Edge-based Runtime Verification for the Internet of Things

Christos Tsigkanos, Marcello M. Bersani, Pantelis A. Frangoudis, and Schahram Dustdar

Abstract—Complex distributed systems such as the ones induced by Internet of Things (IoT) deployments, are expected to operate in compliance to their requirements. This can be checked by inspecting events flowing throughout the system, typically originating from end-devices and reflecting arbitrary actions, changes in state or sensing. Such events typically reflect the behavior of the overall IoT system – they may indicate executions which satisfy or violate its requirements. This paper presents a service-based software architecture and technical framework supporting runtime verification for widely deployed, volatile IoT systems. At the lowest level, systems we consider are comprised of resource-constrained devices connected over wide area networks generating events. In our approach, monitors are deployed on edge components, receiving events originating from end-devices or other edge nodes. Temporal logic properties expressing desired requirements are then evaluated on each edge monitor in a runtime fashion. The system exhibits decentralization since evaluation occurs locally on edge nodes, and verdicts possibly affecting satisfaction of properties on other edge nodes are propagated accordingly. This reduces dependence on cloud infrastructures for IoT data collection and centralized processing. We illustrate how specification and runtime verification can be achieved in practice on a characteristic case study of smart parking. Finally, we demonstrate the feasibility of our design over a testbed instantiation, whereupon we evaluate performance and capacity limits of different hardware classes under monitoring workloads of varying intensity using state-of-the-art LPWAN technology.

1 Introduction

Our life is increasingly dependent on the correct functioning of complex distributed systems, which are expected to operate in compliance to their requirements. Such is the case within contemporary pervasive systems, as the ones induced by Internet of Things (IoT) deployments, composed of resource-constrained devices, edge and cloud services alike. Software components within such distributed systems typically produce events, in terms of which overall system requirements describe intended behaviors. If a system's runtime operation is found to violate them, corrective actions may need to be taken.

Runtime verification (RV) is a lightweight verification technique based on observing information from a system while in operation, and identifying if the observed behaviors satisfy or violate certain properties. It has emerged as a practical application of formal verification, checking properties upon a sequence of events arising from system execution, thus scaling well in systems involving complex and high-throughput events. It is particularly useful when exhaustive design time verification is impractical or infeasible, or a system's formal model is difficult to construct. Monitors are instead constructed from formal property specifications, which detect if an incoming event sequence violates them in an online manner.

Within an IoT setting, events typically originate from low-end devices and flow throughout the system to software components in charge of processing them. Such events may reflect arbitrary actions, changes in state or, quite importantly, be originating asynchronously from the external environment recognized by software, commonly referred to as *sensing*. Events within an IoT system typically define a behavior of the overall system – events may indicate system executions which satisfy or violate its requirements. Supporting such validation in practice however, is challenging. Firstly, the sheer number of devices and their heterogeneity inherent in IoT systems require dedicated software architectures – typical cloud-based deployments are often not applicable. Secondly, communication particularities of the IoT domain need to be taken into account, since devices may be connected through a plethora of networking technologies exhibiting different characteristics, such as low-power wide-area networks (LPWAN). Finally, the volume and velocity of the events generated in realistic IoT systems can saturate network links and centralized processing schemes.

To address these difficulties, this paper presents a service-based software architecture and technical framework supporting runtime verification for decentralized edge-intensive systems. Monitors are deployed on edge components, receiving events originating from end-devices and other edge nodes. The properties expressing desired system requirements are evaluated on each edge monitor in a runtime fashion. The system exhibits decentralization, since property evaluation occurs locally on an edge node, while evaluation verdicts possibly affecting satisfaction of other properties on other edge nodes are propagated accordingly. We assume that requirements to be satisfied by the system under design are specified in terms of assertions in a temporal logic. We subsequently leverage results on runtime verification [1], [2] and devise a practical distributed systems architecture and framework that can support evaluation of properties. Our framework achieves decentralization in two dimensions: (i) events are evaluated locally within the scope of an edge node, avoiding central or cloud-based collection that can incur cross-network overhead, and (ii) properties evaluated in edge nodes that affect satisfaction of others are propagated

• *C. Tsigkanos, P.A. Frangoudis and S. Dustdar are with TU Wien, Austria. M.M. Bersani is with Politecnico di Milano, Italy.*

throughout the hierarchically structured system.

Systems we consider are composed of i) (possibly resource-constrained) edge computers placed near ii) sensing end-devices, as well as potentially iii) cloud infrastructure. We advocate decentralization, as the edge is a first-class entity in our approach, responsible for evaluating properties on events originating from IoT devices within its scope (such as a local administrative domain or wireless network) but bearing no dependencies besides events needed for checking in other edge nodes or the cloud. Adopting the classification of [3], our monitoring approach can be characterised as follows. Firstly (i), we utilize state-based specification, in the sense that previous values are used to compute some current actions to be taken. Secondly (ii), we adopt a discrete view on time, meaning that values of an output stream depend on certain previous values of some other stream; this is in contrast to sliding window approaches, which target aggregation of events over a fixed period of real time. Finally (iii), computation of a verdict may not depend on the arrival times of input values, in an asynchronous manner: in the widely deployed and volatile systems we target, sensing devices may operate at different frequencies (e.g., due to different energy cycles), and thus events received do not necessarily arrive at synchronized rates [4].

Our framework utilizes Metric First-Order Temporal Logic (MFOTL [2]), a formalism operating on traces of events. Our motivations for choosing the formalism are (i) its expressiveness, as MFOTL has seen applications on systems ranging from financial to cyber-physical, and (ii) the fact that it is well-defined and has well-studied theoretical semantics and properties, both rendering practical adoption and tool support easier. We contextify the framework advocated within the wide domain of IoT monitoring, but stress particularly runtime verification as its technical domain.

To provide concrete evidence of the applicability of the proposed architecture and technical framework, we first illustrate how runtime verification can be achieved in practice on a case study of a spatially-distributed parking system in a smart city. We then demonstrate the feasibility of our design to operate in resource-constrained edge computing environments over a testbed instatiation. Thereupon, we evaluate performance and capacity limits of different hardware classes, from small single-board computers (SBC) to server-class data center hosts, under monitoring workloads of varying intensity, for end-devices communicating using state-of-the-art LPWAN technology [5].

The rest of the paper is structured as follows. After setting the stage with a characteristic example, Sec. 2 describes the design of a system architecture to support runtime verification. Sec. 3 outlines property specification, and Sec. 4 describes the monitor as the basic architectural building block in detail. Sec. 6 first provides an applicability assessment over a smart city scenario, before presenting a testbed, whereupon performance, capacity and latency of the proposed solution are investigated. Related work is considered in Sec. 7, and Sec. 8 concludes the article.

2 MONITORING ARCHITECTURE

Monitoring events from wide IoT deployments requires a dedicated software architecture, capable of coping with large volumes and velocities of events, as well as heterogeneous devices for deployment, and able to accommodate the different particularities of networking technologies and overall setting. To this end, this section first illustrates a characteristic motivating scenario. Subsequently, design requirements for a monitoring architecture are distilled, before presenting its materialization.

2.1 Example IoT System

Consider the wide deployment of an environmental monitoring system. Sensing infrastructure deployed in remote and wide areas produces readings which are aggregated and processed both locally and globally. Sensors themselves may be heterogeneous, and may range from temperature and humidity to gas and biochemical. Those may be connected through low-power wide-area (LPWA) network technologies. Within our example, they may be deployed throughout a country-wide setting, performing sensing in remote forest and mountain regions [6]. We consider sensors as IoT devices, subject to small size, energy constraints and limited range – in particular, we consider (i) temperature sensors, (ii) wind-direction sensors and (iii) CO2 sensors. The objective of the overall system is to provide monitoring infrastructure to stakeholders in both local (e.g., region or municipality) and global (e.g., country government) levels. Such monitoring may be intended for declaring emergency situations due to extreme weather-related events, such as forest fires, as typically employed.[1] The exemplary system should alert in the following situations:

ER1 If CO2 and temperature sensors within a local area report on average readings over thresholds of 400ppm and 60 C respectively, a fire may be taking place and this must be reported to municipality emergency services.

ER2 Given a probable fire in a local area, if wind direction is towards an adjacent area where no fire is reported, the region's authorities need to be contacted within 10 minutes from the alarm, as the fire may spread.

ER3 If more than 3 regions report fires in the last hour, the country is in a state of emergency.

The system describes a characteristic case where decentralized monitoring is required. Edge devices in relative vicinity of IoT sensors typically serve as gateways, collecting and processing readings and exchanging data with the cloud. However, the volume, velocity and transmission cost of the data produced renders transmission to a central point (such as the cloud) impractical; one would seek to decentralize monitoring processing as much as possible, to avoid saturating communication channels, and only propagate information critical to levels upwards, while maintaining networking infrastructure cost minimal. Within our exemplar setting this is evident – sensors in the order of thousands may be deployed throughout remote forested regions, communicating over some low-power wide-area networking technology. Events coming from sensors should be processed locally in edge nodes and gateways, and only information resulting from the elaboration of sensors' data is propagated (e.g., when there is a fire alert). Finally, the actual system structure depends on the specific deployment considered (e.g., a country), meaning that the granularity of distribution and actual requirement specification should be

1. For an example, see meteofrance.com.

defined by the system designer, as nodes may be within local administrative domains.

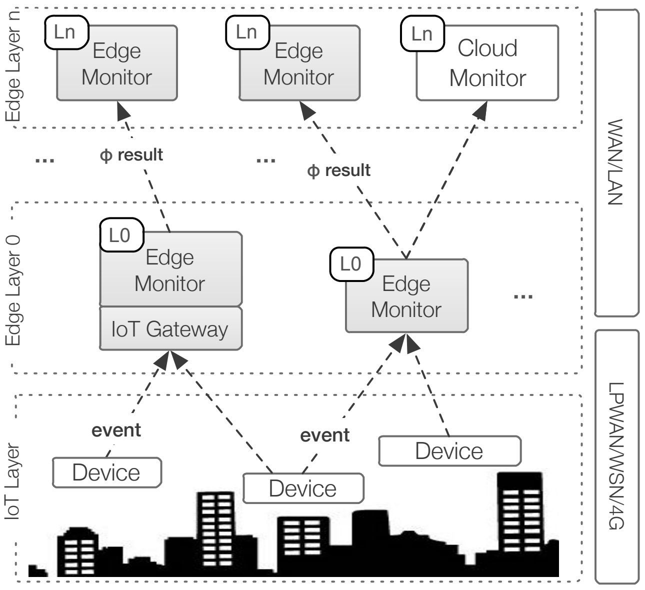

Fig. 1. Edge-based Monitoring Architecture. IoT devices in the range of local edge monitors emit events. Edge monitors are responsible for runtime verification and are deployed in a hierarchical structure, where verdicts of property evaluations are propagated.

2.2 Design Requirements

From a software engineering perspective, the distributed systems we target are made up of small form-factor IoT devices performing sensing, which live within wide spatial domains, and communicate over different networking technologies which should be accommodated uniformly. Moreover, the events they produce may be intermittent and concurrent, while evaluation of higher-order properties based on the sequences (i.e., traces) they induce should occur as timely as possible. We distill the following design requirements:

DR1 *Lightweight communication methods*. Events produced by devices typically involve a low payload per event. This is due to the low-powered, scarcely online devices that make up the system in its lower end. Furthermore, communication may be unstable – events (and thus datapoints) may arrive sparsely and intermittently.

DR2 *Interoperability*. Events propagation should adopt well-defined, lightweight APIs to ensure compatibility with heterogeneous devices talking over different communication technologies. Multiple devices may talk to a single endpoint, while maintaining open connections is atypical. This is because devices may wake up and report sensor readings, and then sleep again. Moreover, devices may move around, and be found in unknown networks or administrative domains. Loose coupling within the system is thus desired.

DR3 *Non-blocking event propagation*. Events may arrive at unknown rates, since no assumptions can be made about the device cycles – end-devices may decide to produce events at any time. This is exacerbated by LPWAN protocols typically employed in the IoT layer, which impose strict fixed *time on air* restrictions (e.g., LoRa, SigFox). As such, processing and propagation of events throughout the system should occur in a non-blocking, asynchronous manner. Events – leveraging loose coupling – should not block due to processing.

DR4 *Scalability*. Numerous sensing end-devices may constitute the system, something that requires handling increased throughput that may saturate single network links. Moreover, IoT devices are typically in the range of local edge gateways, themselves being resource-constrained – typically single-board computers. Thus, decentralization emerges as the way to dominate size, volume, and communication costs.

In the following, we describe the design of a monitoring architecture, tailored to satisfy the above requirements.

2.3 Edge-based Monitoring Architecture

IoT system architectures bridge the gap between the physical and the virtual worlds, but entail multiple challenging factors. Design of IoT architecture involves networking, communication, extensibility, scalability, and interoperability among heterogeneous devices [7]. These (end- or edge-) devices may be heavily resource-constrained, e.g., in terms of computational power and battery. As (possibly numerous) devices may span wide spatial domains, and produce events in real-time, an IoT architecture should be able to accommodate technology-agnostic event-based communication between heterogeneous devices in a decentralized manner. We contextualize our problem within a layered IoT architecture illustrated in Figure 1, consisting of:

Sensing IoT layer, referring to devices responsible for emitting events, which may be periodic or irregular.

Monitoring edge layer, of which there may be multiple; each delimits a set of edge nodes, receives events and may propagate evaluation verdicts to nodes in other layers. The set of devices immediately in range of IoT devices is referred as L0 (layer 0), the next in the hierarchy as L1, etc.

The architectural design described intends to accommodate the various networking technologies which may be found within the IoT setting. Low-power IoT devices typically employ low-power networking technologies. At the monitoring layer, edge nodes may be connected through high-throughput wired or wireless links, such as WLAN or 4G/5G. The common denominator is that lightweight services (in our case, services implementing monitoring functionality) can be used to establish communication throughout the architectural components. In fact, Service-Oriented Architectures (SOA) ensure interoperability among heterogeneous devices [7] making up the system, and abstract functionality as a set of well-defined services [8]. SOA has been widely applied as a mainstream architecture, for example in the context of Wireless Sensors Networks (WSNs) [9]. SOA applied to IoT provides extensibility, scalability, modularity and interoperability among heterogeneous software components; functionalities and capabilities are abstracted as a common set of services.

Regarding infrastructure and the overall design process, we contextify our approach after the system operator has performed dimensioning and allocation of network resources. Thus, we assume that the infrastructure is static and already configured, with monitors in fixed physical locations connected with links with known network properties.

Back to our running example, events are emitted by the temperature and CO2 sensors (Sensing layer), which transmit heartbeats with readings, either periodically or upon certain interrupts (corresponding to e.g., device power cycles). Those events should trigger the evaluation of the properties in the local area range in order to determine whether a fire property holds. The event travels across some wireless link to a local edge node (Monitoring layer), reaching the monitor in charge of processing it. Processing the event may entail another event produced by the edge node, forwarded as input to another edge node (in a higher order Monitoring layer).

3 Logic-based Specification

In this section, we introduce the logical language which is used within our framework for formal specification of requirements. The properties capturing those requirements are allocated to edge monitors and are evaluated automatically, based on the input events that each monitor receives. Logical languages are useful to reason with facts that can be endowed with a truth value and to learn new facts from known ones. In our work, we adopt the classic interpretation of truth, which is based on the Boolean two-valued set including only true and false. Basic facts are elaborated at the bottom level of the monitoring application, where IoT devices sense the environment to extract basic information that manifests through events occurring therein.

The adopted logical language is Metric First Order Temporal Logic (MFOTL [2]) with aggregating modalities. We select this language for several reasons. First, the language offers syntactical elements, called temporal modalities, that allow us to express temporal relationships between two (or more) events. Moreover, the available modalities also support metrics, that is, they can express timing constraints such as “event A follows event B by exactly one time unit”. Secondly, the language also includes first order quantifiers over a set of elements, so it is possible to write constraints such as “for any monitored area, if there is a fire, then notify the local authority”. Finally, aggregating modalities are part of the language, enabling “calculation” over sequences of events, such as counting, averaging or summation of values found within events. What follows is an informal presentation of logic-based specification as used in the scope of this paper, recalling definitions in [2].

The occurrence of events naturally entails an order (before/after) among them, and naturally associates every event with a time position that specifies the ordering. A finite sequence of elements in fact, inherently determines a bijection between the set of elements and a subset of the natural numbers. Every event can also be associated with a timestamp which indicates the exact time instant when the event occurs with respect to the origin of time, commonly associated with 0. For instance, consider the following simplified sequence showing events emitted when an IoT sensor within our example setting may be initialized:

$$(\mathsf{boot}; 4{:}5); (\mathsf{healthcheck}; 5); (\mathsf{heartbeat}; 7); (\mathsf{calibrate}; 10{:}2); (\mathsf{heartbeat}; 11); (\mathsf{heartbeat}; 14{:}3); (\mathsf{heartbeat}; 18); (\mathsf{calibrate}; 20){:}{:}$$

Temporal Modalities. Temporal modalities in MFOTL are those of Linear Temporal Logic (LTL [10]), namely, “neXt” (written X) and “Until” (written U). In the previous sequence, event healthcheck follows boot. Hence, when boot occurs, the fact “healthcheck holds next” is true. Based on this knowledge, the following implication holds in the sequence, $\mathsf{boot}\ )\ X(\mathsf{healthcheck})$. In fact, in every position where boot does not hold, the implication is true because the antecedent is violated; whereas in the first position, where boot does hold, the formula $X(\mathsf{healthcheck})$ is true, and so does the implication. The Until modality expresses a duration, i.e., the occurrence of an event until the occurrence of a different one. In the example sequence, it is true that event heartbeat holds “until” event calibrate, in every position. Formally, formula $(\mathsf{heartbeat}\ U\ \mathsf{calibrate})$ holds in the third position and from the fifth position afterwards. By definition, the same formula is true also at the fourth and eight position, where only calibrate holds.

Metric Modalities. MFOTL includes also metric modalities, allowing reasoning with timestamped sequences of events. Metric modalities are endowed with an interval of the $\mathbb{R}$eals that can be left closed/open (commonly indicated with ‘[’ / ‘(’) and right closed/open (commonly indicated with ‘]’ / ‘)’). For instance, formula $\mathsf{boot}\ )\ X_{(0;1)}(\mathsf{healthcheck})$ holds in (every position of) the sequence because event healthcheck occurs exactly one position, and earlier than one time unit, after boot. Formula $(\mathsf{heartbeat}\ U_{[1;6]}\ \mathsf{calibrate})$ is true at position 3, 6 and 7, because event calibrate occurs 3.2, 5.7 and 2 time units, respectively, after events heartbeat occurring therein; but the same formula does not hold at position 5 because event heartbeat is 9 time units earlier than event calibrate at position 8.

MFOTL includes the past version of metrics next and until, respectively “previous” (or Yesterday) indicated with Y, and “Since”, indicated with S. By using standard equivalences, until and since operators can be used to derive the following modalities: “eventually” (F) and “globally” (G) derive from until and mean, respectively, that formula holds eventually/always in the future; and “once” (O) and “historically” (H) derive from since and mean, respectively, that formula held once/always in the past. The metric variants are straightforward.

Aggregation Operators. MFOTL is further equipped with aggregation operators that allow one to express properties on counting (C) occurrence of events, summation (S) and averaging (A) of a certain numerical characteristic brought by the events in the sequence. Recall the scenario in Sec. 2.1 and consider the following sequence of measuring events called “co2” and “temp,” which respectively indicate readings parts per million (ppm) of Carbon Dioxide (CO2) and temperature obtained by sensors deployed in the monitored forest area. Events are of the form $(timest; a; e; v)$, where $timest$ is a timestamp, a is the identifier of a monitored area, e is either “co2” or “temp” and v is a numerical value for the measurement. A possible sequence of events occurring in the system can be the following trace (1):

$$\begin{array}{l}
(1{:}3; \mathsf{area}_1; \mathsf{co}_2; 370{:}6)\\
(2{:}8; \mathsf{area}_2; \mathsf{co}_2; 450{:}8)\\
(2{:}9; \mathsf{area}_2; \mathsf{temp}; 40{:}6)\\
(3{:}2; \mathsf{area}_1; \mathsf{temp}; 22{:}6)\\
(4{:}5; \mathsf{area}_2; \mathsf{co}_2; 790{:}2)\\
(4{:}9; \mathsf{area}_2; \mathsf{temp}; 52{:}2)\\
(7{:}5; \mathsf{area}_1; \mathsf{co}_2; 390{:}6)\\
(8{:}2; \mathsf{area}_1; \mathsf{temp}; 22{:}9)\\
(10{:}9; \mathsf{area}_2; \mathsf{temp}; 90{:}8)\\
(11{:}2; \mathsf{area}_1; \mathsf{temp}; 22{:}4)\\
(11{:}5; \mathsf{area}_2; \mathsf{co}_2; 15000)\ {:}{:}{:}
\end{array} \tag{1}$$

The atomic MFOTL formulae that can be interpreted over trace (1) are $\mathtt{co2}(a, v)$ and $\mathtt{temp}(a, v)$, being a and v free–non quantified– variables. For instance, the interpretation of $\mathtt{co2}(a, v)$ at position 2 is the pair $(\mathsf{area}_2, 450.8)$, whereas no interpretation can be given for $\mathtt{co2}$ at position 3. The evaluation of $\mathtt{co2}(\mathsf{area}_2, v)$ at position 2 is the value 450.8.

Using aggregation operators, we can incrementally build MFOTL formulae capturing the requirements of the example of Sec. 2.1 – formulae will be evaluated in event traces of the form of trace (1). The following Formula (2), using the aggregating operator A, determines the average value of CO2, indicated with symbol av, in a given area area_2 over the last hour (av is actually a fresh non quantified variable).

$$[A_v(O_{[0,3600]}[\mathtt{co2}(\mathsf{area}_2, v)])](av). \quad (2)$$

The evaluation of the Formula (2) is carried out for every element of trace (1), and the result of every evaluation is a value av corresponding to the average of all the readings of CO2 in the last hour for area_2. The time window is determined with respect to the timestamp specified in every event. To compare the average with a threshold, Formula (2) can be combined with a formula of the form $av > T$, where T is a constant value. The resulting Formula (3) has therefore a boolean value, and can be used to express whether the detected CO2 level is exceeding a threshold T of 400ppm:

$$[A_v(O_{[0,3600]}[\mathtt{co2}(\mathsf{area}_2, v)])](av) \wedge (av > 400) \quad (3)$$

Finally, assume that formulae $_{\mathrm{CO}_2}(a, t)$ and $_{\mathrm{temp}}(a, t)$ express, respectively, that the average amount of CO2 and temperature in area a over the last hour is greater than threshold t. Given some thresholds for CO2 $T_{\mathrm{CO}_2} = 400$ ppm, and for temperature $T_{\mathrm{temp}} = 60$ C, we can concretely specify the requirement ER1 of Sec. 2.1, where $\mathsf{re}(a)$ means that the incidence of fire is notified to emergency services of the considered area a. The universal quantifier expresses the constraint for every area a.

$$\textbf{ER1}: \forall a.\left(\begin{array}{c} _{CO_2}(a, T_{\mathrm{CO}_2}) \\ \wedge \\ _{temp}(a, T_{\mathrm{temp}}) \end{array}\right) \Rightarrow \mathtt{fire}(a)).$$

The requirement corresponding to the regional level can be expressed on top of Formula ER1, given the outcome of the evaluation defining event re. Formula ER2 makes use of event $\mathtt{wind}(a_1, a_2)$ that indicates the presence of wind from a generic area a_1 to area a_2 if the two are adjacent. Observe that the formula can be instantiated for every region – as such, all the areas that are quantified by the universal quantifier belong to the same region R, known a-priori (a constant value in the formula).

$$\textbf{ER2}: \forall a_1.\forall a_2.\left(\begin{array}{c} \mathtt{fire}(a_1) \\ \wedge \\ \mathtt{wind}(a_1, a_2) \\ \wedge \\ \neg\mathtt{fire}(a_2) \end{array}\right) \Rightarrow F_{(0,600)}(\mathtt{fire\_region}(\mathsf{R})).$$

Finally, the requirement describing the country-wide predicate is captured by Formula ER3, where if more than 2 regions report fires in the last hour, a state of emergency is declared.

$$\textbf{ER3}: \forall r_1.\forall r_2.\left(\begin{array}{c} O_{(0,3600)}\mathtt{fire\_region}(r_1) \\ \wedge \\ O_{(0,3600)}\mathtt{fire\_region}(r_2) \\ \wedge \\ r_1 \neq r_2 \end{array}\right) \Rightarrow \mathtt{emergency}.$$

4 MONITORING

Given a specification of requirements, in this section we discuss how a functional unit wrapping suitably a verification tool for execution in an edge setting can be devised. Subsequently, we outline a reference implementation indicating technological choices that may be used for its instantiation.

4.1 Monitoring Package

The basic building block of the decentralized monitoring approach we advocate is the *monitor* – the functional unit that receives events, verifies if they violate some stated property, and if needed, propagates results to other monitors (i.e., in higher layers of Fig. 1). This abstract functionality can be exploited to design a *grey box* monitoring package, which constitutes a single unit that must be deployed on edge nodes in range of IoT devices. The monitoring package encapsulates a runtime verification process, exposed through an API. This grey box configuration can be easily replicated for multiple types of logics and verification procedures, hence making the monitor a flexible basic block for complex architectures.

Figure 2 illustrates the key components of the monitor package, as well as dataflows inherent in it. The monitor is deployed on some edge node, which interacts with it through a lightweight API, whereby all functionality is exposed. Initially, a property to be monitored ( ) is registered to the monitor, before the monitoring procedure (as the *monitor runtime*) is initialized, utilizing an external runtime verification tool. Afterwards, the monitor is ready to receive and process events from IoT devices. IoT devices may submit events through the API. Received events – recall that they may be intermittent and typically arrive in unknown rates – are kept in an in-memory cache. This design choice ensures non-blocking communication (from the point of view of the devices), as events are simply stored in the in-memory queue and the API transaction is kept minimal. The monitor runtime consumes events from the cache and submits them for verification to the external runtime verification procedure ("verifier" in Fig. 2). Processing an event may trigger a change in the state of satisfaction of the monitored property . If this occurs, the value of may need to be propagated to other monitors living in other edge nodes, through lightweight API methods. This way, any type of complex analysis (including other external verification tools) can be encapsulated, albeit using different event models and property formats. We consider the selection of a particular model checker, event format and property to be orthogonal to our approach, as this would be defined per application and type of reasoning required.

Finally, recall that the overall monitoring package of Figure 2 can be adopted to compute different properties, even for the same event flow produced by a single device – e.g., for every sensing IoT device in the scenario of Section 2.1. From a deployment perspective, monitoring functionality is hosted at edge nodes and exposed as lightweight services. This renders the overall monitoring architecture *recursive*, in the sense that monitoring packages (of the same type) are deployed at different hierarchy levels within the system – every layer computes a specific class of properties that cannot be determined by the lower layers only. More advanced arrangements reminiscent of complex event processing may thus be additionally investigated [11].

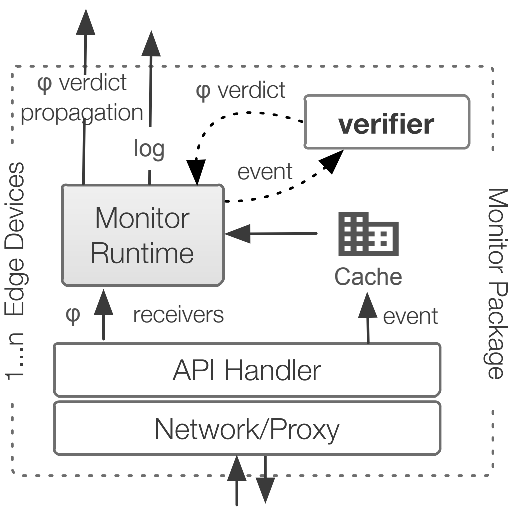

Fig. 2. Edge monitoring package and its dataflow. Events arrive over a lightweight API and are cached before being pushed to a monitoring procedure; verdicts (ϕ) are propagated to other monitors.

4.2 Monitor Instantiation

The grey-box design of Figure 2 can be implemented in different ways, depending on the demands of particular applications. Our approach intends to support different applications; those may require different types of properties, different logics supporting reasoning, or system deployments on diverse edge nodes. In the following, we outline a reference implementation, solely indicating technological choices that may be used to instantiate the monitor package.

API and network communication methods. Given DR1, we advocate lightweight methods such as REST, which are able to be implemented in a wide array of IoT devices or IoT gateways (on one end), and also more resourceful edge nodes (on the other end). A typical setup we highlight is composed of end-devices communicating over some e.g., LPWAN to a gateway. The network gateway collects events from various devices over its radio interface, and encapsulates them in IP packets. Event data are eventually delivered to monitors as REST payloads.

Events cache. For compliance with DR3, events emitted by devices should be essentially fire-and-forget – events should propagate throughout the system without blocking – barring network overhead. To this end, on the edge nodes, we recommend data structures implementing in-memory key-value stores where insertion operations are asymptotically done in constant time ($O(1)$).

RV interfacing. Faithful to our grey-box approach, different runtime verification tools and logics may be supported. We especially point out that different conceptions of time (and its reasoning) exist within literature, offering different assurances on property satisfaction with respect to an unknown (in principle) event trace. For example, handling out-of-order events (an event with an earlier timestamp arriving after an event with a later timestamp) requires fundamentally different notions of time (and report of verdict). We distinguish two general categories; approaches that operate upon *windowing* of event traces and ones which do not, such as [12]. In the former case, a moving window is maintained, where out-of-order events received can be rearranged before submitting them to a verification procedure. In practice, this occurs within the cache (Fig. 2). In the latter case, out-of-order events are either discarded or treated (i.e., rewritten) as just-in-time events. In both cases, the responsibility lies within the *monitor runtime* software component, which is responsible for presenting the external RV procedure (and tool) with the appropriate event format.

Multi-tenancy. Since we adhere to SOA principles, we can define arbitrary monitoring designs, where monitor packages are deployed in various hosts – but also upon the same host. Thus, multiple properties may be evaluated on a single edge node. Moreover, since the monitoring package is a service, it can be readily migrated (e.g., through containerization), while the mode of interaction (e.g., a web API) or API endpoint (e.g., through some localized proxying) stays unchanged. A consequence of observing such SOA principles is that decentralization emerges, also to the tightly bound property specification and evaluation phases.

5 Device Connectivity

In the following we discuss the networking context within the Sensing/IoT layer of our architecture (see Fig. 1). The IoT layer is integrated with end devices and is responsible for emitting and delivering events to some L0 monitor. Subsequently, we illustrate a concrete network pipeline.

5.1 Networking technology selection

We focus on wide area IoT services where key requirements include long range and extended battery lifetime, and thus very low power consumption. For this class of services, LPWAN has emerged as the major connectivity solution of choice. LPWAN protocols are designed to connect massive numbers of low-end battery-powered devices, such as sensors attached to micro-controller units, for delay-tolerant applications that require low throughput per device, such as for environmental monitoring, smart agriculture and smart city services. Multiple competing LPWAN technologies exist [13]. At the moment, the market is largely dominated by LoRaWAN and NB-IoT [14].

LoRaWAN operates on top of the proprietary LoRa physical layer, and it is an open specification maintained by the LoRa Alliance. The main components of the LoRaWAN architecture are end devices, gateways, network servers and application servers. On the other hand, NB-IoT is a cellular-based alternative. It is standardized by the 3GPP and can coexist with 4G LTE and 5G networks. End devices attach to cellular base stations, have IP connectivity, and can directly address remote IoT application components.

Technical details aside, which are beyond the scope of this article and have been thoroughly treated in the literature [15], [16], a key difference between the two technologies lies in their mode of operation: LoRaWAN uses unlicensed spectrum (ISM bands), while NB-IoT is a cellular technology, subset of the LTE standard. Therefore, LoRaWAN makes it possible for the IoT service provider to deploy and manage full end-to-end private networks, while NB-IoT mandates a subscription with a network operator. In this work, we have selected LoRaWAN as the underlying device connectivity scheme for the following reasons [5].

- **More flexibility in the utilization of edge computing resources:** A private LoRaWAN network opens up a

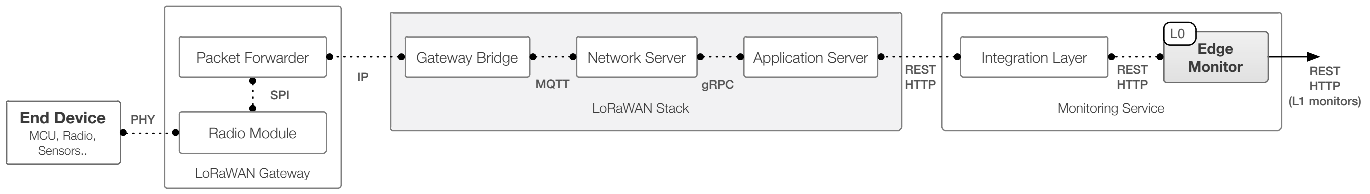

Fig. 3. Components, interfaces, and service invocation pipeline from device to monitor. Events generated at the end device level, and broadcast over the LoRa physical layer, are received by LoRaWAN gateways and pushed through the LoRaWAN stack until they are consumed by a L0 monitor.

richer set of monitor deployment options. Most importantly, the service provider can select to deploy monitors at the edge, either co-located with the LoRaWAN gateway or at other on-premise edge hosts. On the contrary, with NB-IoT and other cellular alternatives, unless there is an operator-based edge cloud deployment in place, as specified by the recent ETSI Multi-access Edge Computing (MEC) standards [17], device data have to go through the operator's core network before they are delivered to an L0 monitor. This limits edge deployment options and cancels potential traffic savings due to data aggregation at L0 monitors. ETSI MEC is expected to significantly enhance IoT services [18], [19], but, as of this writing, MEC technologies are still maturing and commercial MEC offerings by operators are practically inexistent.

Reduced capital and operational expenses: Despite the investment necessary for the installation and operation of LoRaWAN gateways, the operational expenses associated with NB-IoT subscriptions may dominate after only a short time of operation. Also, NB-IoT end devices are slightly more expensive than their LoRaWAN counterparts due to the increased protocol complexity and the need for a SIM card per device.

We should note that NB-IoT comes with its own advantages. It relieves the service provider of the managerial overhead of operating the network infrastructure and, contrary to LoRaWAN, it brings an IP connectivity endpoint directly to the IoT device. These may make it more attractive for some application scenarios and service providers. The choice among connectivity technologies is left to the service provider, noting that that our service design is *networking technology-agnostic*. If the application requires it, alternative network topologies and technologies can be supported. For example, when the required device throughput is high, as would be the case for multimedia-oriented IoT services, or if ultra-reliable and low latency communication is mandated, typical of vehicular IoT, standard 4G or 5G connectivity could be more suitable.

5.2 LoRaWAN-based network pipeline

Figure 3 illustrates the pipeline that events have to go through from an end device to a (first-level, L0) monitor over LoRaWAN access. We follow the lifecycle of an event from its generation. The end device is generally assumed to be highly resource-constrained in the range of a microcontroller (MCU), equipped with sensing and radio capabilities. End devices periodically (due to sleep frequencies) or on-demand, and in an asynchronous fashion, broadcast event data over the LoRa physical layer. Gateways equipped with LoRa radio modules (concentrators), receive data frames from devices in range. Typical LoRaWAN concentrator chipsets provide a Serial Peripheral Interface (SPI) for communication with the host platform. Gateways decode frames received by end devices and propagate them upwards to the network server via a packet forwarder software module over an IP connection.

The network server implements Medium Access Control (MAC) and the upper layers of the WAN stack. This includes functionality such as downlink transmission scheduling and device data rate configuration, but also frame deduplication and security functions. Importantly, the network server is responsible for routing uplink data to the subscribing application servers. This takes place often via lightweight protocols such as MQTT or gRPC. The application server can be considered the uppermost layer of the LoRaWAN stack and in typical implementations it is a lightweight service offering basic customer server logic, i.e., application-layer encryption and integration with the rest of the IoT application components. Often, as is also the case for our service, an integration layer is necessary to relay the data from the application server to the IoT service component (i.e., the monitor), including the necessary API translation. Events are finally delivered over REST to the edge monitor, which may in turn push evaluations to the upper L1 layer. Fig. 3 accurately reflects the network and service components, and the protocols used in our testbed implementation (Sec. 6.2).

6 EVALUATION

To provide tool support for our decentralized monitoring framework, we realized a prototypical monitoring package adapting MONPOLY [2] (an offline verification tool), in order to able to verify MFOTL properties in an online manner. Thereupon, we evaluate our approach over a smart parking scenario; the experimental setup and results obtained are subsequently presented. We conclude with a discussion. Our evaluation goals are two-fold; we seek to investigate (i) *applicability* of the proposed solution, in terms that the architecture, system and logic used are able to be used in practice (Sec. 6.1), and (ii) *performance* in realistic settings (Sec. 6.2). The former entails considering a real scenario. The latter requires measurements of propagation of events that include both processing (by edge monitors) but also networking overhead, over various workloads.

6.1 Case Study: Parking availability in LA

Parking IoT devices within Los Angeles, USA are equipped with sensors, and produce readings when a car enters or leaves the corresponding parking spot [20]. We consider LA to be divided into spatial regions, representing districts, such as "FashionDistrict", or "KoreaTown", which are grouped within larger partitions of the city, such as "EastLA", "DownTownLA", etc. We proceed to model the LA parking

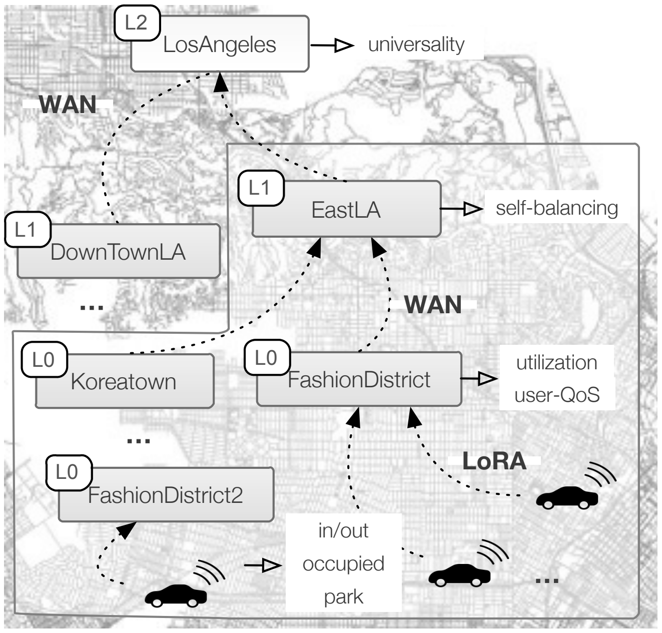

Fig. 4. Monitoring scenario over Los Angeles; events are emitted through LoRa from IoT devices to local edge nodes (in grey). Those in turn propagate property evaluations to a city-wide node (top). The box denotes the architecture-critical levels L0 and L1 considered for evaluation.

setting as a monitoring scenario where the objective is to monitor parking availability at runtime. We incrementally build complex properties capturing requirements that a city operator is typically interested in, namely:

(*utilization*) The number of occupied parking spots in a district is less/more than the average occupied places within some specified time.

(*user-QoS*) The average time to find a parking spot in an district is less/more than some specified time, representing some quality-of-service (QoS) desired for the end user.

(*self-balancing*) Within city partitions, an under-occupied district adjacent to an over-occupied district tends to become equally- or over-occupied in the next future.

(*universality*) Previous requirements should hold for the whole city, across its districts and partitions.

Notice that not only those requirements are non-trivial, but they also predicate about districts, groups of districts (i.e., city partitions) as well as the whole city, rendering the overall monitoring scenario rather challenging.

We consider each district to be within the range of an edge node, to which events from IoT parking sensors may be emitted at any time. Figure 4 illustrates various edge nodes over a map of LA. We assume that parking events are emitted over some LPWAN channel and, following the dataset available [20], are modeled as tuples of the form $(timest, a, parkid, carid)$ where a) $timest$ is a natural number that represents a time instant, b) action $a \in \{\mathtt{in}, \mathtt{out}\}$ represents the act of entering or leaving a parking spot, c) $parkid$ uniquely identifies a parking spot, and d) $carid$ is an identifier of a car. A tuple $(timest, a, parkid, carid)$ signifies that at time $timest$, car $carid$ entered, or left, the parking spot identified with $parkid$.

Specification strategy. The strategy for capturing the scenario requirements is as follows. Firstly, the evaluation of non-trivial properties demands for the definition of sub-properties that are expressed with respect to simpler facts and basic events, thus allowing for compositionality and abstraction – a designer may re-use elementary sub-properties to specify higher-order ones. Secondly, sub-properties are monitored in nodes lower in the edge hierarchy, making evaluation results available to nodes in upper levels. The following basic relations are essential to express such complex properties; the corresponding formulae make use of the constant value $\mathbf{ns}$, that indicates the absence of spots in the parking area (a car c entered a parking area with no available spots–$\mathtt{in}(c, \mathbf{ns})$– and left it shortly after–$\mathtt{out}(c, \mathbf{ns})$). We write φ_n to refer to the formula identified by number n.

Formula (4) expresses the ternary relation $\mathtt{Park}$ between cars, parking spots and timestamps (variables c, p and t are non quantified). Intuitively, $\mathtt{Park}$ indicates that a car has currently been occupying a parking spot since the time instant specified by the timestamp, i.e., the car has not left the spot since the moment it parked. In particular, a tuple $(carid, parkid, timest)$ belongs to $\mathtt{Park}$ if the car identified with $carid$ occupied the parking spot numbered with $parkid$ (in which case $parkid$ is not $\mathbf{ns}$) at time instant $timest$, and the car has not left the parking spot since that moment, when the car took the spot. In Formula (4), predicate $\mathbf{ts}$ is satisfied by all the valid timestamps, hence the evaluation of $\mathbf{ts}$ on a tuple $(timest, a, parkid, carid)$ is $timest$.

$$((\neg\, \mathtt{out}(c,p)\ S\ (\mathtt{in}(c,p) \wedge \mathbf{ts}(t))) \wedge \neg\,(p = \mathbf{ns}) \qquad (4)$$

Subsequently, Formula (5) expresses the unary relation $\mathtt{Occupied}$ on parking spots (only variable p is not quantified). Intuitively, $\mathtt{Occupied}$ indicates that a parking spot is currently not available, i.e., it has not been released since the last time a car took it. In particular, a parking spot identifier $parkid$ belongs to $\mathtt{Occupied}$ if the parking spot numbered with $parkid$ (in which case $parkid$ is not $\mathbf{ns}$) was occupied (hence, there exists a car that took the spot) and there is no car that has left the parking spot since that moment, when the spot was occupied.

$$(\neg\, \exists c.\mathtt{out}(c,p)\ S\ \exists c.\mathtt{in}(c,p)) \wedge \neg\,(p = \mathbf{ns}) \qquad (5)$$

District properties. By using the previous predicates $\mathtt{Park}$ and $\mathtt{Occupied}$, we can define the formulae that L0 edge nodes installed in districts evaluate. Formulae (6) and (7) express two different properties: the first is a counting property referring to the parking spots in an area, whereas the second is a timing property involving the average duration of parking. Formula (6) holds when the number of occupied parking spots in a region is $\bowtie$ than the average occupied places throughout the last T time units, with $\bowtie\ \in \{<, >\}$.

$$\exists a \exists c. \left(\begin{array}{c} [A_x(O_{[0,T]}[C_p \mathtt{Occupied}(p)](x))](a) \\ \wedge \\ [C_p \mathtt{Occupied}(p)](c) \wedge (a \bowtie c) \end{array} \right) \qquad (6)$$

Finally, Formula (7) holds when the average parking time, throughout the last T time units, is $\bowtie$ than the constant T_{avg}.

$$\exists a. \left(\begin{array}{c} [A_d(O_{[0,T]}(\exists t \exists t' \exists c \exists p \left(\begin{array}{c} \mathtt{out}(c,p) \wedge \mathbf{ts}(t') \\ \wedge \\ Y(\mathtt{Park}(c,p,t)) \\ \wedge \\ (d = t' - t) \end{array} \right)))](a) \\ \wedge \\ (a \bowtie T_{\text{avg}}) \end{array} \right) \qquad (7)$$

City Partition Properties. Not only requirements over single districts exist; in fact, requirements may predicate about availability of parking in multiple districts across a city partition – monitoring nodes deployed in partitions constitute L1 edge nodes. In the following, we take advantage of the specification of previous Formulae (6) and (7), whose evaluations are emitted by nodes at layer L0. Let ${}_{(6)}(z)$ and ${}_{(7)}(z)$ be two predicates respectively associated with Formula (6) and Formula (7), z be a free variable indicating the district in LA to which the formula refers, and be a relation in $f<;>g$ defining the order relation expressed by the formula. The intuitive meaning of ${}_{(6)}(z)$ is that the area z is over/under-occupied with respect to the average number of occupied spots; the meaning of ${}_{(7)}(z)$ is that the area z is over/under -occupied with respect to the average parking time. At the edge layer L1, every node receives the evaluation value (either true or false, determined at layer L0) of both ${}_{(6)}(z)$ and ${}_{(7)}(z)$, for a given relation and for every district that is associated with the node.

Formulae (8) and (9) hold when the districts that are monitored by an edge node show a self-balancing trend with respect to the number of occupied places. In particular, this signifies that an under-occupied district z that is adjacent to an over-occupied district z^0 tends to become equally/over -occupied in the next future, within T secs, where T is a constant value. To keep description simple, we assume a specific geography of the areas in the city. This is with no loss of generality, as a more complex layout can always be addressed, provided that more complex formulae are considered. The assumption is that all the districts that are associated with a specific partition of the city (e.g., EastLA) are pairwise adjacent, and the flow of cars among them is relevant (i.e., cars can go from one to another).

Formula (8) refers to the EastLA zone (a similar one can be expressed for the others). We use the constants fd, kt and fd_2 to identify, respectively, the areas called FashionDistrict, KoreaTown and FashionDistrict2, and write ${}_{(6)}(z)$ as a shorthand for ${}^{>}_{(6)}(z)\ \_\ {}^{=}_{(6)}(z)$. The formula is an implication, whose antecedent holds for every time position of the event sequence emitted by node at layer L0. Intuitively, the formula is true when the following two facts hold: if both FashionDistrict and KoreaTown are over occupied then FashionDistrict2 will be over occupied in the next future; and if FashionDistrict2 is over occupied then FashionDistrict or KoreaTown will be over occupied too in the next future.

$$8i{:}tp(i)\ )\ \begin{array}{c}0\\B\\@\end{array}\begin{array}{c}{}^{>}_{(6)}(\mathsf{fd})\ \wedge\ {}^{>}_{(6)}(\mathsf{kt})\ )\ F_{[0;T]}\ {}_{(6)}(\mathsf{fd}_2)\\ \wedge\\ {}^{>}_{(6)}(\mathsf{fd}_2)\ )\ F_{[0;T]}(\ {}_{(6)}(\mathsf{kt})\ \_\ {}_{(6)}(\mathsf{fd}))\end{array}\begin{array}{c}1\\C\\A\end{array}\qquad(8)$$

Next Formula (9) refers to all the areas in a partition of the city, indicated with set $A(p)$ where p is a partition; e.g., $A(\mathsf{EastLA})$ is the set including KoreaTown and the two FashionDistricts. In particular, the formula holds when the parking time, for every area belonging to a partition, shows the same order relation $2\ f<;=;>g$ with respect to the average parking time evaluated in that area. Formula (9) is parametric with respect to p and relation ; and similarly to Formula (8), it is an implication, where the antecedent holds in every time position of the event log. The consequent is a conjunction, true if all the areas in $A(p)$ are over/under -occupied with respect to the average parking time.

$$8i{:}tp(i)\ )\ \begin{array}{c}0\\@\end{array}\ \bigwedge_{z\,2\,A(p)}\ {}_{(7)}(z)\begin{array}{c}1\\A\end{array}\qquad(9)$$

City-wide properties. Finally, at the top "LosAngeles" global node level associated with the entire city, the formulae are expressed in terms of predicates coming from the underlying edge layer (L1 in Fig. 4). The predicates are ${}_{8}(p)$ and ${}_{9}(p)$, where p is a free variable indicating a partition of LA. Let P be the set of partitions WestLA, EastLA, DownTownLA:

$$8i{:}tp(i)\ )\ \bigwedge_{p\,2\,P}\ {}_{8}(p)\ !\ ;\ \text{and}\ 8i{:}tp(i)\ )\ \bigwedge_{p\,2\,P}\ {}_{9}(p)\ !\qquad(10)$$

6.2 Testbed experiments

In this section, we present a set of experiments aiming (i) to demonstrate the feasibility of our design and implementation to operate in resource-constrained edge computing environments, (ii) to evaluate the performance and capacity limits of different hardware classes, from small single-board computers (SBC) to server-class data center hosts under monitoring workloads of varying intensity, and under different deployment strategies. We specifically focus on the lower end of the architecture of Fig. 4, where the critical interplay between resource-constrained end-devices and L0 runtime verification occurs over LPWAN.

6.2.1 *Event processing throughput*

We first focus on the capacity of a monitor in terms of the rate at which it is processing incoming monitoring events. We deploy our monitor package on three different hardware architectures, namely Raspberry Pi (RPi) Zero (1GHz single-core ARMv11 CPU, 512 MB RAM), RPi 3 Model B (Quad Core 1.2GHz Broadcom BCM2837 CPU, 1 GB RAM) and virtual machines running on server-class x86 hosts in a small-scale data center. We configure the monitor to verify the Occupied property (Formula (5)) and submit parking events to the monitor over its HTTP API endpoint.

In this experiment we are interested in evaluating the performance bounds of each candidate technology to host a monitor, and we thus focus on a single monitor *in isolation*. In order for our results not to be affected by the CPU resources necessary by the load generator, we generate events from a separate host connected with the monitor over a high-capacity network link where minimal other traffic is present. This way we also ensure that we are not hitting network-related bottlenecks. For the lower end RPi Zero, featuring only a Wi-Fi interface, its lower computational capabilities ensure that our results are CPU- and not network-bound. RPi 3 is connected with the load generation host over 1 Gbps Ethernet, while in our experiments over x86-based servers, events were emitted to the monitor VM from another VM co-located in the same data center host.

Figure 5 presents the achieved event processing throughput of a single monitor hosted on different edge node types under increasing event workloads. Beyond a specific workload level, up to which event processing throughput increases in a linear fashion, the monitor reaches saturation.

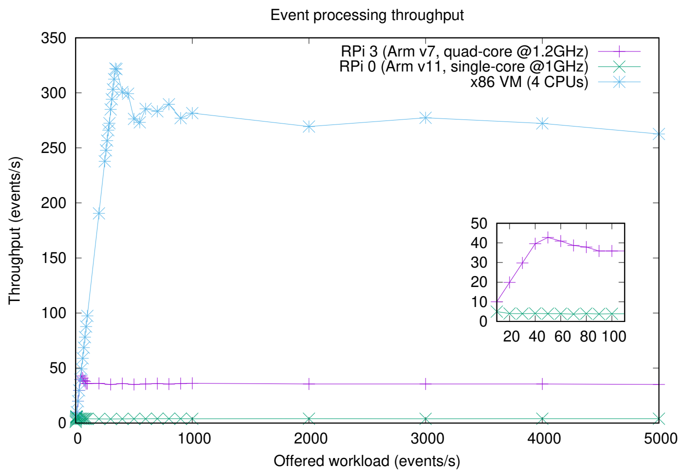

Fig. 5. Event processing throughput for different candidate edge host technologies under increasing workload.

We notice that an RPi 3 can keep up with a workload of up to approximately 45 events per second. While this performance may sound modest, we should note that in the IoT use cases we target, where end devices emit events over LPWAN infrequently (e.g., once per tens of minutes), this workload may translate to thousands of IoT devices. When the monitor is deployed inside a Cloud VM, we observe a significant performance increase, where a 4-CPU VM can handle workloads of up to 330 requests/s.

6.2.2 End-to-end latency

We now turn our attention to the latency perceived end-to-end, from emission (IoT device) to the (resulting) verification of a higher-level property. For this purpose, we set up an edge computing testbed that features all our technologies of interest, including a full LPWA network architecture and low-end SBC nodes hosting our monitor package and LPWA network service components.

Testbed setup. We have deployed a fully functional end-to-end LoRaWAN testbed with one end device and one gateway. Our end device is a Libelium WaspMote, featuring a Microchip RN2483 LoRa radio module on top of an ATmega1281 MCU. The host communicates with the radio module via UART, and a high-level software interface (e.g., to configure the radio, transmit and receive packets) is provided by the open-source Libelium libraries. At the gateway end we attached a Dragino PG1301 LoRaWAN concentrator with a Semtech SX1301 baseband unit and SX1257 RF front-end on a RPi 3. We used the ChirpStack[2] network server stack, which is an open source implementation of a LoRaWAN gateway bridge (for collecting data from multiple gateways), network server and application server. Fig. 3 presents the pipeline that data have to go through from an end device to a (first-level) monitor. Note that an invocation chain of a similar depth also characterizes other LoRaWAN implementations, such as The Things Stack[3]. We assume a two-level hierarchy of properties (L0 and L1) and define end-to-end latency as the time it takes from the moment an event is generated at an IoT device until the verification of the L1 property, as a result of the event, has taken place. Note that every event in the IoT layer corresponds to a change in L0 determined by the verification of the respective property. We measure end-to-end latency for a single event source (IoT device) in the presence of increasing workloads. Contrary to our throughput experiments, where we only considered the performance of the monitoring package in isolation, here we aim to also capture the overheads associated with all the networking and service-based verification components in the end-to-end path, such as the LoRaWAN stack. Therefore, we inject workloads in the form of LoRaWAN frames sent directly to the LoRaWAN server (i.e., bypassing the LoRaWAN gateway, but using the rest of the stack) using a LoRaWAN simulator[4] which we have appropriately extended.

The parking events whose latency we track are generated periodically and infrequently (every 30s) on a measurement host; they are timestamped and are sent to the end device over a serial connection. Our code running on the end device MCU encodes the messages in a concise wire format and transmits them over the LoRaWAN link. The L0 monitor is deployed according to the scenarios described in the following paragraph. The L1 monitor is deployed on the host where the IoT device events are generated, where the final timestamp is recorded after the processing of the event by the monitor has completed. Timestamping event generation and processing completion on the same host avoids time synchronization issues.

Deployment scenarios and latency performance. In order to explore the capabilities of different candidate technologies to support our design at the edge, we compare three different scenarios, each demonstrating a different configuration and trade-off between deployment cost and performance.

All-in-one: all components, from the LoRaWAN stack up to the monitor, are deployed on a single small form-factor device. Our reference device is an RPi 3 Model B.
SBC cluster: we assume the availability of a cluster of small SBCs as in [21], allowing for different components of our architecture to be deployed on dedicated SBC nodes. In particular, in the experiments presented, the full LoRaWAN stack is co-located with the LoRaWAN gateway running on a RPi 3; the frames received are then pushed to a monitor executed on an identical device.
Edge DC: This deployment scenario assumes a lightweight LoRaWAN gateway running on a single-board edge compute device, while the LoRaWAN stack and our monitor are deployed in separate virtual machines instantiated at an edge cloud data center. The latter may be managed by the network operator, by cloud providers, or may be within a city's administrative realm. This scenario is in line with recent works [22], [23] that study the interplay between LoRaWAN and edge clouds, and represents an edge-centric approach to the typical (cloud-centric) deployment strategy of LoRaWAN applications.

Figure 6 presents a latency comparison across the three scenarios. Each point is the mean of the end-to-end latency values of 100 event transmissions presented with 95% confidence intervals. In the "All-in-one" and "SBC cluster" experiments, load was injected from a host connected to the

2. chirpstack.io 3. thethingsstack.io

4. github.com/brocaar/chirpstack-simulator

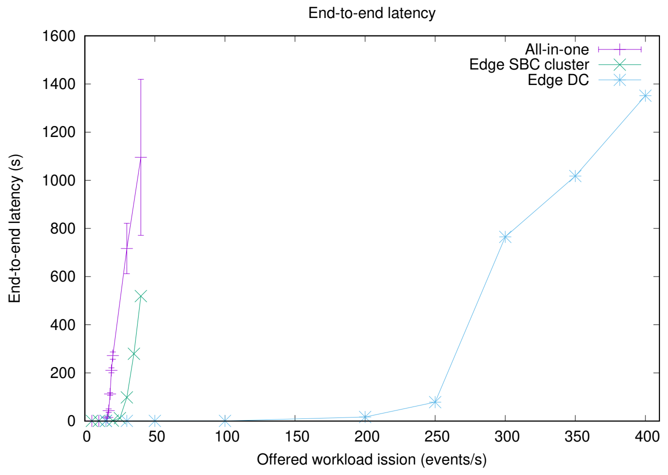

Fig. 6. Mean end-to-end latency (over 95% confidence intervals) as a function of event workload for different deployment scenarios.

same LAN over Ethernet. In the "Edge DC scenario," the LoRaWAN stack and the monitor are deployed in separate identical VMs, each with 4 CPUs and 8 GB RAM running on the same data center host, the latter with a capacity of 128 CPUs. A third VM with 2 CPUs on the same host was used for load generation.

Our results indicate that even an all-in-one deployment, which is the least costly, is capable of supporting workloads of up to 15 events/s with end-to-end latencies in the order of 1 s or less. A factor contributing to latency (something significant in lightly loaded settings) is the UART-based communication between the end-device MCU and the LoRaWAN radio. We carried out optimizations in Libelium's libraries and reduced this latency component to approximately 200 ms. While prior experiments (Fig. 5) showed that a monitor deployed at a RPi 3 device could sustain workloads of about 45 events/s, when we put the LoRaWAN stack in the picture this capacity drops: deploying this stack on SBC comes with unacceptable latencies of hundreds of seconds when load exceeds 30 events/s, even when the LoRaWAN stack is deployed on a dedicated RPi 3. On the contrary, as the "Edge DC" curve indicates, deploying the components on edge cloud VMs makes latencies in the order of seconds to appear only for workloads exceeding 200 events/s.

6.3 Discussion & Limitations

As illustrated in the case study and experimental evaluation, the service-based architecture supporting runtime verification can be achieved in practice, fulfilling design requirements of adoption of lightweight communication methods, interoperability with a variety of devices, non-blocking event propagation and scalability (Sec. 2) in a real-world technological setting.

Modelling and specification of a complex, realistic case study as the one presented showed the applicability of our approach and the success of the design choice of adopting MFOTL as the target logical language. However, from our experience and considering the perspective of practitioners aiming to use our approach, property specification patterns, domain specific languages, and incremental property specification facilities would go a long way in supporting specification – a typical concern raised with application of formal methods in general systems settings.

Regarding distributed systems underpinnings of our approach, we note that (i) different conceptions of time (and its reasoning thereof) may be further adopted, and facilities to deal with (ii) out-of-order and (iii) loss of events critical to property satisfaction can be integrated. There is vast research on such topics and as such a plethora of options to employ in our grey-box design. We excluded data repair and complex event processing techniques in order to reduce complexity and load, instead focusing on architectural, feasibility and design aspects. Naturally, those design choices are coupled with the verification approach used, including the logical language and accompanying model checker. We note however that the primitives to integrate those to our framework (i.e., Fig. 2) are available and can be reused, namely a cache for incoming events (useful e.g., for a moving window rearranging out-of-order events due to radio collisions) and model checker interface. We identify investigating verification design choices, problem domains and logics as a major avenue of future work.

We demonstrated the feasibility of our design and implementation to operate in a resource-constrained edge setting. By evaluating performance and capacity limits of different hardware classes (from SBC to server-class hosts), we showed that the *edge* can be concretized in various hardware backends, with different latencies incurred. The obtained results are useful from a dimensioning perspective: they allow the application provider to plan the amount of computational resources (CPUs, SBC nodes) that should be allocated per monitor, based on the expected event workload, and scale them elastically to match workload variations in case that is desired by the use case or deployment scenario – monitors can be deployed in containers or VMs in edge clouds. Furthermore, we illustrated an end-to-end system integrating LPWAN and the effects this incurs on latency.

We note that the events we consider permeating a deployed system are uncorrelated and may come from heterogeneous sources, a difference from traditional stream processing approaches which assume homogeneous streams with possibly complex data structures. We view our work as complementary to those; we focus on architectural aspects, where monitors are deployed within an IoT system with particular characteristics. The systems we target are highly volatile, as devices may roam, appear or disappear and events emitted may be intermittent. Thus, we focus on opportunistic evaluation of properties within the IoT architecture, targetting large-scale and widely deployed systems. Moreover, our choice of lightweight communication protocols reflects the variety and modes of communication used to emit events to an edge node serving as the endpoint. We identify employing advanced strategies [11] defining how processing load may be distributed and how edge nodes interact as another promising avenue of future work.

Finally and regarding the overall design process, the proposed architecture can be instantiated in a variety of settings where monitoring is desired. Different IoT domains and system operators however may require different instantiations or configurations of the architecture proposed – a smart city application would have vastly different prerequisites from e.g., critical infrastructure monitoring. To this end, usage and domain models, along with appropriate integration of domain-specifics should be further investigated.

7 Related Work

We presented an architecture and technical framework for verification of temporal properties within IoT systems. Consequently, we classify related work into two major categories. First, we discuss relevant approaches tackling complex event processing and recent works operating upon data streams. Subsequently, we present an overview of the state of the art in runtime verification as applied in IoT contexts.

7.1 Complex event processing & Data Streams

The Complex Event Processing (CEP) community has long focused on tracking and analyzing streams of information and deriving meaningful conclusions [24] – successful applications include stock market trading systems, databases, internet operations, and fraud detection. A variety of languages can be used; from formal languages with strong guarantees, to more expressive programming languages allowing for richer verdicts or operations upon data streams. Naturally, there exists a trade-off between language expressiveness and guarantees it provides on the constructed monitors (see [3] for an extensive discussion). We adopt MFOTL as particularly fitting to our problem domain – metric properties with aggregates are typical within IoT applications, while retaining the benefits of a concisely defined language.

Stream processing typically focuses on cloud applications, extending the original database foundations with advanced features such as fault tolerance, adaptive query processing, or extended operator expressiveness, while striving for scalability and robustness (see [25] for a comprehensive overview). However, our context is not big data or horizontal scalability, and our volatile domain hampers robustness. Advanced techniques from stream processing systems like shedding (i.e., dropping events to cope with the load [26]) or elasticity [27] can be integrated within our framework. We assumed a hierarchical organization of edge nodes; multiple event sinks within non-hierarchical systems [28] can achieve highly decentralized systems and compositional applications. Moreover, we ignored management of out-of-order events [29] and their effect [30], rendering our approach opportunistic. Furthermore, hyperproperties relating multiple computation traces with each other have been theoretically investigated; in such properties, it is necessary to store previously seen traces, and to relate new traces to the traces seen so far. Properties of monitoring specifications such as reflexivity, symmetry, and transitivity can be used to reduce the number of comparisons and achieve more scalable monitoring with lower memory demands [31]. We identify integration of such further techniques as avenues of possible future work. Advanced approaches can target scalability by performing slicing upon the event stream, by identifying substreams that can be independently monitored, or by exploiting hash-based partitioning techniques from databases research [32]. We note that our opportunistic, lightweight method of dealing with events excludes data repair techniques [33] in order to reduce complexity and load. More advanced arrangements within complex event processing may be further investigated [11].

7.2 Runtime verification for the IoT

Runtime verification has recently been attracting attention in the context of the IoT. Inçki and Ari [34] focus on verifying the correctness of message exchanges in IoT systems where devices communicate over lightweight application-layer protocols such as CoAP. The authors propose a domain-specific event calculus for the specification of the network interactions in such a system drawing from Message Sequence Charts, and use CEP techniques for runtime verification. Notably, in another work by the same authors, this approach is applied for failure detection in a smart parking system [35]. Lee et al. [36] propose mechanisms to improve the efficiency of model checking at runtime, casting their work to an IoT context. Their approach applies to systems modeled as Finite State Machines and state reachability properties are verified; temporal semantics are not directly supported. Leotta et al. [37] propose a RV system applied to an eHealth use case which builds on the formalism of trace expressions, a more expressive form of LTL. The work focuses on IoT software quality assurance, aiming at facilitating the software development and testing phases by detecting bugs.

Tsigkanos et al. turn their attention to spatio-temporal properties of IoT systems, focusing on use cases such as location-based smart city applications. Relying on device trajectories, they infer models of spatially distributed systems [38], and apply spatio-temporal model checking at runtime to verify that QoS and other goals of such systems are met [39]. To deal with the computational requirements of model maintenance and reasoning at runtime, the authors propose their materialization as a set of cloud-based microservices [40]. In a similar context, Ma et al. [41] study a wealth of smart city requirements and propose SaSTL, a new spatio-temporal logic with increased expressiveness and the capability for spatially-parallelized runtime monitoring.

In a work that bears similarities with ours and with Industrial IoT (IIoT) as the application domain, Grochowski et al. [42] devise an external monitoring architecture whose purpose is to verify the safety and liveness of IIoT production processes. Metric Temporal Logic is used to express properties (runtime requirements) over execution traces of the monitored system. While the proposed design appears to be suitable for execution at edge and cloud computing environments, composing arbitrary complex IoT monitoring service hierarchies spanning the compute continuum, as well as connectivity aspects, are not treated in depth. Lesi et al. [43] also target IIoT scenarios, where they distribute automation/control functions of reconfigurable manufacturing systems across multiple local controllers. They use a variant of Stochastic Petri Nets for system modeling; models are verified at runtime for safety and liveness on real-time process and network measurements. To reduce response times, RV takes place at an edge host in the facility. Akili and Lorenz [44] address the RV problem in the context of collaborative embedded systems, where they equip each component (embedded system, e.g., a member of a robot fleet) with its own monitor. As in our case, the authors deal with distributed RV, where monitors collaboratively verify properties by exchanging messages. At the same time, components exchange messages to collaborate towards achieving global and group-wise goals. This behavior is modeled using Agent UML sequence diagrams, and monitors aim to verify its correctness via distributed RV. The authors tackle critical challenges such as real-time constraints and reliable message passing. The distribution of tasks and monitoring at different hierarchy levels shares our spirit of decentralization and our aim to offer different monitoring granularities and abstraction levels per layer. However, the design and implementation aspects of a distributed, service-based monitoring architecture based on RV, and the mapping of its components to the edge/fog/cloud continuum, which are at the core of our work, are not given consideration.

Overall, a common theme in many of the aforementioned works is the use of RV to attest the correct functioning of the IoT software, or of the underlying communication protocols and infrastructure. While our approach also supports such cases, our focus is on applying RV as a monitoring facility in the *application domain*. Furthermore, our work can be considered complementary to those that study different RV techniques and languages for IoT systems: with our grey-box design, we can support different logics and verification tools with minimal impact on the service architecture itself. Various approaches and RV tools exist in the literature [45], [46], [47]; as we show in Sec. 4, in order to support them, it would suffice to replace the MFOTL verification engine and provide the appropriate application model. Finally, and most importantly, none of the aforementioned works in the IoT domain address architectural and networking aspects in depth, and in particular deployment and operation across the device-to-cloud continuum.

8 Conclusion and Future Work

Within an IoT setting, events typically originate from end-devices and flow throughout the system; since events define some behavior, they may indicate satisfaction or violation of requirements. Supporting such verification in practice within IoT is challenging, where heterogeneity, lightweight communication and decentralization are key domain characteristics. We presented an architecture and technical framework supporting runtime verification where monitors are deployed on edge components, evaluating temporal properties. The monitoring architecture fulfills necessary design requirements appropriate for the IoT/edge setting, adopting lightweight communication methods, device interoperability and scalability. The architecture can be instantiated in a variety of settings; we further demonstrated the feasibility of our design and implementation to operate in a resource-constrained edge setting by evaluating performance and capacity limits of different hardware classes, and latencies that different designs entail.

In future work, we plan to address core distributed systems underpinnings in order to support domains where out-of-order and loss of events are critical. Moreover, investigation of other logics and verification tools will render the approach more widely applicable. Regarding deployment, we assumed that the edge topology is static. However, monitors on edge nodes may need to be scaled or migrated to comply with other constraints like energy, latency or moving IoT devices, introducing dynamicity. A promising avenue of future work can benefit from techniques developed within stream operator placement [11], [48], but accounting for the particularities of verification and the edge/fog domain. Finally, since event frequency, type, volume and velocity are factors that influence variation points in the runtime verification architecture, they warrant further study.

Acknowledgments

Research partially supported by Austrian Science Foundation (FWF) project M 2778-N “EDENSPACE”.

References

[1] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[2] D. A. Basin, F. Klaedtke, and E. Zalinescu, “The monpoly monitoring tool,” ser. Kalpa Publications in Computing, G. Reger and K. Havelund, Eds., vol. 3. EasyChair, 2017, pp. 19–28.

[3] H. Torfah, “Stream-based monitors for real-time properties,” in *Intl. Conf. on Runtime Verification*. Springer, 2019, pp. 91–110.

[4] F. Gorostiaga and C. Sánchez, “Striver: stream runtime verification for real-time event-streams,” in *International Conference on Runtime Verification*. Springer, 2018, pp. 282–298.

[5] P. A. Frangoudis, C. Tsigkanos, and S. Dustdar, “Connectivity technology selection and deployment strategies for iot service provision over LPWAN,” *IEEE Internet Comput.*, vol. 25, no. 1, pp. 61–70, 2021.

[6] W. Krüll, R. Tobera, I. Willms, H. Essen, and N. von Wahl, “Early forest fire detection and verification using optical smoke, gas and microwave sensors,” *Procedia Engineering*, vol. 45, pp. 584–594, 2012.

[7] S. Li, L. Da Xu, and S. Zhao, “The internet of things: a survey,” *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[8] L. Baresi, M. Garriga, and A. d. De Renzis, “Microservices identification through interface analysis,” in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2017, pp. 19–33.

[9] Q. Chi, H. Yan, C. Zhang, Z. Pang, and L. Da Xu, “A reconfigurable smart sensor interface for industrial wsn in iot environment,” *IEEE trans. on industrial informatics*, vol. 10, no. 2, pp. 1417–1425, 2014.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT press, 1999.

[11] G. Cugola and A. Margara, “Deployment strategies for distributed complex event processing,” *Computing*, vol. 95, pp. 129–156, 2013.

[12] M. M. Bersani, D. Bianculli, C. Ghezzi, S. Krstic, and P. S. Pietro, “Efficient large-scale trace checking using mapreduce,” in *Proc. of the 38th Intl Conf. on Software Engineering,*. ACM, 2016, pp. 888–898.

[13] F. Gu, J. Niu, L. Jiang, X. Liu, and M. Atiquzzaman, “Survey of the low power wide area network technologies,” *Journal of Network and Computer Applications*, vol. 149, p. 102459, 2020.

[14] L. Ratliff, “Unlocking Captive Value: LPWAN Enables Emerging IoT Applications,” lora-alliance.org/sites/default/files/2019-07/ihsmarkit_berlin_2019_0.pdf, IHS Markit, Tech. Rep., Jun. 2019.

[15] J. Haxhibeqiri, E. D. Poorter, I. Moerman, and J. Hoebeke, “A Survey of LoRaWAN for IoT: From Technology to Application,” *Sensors*, vol. 18, no. 11, p. 3995, 2018.

[16] Y. E. Wang, X. Lin, A. Adhikary, A. Grövlen, Y. Sui, Y. W. Blankenship, J. Bergman, and H. Shokri-Razaghi, “A Primer on 3GPP Narrowband Internet of Things,” *IEEE Commun. Mag.*, vol. 55, no. 3, pp. 117–123, 2017.

[17] *Mobile Edge Computing (MEC); Framework and Reference Architecture*, ETSI Group Specification MEC 003, V2.1.1, Jan. 2019.

[18] L. Zanzi, F. Cirillo, V. Sciancalepore, F. Giust, X. P. Costa, S. Mangiante, and G. Klas, “Evolving Multi-Access Edge Computing to Support Enhanced IoT Deployments,” *IEEE Commun. Stand. Mag.*, vol. 3, no. 2, pp. 26–34, 2019.

[19] Y. Liu, M. Peng, G. Shou, Y. Chen, and S. Chen, “Toward Edge Intelligence: Multiaccess Edge Computing for 5G and Internet of Things,” *IEEE Internet Things J.*, vol. 7, no. 8, pp. 6722–6747, 2020.

[20] (2020) LADOT Parking Meter Occupancy dataset . data.lacity.org/A-Livable-and-Sustainable-City/LADOT-Parking-Meter-Occupancy/e7h6-4a3e.

[21] T. Rausch, C. Avasalcai, and S. Dustdar, “Portable energy-aware cluster-based edge computers,” in *2018 IEEE/ACM Symposium on Edge Computing*, 2018, pp. 260–272.

[22] R. Sanchez-Iborra, J. Sanchez-Gomez, and A. F. Skarmeta, “Evolving IoT networks by the confluence of MEC and LP-WAN paradigms,” *Future Gen. Comp. Syst.*, vol. 88, pp. 199–208, 2018.

[23] A. Ksentini and P. A. Frangoudis, “On Extending ETSI MEC to Support LoRa for Efficient IoT Application Deployment at the Edge,” *IEEE Comm. Standards Magazine*, no. 2, pp. 57–63, 2020.