Problem Detection in the Edge of IoT Applications

Iván Bernabé-Sánchez, Alberto Fernández, Holger Billhardt, Sascha Ossowski *

CETINIA, Universidad Rey Juan Carlos, Madrid (Spain)

Received 5 May 2022 | Accepted 20 July 2023 | Published 26 July 2023

Abstract

Due to technological advances, Internet of Things (IoT) systems are becoming increasingly complex. They are characterized by being multi-device and geographically distributed, which increases the possibility of errors of different types. In such systems, errors can occur anywhere at any time and fault tolerance becomes an essential characteristic to make them robust and reliable. This paper presents a framework to manage and detect errors and malfunctions of the devices that compose an IoT system. The proposed solution approach takes into account both, simple devices such as sensors or actuators, as well as computationally intensive devices which are distributed geographically. It uses knowledge graphs to model the devices, the system's topology, the software deployed on each device and the relationships between the different elements. The proposed framework retrieves information from log messages and processes this information automatically to detect anomalous situations or malfunctions that may affect the IoT system. This work also presents the ECO ontology to organize the IoT system information.

Keywords

Complex Event Processing, Intelligent Agents, Internet Of Things (IoT), Ontologies.

DOI: 10.9781/ijimai.2023.07.007

I. Introduction

According to [1], 2023 some 29.3 billion devices will be connected to IP networks. This more than triples the world's population. In fact, there will be 3.6 devices per person, a considerable growth of over 50% compared to 2018. Half of these devices will make machine-to-machine (M2M) connections, totalling 14.7 billion M2M connections. This increase in the number of devices and connections will produce an enormous amount of data and create new opportunities for innovative applications in domains such as healthcare [2], environmental sciences and industrialization [3], etc. The inclusion of IoT in these domains requires caution in large-scale implementations because of the risks of saturation of system resources and due to security issues. In many cases, IoT devices are used to improve people's daily activities or optimize important processes in companies, which may expose data [4].

Albeit security of IoT is a major topic addressed in literature [5], there are other important problems that condition the expansion and implementation of IoT solutions. For example, managing a large volume of devices requires dealing with problems like communications interruptions (network connectivity) [6], discontinuity of services, discharge of batteries (energy saving), and problems with the operating environment (overheating, storage management, cybercrime) [7]. All of these issues may apply to any of the devices that integrate an IoT system.

Self-repair or self-healing is defined as a property of systems that are able to identify and diagnose problems that appear during their operation and to determine and propose solution strategies in an autonomous way [8]. More specifically, self-healing provides reliability to a system through responsibility and awareness of the environment. This allows to automatically detect problems and to propose solutions to unwanted situations. In order to do so, a self-healing IoT system must incorporate monitoring, awareness, and knowledge to detect unwanted states. When a problem is detected, the system generates and executes plans with appropriate corrective actions [9], [10].

In this work, we propose a framework for the specification and automatic detection of problems that may occur in an IoT system. This framework consists of independent agents that are distributed on the different devices that make up a system. Setting out from messages stored in log registers, these agents extract information about the operation of devices and the software deployed on them, and process it to identify operating problems. The proposed framework uses knowledge graphs (ontologies) to structure the information, event stream processing to identify problems, and automatic reasoning to infer additional knowledge related to the operation and potential problems of a system. The edge-cloud ontology (ECO) has been designed to structure the system information and possible problems.

The rest of the article is divided into the following sections. Section II contains the state of the art. Section III shows the proposed architectural solution. Section IV details how semantic technologies are used to represent and process the information for identifying existing problems. An example is presented in Section V. We conclude the paper and point to some future lines of research in Section VI. Table I shows the list of acronyms.

II. Related Work

Failures of system elements in IoT systems are usually considered as something inevitable. It is important to consider this possibility and to integrate mechanisms that ensure that the infrastructure will continue to function without interruption, even if some elements fail.

* Corresponding author.

E-mail addresses: ivan.bernabe@urjc.es (I. Bernabé-Sánchez), alberto.fernandez@urjc.es (A. Fernández), holger.billhardt@urjc.es (H. Billhardt), sascha.ossowski@urjc.es (S. Ossowski).

TABLE I. List of Acronyms and Abbreviations Used

Abbreviations	Explanation
ASS	Action Schedule Service
CEP	Complex Event Processing
CLF	Common Log Format
CMA	Complex Management Agent
DIR	Deployed Infrastructure Repository
DPR	Detected Problem Repository
ECO	edge-cloud ontology
ELFF W3C	Extended Log File Format
DPR	Detected Problems Repository
FD	Fog Devices
IoT	Internet of Things
IS	Inference Service
KG	knowledge graphs
LMA	Lightweight Management Agent
LMS	Log Management Service
MMS	Middleware Management Service
OS	operating system
SAREF	Smart Applications REFerence Ontology
SD	Simple Devices
SmD	Smart Devices
SOSA/SSN	Semantic Sensor Network ontology

Some works, like [11]–[13], propose fault-tolerant solutions to recover IoT systems deployed in the cloud and edge computing. These works are mainly focused on managing problems associated with resource exhaustion and performance degradation. However, fault-tolerant distributed systems must be able to go further and handle finer-grained problems such as error management in applications deployed in cloud and edge computing. To this respect, [14], [15] and [16] propose solutions based on micro-services. These works put forward mechanisms for system recovery, but do not describe the previous error detection process. Still, information on the causes of errors and the elements involved can greatly facilitate the generation of potential solutions to a problem.

A starting point for troubleshooting is to know if errors exist and when they appeared. Usually, applications write status information in log files, which are analyzed manually or automatically to find out if there are any problems. Normally, these logs have to be parsed before their contents can be interpreted [17]. A common approach to parse logs is to detect or match with specific error patterns [18], [19]. Other solutions use data mining techniques such as SLCT [20], and its extension LogCluster [21]. These works require a large data set with a large log history to generate efficient log patterns. Still, in recent and very specific or uncommon systems it is complicated to apply these solutions, because there may not be enough information to generate efficient patterns. Some works, such as [14], [15] and [16], also consider mechanisms to recover systems from errors and bring them to the desired operating status. These works use information repositories and catalogues to have a record of the architecture of services that make up an IoT systems. However, such repositories are specifically designed for the proposed solutions and do not have a formal specification of how their information is structured.

We claim that working with structured information can largely facilitate error detection because it allows specifying explicitly the relationships among the different information elements at a conceptual level, and enables the reuse of information across different systems. In particular, knowledge graphs and ontologies can help structuring the failure-related information available in an IoT system. Furthermore, inference based on ontologies may even infer additional information that is not explicitly available. As described in Noy and McGuiness [22], using ontologies provides several benefits: (1) they share a common understanding of information structure among software agents; (2) allow reuse of domain knowledge; (3) domain assumptions are made explicit; (4) domain knowledge can be analyzed. Taking advantage of all these benefits is key to interoperability. As indicated by Bittner et al. [23], as well as by Jasper and Uschold [24], ontologies facilitate the semantic interoperability between humans, computers, and systems. They consider them as a facilitating technologies to achieve communication interoperability between software systems.

The use of ontologies in cloud systems [25] has already been applied to different areas such as resource management [26], service discovery [27], [28], security [29] or even to improve system interoperability. In this line, mOSAIC [30] presents a cloud ontology that provides a detailed description of cloud computing resources. mOSAIC focuses on promoting transparency in accessing multiple clouds. However, mOSAIC has not been updated since its development more than 10 years ago, which implies that new elements that have appeared in the area during this time are not contemplated in this ontology. As a result, other works such as [31] and [32] have appeared so as to try to address these limitations. In [31], a solution for deploying applications on public and private clouds is shown. The solution uses a set of rules to control the deployment of applications. This rule set uses the CAMEL modeling language[1]. ModaClouds [32] is another work that uses ontology-based models to perform semi-automatic code transformations allowing to obtain compatible implementations in public and hybrid cloud provider platforms. The aforementioned works propose mechanisms to improve the interoperability of services hosted by different service providers, improve the description of existing interfaces and even provide decision-making support. However, these projects do not support a broad heterogeneous environment, i.e., they are limited to resource management, hardware accelerators and provide resource abstractions in the cloud. In general, these works are not oriented to work with IoT devices such as sensors, actuators, gateways, etc.

In this line, specific ontologies have been developed to model the capabilities, characteristics and descriptions of systems that integrate IoT devices. The Semantic Sensor Network Ontology (SOSA/SSN) [33] is one of the most prominent efforts in this area. SOSA/SSN describes sensor and actuator networks, their capabilities, features of interest, and observations and serves as a starting point for the creation of new ontologies that integrate these devices. Another effort similar to SOSA is the Smart Applications REFerence Ontology (SAREF) [34] developed by the ETSI's SmartM2M technical committee. SAREF allows the description of devices and their functions and is aligned with the oneM2M ontology [35], which allows syntactic and semantic interoperability between devices and external systems. SAREF and SSN are ontologies that are widely used and there are works that extend their scope of application to other more specific domains. For example, CASO [36] and EEPSA [34] extend SAREF for its application in agriculture and smart buildings domains, respectively. In the case of the SSN ontology, the SSN System module allows modeling systems, capabilities and things.

Still, despite the number of existing original ontologies and their extensions, to the best of our knowledge, there are no ontologies capable of providing mechanisms that integrate information from systems in the cloud, at the edge computing with IoT devices. For this reason, we created and present in this article an ontology for this purpose.

[1] https://camel-dsl.org/

III. Proposed Solution

Fig. 1 shows the type of infrastructures we aim at in this work. The architecture depicts a generic IoT system made up of sensors and other devices that allow the collection of information and interaction with the physical world. In general, the information collected by IoT devices in lower layers is processed, filtered out, and sent to higher levels for further analysis. As we will present in this section, we propose the inclusion of intelligent agents that will monitor the operation and status of the different devices in the system.

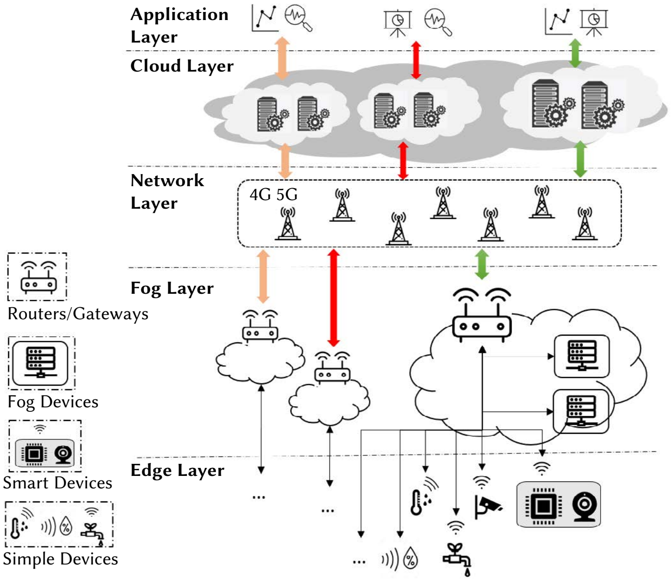

Fig. 1. Typical layered architecture with the distribution of devices that form a generic IoT system. Higher layers include devices with higher computational resources. Green, red and orange arrows represent interactions within different applications.

In the following, we begin with a description of the reference edge-cloud architecture we focus on in this work. Then, we present our architectural proposal for problem identification.

A. Reference Architecture

In the structure of the IoT system described in Fig. 1, there are several types of devices distributed across different layers. A basic IoT system is composed of IoT devices deployed at some physical location, for example, collecting data from the environment and forwarding them to some remote machine for their processing. With this basic infrastructure, systems can become more complex by adding devices, services or applications.

The basic architecture is composed of five layers, each of them grouping different systems, devices and other computational resources, which may involve different service, computation and communication providers. These layers are common to different IoT systems; in fact, Fig. 1 shows different IoT systems (identified by red, green and orange arrows) that are extended across different processing layers. The *Edge Layer* is composed of IoT devices such as sensors, actuators, etc. It is the layer that interacts with the physical world. The *Fog Layer* connects the devices at the Edge Layer with upper layers and can provide basic processing services. The *Network Layer* is in charge of managing communications with data centres located in remote locations. Large-scale computational resources are provided on the Cloud Layer. Finally, applications, typically processing and producing high-level information, are running on the *Application Layer.*

For the two lower layers, in this work, we focus on three types of devices, namely simple, smart and fog devices. Their functionality highly depends on their computational resources. *Simple Devices* (SDs) are low-cost devices (e.g. sensors and actuators). SDs have low computational resources and basic capabilities, for example, to take measurements (depending on the type of physical sensor installed) and send those values to other remote devices where that information is processed. SDs usually use batteries and are typically deployed in remote physical environments.

Smart Devices (*SmDs*) have functions similar to SDs, but with more computing power, which allows them to process the information collected on the same device. For example, in a cultivated field, devices can be deployed to monitor the appearance of imperfections on plant leaves. In that case, the device would have a camera to take pictures of plant leaves, and a running algorithm to detect biological problems (e.g. musty, dry, etc.), which would be forwarded to a processing node in the upper layers.

Finally, *Fog Devices* (*FDs*) are located at the fog layer and have some computing capacity deployed somewhere on a local network. These devices receive information from SmDs and SDs and carry out processing tasks such as aggregation, integration, filtering, statistics, etc.

B. Proposed Architectural Solution

During normal operation, the software deployed on SDs, SmDs and FDs might be subject to errors and malfunctioning. Intelligent management techniques are required to deal with such errors and to make the systems efficient and stable. For this purpose, in our work we propose to use distributed intelligent agents with the aim of monitoring and controlling the software deployed on each of the devices. We focus on distributed systems where software is deployed on lower-level devices as well as on data centres to perform the assigned functions.

Usually, the software is installed and deployed on each device in the traditional way. However, we recommend encapsulating such software in software containers and then deploying such containers on devices. Software containers are a type of lightweight virtualization [37] that allows running multiple isolated software instances on a single operating system (OS) without the need to have an OS for each instance. This type of virtualization is also called containerization and provides encapsulation for each container and resource management. It makes this technology lighter and more efficient than traditional virtualization technologies which require an operating system on each instance. The encapsulation offered by containers does not affect the normal operation of the software running inside them and facilitates their deployment and management. Containers are managed independently of other containers and the OS installed on the device. Generally, a middleware (also called framework) is installed between the OS and the containers. The middleware is responsible for the management of containers and provides mechanisms and interfaces to obtain information and control them.

Using software containers is not only an advantage in terms of ease of management but also offers heterogeneity in terms of being able to run the same container on different devices. As software containers require to work a middleware placed between the device's operating system and the containers, the same container can be executed on different devices if these devices have the middleware installed. The container will be executed on the device regardless of the type of OS and hardware that integrates the device. This is important for the solution proposed in this work because it facilitates moving and running software services between devices.

The container middleware provides mechanisms for starting, stopping and deployment of containers regardless of the device where it is deployed or the software it encapsulates. Fig. 2 shows our proposal of a container-based software architecture that is to be used on the devices deployed in the architecture of Fig. 1. Using makes it easy to deploy software on any device in a system. This is because most container frameworks can connect to remote repositories where the software has been previously uploaded to easily download, install and run the desired software on any particular device. This feature facilitates the resolution of device failures since software from devices with errors can easily be transferred to other devices.

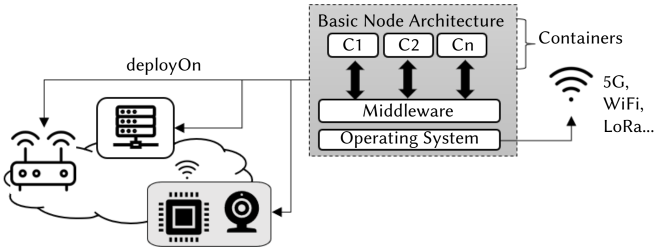

Fig. 2. Device architecture that manages installed software through software containers.

In this paper, we propose two types of intelligent agents that control the functioning of the system. The first one is the *Lightweight Management Agent* (*LMA*), which is responsible for collecting information about the software and device on which it is deployed. LMAs send the collected information and can receive actions to apply on the device. The second type of agent is the *Complex Management Agent* (*CMA*). CMAs are able to carry out more complex reasoning processes, including receiving information from LMAs, detecting existing problems and generating local actions to alleviate existing problems and bring the device operation back to a desired state. CMAs are typically located in the fog or in the cloud. CMAs are deployed at the fog layer to troubleshoot unwanted situations in local device networks. CMAs can also be deployed in the cloud where the CMA is responsible for managing problems that cannot be resolved on the fog level. CMAs are prepared to operate with limited resources but are also capable of dealing with complex problems by scaling the computational resources of the CMA. A CMA deployed in the cloud is capable of addressing problems using a large number of devices and parameters.

Fig. 3 shows the integration of the solution proposed in this work into the general architecture shown in Fig. 1. LMAs and CMAs are described in more detail below and Fig. 4 shows their integration on the different devices.

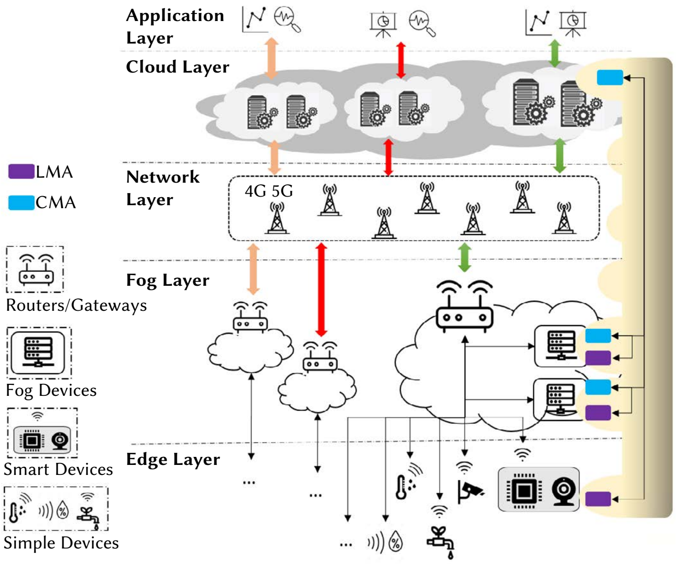

Fig. 3. Integration of the solution proposed in this paper into a generic IoT system.

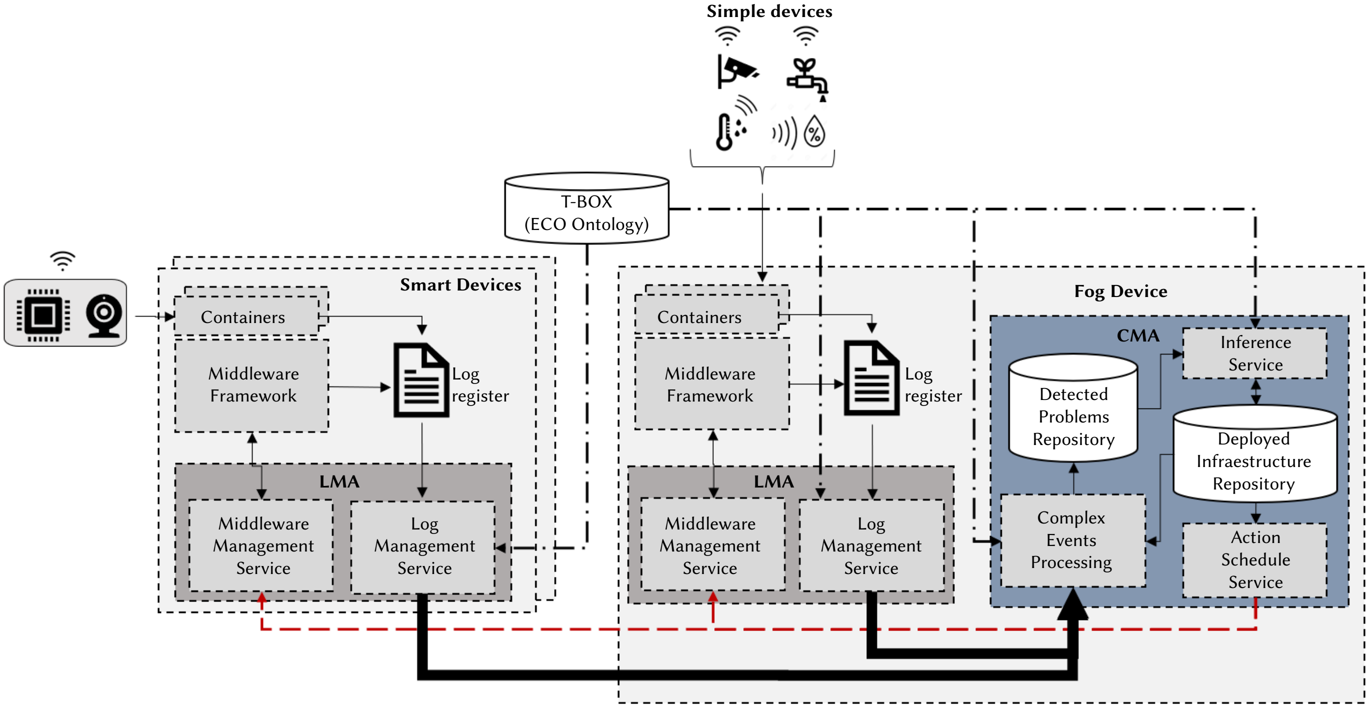

Fig. 4. Disposition and connections of agents, devices, services and software of elements contained in Fig. 1.

1. Lightweight Management Agent

Lightweight Management Agents are deployed on IoT devices with low computational resources, such as SmDs. They read information about the operation and status of the device they control and, if situations are detected that require attention, collect the related information and send it to a CMA, which will process that information and propose corrective actions. LMAs extract information from log files generated by the deployed software, middleware (if installed) or operating system (OS).

The LMA architecture is composed of two services: the Log Management Service (LMS) and the Middleware Management Service (MMS). The Log Management Service is responsible for extracting the data provided from applications, middleware and OS. In addition, it periodically extracts information related to the operating status of the device (e.g. battery charge status, resource usage, etc.). This information may indicate whether the device is operating within pre-established security limits or may stop working when resources collapse. The LMS structures all the collected information following the RDF data model as a list of events using the ECO ontology (described below). Then, all the events are sent to be analyzed by a CMA.

The LMA can also receive actions to be applied to the device. Most actions are expected to be executed via the Middleware Management Service.

2. Complex Management Agent

Complex Management Agents are more complex than LMAs and they are typically deployed on devices located in the Fog layer of an IoT system. These devices are usually advanced routers, gateways or other network devices that, due to their computing power, can provide additional services to the local network. CMAs are in charge of processing the information obtained from LMAs, either related to detected malfunctions or to any other events. A CMA consists of five components: Complex Event Processing (CEP), Detected Problems Repository (DPR), Inference Service (IS), Deployed Infrastructure Repository (DIR) and Action Schedule Service (ASS).

The Complex Events Processing component introduces the events received from LMAs into a stream of events. CEP [38] is a technology that analyses continuous streams of events to identify complex patterns. CEP systems use elements such as timestamps and sliding windows. Several filters are continuously analyzing the stream in order to identify critical operating states, which are registered in the DPR. The DIR contains information about the local network topology (e.g. connections among devices, dependencies among software, etc.). The Inference Service inserts into the DIR additional information which is inferred from the identified problems (available in the DPR) and the current infrastructure status (available in the DIR). Finally, the ASS is in charge of proposing actions with the aim of reducing the impact of the identified problems.

IV. Semantic Technologies Supporting Problem Identification

As mentioned above, we propose a solution to detect problems or undesired operating states in distributed edge-to-cloud infrastructures based on collaborative intelligent agents. Lightweight agents at the edge collect basic pieces of information (raw events) that in correlation may lead to the identification of problems or undesired operating states. In this context, information about dynamic events and the system topology (e.g. physical and/or logical connections among devices and processes running on them) has to be represented and processed. We opt for using knowledge graphs (KG) [39] to represent such information. A knowledge graph is a way of describing information in a graph structure where nodes represent entities (individuals or types of elements) and edges represent relations between them. While KGs have been used in AI for a long time (also known as *semantic networks*), they have been gaining popularity in the last years [40]. A knowledge graph is a flexible and easy-to-extend representation model, which can be endowed with a schema or ontological model (aka T-Box), thus facilitating automatic inference processes.

In the rest of this section, we first (A) present an ontology for representing the information about the topology of an edge-cloud system and the problems that may occur during its operation. Then (B), we describe how to extract and represent basic information (events) about the status of devices while the IoT system is running. Finally, we show (C) how the combination of the knowledge graph and the generated events are used to identify existing problems in the system.

A. The Edge-Cloud Ontology (ECO)

The proposed solution uses a KG and it requires advanced mechanisms to manage that KG in a viable way. The KG organizes data related to the devices connected to the IoT system, the network topology that interconnects different devices, and the software deployed on the devices. Instead of developing an ontology from scratch, we have considered reusing existing ontologies and if necessary adapting them to meet our needs. In particular, the ECO ontology [41] is appropriate for the needs of the proposed system because it provides concepts and properties to represent the state of each of the devices that make up an IoT system.

Fig. 5 shows the main concepts and properties of the ECO ontology. The ECO ontology is based on the SEAS ontology [42] and adds new entities. These new entities allow for specifying the current state of an IoT system thanks to events generated during the operation of the integrated devices. When events are processed, it is possible to identify problems or undesired operating states that are modelled into the system by the *eco:Problem* entity.

The ontology classes and properties can be organised into three main groups describing: (i) the connections among physical devices forming the network topology,(ii) the software deployed on devices and their logical dependencies, and (iii) the events representing relevant states of devices and/or software, and the problems that define critical situations. In the following, we describe the main elements of each group.

The topology of the IoT infrastructure representing computational systems and how they are interconnected can be represented with the following classes:

- *seas:System*. This class describes systems that share connections with other systems;
- *seas:Connection*. A connection describes potential interactions between systems.
- *seas:ConnectionPoint*. This class models the connection between systems.
- *eco:ComputingNode*. This class represents any device with processing capability.

Knowing which software is deployed on each device and how software components logically depend on each other can be important in certain situations in which unexpected problems (e.g. connectivity failures) on one device may affect the behaviour of others.

- *eco:Software*. This class represents any type of software and can be instantiated through three different types of subclasses: *eco:Application*, *eco:Service* and *eco:Middleware*.
- *eco:Application*. This class is a type of software that represents a particular application. An application may be composed of one or more services of type *eco:Service*.

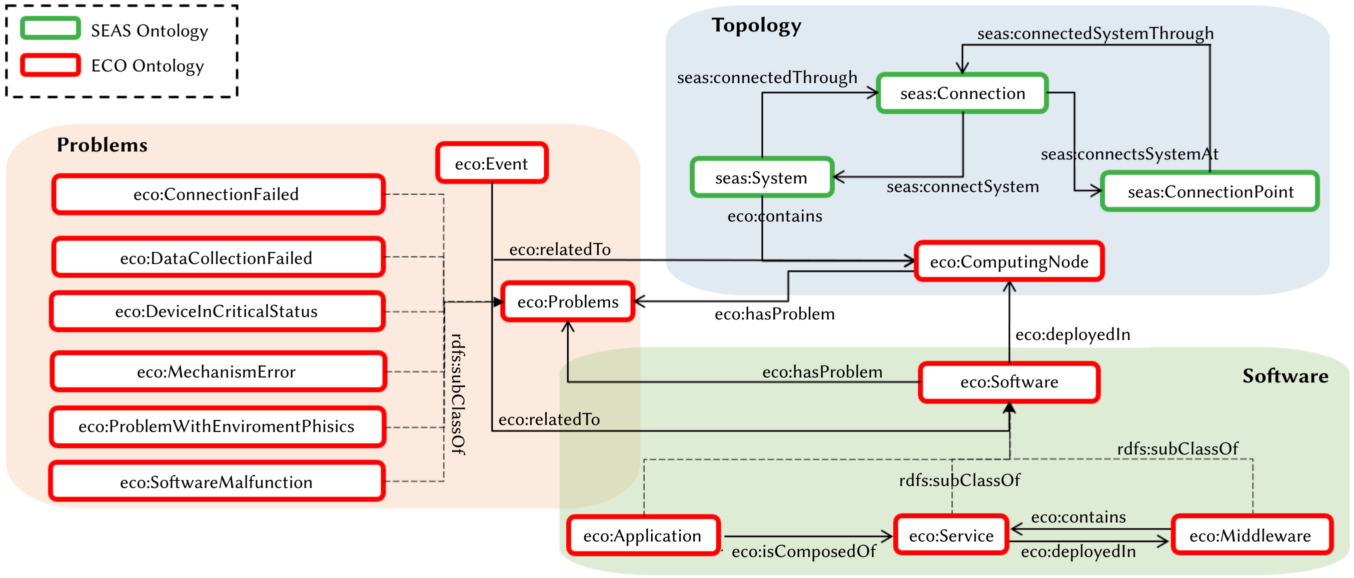

Fig. 5. Main concepts and properties of the ECO ontology. Green concepts are reused from the SEAS ontology, whereas red nodes represent new concepts defined in ECO.

- *eco:Service*. The Service class represents software services that are independent and have been designed to perform a specific function.
- *eco:Middleware*. This class is a type of Software that may contain applications or services. In practice, this class can refer to software frameworks on which applications run. An example of frameworks can be the Java virtual machine, the .NET framework, software container frameworks such as Docker, etc.

Finally, the ECO ontology allows the representation of events and problems that define critical situations.

- *eco:Problem*. This class models problems that may appear in the system. Six types of problems have been identified in this work.
- *eco:ConnectionFailed*. The class represents problems related to the connections between devices and refers to operating states related to the lack of connection between devices or software.
- *eco:DataCollectionFailed*. The DataCollectionFailed class is expected to be instantiated when a sensor has taken a measurement but the result obtained is erroneous (non-consistent value or values out of limits). The *DataCollectionFailed* class is oriented towards simple sensors measuring attributes like temperature, humidity, atmospheric pressure, etc.
- *eco:DeviceInCriticalStatus*. This class models a problem that represents a device that is in a critical state (that could stop working at any time). This situation can occur when the CPU is saturated, the available RAM memory is low, the free disk space is almost exhausted, the device battery is almost discharged or even if the device temperature is relatively high. All these situations are indicators that the device may not be working properly and may affect its performance.
- *eco:MechanismError*. This class models a problem on an actuator device, robot or any other device that has some physical mechanism. The entity models a physical operating problem. i.e. the device receives and processes the indicated actions, but cannot carry them out due to mechanical problems of the device itself.
- *eco:ProblemWithPhysicalEnviroment*. This class models real-world conditions that can adversely affect the functioning of specific devices. It is used to model physical aspects of the environment that may present a problem for specific devices. An example could be the level of luminosity of the environment in which a sensor or camera operates.
- *eco:SoftwareMalfunction*. This entity reflects problems related to the normal operation of software. It may also indicate that a software component is stopped.
- *eco:Event*. This entity represents an event that has been generated during system operation. The event contains relevant information that must be processed to evaluate whether the system is functioning correctly or whether there is an associated problem.

B. Raw Events Generation

The proposed solution requires knowing the current status of devices and software that is deployed in an IoT system. In this sense, log registers can be an important source of information since they will usually include information related to errors and operating status. There have been some efforts to standardize log files such as Common Log Format (CLF) [43], W3C Extended Log File Format[2] (ELFF), RFC5424 [44] or RFC3164 [45]. CLF and ELFF provide guidelines for organizing the information contained in log files generated by web servers and RFC5424 or RFC3164 are oriented to define the transmission of messages generated by log systems. However, there is no general standard nor guidelines for defining log messages or how to structure them. This implies that developers are responsible for designing the structure of the log messages generated by their applications. To improve this situation, we propose some indications to take into account when software has to generate log messages. Basically, we propose to specify: i) what information will be introduced in the log file. ii) how this information will be structured and iii) where the log information will be available.

Specifying the information registered in each log message is not easy. Ideally, the information should be as complete as possible because it will describe in detail why the message was inserted in the log register. However, the nature of applications is very varied and this opens up a wide range of possibilities. We propose the use of a limited set of parameters, grouped by field of application or category, as shown in tables II, III and IV.

Table II contains parameters related to the operation of applications and, whether or not an application is encapsulated on containers. The

[2] https://www.w3.org/TR/WD-logfile.html

table also presents some parameters related to the management of these containers. This information is published in the log register by the applications and by the container framework.

TABLE II. Parameters Registered by the Middleware

Params	Description
softwareIsRunnig	Whether the software is running.
actionsCarriedOut	Last command executed in the Middleware (e.g. stop, start, ...)
numContainersDeployed	Number of containers deployed in the framework.
numContainersRunning	Number of containers running in the framework.

Table III presents specific domain application parameters, usually related to perception and actuation elements (e.g., sensors or actuators). These parameters try to collect common problems that occur when applications use such elements.

TABLE III. Parameters Registered by Domain Applications. They Indicate Problems With Measurements or Actuation Orders

Params	Description
errorDescription	Error message generated by the software.
OutOfRangeReading	This error indicates that the measurement received from a sensor is outside specified limits.
abnormalReading	Measurement taken from a sensor is invalid.
highLightConditions	There is too much light in the physical environment.
lowLightConditions	There is insufficient brightness in the physical environment.
errorInPhysicalMechanism	There was a failure to activate an actuator. This error is usually associated with actuators rather than sensors, since actuators usually have physical mechanisms to interact with the physical environment.

Table IV presents a set of parameters that represent the current state of a device. Essentially, these parameters are intended to indicate the availability of resources. In our proposed solution, these values are obtained by the LMS by querying the operating system.

The proposed set of parameters is oriented to an IoT system (as shown in Fig. 1) and is intended to cover potential unexpected situations that may arise. The ECO ontology includes properties to represent those parameters.

TABLE IV. Parameters Related to Device Operation

Params	Description
connectedToTheNetwork	Whether the device is connected to a network.
droppedPackets	Number of packets discarded per second.
receivedPackets	Number of network packets received per second.
sentPackets	Number of network packets sent per second.
deviceTemperature	Device temperature.
meanPercentageUseOfCPU	Percentage of CPU usage.
batteryChargeLevel	Battery charge percentage.
freeRAM	It is the free RAM memory space. It is measured in MBs.
freeDiskSpace	It is the free disk space. It is measured in MBs.

Normally, applications record log messages to dedicated log files. However, in this paper, we propose using the log register provided by the operating systems since it is possible to access all the information about errors from a single point. Each log entry is organized into the following ordered list of attributes:

- *Time*: when a message is generated in the log register.
- *Machine*: indicates the name of the machine.
- *Application*: refers to the application, service, process, or software that generates the message
- *Message*: information regarding the event registered in the log. The content is represented as key-value pairs in JSON format.

The structure formed by the fields of each log entry facilitates the understanding of the information by humans and machines.

Table V shows an example extracted from a log register. In Linux the log register is called Syslog and contains all the log entries generated by the operating system. It has a semi-structured format where spaces separate multiple segments (timestamp, machine name, application name and message). Similarly, other operating systems have their own log systems (e.g. logcat in Android).

Line 1 has been generated by a Tomcat application server that shows, in the body of the message, information that the application has been started. Line 2 corresponds to the execution of a task that the OS had scheduled. Line 3 shows a message from the Docker (middleware) framework, structured as key-value parameters. This line specifies that a deletion task has been performed ("topic= /tasks/ delete type=event.TaskDelete") on a container ("container=ad9c..."). Line 4 presents information that has been entered by the LMA (called LMA1 in table V).

TABLE V. Parameters Related to Device Operation

	Header entry			Message entry
	Time	Machine	Application	Messages
1	Jan 17 17:11:36	machine 1	tomcat[8944]	17-Jan-2023 14:17:29.850 INFORMATION [main] org.apache.catalina.startup.Catalina.start Server startup in [560] milliseconds
2	Jan 17 17:17:01	machine 1	CRON[9184]	(root) CMD (cd / && run-parts –report /etc/cron.hourly)
3	Jan 17 17:19:39	machine 1	dockerd[751]	time="2023-01-17T17:19:39.911663169+01:00" level=info msg="ignoring event" container=ad9c3e4f269aff56c60fb3558655de1c3703be4 8b86b848ba62d8510261e8ffe module=libcontainerd namespace=moby topic=/tasks type="*events.TaskDelete"
4	Jan 17 17:20:00	machine 1	LMA1	meanPercentageUseOfCPU="7.24", freeRAM="63.4", freeDiskSpace="86.5", batteryChargeLevel="100.00"
5	Jan 17 17:25:00	machine 1	LMA1	errorInPhysicalMechanism

The LMA reads the log register and generates an event with the necessary information and formats it in RDF triples according to the ECO ontology. Listing 1 shows three events (event1, event2 ad event3) in RDF format[3] generated from lines 3, 4 and 5 of Table V, respectively.

Listing 1: Example of several events in RDF Turtle format

```
: event1 rdf : type eco : Event ;
     eco : date " Jan 17 17:19:39" ;
     eco : relatedToDevice : machine1 ;
     eco : relatedToSoftware : dockerd ;
     eco : actionsCarriedOutExecuted "* events . TaskDelete
     ".
: event2 rdf : type eco : Event ;
     eco : date " Jan 17 17:20:00" ;
     eco : relatedToDevice : machine1 ;
     eco : relatedToSoftware : LMA1 ;
     eco : freeRAM "63.4";
     eco : meanPercentageUseOfCP "9" ;
     eco : batteryChargeLevel "100" ;
     eco : freeDiskSpace "86.5".
: event3 rdf : type eco : Event ;
     eco : date " Jan 17 17:25:00" ;
     eco : relatedToDevice : machine1 ;
     eco : relatedToSoftware : LMA1 ;
     eco : errorInPhysicalMechanism " TRUE ".
```

In principle, ad-hoc parsers are needed for the different applications and operating systems used. However, as we mentioned above, in order to facilitate this task, we propose a *key-value* parameter representation and a set of specific parameters (Tables II, III and IV). LMAs can interpret those parameters as well as some popular application formats such as docker and tomcat. Other log formats are not considered. Thus, the systems that want to integrate our solution approach in the future must follow our proposed log format.

C. Stream Processing

Event processing is mainly carried out by the CEP component, which is responsible for analyzing the events and is able to identify undesirable situations or system problems from the information contained in an event stream. In the case of identifying any undesirable situation, the CEP is able to classify the type of situation and to label that situation with the corresponding problem entity from the ECO ontology. Detecting problems is generally a complex task that may depend on several parameters. Thus, the origin of a problem may be associated with one or several pieces of evidence. In this work, an evidence is an event; therefore, the information contained in the events helps to identify problems. One of the main characteristics of systems based on event processing is that events are not independent of each other, but are related to each other. In the case of systems formed by sensor devices, the data generated by the sensor network are usually related in time and space. For example, in agro-IoT systems, the data measured by a sensor is usually strongly related to the data of a nearby sensor. In a similar way, the measures observed at a particular moment in time are generally correlated to the measures taken in the next unit of time. This is important because applications may not be interested in measurements taken from a sensor at a particular time and place, but in aggregated information in space and time [46].

The main task of event processing is to identify within event streams those event patterns that are of interest in a particular domain. For example, in an agro-IoT system consisting of sensors to measure the conditions of cropland, several sensors may emit events that must be analyzed to discover patterns identified with problems related to plant health. In the context of the work presented in this paper, the events generated from the log messages would also be analyzed in order to discover problems related to software or devices. It will even be possible to predict problems before they actually happen such that corrective actions could be applied before a particular problem appears.

The LMA is responsible for generating the corresponding events and sending them to the CMA. Then, the CEP component located in the CMA filters the events trying to identify problems. This process is done through queries. Each implemented filter returns a problem instance according to the ECO ontology, which is inserted into the Detected Problem Repository (DPR). For example, the

DeviceInCriticalStatus problem instance could be triggered in those cases where a device has a battery below 15% in addition to having high CPU and RAM consumption. This situation could cause a device to consume its low battery power in a short period of time.

For stream processing, we use C-SPARQL [47], a continuous query language that extends SPARQL [48] to work with RDF data streams such as the example shown in listing 1. C-SPARQL queries are continuously monitoring recent events/triples (time windows are specified) to detect particular patterns that correspond to identified problems. When the query is matched, it generates a result as RDF triples.

Listing 2: C-SPARQL query that identifies a problem that a device is in a critical state (CPU and RAM usage higher than 90% and battery charge lower than 15%)

```
CONSTRUCT {
     _: prob rdf : type eco : DeviceInCriticalStatus .
     _: prob eco : relatedTo ? device .
}
FROM STREAM : streamExample1
          [ RANGE 60s STEP 30s]
FROM < instancesTopology .owl >
WHERE {
     ? event rdf : type eco : Event .
     ? event eco : relatedTo ? device .
     ? event eco : percentCPUUsage ? cpuL .
     ? event eco : freeRAM ? ram .
     ? event eco : batteryChargeLevel ? batt .
FILTER (? percentCPUUsage > 90
     && ? freeRAM < 10
     && ? batteryChargeLevel < 15)
}
```

Listing 2 shows an example of a query that returns the instance of *eco:DeviceInStatusCritical* problem, with its associated device. The query checks the value of several parameters such as CPU, RAM consumption and the battery charge level of a specific device. The query only takes into account events occurring in the specified time window (60 seconds).

V. Use Case

This section presents a use case that shows the potential of the error detection framework proposed in this work. We consider a scenario composed of two local Wi-Fi networks, each containing two sensors deployed on SmartDevices (e.g., Smartphones), a computing device (Raspberry Pi4 model B equipped with 4GB of RAM) that acts as a Fog Device and a router that provides the Wi-Fi network to which each of the devices is connected. Fig. 6 shows the example

[3] We use the Turtle RDF serialization. In short, each triple (subject predicate object) is written in a line, ending in '.'. A ';' can be used to avoid repeating the same subject in consecutive lines.

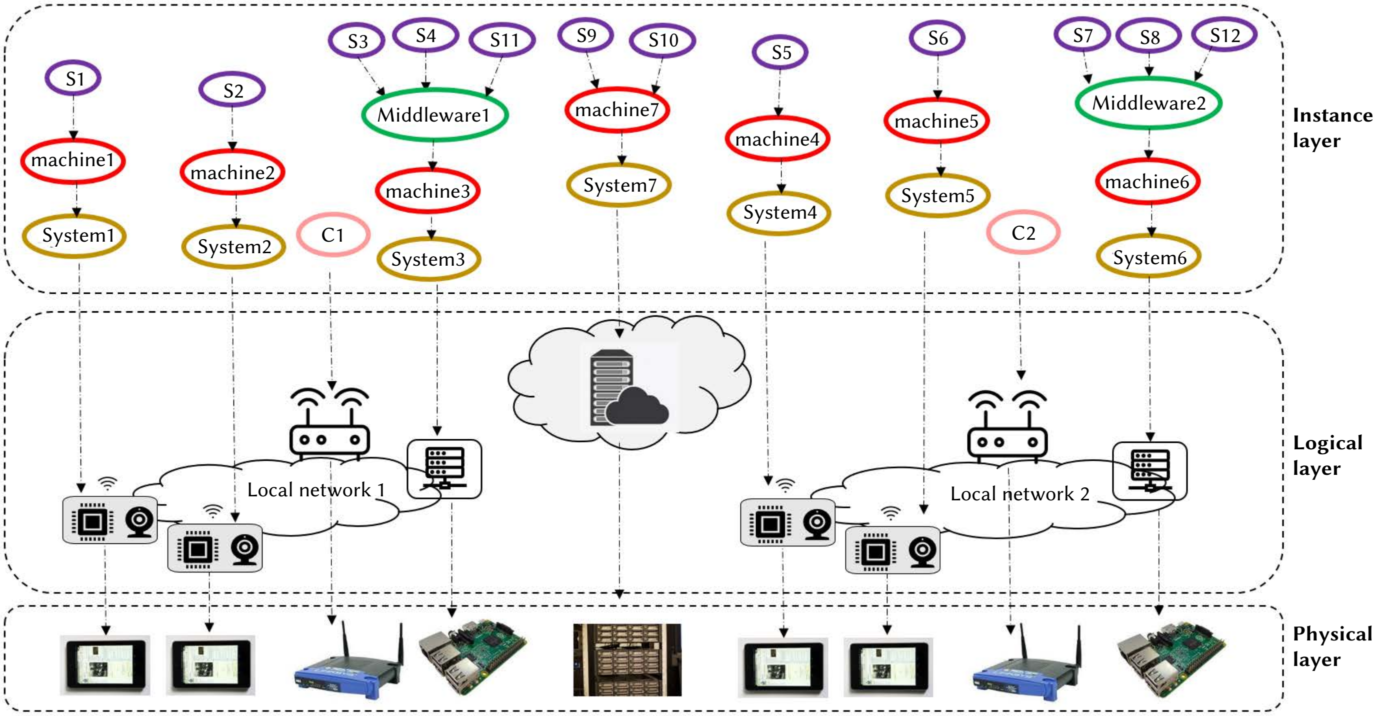

Fig. 6. Infrastructure and layout of the devices used in the use case.

infrastructure. The devices host software services and could provide data to other services. This data is processed by services to generate reports, propose actions or generate new information. According to Fig. 6, *S1*, *S2*, *S5*, *S6* retrieve data from the sensors integrated with them. *S1*, *S2*, *S5*, *S6* are distributed in geographic positions. *S1* and *S2*, located in local network 1, send data to *S4* and *S3* correspondingly (*S3*, *S4* and *S11* are part of *APP1*). *S11* sends to *S9* information processed from data received by *S3*. *S9* is hosted in the cloud (*machine7*). It unifies the information received by remote services and *S10* runs high-level tasks and provides results to end users. In the case of local network 2, the disposition of the elements and their functions are similar to local network 1.

The connections between the services as well as their relationships are shown in Fig. 7. This figure represents the knowledge graph contained in the Deployed Infrastructure Repository hosted in the CMA.

LMAs are installed on *machine1*, *machine2*, *machine4* and *machine5*. LMAs monitor the software deployed on those devices and, if necessary, generate the corresponding events. Listing 3 shows an example of events generated by LMA1, located on *machine1*, and which sends the events to CMA1 (deployed on *machine3*).

The events have parameters related to the operation of the sensor connected to *S1*. The CEP component of CMA1 (at *machine3*) continuously reads and processes the received events in order to detect abnormal situations. If a sensor is damaged, it will produce different types of errors that are reflected in the event stream. For example, if the sensor generates errors of the type *highLightConditions*, *lowLightConditions*, *abnormalReading*, *OutOfRangeReading*, etc. the reason could be that the sensor is damaged. In this case, the CEP will detect this situation and will generate a *eco:DataColletionFailed* entity. In particular, event1 indicates that service *S1* is notifying errors related to measurements taken from a sensor. Event1 has active the abnormalReading and *OutOfRangeReading* parameters, which indicate some problem with the measurement taken from the sensor. Event1 also indicates that service *S1* is generating the error and that *S1* is deployed on *machine1*.

Listing 3: Events generated by the use case shown in Fig. 7

```
: event1 rdf : type eco : Event ;
    eco : date " Jan 20 18:10:29" ;
    eco : relatedToDevice : machine1 ;
    eco : relatedToSoftware :S1 ;
    eco : abnormalReading " TRUE ";
    eco : OutOfRangeReading " TRUE ".
: event2 rdf : type eco : Event ;
    eco : date " Jan 20 18:11:41" ;
    eco : relatedToDevice : machine1 ;
    eco : relatedToSoftware :S1 ;
    eco : lowLightConditions " TRUE ".
: event3 rdf : type eco : Event ;
    eco : date " Jan 20 18:11:57" ;
    eco : relatedToDevice : machine1 ;
    eco : relatedToSoftware :S1 ;
    eco : batteryChargeLevel "10".
: event4 rdf : type eco : Event ;
    eco : date " Jan 20 18:11:41" ;
    eco : relatedToDevice : machine1 ;
    eco : relatedToSoftware :S1 ;
    eco : highLightConditions " TRUE ".
```

In addition to the detected problem of the sensor, the indications that the battery is low (10%) contained in event3 might be relevant. This event indicates that the sensor failure could be associated with the fact that the sensor does not have enough battery.

The CEP component will evaluate the events and their implications on the system depending on the information contained in the event stream. For example, Listing 4 detects the *eco:DataCollectionFailed* problem if the light conditions change between low and high in a short period of time (60s) and the reported sensor values are out of range. In that case, the *eco:DataCollectionFailed* entity is added to the Detected Problems Repository.

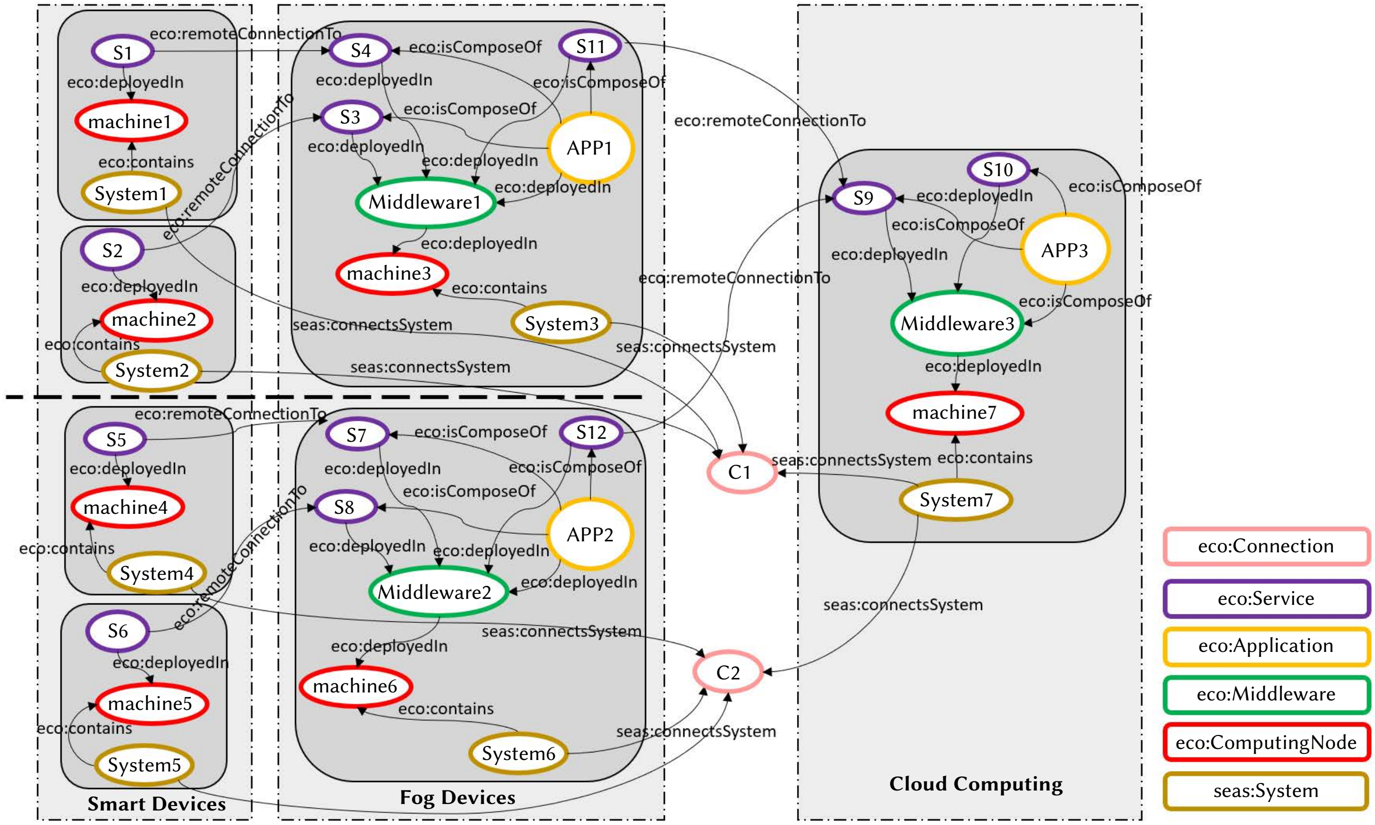

Fig. 7. Knowledge graph representing the deployed architecture. The colour of a node indicates the ontological concept (on the right side of the figure) it instantiates. E.g. S1 is an instance of eco:Service.

Listing 4: C-SPARQL query to identify data collection problems

```
CONSTRUCT {
     _: prob rdf : type eco : DataCollectionFailed .
     _: prob eco : relatedTo ? service
}
FROM STREAM
     : streamExample1
          [ RANGE 60s STEP 30s]
FROM < instancesTopology .owl >
WHERE {
     ? event rdf : type eco : Event .
     ? event eco : relatedTo ? machine .
     ? event eco : lowLightConditions " TRUE ".
     ? event eco : highLightConditions " TRUE ".
     ? event eco : OutOfRangeReading " TRUE ".
}
```

The Inference Service tries to find other elements of the IoT system that might be affected by the problems registered in the DPR. For this, the IS analyses the system architecture described in the Deployed Infrastructure Repository. As a result, the IS infers that *S4* is affected since it is connected to *S1*. Thus, the software application *APP1* is also affected. Furthermore, the problem could be propagated to *S9*, because *S11* (*S11* forms part of *APP1*) is connected to *S9* (by *eco:remoteConnecionTo* property), and this could affect APP3. The affected software components can be detected with standard SPARQL queries. Listing 5 shows an example query to retrieve the applications affected by the malfunction of *S1*.

The query considers four cases: (1) the problematic service is directly part of an application, (2) an application contains a service that is remotely connected to a problematic service (APP1 is identified), (3) an application is remotely connected to an application that contains the problematic service, and (4) an application is remotely connected to another application that includes a service connected to the problematic service (*APP3* is identified).

Listing 5: Query identifying applications affected by remote unconnected services

```
SELECT ? affectedApp
WHERE {
   ? faultServ eco :id "S1" .
   { # Case 1
   ? affectedApp eco : isComposedOf ? faultServ .}
UNION { # Case 2
   ? affectedApp eco : isComposedOf ? servAux .
   ? faultServ eco : remoteConnectionTo + ? servAux .}
UNION { # Case 3
   ? appAux eco : isComposedOf ? faultServ .
   ? appAux eco : isComposedOf ? serv .
   ? serv eco: remoteConnectionTo + ? serviceAux2 .
   ? affectedApp eco : isComposedOf
   ? serviceAux2 .}
UNION { # Case 4
   ? faultServ eco : remoteConnectionTo +
   ? serviceAux4 .
   ? appAux eco : isComposedOf ? serviceAux4 .
   ? appAux eco : isComposedOf ? serv .
   ? serv eco: remoteConnectionTo + ? servAux .
   ? affectedApp eco : isComposedOf ? servAux .}
}
```

Once a problem and its scope have been identified, the Action Schedule Service (ASS) will decide what actions should be taken to resolve or alleviate the problem. The possible solutions to problems will be context dependent. For the above case, for example, the ASS may decide whether or not to stop *APP1* and *APP3* until the problem with the sensor connected to S1 is solved. Alternatively, it may replace some of the services that are affected by the problem. For example, *S4* could be replaced by another service that does not require the information from *S1*. In this way, *APP1* could adapt to the new situation and operate consistently. If *APP1* works correctly then *APP3* would also not be affected by the problem with *S1* and would also work correctly.

The ASS is an independent component that takes information from the DIR, interprets it and proposes corrective actions. The architecture has been designed with the objective that the ASS has a low coupling, this allows to have several implementations of the ASS with different AI mechanisms facilitating to experiment with different AI techniques. The complete design of the ASS is part of our future work. In this work, we propose using the Jena[4] rule-based system, which allows easy integration of ontologies and rules. The proposed rules are activated depending on the information about the current status of the infrastructure (especially malfunctioning issues) contained in the DIR. When a rule is fired the specified actions are executed, which typically define the changes to be applied in the system.

VI. Conclusion

In this work, we have proposed a solution approach that aims at detecting and eventually resolving anomalous situations or malfunctions in IoT systems. The approach uses several mechanisms distributed on independent intelligent agents that collaborate with each other. These agents process the log registers generated by software installed in IoT devices and detect problems and malfunctions that may compromise the operation of the IoT system. The ability to understand messages contained in a log register is complex. For this reason, we propose using a list of parameters that help to identify and describe undesired situations of the elements that compose an IoT system. The Lightweight Management Agent generates events from messages contained in log registers and each event contains information about the status of an IoT device. LMAs send those events to Complex Management Agents, which process them in order to identify problems. CMAs use knowledge graphs (based on the ECO ontology) to structure the system information such as the topology, the deployed software and possible problems (undesired situations). They use this knowledge to infer new information, in particular, to identify the scope to which an identified problem affects the entire IoT system. Based on this information, corrective actions can be carried out to bring back the IoT system to a desired state. All these mechanisms provide a viable solution for the auto-maintenance of IoT systems. The proposed approach can be deployed in conjunction with third-party IoT systems since it can be adapted and integrated with existing solutions that have been designed and deployed for specific tasks.

Our work is subject to some limitations that we plan to address in future work. In a first step, we will extend the list of parameters proposed in this work. We will also apply our solution to more complex real-world environments so as to further back its versability and analyse its performance. To this end, we rely on the Mininet5 simulator for large-scale experiments. Also, in this work, we focused on problem detection. As a next step, we will concentrate on analysing the automatic execution of corrective actions to resolve detected problems (Action Schedule Service in our architecture).

[4] https://jena.apache.org

Acknowledgment

This work has been supported by grant VAE: TED2021-131295B-C33 funded by MCIN/AEI/ 10.13039/501100011033 and by the “European Union NextGenerationEU/PRTR”, by grant COSASS: PID2021-123673OB-C32 funded by MCIN/AEI/10.13039/501100011033 and by “ERDF A way of making Europe”, and by the AGROBOTS Project of Universidad Rey Juan Carlos funded by the Community of Madrid, Spain. Iván Bernabé has been funded by the Spanish Ministry of Universities through a grant related to the Requalification of the Spanish University System 2021–23 by the University Carlos III of Madrid.

References

[1] U. Cisco, “Cisco annual internet report (2018–2023) white paper,” *Cisco: San Jose, CA, USA*, vol. 10, no. 1, pp. 1–35, 2020.

[2] S. Qiu, K. Cheng, T. Zhou, R. Tahir, L. Ting, “An eeg signal recognition algorithm during epileptic seizure based on distributed edge computing,” *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 7, no. 5, pp. 6–13, 2022, doi: 10.9781/ijimai.2022.07.001.

[3] S. Pan, X. Gu, Y. Chong, Y. Guo, “Content-based hyperspectral image compression using a multi- depth weighted map with dynamic receptive field convolution,” *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 7, no. 5, pp. 85–92, 2022, doi: 10.9781/ijimai.2022.08.004.

[4] M. M. Ogonji, G. Okeyo, J. M. Wafula, “A survey on privacy and security of internet of things,” *Computer Science Review*, vol. 38, p. 100312, 2020, doi: https://doi.org/10.1016/j.cosrev.2020.100312.

[5] H. Mrabet, S. Belguith, A. Alhomoud, A. Jemai, “A survey of iot security based on a layered architecture of sensing and data analysis,” *Sensors*, vol. 20, no. 13, p. 3625, 2020.

[6] K. Gulati, R. S. K. Boddu, D. Kapila, S. L. Bangare, N. Chandnani, G. Saravanan, “A review paper on wireless sensor network techniques in internet of things (iot),” *Materials Today: Proceedings*, vol. 51, pp. 161–165, 2022.

[7] S. Rani, A. Kataria, V. Sharma, S. Ghosh, V. Karar, K. Lee, C. Choi, “Threats and corrective measures for iot security with observance of cybercrime: A survey,” *Wireless communications and mobile computing*, vol. 2021, pp. 1–30, 2021.

[8] J. Seeger, A. Bröring, G. Carle, “Optimally self-healing iot choreographies,” *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 3, pp. 1–20, 2020.

[9] D. Weyns, *Software Engineering of Self-adaptive Systems*, pp. 399–443. Cham: Springer International Publishing, 2019.

[10] O. Gheibi, D. Weyns, F. Quin, “Applying machine learning in self-adaptive systems: A systematic literature review,” *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 15, no. 3, pp. 1–37, 2021.

[11] C. Zhu, G. Pastor, Y. Xiao, Y. Li, A. Ylae-Jaeaeski, “Fog following me: Latency and quality balanced task allocation in vehicular fog computing,” in *2018 15th Annual IEEE international conference on sensing, communication, and networking (SECON)*, 2018, pp. 1– 9, IEEE.

[12] Z. Liu, X. Yang, Y. Yang, K. Wang, G. Mao, “Dats: Dispersive stable task scheduling in heterogeneous fog networks,” *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 3423–3436, 2018.

[13] G. Zhang, F. Shen, N. Chen, P. Zhu, X. Dai, Y. Yang, “Dots: Delay-optimal task scheduling among voluntary nodes in fog networks,” *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 3533–3544, 2018.

[14] L. Sun, Y. Li, R. A. Memon, “An open iot framework based on microservices architecture,” *China Communications*, vol. 14, no. 2, pp. 154–162, 2017.

[15] A. Celesti, L. Carnevale, A. Galletta, M. Fazio, M. Villari, “A watchdog service making container- based micro-services reliable in iot clouds,” in *2017 IEEE 5th international conference on future internet of Things and Cloud (fiCloud)*, 2017, pp. 372–378, IEEE.

[16] A. Krylovskiy, M. Jahn, E. Patti, “Designing a smart city internet of things platform with microservice architecture,” in *2015 3rd international conference on future internet of things and cloud*, 2015, pp. 25–30, IEEE.

[17] S. He, J. Zhu, P. He, M. R. Lyu, “Experience report: System log analysis for anomaly detection,” in *2016 IEEE 27th international symposium on*

software reliability engineering (ISSRE), 2016, pp. 207–218, IEEE.

[18] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, X. Qu, *et al.*, "Syslog processing for switch failure diagnosis and prediction in datacenter networks," in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, 2017, pp. 1–10, IEEE.

[19] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, R. Sasnauskas, "A search-based approach for accurate identification of log message formats," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 167–177.

[20] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of the 3rd IEEE Workshop on IP Operations and Management (IPOM 2003) (IEEE Cat. No. 03EX764)*, 2003, pp. 119–126, Ieee.

[21] R. Vaarandi, M. Pihelgas, "Logcluster-a data clustering and pattern mining algorithm for event logs," in *2015 11th International conference on network and service management (CNSM)*, 2015, pp. 1–7, IEEE.

[22] N. F. Noy, D. L. McGuinness, *et al.*, "Ontology development 101: A guide to creating your first ontology," 2001.

[23] T. Bittner, M. Donnelly, S. Winter, "Ontology and semantic interoperability," in *Large-scale 3D data integration*, CRC Press, 2005, pp. 139–160.

[24] R. Jasper, M. Uschold, *et al.*, "A framework for understanding and classifying ontology applications," in *Proceedings 12th Int. Workshop on Knowledge Acquisition, Modelling, and Management KAW*, vol. 99, 1999, pp. 16–21, Citeseer.

[25] J. Agbaegbu, O. T. Arogundade, S. Misra, R. Damaševičius, "Ontologies in cloud computing—review and future directions," *Future Internet*, vol. 13, no. 12, p. 302, 2021.

[26] S. Jaskó, A. Skrop, T. Holczinger, T. Chován, J. Abonyi, "Development of manufacturing execution systems in accordance with industry 4.0 requirements: A review of standard- and ontology-based methodologies and tools," *Computers in Industry*, vol. 123, p. 103300, 2020, doi: https://doi.org/10.1016/j.compind.2020.103300.

[27] A. Heidari, N. Jafari Navimipour, "Service discovery mechanisms in cloud computing: a comprehensive and systematic literature review," *Kybernetes*, vol. 51, no. 3, pp. 952–981, 2022.

[28] M. M. Al-Sayed, H. A. Hassan, F. A. Omara, "Cloudfnf: An ontology structure for functional and non- functional features of cloud services," *Journal of Parallel and Distributed Computing*, vol. 141, pp. 143–173, 2020, doi: https://doi.org/10.1016/j.jpdc.2020.03.019.

[29] V. Singh, S. Pandey, "Cloud security ontology (cso)," *Cloud Computing for Geospatial Big Data Analytics: Intelligent Edge, Fog and Mist Computing*, pp. 81–109, 2019.

[30] F. Moscato, R. Aversa, B. Di Martino, T.-F. Fortiş, V. Munteanu, "An analysis of mosaic ontology for cloud resources annotation," in *2011 federated conference on computer science and information systems (FedCSIS)*, 2011, pp. 973–980, IEEE.

[31] K. U. Sri, M. B. Prakash, J. Deepthi, "A frame work to dropping cost in passage of cdn into hybrid cloud," *Int.J. Innov. Technol. Res*, vol. 5, no. 2, pp. 5829–5831, 2017.

[32] E. Di Nitto, G. Casale, D. Petcu, *et al.*, "On modaclouds' toolkit support for devops," in *4th European Conference on Service Oriented and Cloud Computing Workshops (ESOCC)*, 2016, pp. 430–431.

[33] K. Taylor, A. Haller, M. Lefrançois, S. J. Cox, K. Janowicz, R. Garcia-Castro, D. Le Phuoc, J. Lieberman, R. Atkinson, C. Stadler, "The semantic sensor network ontology, revamped.," in *JT@ ISWC*, 2019.

[34] L. Daniele, F. den Hartog, J. Roes, "Created in close interaction with the industry: the smart appliances reference (saref) ontology," in *Formal Ontologies Meet Industry: 7th International Workshop, FOMI 2015, Berlin, Germany, August 5, 2015, Proceedings 7*, 2015, pp. 100–112, Springer.

[35] B. Ontology, "onem2m technical specification: Ts-0012- v3.7.3.."

[36] Q.-D. Nguyen, C. Roussey, M. Poveda-Villalón, C. de Vaulx, J.-P. Chanet, "Development experience of a context-aware system for smart irrigation using caso and irrig ontologies," *Applied Sciences*, vol. 10, no. 5, p. 1803, 2020.

[37] S. R. U. Kakakhel, L. Mukkala, T. Westerlund, J. Plosila, "Virtualization at the network edge: A technology perspective," in *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, 2018, pp. 87– 92.

[38] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Boston, MA: Addison-Wesley, 2002.

[39] A. Hogan, E. Blomqvist, M. Cochez, C. D'amato, G. D. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, A.-C. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, A. Zimmermann, "Knowledge graphs," *ACM Computing Surveys*, vol. 54, pp. 1–37, jul 2021, doi: 10.1145/3447772.

[40] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, J. Taylor, "Industry-scale knowledge graphs: Lessons and challenges," *Communications of the ACM*, vol. 62 (8), pp. 36–43, 2019.

[41] I. Bernabé, A. Fernández, H. Billhardt, S. Ossowski, "Towards semantic modelling of the edge-cloud continuum," in *Highlights in Practical Applications of Agents, Multi-Agent Systems, and Complex Systems Simulation. The PAAMS Collection*, Cham, 2022, pp. 71– 82, Springer International Publishing.

[42] M. Lefrançois, J. Kalaoja, T. Ghariani, A. Zimmermann, *The SEAS Knowledge Model.* PhD dissertation, ITEA2 12004 Smart Energy Aware Systems, 2017.

[43] G. Salgueiro, V. Gurbani, A. Roach, "Format for the session initiation protocol (sip) common log format (clf)," 2013.

[44] R. Gerhards, "Rfc 5424: The syslog protocol," 2009.

[45] C. Lonvick, "The bsd syslog protocol," 2001.

[46] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, J. Widom, "A pipelined framework for online cleaning of sensor data streams," in *22nd International Conference on Data Engineering (ICDE'06)*, 2006, pp. 140–140, IEEE.

[47] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, M. Grossniklaus, "Querying rdf streams with c- sparql," *ACM SIGMOD Record*, vol. 39, no. 1, pp. 20–26, 2010.

[48] B. DuCharme, *Learning SPARQL: querying and updating with SPARQL 1.1.* " O'Reilly Media, Inc.", 2013.

Iván Bernabé-Sánchez

Iván Bernabé-Sánchez received a degree in information technology engineering from the Carlos III University of Madrid, Leganés, Spain, in 2007 and a PhD degree in 2021. His research interests include the virtualization of devices and infrastructures defined through code, cloud and edge computing architectures, and self-configuration systems mechanisms based on knowledge representation and semantic technologies. He has participated in several nationally or internationally funded research projects.

Alberto Fernández

Alberto Fernández is a full professor at the University Rey Juan Carlos (URJC) in Madrid, where he is a member of the Artificial Intelligence Group of the CETINIA research centre. He obtained a PhD in Computer Science from the URJC. His main research lines are multi-agent systems, knowledge representation, semantic technologies, open systems, etc. He is especially interested in the application of previous technologies in domains such as intelligent transportation systems, fleet management, etc. He has participated in many national and international projects on the above topics and has published more than 80 articles in international journals, books and conferences.

Holger Billhardt

Holger Billhardt received his M.Sc. in computer science from the TH Leipzig, Germany, and his PhD in computer science at the Universidad Politécnica in Madrid. He is currently a full professor of computer science at Universidad Rey Juan Carlos in Madrid, where he is a member of the Artificial Intelligence Group at the Centre for Intelligent Information Technologies (CETINIA). His research is concerned with multi-agent systems, especially with the coordination of agents in distributed, open and dynamic environments. He is the author or co-author of more than 100 research papers and has participated in several nationally or internationally funded research projects.

Sascha Ossowski

Sascha Ossowski is a full professor of computer science and director of the CETINIA research centre at the University Rey Juan Carlos in Madrid. He received a MSc degree in informatics from U Oldenburg (Germany) and a PhD in artificial intelligence from TU Madrid (Spain). The main themes of his research refer to models and mechanisms for coordination in all sorts of agent systems and environments. He was co-founder of the European Association for Multiagent Systems (EURAMAS), chaired the European COST Action on Agreement Technologies, and is an emeritus board member of the International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS).