle j\rangle$ will not be needed
47:     for all $u\langle j\rangle \mapsto c \in R_n \mid$ confirmed($u\langle j\rangle \mapsto c$) do
48:         $R_n.del(u\langle j_i\rangle \mapsto c_i)$; $U_n.del(u\langle j_i\rangle)$
```

remote nodes. This is beneficial in cases where the value to be requested is rarely needed (for example, it is only needed in the *else* part where the test is typically true and locally checkeable). We call these *lazy streams*.

To introduce the modified algorithm we need to introduce a new type of message: the request message. We also call a response message to the messages containing the value of an instant variable.

- **Response** messages: (**resp**, $s\langle k\rangle, c, n_s, n_d$) where $s\langle k\rangle$ is an instant variable, c is a constant of the same datatype as $s\langle k\rangle$, n_s is the source node and n_d is the destination node of the message.
- **Requests** messages: (**req**, $s\langle k\rangle, n_s, n_d$) where $s\langle k\rangle$ is an instant variable, n_s is the source node and n_d is the destination node of the message.

Again, if $msg = (\textbf{req}, s\langle k\rangle, n_s, n_d)$, then $msg.src = n_s$, $msg.dst = n_d$, $msg.type = \textbf{req}$, $msg.stream = s\langle k\rangle$. Similarly, for a response message we have the same, the only difference is that we add $msg.val = c$.

Each stream variable v can be assigned one of the following two *communication strategies* to denote whether an instant value $v\langle k\rangle$ is automatically communicated to all potentially interested nodes, or whether its value is provided upon request only. Let v and u be two stream variables such that v appears in the equation of u and let $n_v = \mu(v)$ and $n_u = \mu(u)$.

- **Eager communication**: the node n_v informs n_u of every value $v\langle k\rangle = c$ that it resolves by sending a message (**resp**, $v\langle k\rangle, c, n_v, n_u$). This is what we have used previously in the paper.
- **Lazy communication**: node n_u requests n_v the value of $v\langle k\rangle$ (in case n_u needs it to resolve $u\langle k'\rangle$ for some k') by sending a message (**req**, $v\langle k\rangle, n_u, n_v$). When n_u receives this message and resolves $v\langle k\rangle$ to a value c, n_u will respond with (**resp**, $v\langle k\rangle, c, n_v, n_u$).

Each stream variable can be independently declared as eager or lazy. We use two predicates $eager(u)$ and $lazy(u)$ (which is defined as $\neg eager(u)$) to indicate the communication strategy of stream variable u. Note that the lazy strategy involves two messages and the eager strategy only one, but the eager strategy sends a message every time an instant variable resolved, while lazy will only sends those that are requested. In case the values are almost always needed, eager is preferable while if values are less frequently required lazy is preferred. We now need to add the communication strategy to the definition of the decentralized SRV problem. A decentralized SRV problem $\langle \varphi, \mathcal{T}, \mu, eager\rangle$ is now characterized by a specification φ, a topology $\mathcal{T}$, a stream assignment μ and a communication strategy for every stream variable.

A. LAZY DSRV ALGORITHM FOR TIMED ASYNCHRONOUS NETWORKS

We extend our local monitor to $\langle Q_n, U_n, R_n, P_n, W_n\rangle$ adding the following two storages:

- **Pending** requests P_n, where n records instant variables that have been requested from n by other monitors but that n has not resolved yet.

- **Waiting** for responses W_n, where n records instant variables that n has requested from other nodes but has received no response yet.

The storage W_n is used to prevent n from requesting the same value twice while waiting for the first request to be responded. An entry in W_n is removed when the value is received, since the value will be subsequently fetched directly from R_n and not requested through the network. The storage P_n is used to record that a value that n is responsible for has been requested, but n does not know the answer yet. When n computes the answer, then n will send the corresponding response message and remove the entry from P_n. Finally, request messages are generated for unresolved lazy instant variables and inserted in the queues of the corresponding neighbors.

More concretely, every node n will execute the procedure Monitor shown in Algorithm 2, which invokes Step in every clock tick until the input terminates or ad infinitum. Procedure Finalize is used to resolve the pending values at the end of the trace to their default if the trace ends. Procedure Step now executes some modified procedures and additional steps:

1) **Process Messages**: Line 27 annotates requests in P_n, which will be later resolved and responded. Lines 28-29 handle response arrivals, adding them to R_n and removing them from W_n.
2) **Send Responses:** Lines 34-37 deal with pending lazy variables. If a pending instant variable is now resolved, the response message is sent and the entry is removed from P_n.
3) **Send new Requests:** Lines 38-42 send new request messages for all lazy instant streams that are now needed.
4) **Send Confirmations:** Line 43-45 send confirmations for the newly resolved instant variables. See section VII-C1.d.
5) **Prune:** Line 46-48 prune the set R and U from information that is no longer needed. See section VII-C1.d.

B. FORMAL CORRECTNESS

We now show that our solution is correct again by proving that the output computed is the same as in the denotational semantics, and that every output is eventually computed.

Theorem 4: All of the following hold for every instant variable $u\langle k\rangle$:

(1) The value of $u\langle k\rangle$ is eventually resolved.
(2) The value of $u\langle k\rangle$ is c if and only if $(u\langle k\rangle, c) \in R$ at some instant.
(3) If $eager(u)$ then a response message for $u\langle k\rangle$ is eventually sent.
(4) If $lazy(u)$ then all request messages for $u\langle k\rangle$ are eventually responded.

Proof: The proof proceeds by induction in the evaluation graph, showing simultaneously in the induction step (1)-(4) as these depend on each other (in the previous inductive steps). Let M be a length of a computation and σ_I be an input of length M. Note that (1) to (4) above are all statements about instant variables $u\langle k\rangle$, which are the nodes of the evaluation graph $G_{\varphi,M}$. We proceed by induction on $G_{\varphi,M}$ (which is acyclic because D_φ is well-formed).

- **Base case**: The base case are vertices of the evaluation graph that have no outgoing edges, which are either instant variables that correspond to inputs or to defined variables whose instant equation does not contain other instant variables. Statement (1) follows immediately for inputs because at instant k, $s\langle k\rangle$ is read at node $\mu(k)$. For output equations that do not have variables, or whose variables have offsets that once instantiated become negative or greater than M, the value of its leafs is determined either immediately or at M when the offset if calculated. At this point, the value computed is inserted in R, so (2) also holds at $\mu(u)$. Note that (2) also holds for other nodes because the response message contains $u\langle k\rangle = c$ if and only if $(u\langle k\rangle, c) \in R_n$, where $\mu(u) = n$. Then the response message is inserted exactly at the point it is resolved, so (1) implies (3). Finally, (4) also holds at the time of receiving the request message or resolving $u\langle k\rangle$ (whatever happens later).
- **Inductive case**: Consider an arbitrary $u\langle k\rangle$ in the evaluation graph $G_{\varphi,M}$ and let $u_1\langle k_1\rangle \ldots u_l\langle k_l\rangle$ the instant variables that $u\langle k\rangle$ depends on. These are nodes in $G_{\varphi,M}$ that are lower than $u\langle k\rangle$ so the inductive hypothesis applies, and (1)-(4) hold for these instant variables. Let $n = \mu(u)$. At instant k, $u\langle k\rangle$ is instantiated and inserted in U_n. At the end of cycle k, lazy variables among $u_1\langle k_1\rangle \ldots u_l\langle k_l\rangle$ are requested. By induction hypothesis, at some instant all these requests are responded by (1) and (4). Similarly, the values of all eager variables are calculated and sent as well (by (1) and (3) which hold by IH). At the latest time of arrival, the equation for $u\langle k\rangle$ has no more variables and it is evaluated to a value, so (1) holds and (2) holds for $u\langle k\rangle$ at n. At this point, if $eager(u)$ then the response message is sent (so (1) holds for $u\langle k\rangle$) and if $lazy(u)$ then all requests (previously received in P_n or future requests) are answered, so (1) also holds.

This finishes the proof.

C. COMPLEXITY ANALYSIS FOR LAZY

Analyzing the lazy case requires modifications.

1) MEMORY COMPLEXITY FOR LAZY

In timed asynchronous networks we need to introduce a new kind of message to provide *confirmations* that are only used to inform the receiving node that some instant variables are not needed so they can be pruned. This new message have the following form:

- **Confirmation** messages: (**confirm**, $s\langle k\rangle, n_s, n_d$) where $s\langle k\rangle$ is an instant variable, n_s is the source node and n_d is the destination node of the message.

This message will be interpreted as the source node n_s has resolved instant variable $s\langle k\rangle$. This information allows the destination node to conclude that instant variables required at the remote node for instant variables that have been resolved are no longer necessary. We change MTR_{rem} to include that the response gets emitted when the request arrives or when the remote instant variable gets resolved, whichever happens later.

$$MTR_{rem}^{lazy}(s\langle t\rangle) \overset{\text{def}}{=} \max(t, M(s\langle t\rangle)$$

where

$$M(s\langle t\rangle) = \{arr_{r\to s}(t') \text{ s.t.} s \xrightarrow[rem]{w} r \text{ and } t+w \geq 0\})$$

and

$$t' = \max(arr_{s\to r}(t), MTR^{lazy}(r\langle t+w\rangle))$$

Here $arr_{s\to r}(t)$ is the time when the request is sent, that is, when the instant variable s gets instantiated and stored in U. $MTR_{rem}^{lazy}(r\langle t + w\rangle)$ is when the remote instant stream gets resolved. Finally $arr_{r\to s}(t')$ is the moment at which the response of the lazy instant stream variable arrives at the requesting node.

a: INSTANTIATION TO SYNCHRONOUS

Again, we first consider the case where the delay of any link to be a constant throughout the execution. This constant is useful to simplify the equations but we need to consider now that for each instant variable we need a request and afterwards a response, in order to get the remote value. Again, $dist_{r\_s}$ is used to represent the delay that every message will take from $\mu(r)$ to $\mu(s)$, independently of the time instant at which the message is sent. We use this knowledge to simplify MTR_{rem}^{lazy} for synchronous networks as follows

$$MTR_{rem}^{synclazy}(s\langle t\rangle) \overset{\text{def}}{=} \max(t, t') \text{ s.t.} s \xrightarrow[rem]{w} r; t+w \geq 0\})$$

where

$$t' = \{dist_{r\_s} + \max(t + dist_{s\_r}, MTR^{synclazy}(r\langle t+w\rangle))$$

where the value of the remote instant variable arrives when the response message arrives $dist_{r\_s}$, which is emitted either when the request arrived $t + dist_{s\_r}$ or when the remote value is resolved $MTR^{sync}(r\langle t+w\rangle)$, whichever occurs later.

b: ***AETERNALLY*** BOUNDED DELAYS

Now we consider that case where we know a maximum delay in the network that upper bounds all the other delays in the network behavior. Substituting the upper-bound value d in the equations for MTR, we obtain an constant upper-bound on the MTR (although this value can be a gross over-approximation):

$$MTR_{rem}^{aeternal\ lazy}(s\langle t\rangle) \overset{\text{def}}{=} \max(t, M(s\langle t\rangle))$$

where

$$M(s\langle t\rangle = \{d + \max(t+d, t') \text{ s.t. } s \xrightarrow[rem]{w} r; t+w \geq 0\})$$

and

$$t' = MTR^{aeternal\ lazy}(r\langle t+w\rangle)$$

c: ***TEMPORARILY*** BOUNDED DELAYS

Finally, we do not assume an ***aeternal*** bound on the delays of the network. Instead, we can just look at what affects the computation of the instant variables, that is, other instant variables that it depends on and the network delays that affect the messages to compute those instant variables. We take into account again the window $win(s\langle t\rangle)$, which contains the interval that includes all the instants at which values that influence $s\langle t\rangle$ are resolved and sent. This window always ends at most at $MTR(s\langle t\rangle)$. Inside this window we can find the worst delay of a message sent for the computing of the instant variable: $d_{worst}(s\langle t\rangle)$. Then, we can bound $MTR(s\langle t\rangle)$ as follows for the lazy case:

$$MTR_{rem}^{temp\ lazy}(s\langle t\rangle) \overset{\text{def}}{=} \max(t, \{t' \text{ s.t.} s \xrightarrow[rem]{w} r; t+w \geq 0\})$$

where

$$t' = d_{worst}(s\langle t\rangle) + \max\begin{pmatrix} d_{worst}(s\langle t\rangle) \\ MTR^{temp\ lazy}(r\langle t+w\rangle)\end{pmatrix}$$

Here, $d_{worst}(s\langle t\rangle)$ is the time for worst message affecting the computation of $s\langle t\rangle$, so the window for obtaining this value considers both request and response messages. We use this value to bound both the request and the response. First, we obtain the latter instant at which either the request arrives or the remote dependency is resolved in $\max(d_{worst}(s\langle t\rangle), MTR^{temp\ lazy}(r\langle t + w\rangle))$ and then we add the time for the response message to arrive with the value in $d_{worst}(s\langle t\rangle)$. Obtaining the moment at which we know that the remote dependency is guaranteed to be resolved and its value arrived at the requesting network node.

d: PRUNING THE RESOLVED STORAGE

We are finally ready to prune R_n for the lazy algorithm case because we know now when every instant variable will be resolved.

Theorem 5: Every unresolved instant variable $s\langle t\rangle$ that is lazy in U_n is resolved at most at $MTR^{lazy}(s\langle t\rangle)$.

As soon as this moment is reached, considering that the network delays are bounded, a confirmation message will be sent to those monitors where lazy instant variables that are dependencies to the resolved instant variable are computed and this message will arrive in bounded time. Then the receiving node can prune the corresponding instant variables from its memory. Now we need to add $tconf$ in this theorem which is the time for the confirmation message to arrive:

Every unresolved $s\langle k\rangle = e$ in U_n is pruned at most at $max(\{MTR^{lazy}(u\langle k-w\rangle) + tconf_u\})$. Where $u\langle k-w\rangle$ is a remote instant variable that contains $s\langle k\rangle$ in its equation and $tconf_u$ is the time for the confirmation message to travel from $\mu(u)$ to $\mu(s)$ sent at time $MTR^{lazy}(u\langle k-w\rangle)$. This message arrives at destination in bounded time and the instant variable gets pruned. Because at that point the receiving node knows that the instant variable is no longer needed and we say that the instant variable is confirmed. At that point the monitor can prune the instant variable even if it is not resolved yet, as shown in lines 47-48 of Algorithm 2.

With this mechanism, we can assure that every instant variable will be in memory (U_n, R_n) for a bounded amount of time. This implies that decentralized efficiently monitorable specifications in timed asynchronous networks can be monitored with bounded resources. The bound depends only linearly on the size of the specification, the diameter of the network and the delays among the nodes of the network.

2) TIME COMPLEXITY ANALYSIS FOR LAZY

The Time complexity remains unchanged from the eager case.

3) MESSAGE COMPLEXITY ANALYSIS FOR LAZY

Here we analyze the total number of messages amortized and it is different from the eager case. For each instant variable, we will have a request, a response and a confirmation message traversing the network.

Proposition 2: Let φ be a specification with s streams and N be the trace length, then there will be at most $3 * s * N$ messages during the monitoring.

Proof: As we have s streams and a tracelength of N, then we have $s * N$ instant variables by instancing the streams at each time point. By procedure SendRequests in line 12 of Algorithm 2 we observe that line 41 will be invoked once per instant variable. Also, by procedure SendResponses in line 11 of Algorithm 2 we observe that line 35 will be invoked once per instant variable. Finally, by procedure SendConfirmations in line 42 of Algorithm 2 we observe that line 41 will be invoked once per instant variable.

So, now we have a total of 3 messages(request, response and confirmation) per instant variable. The worst case(when most messages are needed) occurs when each stream is mapped to a different node. In that case we have up to $3 * s * N$ messages.

VIII. CONCLUSION AND FUTURE WORK

We have studied the problem of decentralized stream runtime verification for timed asynchronous networks where messages can take an arbitrary amount of time to arrive. This problems starts from a specification and a network. Our solution consists of a placement of output streams and an online local monitoring algorithm that runs on every node. We prove the termination and correctness of the proposed algorithm. We have studied the algorithms complexity and captured specifications and network assumptions (synchronous, **aeternal** and **temporary** bounds) that guarantee that the monitoring can be performed with constant memory independently of the length of the trace showing that our solution subsumes the previous synchronous algorithm. We report on an empirical evaluation of our prototype tool tadLola. Our empirical evaluation shows that placement is crucial for performance and suggest that in most cases careful placement can lead to bounded costs and delays.

As future work we plan to extend our solution to disaster scenarios where some links may present a delay ad infinitum, so no message can traverse that link. Our intuition is that we could use redundancy in the specifications and the network topology to provide resilience against faulty network links while also providing better performance than just by replicating the time asynchronous algorithm and running them in parallel isolated from each other.

REFERENCES

[1] D. Ancona, A. Ferrando, and V. Mascardi, "Exploiting probabilistic trace expressions for decentralized runtime verification with gaps," in *Proc. 37th Italian Conf. Comput. Log.*, vol. 3204, R. Calegari, G. Ciatto, and A. Omicini, Eds. Bologna, Italy, 2022, pp. 154–170.

[2] E. Asarin, P. Caspi, and O. Maler, "Timed regular expressions," *J. ACM*, vol. 49, no. 2, pp. 172–206, 2002.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *Proc. 5th Int. Conf. Verification, Model Checking Abstract Interpretation* (Lecture Notes in Computer Science), vol. 2937. Cham, Switzerland: Springer, 2004, pp. 44–57.

[4] D. Basin, D. S. Dietiker, S. Krstic, Y.-A. Pignolet, M. Raszyk, J. Schneider, and A. Ter-Gabrielyan, "Monitoring the internet computer," in *Formal Methods*, M. Chechik, J.-P. Katoen, and M. Leucker, Eds. Cham, Switzerland: Springer, 2023, pp. 383–402.

[5] D. Basin, F. Klaedtke, and E. Zalinescu, "Failure-aware runtime verification of distributed systems," in *Proc. 35th IARCS Annu. Conf. Foundations Softw. Technol. Theor. Comput. Sci.*, vol. 45, 2015, pp. 590–603.

[6] D. A. Basin, F. Klaedtke, and E. Zalinescu, "The MonPoly monitoring tool," *RV-CuBES*, vol. 3, pp. 19–28, Dec. 2017.

[7] A. Bauer, J.-C. Kuster, and G. Vegliach, "From propositional to first-order monitoring," in *Runtime Verification* (Lecture Notes in Computer Science), A. Legay and S. Bensalem, Eds. Rennes, France: Springer, 2013, pp. 59–75.

[8] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, p. 14, 2011.

[9] A. K. Bauer and Y. Falcone, "Decentralised LTL monitoring," in *Proc. 18th Int. Symp. Formal Methods* (Lecture Notes in Computer Science), vol. 7436. Cham, Switzerland: Springer, 2012, pp. 85–100.

[10] B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, D. Rosenblueth, and C. Travers, "Decentralized asynchronous crash-resilient runtime verification," *J. ACM*, vol. 69, no. 5, pp. 1–31, Oct. 2022.

[11] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, Dec. 2015.

[12] A. Castañeda and G. Valeria Rodríguez, "Asynchronous wait-free runtime verification and enforcement of linearizability," 2023, *arXiv:2301.02638*.

[13] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, "TeSSLa: Temporal stream-based specification language," in *Proc. 21st Brazilian Symp. Formal Methods* (Lecture Notes in Computer Science), vol. 11254. Cham, Switzerland: Springer, 2018, pp. 144–162.

[14] F. Cristian and C. Fetzer, "The timed asynchronous distributed system model," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 642–657, Jun. 1999.

[15] J. Cumin, G. Lefebvre, F. Ramparany, and J. Crowley, "A dataset of routine daily activities in an instrumented home," in *Proc. Int. Conf. Ubiquitous Comput. Ambient Intell.*, 2017, pp. 413–425.

[16] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "LOLA: Runtime monitoring of synchronous systems," in *Proc. 12th Int. Symp. Temporal Represent. Reasoning*, 2005, pp. 166–174.

[17] L. M. Danielsson and C. Sanchez, "Decentralized stream runtime verification," in *Runtime Verification* (Lecture Notes in Computer Science), vol. 11757, B. Finkbeiner and L. Mariani, Eds. Porto, Portugal: Springer, 2019, pp. 185–201.

[18] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout, "Reasoning with temporal logic on truncated paths," in *Proc. 15th Int. Conf. Comput. Aided Verification* (Lecture Notes in Computer Science), vol. 2725. Cham, Switzerland: Springer, 2003, pp. 27–39.

[19] A. El-Hokayem and Y. Falcone, "Monitoring decentralized specifications," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2017, pp. 125–135.

[20] A. El-Hokayem and Y. Falcone, "THEMIS: A tool for decentralized monitoring algorithms," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2017, pp. 372–375.

[21] A. El-Hokayem and Y. Falcone, "On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 1, pp. 1–57, Jan. 2020.

[22] A. El-Hokayem and Y. Falcone, "Bringing runtime verification home: A case study on the hierarchical monitoring of smart homes using decentralized specifications," *Int. J. Softw. Tools Technol. Transf.*, vol. 24, no. 2, pp. 159–181, Apr. 2022.

[23] P. Faymonville, B. Finkbeiner, S. Schirmer, and H. Torfah, "A stream-based specification language for network monitoring," in *Proc. 16th Int. Conf. Runtime Verification* (Lecture Notes in Computer Science), vol. 10012. Cham, Switzerland: Springer, 2016, pp. 152–168.

[24] P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, and T. Hazem, "StreamLAB: Stream-based monitoring of cyber-physical systems," in *Proc. 31st Int. Conf. Comput.-Aided Verification* (Lecture Notes in Computer Science), vol. 11561. Cham, Switzerland: Springer, 2019, pp. 421–431.

[25] A. Francalanza, J. A. Perez, and C. Sanchez, "Runtime verification for decentralised and distributed systems," in *Lectures Runtime Verification* (Lecture Notes in Computer Science), vol. 10457, E. Bartocci and Y. Falcone, Eds. Cham, Switzerland: Springer, 2018, pp. 176–210.

[26] F. Gallay and Y. Falcone, "Decent: A benchmark for decentralized enforcement," in *Runtime Verification*, T. Dang and V. Stolz, Eds. Cham, Switzerland: Springer, 2022, pp. 293–303.

[27] R. Ganguly, A. Momtaz, and B. Bonakdarpour, "Distributed runtime verification under partial synchrony," in *Proc. 24th Int. Conf. Princ. Distrib. Syst.* (Lecture Notes in Computer Science), vol. 184, Q. Bramas, R. Oshman, P. Romano, Eds. Dagstuhl, Germany: Schloss Dagstuhl-Leibniz Center for Informatics, 2021, pp. 1–17.

[28] 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 *Proc. IEEE 42nd Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2022, pp. 23–33.

[29] F. Gorostiaga, L. M. Danielsson, and C. Sanchez, "Unifying the time-event spectrum for stream runtime verification," in *Runtime Verification* (Lecture Notes in Computer Science), vol. 12399, J. Deshmukh and D. Nickovic, Eds. Los Angeles, CA, USA: Springer, 2020, pp. 462–481.

[30] F. Gorostiaga and C. Sanchez, "Striver: Stream runtime verification for real-time event-streams," in *Proc. 18th Int. Conf. Runtime Verification* (Lecture Notes in Computer Science), vol. 11237. Cham, Switzerland: Springer, 2018, pp. 282–298.

[31] K. Havelund and G. Roşu, "Synthesizing monitors for safety properties," in *Proc. 8th Int. Conf. Tools Algorithms Construction Anal. Syst.* (Lecture Notes in Computer Science), vol. 2280. Cham, Switzerland: Springer, 2002, pp. 342–356.

[32] M. Jaber, Y. Falcone, P. Attie, A.-A. Khalil, R. Hallal, and A. El-Hokayem, "From global choreographies to verifiable efficient distributed implementations," *J. Log. Algebr. Methods Program.*, vol. 115, Oct. 2020, Art. no. 100577.

[33] L. Kaupp, H. Webert, K. Nazemi, B. Humm, and S. Simons, "CONTEXT: An industry 4.0 dataset of contextual faults in a smart factory," *Proc. Comput. Sci.*, vol. 180, pp. 492–501, Jan. 2021.

[34] S. Kazemlou and B. Bonakdarpour, "Crash-resilient decentralized synchronous runtime verification," in *Proc. IEEE 37th Symp. Reliable Distrib. Syst. (SRDS)*, Oct. 2018, pp. 207–212.

[35] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann, 1996.

[36] A. Momtaz, H. Abbas, and B. Bonakdarpour, "Monitoring signal temporal logic in distributed cyber-physical systems," in *Proc. ACM/IEEE 14th Int. Conf. Cyber-Phys. Syst.*, May 2023, pp. 154–165.

[37] F. Pajuelo-Holguera, J. A. Gómez-Pulido, and F. Ortega, "Recommender systems for sensor-based ambient control in academic facilities," *Eng. Appl. Artif. Intell.*, vol. 96, Nov. 2020, Art. no. 103993.

[38] I. Perez, F. Dedden, and A. Goodloe, "Copilot 3," NASA, Washington, DC, USA, Tech. Rep., NASA/TM-2020-220587, Apr. 2020.

[39] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: A hard real-time runtime monitor," in *Proc. 1st Int. Conf. Runtime Verification* (Lecture Notes in Computer Science), vol. 6418. Cham, Switzerland: Springer, 2010, pp. 345–359.

[40] L. Pike, N. Wegmann, S. Niller, and A. Goodloe, "Copilot: Monitoring embedded systems," *Innov. Syst. Softw. Eng.*, vol. 9, no. 4, pp. 235–255, Dec. 2013.

[41] D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe, "PrivApprox: Privacy-preserving stream analytics," in *Proc. USENIX Annu. Tech. Conf.*, Santa Clara, CA, Jul. 2017, pp. 659–672.

[42] G. Rosu and K. Havelund, "Rewriting-based techniques for runtime verification," *Automated Softw. Eng.*, vol. 12, no. 2, pp. 151–197, Apr. 2005.

[43] V. Roussanaly and Y. Falcone, "Decentralised runtime verification of timed regular expressions," in *Proc. 29th Int. Symp. Temporal Represent. Reasoning*, vol. 247, A. Artikis, R. Posenato, and S. Tonetta, Eds. Dagstuhl, Germany: Schloss Dagstuhl-Leibniz Center for Informatics, 2022, pp. 1–18.

[44] M. Samadi, F. Ghassemi, and R. Khosravi, "Decentralized runtime verification of message sequences in message-based systems," *Acta Inf.*, vol. 60, no. 2, pp. 145–178, 2022.

[45] C. Sanchez, "Online and offline stream runtime verification of synchronous systems," in *Proc. 18th Int. Conf. Runtime Verification* (Lecture Notes in Computer Science), vol. 11237. Cham, Switzerland: Springer, 2018, pp. 138–163.

[46] K. Sen and G. Roşu, "Generating optimal monitors for extended regular expressions," in *Electronic Notes in Theoretical Computer Science*, vol. 89, O. Sokolsky and M. Viswanathan, Eds. Amsterdam, The Netherlands: Elsevier, 2003.

[47] K. Sen, A. Vardhan, G. Agha, and G. Rosu, "Efficient decentralized monitoring of safety in distributed systems," in *Proc. 26th Int. Conf. Softw. Eng.*, 2004, pp. 418–427.

[48] P. Zorin and O. Stukach, "Data of heating meters from residential buildings in Tomsk (Russia) for statistical modeling of the thermal characteristics of buildings," 2020, doi: 10.21227/3r4e-ch18.

LUIS MIGUEL DANIELSSON was born in Madrid, Comunidad de Madrid, España, in 1992. He received the B.S. degree in computer engineering and the M.S. degree in software and systems from Universidad Politécnica de Madrid, Madrid, Spain, in 2015 and 2018, respectively, where he is currently pursuing the Ph.D. degree in software, systems and computing.

From 2015 to 2016, he was working at the consulting firm Management Solutions. From 2016 to 2017, he was with Speex as a Web Developer. From 2017 to 2018, he was a Research Intern with the Reactive Systems Group, IMDEA Software Institute. Since 2018, he has been a Research Assistant with the Reactive Systems Group. His research interests include software verification, formal methods, runtime verification, stream runtime verification, distributed systems, and fault tolerance.

CÉSAR SÁNCHEZ (Senior Member, IEEE) received the M.S. degree in electrical engineering from Ingenería Superior de Telecomunicación, Universidad Politécnica de Madrid, in 1998, and the M.S. and Ph.D. degrees in computer science from Stanford University, in 2001 and 2007, respectively. In 2007, he was a Postdoctoral Researcher with the University of California at Santa Cruz. He joined the IMDEA Software Institute, in 2008, as an Assistant Professor, and he was promoted to an Associate Professor, in 2013. From 2009 to 2020, he was a Research Scientist with the Spanish National Council for Research (CSIC). His research interests include the applications of logic to computer science, and applicable formal methods for the design, the analysis and verification of distributed systems, real-time systems, and embedded systems. He is a Senior Member of ACM and a member of Mensa. He was a recipient of the Frank Anger Memorial Award, in 2006.