Computers & Security 120 (2022) 102829

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Efficient ABAC based information sharing within MQTT environments under emergencies

Pietro Colombo*, Elena Ferrari, Engin Deniz Tümer

Department of Theoretical and Applied Science, University of Insubria, Via O. Rossi, 9, Varese 21100, Italy

ARTICLE INFO

Article history:
Received 24 December 2021
Revised 2 June 2022
Accepted 4 July 2022
Available online 6 July 2022

Keywords:
ABAC
Emergency policies
MQTT environments
Emergency detection
Complex event processing

ABSTRACT

Recent emergencies, such as the COVID-19 pandemic have shown how timely information sharing is essential to promptly and effectively react to emergencies. Internet of Things has magnified the possibility of acquiring information from different sensors and using it for emergency management and response. However, it has also amplified the potential of information misuse and unauthorized access to information by untrusted users. Therefore, this paper proposes an access control framework tailored to MQTT-based IoT ecosystems. By leveraging Complex Event Processing, we can enforce controlled and timely data sharing in emergency and ordinary situations. The system has been tested with a case study that targets patient monitoring during the COVID-19 pandemic, showing promising results.

1. Introduction

Recent emergencies, such as the COVID-19 pandemic, have shown that, due to scarcely information sharing, emergency protocols often fail in fully achieving their goals. For instance COVID-19 contact tracing has been pointed out by the World Health Organization as a strategic tool for contrasting SARS-CoV-2 diffusion and reducing COVID-19 mortality.[1] However, manual contact tracing methods proved scarcely applicable, highly demanding in terms of time and human resources, and overall impractical with a high number of new daily cases. Moreover, people's ability and willingness to derive and disclose sensitive information, as visited places and persons met, have further hindered their application and efficacy. Contact tracing apps have addressed the scalability and performance issues of manual methods. However, due to a scarce perception of enforced data protection, in several western countries, citizens proved unavailable to install and use these apps (Akinbi et al., 2021). As a consequence, the efficacy of contact tracing has been undermined by limited population coverage. These facts suggest that efficient data sharing is a key requirement for emergency management, and should be complemented with proper data protection tools.

Efficient emergency management starts with timely identification of an emergency through the analysis of what has occurred in a target scenario and requires that all resources needed to properly handle the identified emergency are timely accessed by authorized subjects. Internet of Things (IoT) technologies provide valid support to the development of efficient data sharing and analysis services and thus appear well suited for building emergency management applications. Data can be gathered by manifold types of smart devices which are nowadays available for different domains. For instance, for what healthcare is concerned, medical wearables and IoT technologies are enabling new forms of diagnosing and care and allow the detection of emergency situations. As an example, during the COVID-19 pandemic, the OLVG Hospital in the Netherlands started to experiment with wearable biosensors able to detect possible deterioration of suspected or confirmed COVID-19 patients.[2] The proposed monitoring framework aimed at limiting the interaction of patients with the medical personnel, favoring at the same time a prompt intervention, when required.

The management of an emergency typically also requires granting exceptional privileges to subjects, which in an ordinary situation would not be permitted. For instance, in an ordinary situation, a physician responsible to provide treatment has to ensure that valid consent has been obtained from the patient or a delegated person before the treatment can begin. However, if an emergency

* Corresponding author.
E-mail addresses: pietro.colombo@uninsubria.it (P. Colombo), elena.ferrari@uninsubria.it (E. Ferrari), edtumer@uninsubria.it (E.D. Tümer).

[1] https://www.who.int/publications/i/item/contact-tracing-in-the-context-of-covid-19.

[2] https://www.bioworld.com/articles/435384-philips-debuts-wearable-vitals-sign-patch-to-monitor-covid-19-patients-for-early-intervention.

https://doi.org/10.1016/j.cose.2022.102829

occurs and the treatment is finalized to save the patient life, it can be provided without consent. Nonetheless, all granted exceptional privileges have to be immediately revoked as soon as the emergency is over.

The enforcement of similar controls within IoT applications requires mechanisms able to regulate the access to data gathered by IoT devices. Although a variety of access control models for IoT applications have been proposed in the literature (e.g., see Qiu et al., 2020 for a compendium), just a few of them allow regulating data sharing in emergency situations (see Section 9). All these proposals rely on a permission management approach known as *break the glass* (BtG), according to which, during an emergency, a user requests and then gains access to resources that would not be permitted to him/her in normal conditions. Although extremely flexible, BtG approaches have drawbacks. For instance, the accesses executed after breaking the glass are normally traced for later reviews (Schefer-Wenzl et al., 2014), opening the way to possible information leakage. In addition, the abuse of BtG policies can lead the system to an unsafe state (Carminati et al., 2013), thus sufficient ordinary access control policies should always be specified to minimize the necessity of breaking the glass. Although not targeting the IoT domain, complementary approaches to regulate information sharing in emergency situations have been proposed (Carminati et al., 2013; Kabbani et al., 2014), where emergency policies were introduced to grant subjects all privileges needed for the management of specific emergencies, as soon as they occur. Since emergency management plans are expected to be a priori defined, emergency policies could be specified in such a way to fulfill information-sharing requirements elicited from the associated plan. As an example, an a priori defined protocol is expected for the above-mentioned patient monitoring scenario, which, under specific emergencies, allows medical personnel with certain functions to access patients' physiological data (e.g., in case of a severe cardiovascular issue, the privileges should be granted to cardiologists). Permission management based on emergency policies allows shorting data access time, as no request to override permission has to be issued, and data can thus be received by authorized subjects as soon as the emergency begins. However, to the best of our knowledge, none of the previous approaches targets the IoT domain.

In this paper, we do a step to fill this void, by proposing an Attribute-based Access Control (ABAC) framework to regulate data sharing within MQTT-based IoT applications in ordinary and emergency situations. The choice of targeting MQTT is motivated by the wide adoption of this protocol within IoT applications for inter-device communication, whereas ABAC has been selected as it has already been profitably used to regulate data sharing on the basis of context properties (e.g., see Colombo and Ferrari, 2018; Colombo et al., 2021), which makes it a good fit for emergency policy support. As a matter of fact, policy selection requires evaluating access request contexts, checking whether the subject that aims at sending and receiving an MQTT message is involved in emergency situations.

The proposed system is an extension of the framework we proposed in Colombo and Ferrari (2018) that supports ordinary fine-grained access control policy enforcement in MQTT environments. Key novel features introduced by this paper include:

- modeling support required to: i) define the events that trigger an emergency, ii) bind events to MQTT messages, iii) specify emergency situations along with their possible evolution, and iv) specify emergency policies;
- emergency management functionalities, that is, the ability to: i) detect occurrences of modeled events starting from the analysis of MQTT control packets exchanged in a monitored application, and ii) identify the possible evolution of emergency situations. For event detection, we leverage on a complex event processing engine;
- access control capabilities, that is, the ability to enforce both regular and emergency access control policies which apply to an access request issued in a specific context.

To show the feasibility of the proposed approach we apply our framework to a case study of pseudo-realistic complexity related to an MQTT-based health monitoring application employed in a nursing home during the COVID-19 pandemic. Our framework is here employed to regulate information sharing within the considered application, with the aim to ensure that in ordinary and emergency situations data can only be accessed by authorized subjects. The proposed case study allows us to exemplify the definition of all modeling artifacts required to configure the framework for the considered application. The case study has also been employed for an early performance evaluation of the proposed approach, overall showing a reasonably low enforcement overhead.

The remainder of the paper is organized as follows. Section 2 introduces a running scenario that will be used throughout the paper to exemplify basic framework concepts, and which will be also developed into a case study. In Section 3, we introduce background technologies instrumental for the framework definition. Section 4 presents key concepts related to the proposed event modeling approach, whereas Section 5 introduces the foundations of our access control model. Section 6 provides an overview of the framework architecture and shortly presents the rationale of the enforcement mechanism, which is then more thoroughly analyzed in Section 7. In Section 8 we present a case study, an early experimental evaluation of the framework performance, discussing possible addressing strategies for identified framework limitations. Section 9 presents a short survey of related work, whereas Section 10 concludes the paper.

2. Running example

Let us consider an IoT application that aims at monitoring the health conditions and behaviors of patients hosted in a nursing home during the COVID-19 pandemic. IoT devices worn by patients and sensors deployed in the rooms where patients live allow the real-time monitoring of patients' conditions. For instance, body temperature and respiratory rate are vital signs of patients that can be acquired by wearable biosensors, whereas the location of patients can be collected by indoor tracking bracelets. The acquired data are stored, and can therefore be visualized and analyzed by dedicated monitoring apps used by medical personnel of the nursing home, by selected relatives of the patients who can check the conditions of their kin, and even by self-sufficient patients who wish to check their own conditions. Medical personnel has access to physiological and environmental data, whereas patients and relatives have limited authorizations.

Patients can occasionally face emergency conditions, which require a prompt reaction from the medical personnel. To effectively manage some emergencies, it is required to share patients data in critical conditions with external physicians with the aim to promptly identify proper treatment. The monitored data is also used to contrast COVID-19 diffusion. Temperature and oxygen saturation levels reveal potential COVID-19 symptoms and could be used to notify physicians to make a test. The access to proximity data of infected patients, and the immediate isolation of potentially infected guests, allow contrasting COVID-19 diffusion (Ouslander and Grabowski, 2020).

Patients data has to be accessed by authorized users in any possible situation. The considered scenario emphasizes the need for special policies to enforce access control during emergencies that we will illustrate in the following sections.

Table 1
MQTT control packets.

Control packet	Acronym	Description
CONNECT	cp_{CN}	Connection request
CONNACK	cp_{CA}	Connect acknowledgment
PUBLISH	cp_{PB}	Publish message
PUBACK	cp_{PA}	Publish acknowledgement
PUBREC	cp_{PRC}	Publish received
PUBREL	cp_{PRL}	Publish release
PUBCOMP	cp_{PC}	Publish complete
SUBSCRIBE	cp_{SB}	Subscribe to topics
SUBACK	cp_{SA}	Subscribe acknowledgement
UNSUBSCRIBE	cp_{US}	Unsubscribe from topics
UNSUBACK	cp_{UA}	Unsubscribe acknowledgement
PINGREQ	cp_{PRQ}	PING request
PINGRESP	cp_{PRS}	PING response
DISCONNECT	cp_{DS}	Disconnect notification

3. Background

In this section, we shortly present key aspects of MQTT, Complex Event Processing, and the access control model for MQTT environments we proposed in Colombo and Ferrari (2018), since they are instrumental for the definition of the proposed emergency management framework.

3.1. MQTT

MQTT is a popular protocol for IoT applications and can be used in a variety of IoT scenarios (Mishra and Kertesz, 2020). It allows peers of an IoT ecosystem to communicate by means of the publish/subscribe paradigm. In an MQTT environment, multiple clients exchange messages by means of a message broker. MQTT *clients* connect to an MQTT broker to send or receive application messages. Clients can subscribe to the reception of messages on the topics caught by a topic filter specification *tf*, or can request the MQTT broker to publish messages on given topics. MQTT *brokers* route messages on the basis of the associated topic. A topic filter, in turn, is a textual expression, which, by employing special characters, denoted as wildcards, allows referring to multiple topics. Whenever a broker receives a publishing request on a topic *t*, it forwards the message to any client that subscribed to the reception of messages on topics that include *t*.

Broker and clients interact by exchanging control packets (Banks et al., 2019). The list of MQTT control packets is reported in Table 1.

Let us refer to the broker of an MQTT environment as *b*. In order to connect with *b*, for sending or receiving messages, an MQTT client *c* sends a connection request cp_{CN} to b. On receipt of this packet, *b* evaluates the request and sends back an acknowledgment cp_{CA} to *c* which specifies whether the request has been accepted and thus a connection has been opened. Once a connection is established, *c* can request to publish an application message on a topic *t* with a payload *p*, by issuing a publishing request packet cp_{PB}. In addition, *c* can also request the reception of messages on topics that match a topic filter *tf*, by sending a subscription request packet cp_{SB} to the broker.

The topic *t* referred to in a packet cp_{PB} is a string composed of a sequence of / separated tokens, referred to as topic levels. The protocol does not constrain the format of a message payload *p*. However, a predominant data-interchange format adopted in numerous MQTT-based applications is JSON (Bray, 2017). Therefore, in this paper, we assume the payload is formatted as a JSON object[3] and thus represented as a set of hierarchically organized key-value pairs. An interested reader can refer to Banks et al. (2019) for further details on the MQTT protocol.

[3] https://www.json.org/.

Table 2
An abstract event algebra for complex event specification (Giatrakos et al., 2020).

$ce ::=$	pe	Primitive Event
	$ce_1 \; ; \; ce_2$	Sequence
	$ce_1 \vee ce_2$	Disjunction
	$ce_1 \wedge ce_2$	Conjunction
	ce^*	Iteration
	$\neg \; ce$	Negation
	$\sigma_\theta \; (ce)$	Selection
	$\pi_m(ce)$	Projection
	$[ce]_{T_1}^{T_2}$	Windowing from T_1 to T_2

3.2. Complex event processing

A complex event processing (CEP) system is a framework whose primary aim is to understand what is happening in a system under analysis (Giatrakos et al., 2020). A CEP system is composed of a set of *event sources*, a *CEP engine*, and a group of *event sinks* (Cugola and Margara, 2012). Event sources are components devoted to: 1) identify changes of monitored system properties, and 2) notify to the CEP engine of a *primitive event* that denotes the change. A CEP engine is a tool that identifies the occurrence of specific situations in the monitored system. Situations are modeled as patterns of events, referred to as *complex events*, which occur in a time interval in the monitored system. Event sinks are notified of the occurrence of complex events by the CEP engine and are configured to promptly react to the identified conditions. A notification is an object with fields specifying a *time annotation*, which refers to the event generation time, a *payload*, which specifies the event content and is structured as a data record, and a *type*, which constrains the structure of the *payload* (Cugola and Margara, 2015).

Example 1. Let us consider a thermometer *th* which is used to monitor a patient's body temperature. The events generated by this event source may have payloads composed of attributes *temperature* and *deviceId*, which respectively denote the measured temperature and the device identifier. A measured temperature of 37 °C at time *ts* can thus be represented as a JSON object: *Temperature@ts: {"temperature":37, "deviceId":th }*, where *@ts* denotes the time annotation, and *{"temperature": float, "deviceId": string}* the associated type.

Complex events are usually specified using platform-specific languages. Although no universally recognized standard modeling language exists for specifying complex events, the majority of CEP engines allow specifying them within SQL-like queries (Giatrakos et al., 2020).

In order to specify complex events abstracting from platform-specific details, in this paper, we use the abstract event algebra presented in Giatrakos et al. (2020), whose operators are listed in Table 2.

A complex event *ce* is therefore defined by the composition of primitive and complex events, using a variety of operators (e.g, *sequence* (;), *disjunction* ($\vee$), *conjunction* ($\wedge$)).

Additionally, the *iteration* operator (*) allows the specification of a complex event as a set of events of the same type that occur repeatedly. In this case, *ce* occurs when the number of referred occurrences is reached. A complex event *ce* can also be defined by *negation* ($\neg$) of another event *ce'*, meaning that *ce* only occurs if *ce'* does not occur. Finally, *ce* can be modeled as a *selection* or *projection* of other events. The *selection* operator (σ_θ) filters events whose attribute values satisfy a condition θ, whereas the *projec-*

tion operator (π_m) extracts only part of the attributes, according to a set of mapping expressions m.

Any specification of a complex event *ce* can refer to events that occur in a specific time interval, specified through the *window* operator $[\ldots]_{T_1}^{T_2}$.

Example 2. Let us assume that sensors worn by patients periodically issue a primitive event $RespiratoryRate_{pe}$, which simply notifies the observed number of breaths per minute (bpm). A complex event $Breathlessness_{ce}$ which shows shortness of breath episode observed in the last 2 days, can thus be defined as:

$(\sigma_{bpm>25}(RespiratoryRate_{pe}))\ {}^{\text{now}}_{\text{now - 2 days}}$

3.3. Access control in MQTT environments

In this section, we shortly present the ABAC model for MQTT environments we proposed in Colombo and Ferrari (2018).

Subjects, objects, and environments are the basic building blocks of the proposed model. A subject *s* represents a client which, possibly on behalf of a user, connects to an MQTT broker with the aim of sending or receiving messages. *s* is characterized by attributes, such as the client identifier (*cID*) and, optionally, by a user identifier (*uID*), which denotes the user on behalf of whom client *cID* operates. Subjects who have similar access patterns can be classified into subject groups.

Application messages are the protection objects of the considered model. Therefore, control packet fields, such as the message topic *t* and payload *p* are object attributes. Lastly, an environment *e* represents the context within which an access request is issued, and could be characterized by attributes such as location, time, and access purpose. For the sake of simplicity, in this paper, we only consider time as environment attribute.

Data sharing is regulated on the basis of access control policies, specified by security administrators, which grant subjects the read/write access to messages on specific topic(s).[4]

Access control policies grant privileges under satisfaction of boolean expressions, denoted as parametric predicates, built by composition of subject, object and environment attributes, and sets of predefined operators and functions.[5]

Definition 1 (Access control policy (Colombo and Ferrari, 2018))**.** An access control policy *p* is a tuple ⟨*s, tf, exp, pr*⟩, where *s* refers to the subject(s) to whom *p* assigns privileges, denoting the identifier of a client, user, or user group, *tf* specifies a topic filter expression, which allows selecting by topic the protected messages, whereas *pr* specifies the *read* /*write* privilege granted by *p*, if the parametric predicate *exp* is satisfied.

Example 3. Let us consider an access control policy acp_1, which authorizes close relatives of patient p_1 to access his/her physiological data, and a second policy acp_2, which grants medical personnel access to every monitored data of any nursing home patient. Policy acp_1 can be specified as ⟨*relatives::*p_1*, p1/physiological/#, true, read*⟩, where *relatives::*p_1 denotes the group of users authorized by acp_1 to read messages that specify physiological data of *p1*. Similarly, acp_2 can be specified as: ⟨*medical_personnel, #, true, read*⟩.

To empower users with a finer-grained control on their data, the model in Colombo and Ferrari (2018) allows the specification of *user preferences*, namely user-defined policies that allow a user to further constrain the read privileges granted by the access control policies specified by security administrators.

[4] Read and write accesses respectively denote the privileges to send / receive messages on given topics.

[5] In Colombo and Ferrari (2018), we consider mathematical operators (>, <, =, +, −, *, /,%), logical operators (∧, ∨, ¬), set operators (∈, ⊂, ⊆, ∩, ∪, \), logical quantifiers (∀, ∃), and predefined functions that allow the processing of attributes values.

Definition 2 (User preference (Colombo and Ferrari, 2018))**.** A user preference *up* is a tuple ⟨*uid, tf, sub_exp*⟩, where *uid* specifies the identifier of a user who wishes to protect the access to messages published by any of the clients he/she handles, *tf* specifies a topic filter expression which refers to the topics of the messages published on behalf of *uid* to which *up* applies, whereas *sub_exp* is a parametric predicate specifying a precondition to the receiving of these messages.

Example 4. Let us now consider a user preference *up* specified by patient p_1 who wishes to limit the read access to his/her respiratory rate to medical personnel. *up* restricts the privileges granted by acp_1, and can be specified as: ⟨p_1*, p1/physiological/respiratory, s.gid==medical_personnel*⟩.

The ABAC framework proposed in Colombo and Ferrari (2018) provides an enforcement monitor which can be easily integrated into existing MQTT environments and which can operate with any MQTT client and broker.

A high-level view of the framework architecture (Colombo and Ferrari, 2018) is shown in Fig. 1. The monitor operates at the external interface of a trusted network where the broker and a NoSQL datastore, that keeps track of access control metadata, are deployed, whereas clients operate within untrusted external networks.

A client publishing request is intercepted by the monitor and, if at least one applicable policy is satisfied, the monitor authorizes the request. Otherwise, the publishing request is denied and the application message is blocked. If the publishing request is authorized by the specified access control policies, the monitor checks the existence of preferences specified by the user on behalf of whom the publishing request has been issued. The monitor embeds all applicable preferences into the payload of the message and sends it to the broker. The broker, on the basis of the referred topics, routes the received packets to the rightful subscriber clients. Any forwarded packet is again intercepted by the monitor, which checks whether the candidate's receiver client can actually receive the message. The monitor evaluates the user preferences embedded in the message payload. If no preference is satisfied, the packet is immediately blocked. In contrast, if at least one preference is satisfied, the monitor checks the access control policies that regulate the reception of messages by the client. If at least one access control policy authorizes the reception, the monitor removes the embedded user preferences from the message payload and sends it to the client. We refer the interested reader to Colombo and Ferrari (2018) for more details.

4. Event modeling

The proper management of an emergency requires identifying and modeling the events that cause the emergency. Therefore, in this section, we propose an approach to model events related to messages exchanged in a monitored MQTT environment.

Let us start to focus on the modeling of primitive events, which is achieved through the specification of primitive event types. A primitive event type specifies: 1) the structure of a class of primitive events, 2) the binding criteria of the considered events to the control packets exchanged in the ecosystem, and 3) the criteria to derive the event starting from the structural characteristics of the bound control packets.

A primitive event type is therefore modeled as a tuple ⟨*pet, adc, bcr, adf*⟩, where *pet* refers to the name of the event type, *adc* is a set of pairs ⟨*id, type*⟩ that specify the attributes that compose any event of type *pet, bcr* specifies the binding criteria, namely the conditions for a cp_{PB} control packet to trigger the generation of events of type *pet*, whereas *adf* is a set of pairs ⟨*id, exp*⟩, where the iden-

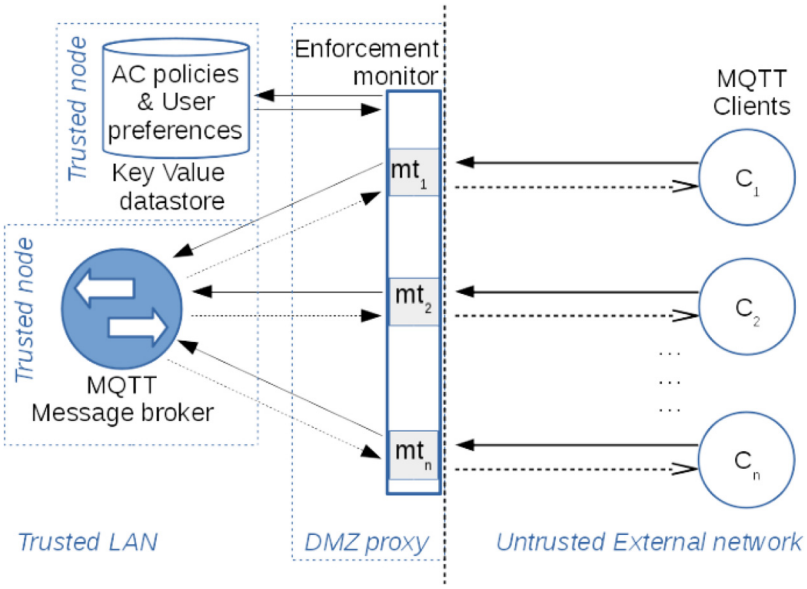

Fig. 1. An high-level view of the architecture in Colombo and Ferrari (2018).

tifier *id* refers to an attribute declared within *adc*, whereas *exp* is an initialization expression.

Boolean expressions that specify binding criteria are defined by referring to any structural property of a candidate control packet cp_{PB}, such as, for instance, the topic, the whole payload, or a payload attribute, and employing arithmetical, set and logical operators and quantifiers, as well as predefined functions. Binding criteria specify the required characteristics of a bound control packet *t*, referring to *t* like it was a JSON object (e.g., *t.payload* refers to the payload of the control packet). The same specification criteria are also employed for the initialization expressions within component *adf*, allowing one to refer to *t*'s properties, as well as to the subject, object and environment attributes related to the publishing request context of *t*. These attributes are referred to as fields of the objects *s, o*, and *e*, which represent the subject, object and environment associated with the publishing request *t*, respectively.

Example 5. Let us now consider the specification of a primitive event type *Temp* for messages published by MQTT thermometers. Let us assume that any publishing request that includes *"temperature"* in the topic name is bound to a primitive event of type *Temp*, in turn defined as a tuple ⟨*Temp, {"temp": float, "time": long, "pID": string}, t.TopicName.includes("temperature"), {"temp":t.Payload.temperature, "pID": o.patientID, "time": e.time}*⟩. It is worth noting that the initialization of the attributes is achieved referring to internal properties of the message payload,[6] and to object and environment attributes.

Let us now consider the modeling of complex events. Similar to primitive events, their modeling is achieved through the specification of an event type.

A complex event type is a tuple ⟨*cet, adc, ets, exp*⟩, where *cet* specifies the name of the event type, *adc* is a set of pairs ⟨*id, type*⟩ which specifies the attributes composing the payload of any event of type *pet, ets* is a set that includes the list of identifiers of primitive and complex event types referred to in the specification of *cet*, whereas *exp* is an expression defined with the abstract event algebra introduced in Section 3.2, which allows initializing the value of attributes declared within *adc. exp* is specified by referring to the event types in *ets*, and employing the event algebra operators (i.e., $;, \vee, \wedge, ^*, \neg, \sigma_\theta, \pi_m[]_{T1}^{T2}$, see Section 3.2).

Example 6. Let us now consider the specification of complex event type *Fever*, used to characterize events denoting that a specified patient has had a fever in the last 2 days. *Fever* can be defined as: ⟨*Fever, {"pID": string, "temp": float}, {Temp}, (*$\sigma_{temp > 38}(Temp_{pe})$*)*$^{\text{now}}_{\text{now - 2 days}}$⟩, where *Temp* is the primitive event type introduced in Example 5, and $Temp_{pe}$ is a primitive event of type *Temp*.

Similarly, let us assume that the primitive event type *RespiratoryRate* is defined as: ⟨*RespiratoryRate, {"bpm": float, "time":long, "pID": string}, t.TopicName.includes("respiratoryrate"), {"bpm": t.Payload.bpm, "pID": o.patientID, "time": e.time}*⟩. Referring to Example 2, the complex event type *Breathlessness* used to represent events notifying that a patient has had shortness of breath episodes in the last 2 days can be specified as:

⟨*Breathlessness, {"pID": string, "bpm": float}, {RespiratoryRate},* $(\sigma_{bpm > 25}(RespiratoryRate_{pe}))^{\text{now}}_{\text{now - 2 days}}$⟩.

The sets of primitive and complex event types specified for an application scenario are hereafter referred to as PET and CET, respectively.

5. Access control model

In this section, we present an extension of the ABAC model introduced in Section 3.3, which allows regulating data sharing within MQTT-based IoT environments in ordinary and emergency situations.

The proposed model is built on top of some key conceptual elements, respectively denoted as *emergency situation, emergency evolution*, and *action*, which are then used to define *emergency development plans, emergency scenarios*, and corresponding *emergency policies*.

An *emergency situation* is a critical situation that happens suddenly and requires prompt management to avoid harmful results. An emergency can evolve into another emergency, possibly more serious or mild, or it can be solved. Any emergency is characterized by a severity level that specifies its severity. In our model, an *emergency situation* is a single stage of an emergency scenario subject to possible evolution. Therefore, we model an emergency

[6] As already mentioned in Section 3.1, we assume that message payloads are structured as JSON objects.

situation *ems* as a pair ⟨*sid, lev*⟩, where *sid* specifies the emergency identifier, whereas *lev* denotes the related severity level. A severity level is an integer value in the range [L_{min}. L_{max}], configurable by the system administrator at specification time (where L_{min},$L_{max} \in \mathbb{N}$, and $1 \leq L_{min} \leq L_{max}$).

In contrast, an *action* is a task that converts an event of complex type into an MQTT publish request packet cp_{PB}^{CEP}, and forwards it to the MQTT environment.[7] An action is modeled as a tuple ⟨*aid, cet, tp, pl*⟩, where *aid* and *cet* respectively specify the identifier of the modeled action and of the referred complex event type, whereas *tp* and *pl* are expressions that allow the specification of the topic and payload of cp_{PB}^{CEP}, respectively. More precisely, *tp* is an initialization expression built referring to any attribute in the payload of events of type *cet* (see Section 4), whereas *pl* is a set of pairs ⟨*id, exp*⟩, each specifying an attribute of the payload of cp_{PB}^{CEP}. Component *id* specifies the name of an attribute, whereas *exp* is the related attribute initialization expression.

Example 7. Let us now consider the specification of action *SevereBreathlessnessNotifier*, which, upon detecting an event of type *SevereBreathlessness* denoting a serious form of shortness of breath, publishes an MQTT message which notifies the detected criticality. *SevereBreathlessness* can be straightforwardly specified by restricting the definition of *Breathlessness* in Example 6. *SevereBreathlessnessNotifier* is specified as ⟨*SevereBreathlessnessNotifier, SevereBreathlessness, "criticality/severebreathlessness", {"patientId": SevereBreathlessness$_{ce}$.pID, "time": SevereBreathlessness$_{ce}$.time, "bpm": SevereBreathlessness$_{ce}$.bpm}*⟩. The execution of this action causes the publishing of a message on topic *criticality/severebreathlessness*, with a payload characterized by fields that map those of the detected event.

An *emergency evolution* is a transition between a pair of emergency situations, which occurs when, due to the continuous analysis of the messages exchanged in the MQTT environment operated by the CEP system, an event of complex type is detected. More formally:

Definition 3 (Emergency evolution)**.** An emergency evolution *ev* is a tuple ⟨*cet, src, trg, act*⟩, where *cet* specifies the identifier of a complex event type in *CET* (see Section 4), *src* and *trg* respectively refer to the identifiers of the emergency situations that are left and entered when an event of type *cet* is detected,[8] whereas *act* refers to the identifier of an action executed when *pr* occurs (or ⊥ if no action has to be executed).

The occurrence of events in the monitored MQTT environment can cause: i) the starting of an emergency situation, ii) the evolution of an emergency situation into a more severe or mild one, or even iii) the resolution of an emergency situation. In order to model the possible evolution of an emergency case we hereby introduce the concept of *emergency development plan*.

Definition 4 (Emergency development plan)**.** An emergency development plan *edp* is a tuple ⟨*edpi, Ev*⟩ where *edpi* is the identifier of the emergency development plan, whereas *Ev* is a set of emergency evolutions depicting the possible developments of an emergency case.

The definition of an emergency development plan *edp* has to satisfy some well-formedness rules. An evolution *ev* in the set *Ev* of *edp*, referred to as *edp.Ev*, can only refer as end points ⊥ or an emergency situation. Moreover, any pair of evolutions that refer to the same emergency situation within component *src*, have to specify events of different type within component *cet*. The same constraint applies to a pair of evolutions which refer to ⊥ as source.

In order to intuitively represent any possible evolution of the emergency situations referred to by an emergency development plan, we represent them as state machine (stm) diagrams, where the *emergency situations* are depicted as *states* and the *evolutions* as *transitions*. Each state is labeled with the identifier of the modeled emergency situation, whereas each transition is labeled with a complex event of a type referred to by the related evolution.

Example 8. Let us consider the emergency development plan *PulmonaryIssues*, which is characterized by the emergency situations *Dyspnea, LowOxygenSaturation*, and *DyspneaOxygen*, where *Dyspnea* denotes a breathing discomfort, *LowOxygenSaturation* a low level of oxygen-saturated hemoglobin in the blood, whereas *DyspneaOxygen* a combination of the previous cases. Two evolutions map the activation of the modeled emergency case, respectively entering the emergency situations *Dyspnea* and *LowOxygenSaturation*, and other two its deactivation, which exits the same emergency situations. Additional evolutions allow the transition from the emergency situation *Dyspnea* to *DyspneaOxygen*, and back, as well as from *LowOxygenSaturation* to *DyspneaOxygen*, and back. Let us assume that the above-mentioned evolutions refer to events of type *Breathlessness, BelowThresholdO2, NormalBreathRate*, and *NormalO2Level*, and, also, for the sake of simplicity, that no evolution refers to actions. This scenario is depicted by the state machine shown in Fig. 2.

In contrast, the concept of *emergency scenario* is used to denote an emergency case that involves a specific set of subjects, and whose evolution is depicted by an emergency development plan.

Definition 5 (Emergency scenario)**.** An emergency scenario *es* is a tuple ⟨*esi, edp, sf*⟩, where *esi* is the emergency scenario identifier, *edp* refers the identifier of the associated emergency development plan, whereas *sf* is a logic predicate, referred to as subject filter, which specifies under which conditions a subject is involved in *es*. Like parametric predicates (see Section 3.3), subject filters are defined by composition of subject attributes using mathematical and logical operators.

At any point of the execution, an emergency scenario *es* is either inactive or in one of the emergency situations referred to by the evolutions of the emergency development plan *edp*. More precisely, at specification time an emergency scenario *es* is *inactive* and maintains this state until an event: (a) of type *cet* referred to by an evolution *ev* in *edp.Ev*, and (b) which refers to ⊥ as *src* component, and to an emergency situation *ems* as *trg* component, occurs. The event triggers the activation of the emergency scenario, and the entrance in the emergency situation *ems*, which is then referred to as the new current stage of *es*. Afterward, when an event occurs, which is referred to by an evolution *ev'* among the possible evolutions of *ems* (i.e., any evolution that refers to *ems* within component *src*), the current stage of the emergency scenario is updated. More precisely, if component *trg* of *ev'* refers to ⊥, the emergency scenario is disabled, whereas if it refers to another emergency situation, such as, for instance, *ems'*, this new emergency is entered, and the current stage of *es* is updated to *ems'*.

It is worth noting that our model allows the specification of multiple emergency scenarios per single application, therefore a subject could be referred to by different emergency scenarios, as well as by no scenario. In addition, multiple emergency scenarios defined for an application could specify the same development plan, but different subjects.

[7] Actions turn MQTT clients into event sinks (see Section 3.2), which possibly could be programmed to react to the detected events.

[8] *src* / *trg* could also refer to value ⊥ to denote that the occurrence of an event of type *cet* activates / terminates an emergency scenario. Further details are provided in the remainder of this section, where we present the concept of emergency scenario.

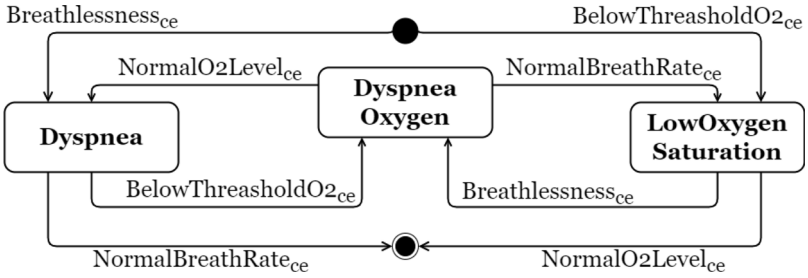

Fig. 2. The stm diagram corresponding to the emergency development plan *PulmonaryIssues*.

Example 9. Let us consider the definition of the emergency scenarios es_1 / es_2, respectively specifying the possible involvement of patient Bob / Mary and of the medical staff operating in the nursing home, in emergency cases modeled by the emergency development plan *PulmonaryIssues* (see Example 8). According to Definition 5, es_1 can be specified as $\langle es_1$, *PulmunaryIssues, (s.gid=patient* $\wedge$ *s.uid=Bob)* $\vee$ *s.gid=medical_personnel*$\rangle$, and similarly, es_2 as $\langle es_2$, *PulmunaryIssues, (s.gid=patient* $\wedge$ *s.uid=Mary)* $\vee$ *s.gid=medical_personnel*$\rangle$. Although these emergency scenarios refer to the same development plan, they represent emergency cases related to two distinct patients, therefore, at any point in time, the current stage of es_1 could be different from the one of es_2.

Let us now consider the case of a subject *s* who issues an access request *ar*. If at *ar* request time no emergency scenario refers to *s*, or all emergency scenarios which refer to *s* are inactive, *s* is said to be in an *ordinary* situation, and therefore *ar* is regulated by the access control policies introduced in Section 3.3, which from now on are referred to as *ordinary policies*.

In contrast, if at *ar* request time at least one of the emergency scenarios that refer to *s* is active the request *ar* is controlled by *emergency policies*. Emergency policies regulate the ability of a subject involved in one or more emergency scenarios to send or receive MQTT messages.

Emergency policies are formally defined as follows.

Definition 6 (Emergency policy)**.** An emergency policy *ep* is a tuple$\langle s, tf, exp, pr, esf, stf\rangle$, where *s, tf, exp, pr* correspond to the homonym components in Definition 1, *esf* is an emergency scenario filter, namely an expression built referring to emergency scenario properties, resulting in a set of emergency scenarios that specify the same emergency development plan, whereas *stf* is a situation filter expression, which, by referring to emergency situation properties, specifies the emergency situations to which *ep* is applied.

An emergency policy *ep* upon satisfaction of the parametric predicate *exp* grants to the subjects referred to by *s* the privilege to send or receive messages on topics included in *tf*, in any emergency situation denoted by *stf* of the emergency scenarios specified by *esf*.

Example 10. Let us now consider the specification of an emergency policy *ep* which grants external specialists access to physiological data of patients in severe conditions, with the aim to consent to timely treatments. Let us assume that *ep* grants access to data of patients involved in emergency scenarios that specify *PulmonaryIssues* as emergency development plan, and who are currently under the emergency situation *DyspneaOxygen*. Policy *ep* can be specified as: $\langle$*specialist, +/physiological/#, true, read, edp="PulmonaryIssues", {DyspneaOxygen}*$\rangle$. Based on Example 9, Mary and Bob are involved in the emergency scenarios es_1 and es_2, which specify *PulmonaryIssues* as emergency development plan. Therefore, according to *ep* a specialist can only access Bob's/Mary's data when es_1/es_2 specify *DyspneaOxygen* as current stage.

Finally, let us shortly consider the process that allows a security administrator to specify emergency policies for a target MQTT environment.

Based on Definition 6, emergency policy specifications can only be achieved after having defined at least one emergency scenario. In turn, at least one emergency development plan should be specified to define an emergency scenario. An emergency development plan can be defined following a step-wise process that starts with the identification of: i) a set of emergency situations depicting possible stages of an emergency case, along with their associated severity levels. Afterward, the security administrator needs to establish, for any considered emergency situation, if it can be entered as first stage of the emergency case, or if it can only be reached as a possible evolution of another emergency situation. Similarly, he/she needs to decide if any of the considered situations can evolve into the resolution of the emergency case. Any evolution is enabled by the occurrence of events. In order to properly label all considered evolutions, it is first required to identify the involved events and model the related event types. Finally, the modeling of the evolutions is completed with the possible specification of actions. Action specification relies on the previously mentioned modeling of complex events types, as well as on simple transformation rules that allow converting complex events into MQTT publishing requests.

Once the definition of emergency development plans has been completed, emergency scenarios are straightforwardly specified indicating the set of subjects potentially involved in any considered emergency case. Afterward, the security administrator can finally focus on emergency policy specifications. Emergency policies are specified as ordinary access control policies, but they make explicit reference to the emergency situations where they apply.

6. System overview

Our proposed framework includes multiple enforcement monitors, that are used to keep a reasonably low enforcement overhead in scenarios where several clients are involved. They regulate the exchange of messages by MQTT clients of a monitored environment, on the basis of the specified ordinary and emergency access control policies. A NoSQL datastore is employed to keep track of metadata related to emergency management, and access control. More precisely, it stores: a) emergency scenarios along with related current stages; b) primitive and complex event types; c) ordinary and emergency policies; and d) subject, object, and environment attributes employed for policy specification. A module, denoted CEP interface, is used to manage the evolution of emergency scenarios, on the basis of interactions with the monitors and a CEP engine. The possible detection of events by the CEP engine is managed by handlers embedded in the CEP interface. Any time an emergency scenario *es* is specified, two handlers are instantiated.

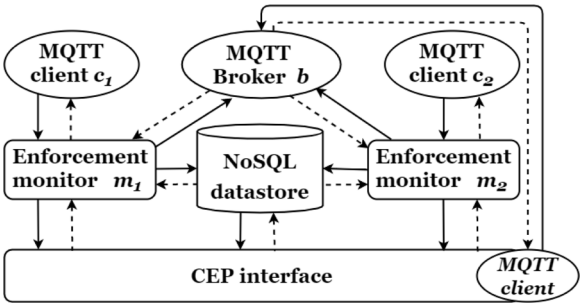

Fig. 3. A high-level view of the system architecture.

The former is employed to manage the possible update of the referred current stage of *es*, whereas, the latter to catch CEP engine notifications denoting that no update is required. Finally, the CEP interface also embeds an MQTT publisher client responsible for issuing selected event notifications (formatted as MQTT messages) to rightful subscriber clients.

A high-level view of the system architecture is shown in Fig. 3.

In order to present the role of each component of the proposed framework, as well as the overall control flow, let us consider a simple scenario where users u_1 and u_2 respectively administer MQTT clients c_1 and c_2, which operate in an MQTT environment that hosts a broker *b*. A high-level representation of the system's control flow for the considered scenario is provided in Fig. 4, where a UML Sequence diagram is used to depict the main interactions that hold among the system's stakeholders and components, along with the executed tasks.

Let us assume that c_1 has been configured to publish messages on topic *t*, whereas c_2 to subscribe the reception of messages referring to *t*. c_1 and c_2 have been set up to connect with broker *b* by means of the enforcement monitors m_1 and m_2, respectively.

In order to exchange messages, c_1 and c_2 need to connect with the MQTT broker *b*. Let us shortly consider the connection process of c_1 mediated by m_1 (the same process allows the connection of c_2 mediated by m_2). The process starts with a connection request issued by c_1 on behalf of u_1, denoted as ${cp_{CN}}^{c}{}_1$. On receipt of ${cp_{CN}}^{c}{}_1$, m_1: 1) opens a communication channel with *b*, and another one with the CEP interface, to be used to convey any communication related to c_1 requests, 2) analyzes subject attributes in the intercepted packet header deriving the identity of the requester subject, 3) forwards ${cp_{CN}}^{c}{}_1$ to *b*. The broker authenticates c_1 and sends back an acknowledgment packet ${cp_{CA}}^{c}{}_1$ to m_1, which in turn forwards it to c_1.

Let us now assume that, once connected, c_1 sends a publishing request ${cp_{PB}}^{c}{}_1$ on behalf of u_1. On receipt of ${cp_{PB}}^{c}{}_1$, m_1 recognizes that ${cp_{PB}}^{c}{}_1$ has been issued by a client, and redirects the request to the CEP interface. More precisely, it prepares a composite packet (i.e., ${cp_{PB}}^{c*}_1$), which includes the intercepted request ${cp_{PB}}^{c}{}_1$, and the objects *s*, *o*, and *e*, with fields corresponding to the subject, object, and environment attributes associated with the request. It then issues the packet to the CEP interface and waits for a response.

The CEP interface instantiates a control task responsible for the analysis of ${cp_{PB}}^{c*}_1$. This task first extracts from ${cp_{PB}}^{c*}_1$ the embedded objects, and the original request ${cp_{PB}}^{c}{}_1$, and identifies the requesting subject u_1 from the subject attributes. Afterward, it checks if ${cp_{PB}}^{c}{}_1$ matches the binding criteria of any specified primitive event type in *PET* (cfr. Section 4). If no criterion is matched, the packet cannot trigger any emergency evolution, thus the control task notifies the monitor of the completion of the analysis. In contrast, if ${cp_{PB}}^{c}{}_1$ is referred to by at least one primitive event type, the task handles the generation of primitive event notifications bound to ${cp_{PB}}^{c}{}_1$. For any primitive event type *pet* in *PET* (see Section 4) which specifies binding criteria satisfied by ${cp_{PB}}^{c}{}_1$, the control task derives an event notification en^{pet}. The generation employs internal properties of ${cp_{PB}}^{c}{}_1$ and attributes extracted from ${cp_{PB}}^{c*}_1$ referred to in the initialization expression *pet.adf* (see Section 4 for more details). Let us denote with EN_{pet} the set of event notifications generated from ${cp_{PB}}^{c*}_1$. The control task delivers EN_{pet} to the CEP engine, and waits for the completion of the analysis of this set of primitive event notifications. As soon as the control task is notified of the analysis completion by the pair of handlers associated with any emergency scenario the control task notifies the monitor that the analysis has been completed. The monitor can thus continue the processing of ${cp_{PB}}^{c}{}_1$ (see Section 7.2 for more details). Upon receiving the acknowledgment, the monitor selects from the NoSQL datastore the emergency scenarios that refer to u_1 as an involved subject. On the basis of the referred emergency situation of u_1 in any of the active scenarios, monitor m_1 selects from the datastore all emergency policies that apply to u_1's request ${cp_{PB}}^{c}{}_1$. In contrast, if there does not exist an active scenario among the selected ones, m_1 selects from the datastore all ordinary policies applicable to ${cp_{PB}}^{c}{}_1$. In both cases, m_1 then employs the enforcement mechanism proposed in Colombo and Ferrari (2018) (see Section 3.3), authorizing the publishing if at least one of the selected policies grants the access.

Let us now assume that the publishing of ${cp_{PB}}^{c}{}_1$ has been authorized by m_1. The message is therefore issued by m_1 to the broker *b*, which, on the basis of the received subscriptions, forwards a copy of this packet, referred to as ${cp_{PB}}^{b}$, to c_2. The packet is then intercepted by m_2, which monitors the communication channel between c_2 and *b*. Since the sender of ${cp_{PB}}^{b}$ is *b*, m_2 derives and enforces the applicable policies without the intervention of the CEP interface. Once the identity of the receiver subject u_2 has been derived, m_2 selects from the datastore the emergency scenarios that refer to u_2 as an involved subject. In any of the selected scenarios which are not referred to as inactive, the monitor derives the current emergency situation of u_2, and then selects from the datastore all emergency policies that regulate the receiving of messages on topic *t* (i.e., the topic of ${cp_{PB}}^{b}$) by u_2 in any of the selected emergency situations. In contrast, if no active scenario is detected, the selection targets all ordinary policies applicable to ${cp_{PB}}^{b}$. In both cases, m_2 then proceeds to apply the enforcement approach proposed in Colombo and Ferrari (2018).

7. Enforcement

Let us now focus in more details on selected aspects of the proposed enforcement mechanism, instrumental to the selection of the emergency policies applicable to an access request. Selected policies are then enforced employing the mechanisms proposed in Colombo and Ferrari (2018).

7.1. Event detection

A key functionality of our framework is its ability to handle the evolution of emergency scenarios, on the basis of detected complex events that trigger the entrance into specific emergency situations. Since an event of complex type can only occur when specific primitive events are observed, each corresponding to an MQTT client's publishing request, we now analyze core activities of the CEP interface instrumental for event detection. These are executed any time an MQTT client's publishing request intercepted by an enforcement monitor is forwarded to the CEP interface. More precisely, let us start to consider the *control task* instantiated by the CEP interface on receipt of a packet $\text{cp}_{\text{PB}}{}^{*}$ issued by an enforcement monitor. We remind that $\text{cp}_{\text{PB}}{}^{*}$ includes a control packet cp_{PB} and three objects, denoted *s*, *o* and *e*, with fields representing the subject, object, and

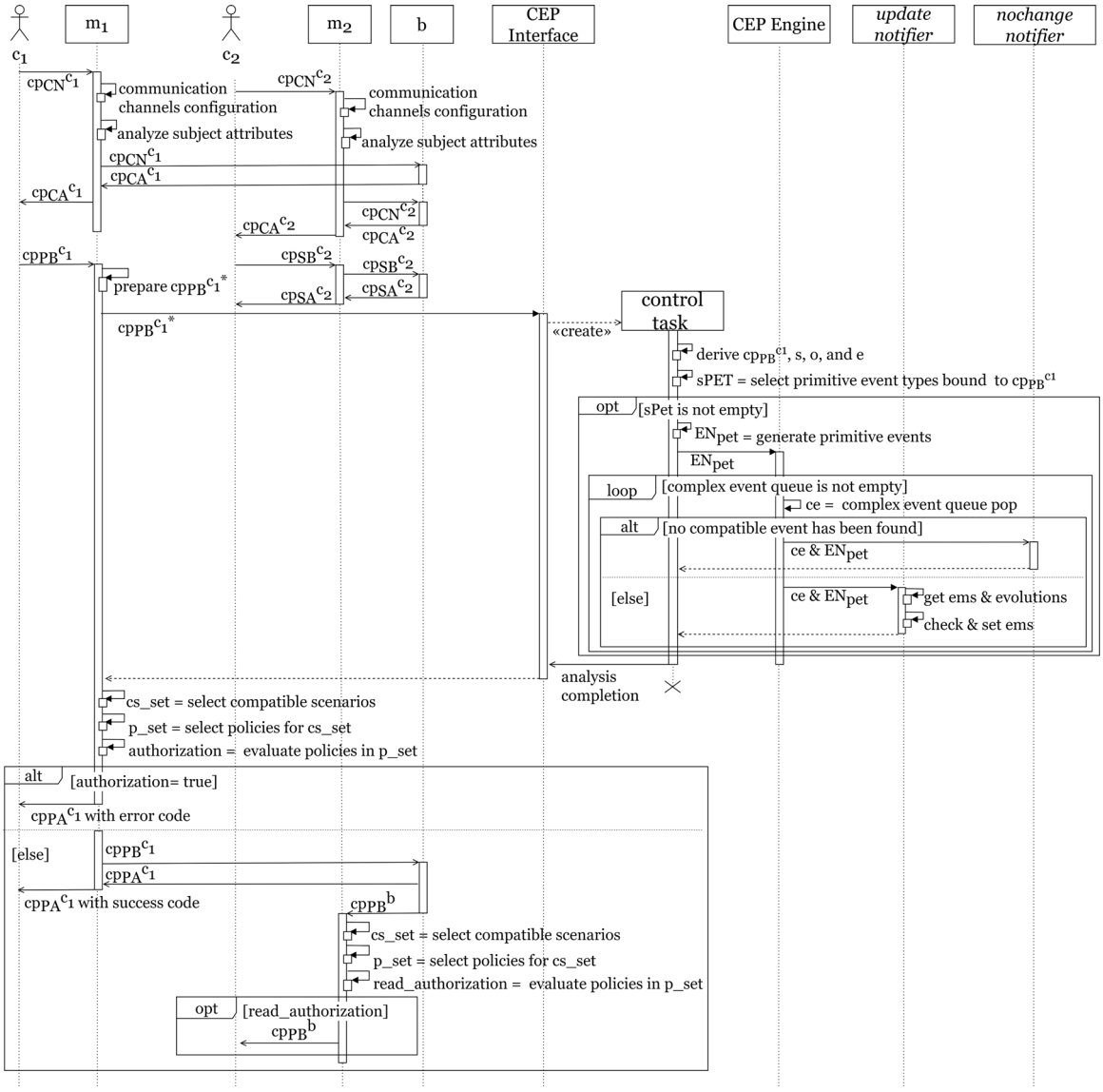

Fig. 4. System control flow for the exemplified scenario.

environment attributes associated with the publishing request context (see Section 6).

The *control task* starts managing the generation of primitive event type notifications. This is achieved for all primitive event types belonging to PET which are bound to cp_{PB}. A primitive event type *pet* is selected iff the evaluation of the binding expression *bcr* of *pet* is satisfied by cp_{PB}.

Any selected primitive event type *spet* is then used to generate event notifications, specifying objects characterized by: 1) a *time annotation*, used for event ordering purposes, 2) a *payload*, which represents the event content, and 3) a *type*, which refers to the related event type name, implying that the structure of the event payload has to match the one specified within component *adc* of *spet* (see Section 3.2). The *time annotation* is straightforwardly derived as the timestamp related to the reception of $cp_{PB}{}^*$. The *type* corresponds to the value referred to by the component *pet* of *spet*. Finally, the *payload* is specified by referring to the content of component *adf* of *spet*. More precisely, the control task initializes any attribute *id* referred to within component *adf* of *spet* (cfr. Section 4) to the value of the corresponding expression *exp*.

Example 11. Let us consider the control task *ct* created at time *rt*, upon receipt of $cp_{PB}{}^*$. Suppose that $cp_{PB}{}^*$ embeds: i) a publishing request cp_{PB} on topic *"physiological/respiratory"*, with a payload that includes field *respiratory* initialized to 27, and ii) subject, object and environment attributes indicating that cp_{PB} has been issued at time *st* by a device that monitors patient Bob's conditions. Finally, let us also assume that PET includes the primitive event type *RespiratoryRate* introduced in Example 6. Since the binding expression of *RespiratoryRate* is satisfied by cp_{PB}, this primitive event type is selected for generating event notifications. The expressions in component *adf* of *RespiratoryRate* are thus evaluated for deriving the notification. As a consequence, the event notification *RespiratoryRate@rt: {"bpm":27, "time":st, "patientID":"Bob"}* is generated.

Once the generation process has been completed, the control task issues the primitive event notifications to the CEP engine, and waits to be notified for the completion of the analysis of the delivered set of notifications. As soon as the pairs of handlers responsible for handling the evolution of any emergency scenario notify the completion, the control task responds to $cp_{PB}{}^*$ with an acknowledge message, terminating the execution.

7.2. Emergency management

The CEP interface manages the evolution of any specified emergency scenario *es* by means of two event handlers, denoted as *update-notifier* and *nochange-notifier* (see Section 6). *update-notifier* manages the detection of complex events and the possible update of *es*'s current stage, whereas, *nochange-notifier* keeps track of CEP engine analysis cycles during which no complex event is detected. These handlers are instantiated by the CEP interface at *es* specification time and then kept active as long as *es* belongs to the set of managed emergency scenarios.

Each time the CEP engine completes the analysis of a delivered set of primitive event notifications one of these handlers is invoked.

update-notifier is invoked when, on receipt of a set of primitive event notifications, a complex event *ce* of type *ecet* is detected by the CEP engine, which is referred to by an evolution *ev* of *es*. The handler is notified of the detected event, as well as of the set of primitive event notifications that have caused the event occurrence. The handler starts to select the current stage *ems* of the emergency scenario *es* from the datastore, as well as the evolution set of *es*. If *es* is not active, *ems* is initialized to $\perp$, whereas if *es* is active, it is set to the current emergency situation of *es*. If there exists an evolution *ev* whose components *src* and *cet* respectively refer to *ems* and *ecet*, *update-notifier* specifies the emergency situation referred to within component *trg* of *ev* as the new current emergency situation of *es*, propagating the update to the datastore. Let us now denote with *ct* the control task that has delivered the set of primitive event notifications which triggered the detection of *ce*. Once the update has been performed, if *ev* specifies an action *ac*, *update-notifier* gets from *ct* a copy of the composite packet $cp_{PB}{}^{*}$ used for generating the notifications that caused the occurrence of *ce*. It then instantiates an *execution manager* task, which asynchronously manages the execution of *ac*, following the process detailed in Section 7.3, to which the derived copy is passed. Finally, *update-notifier* issues the analysis completion notification to *ct*.

In contrast, the handler *nochange-notifier* is invoked if, upon receiving a set of primitive event notifications, the CEP engine does not detect events of complex types referred to by an evolution of *es*. The handler is notified of the analyzed set of primitive event notifications, and, in turn, it notifies the analysis completion to the control task which delivered these notifications.

Example 12. Let us assume that on receipt of the event notification *Respiratory@rt* presented in Example 11, the CEP engine detects an event of type *Breathlessness*, which is referred to by the evolutions of the emergency scenario es_1 (see Example 9), as this specifies *PulmonaryIssues* as emergency development plan (see Example 8). As a consequence, the handler *update-notifier* configured for es_1 is notified of the detected event, as well as of the primitive event notification *Respiratory@rt* that caused the detection. In contrast, the handler *nochange-notifier* is not invoked. Let us now suppose that es_1 is not active. The handler initializes *ems* to $\perp$, and then checks if an evolution exists, which refers to *Breathlessness* within component *cet*, and to $\perp$ within component *src*. As shown in Fig. 2, such evolution exists, and specifies *Dyspnea* as a target emergency situation. As a consequence, *update-notifier* activates es_1 specifying the emergency situation *Dyspnea* as the new current stage of the emergency scenario. Afterward, since the considered evolution does not refer to any action, *update-notifier* notifies *ct* of the completion of the analysis.

7.3. Action execution

Let us now focus on the execution of actions associated with emergency evolutions. We hereby present the process implemented by the *execution manager* task, which is responsible to handle action executions. *Execution manager* is invoked any time the current stage of an emergency scenario *es* is updated, and component *act* (see Section 5) of the emergency evolution *ev* that caused the update refers to an action.

Let us assume that the *execution manager* has been invoked to handle the execution of a generic action *ac*. At start time, the *execution manager* derives three event notifications from the copy of $cp_{PB}{}^{*}$ received as input (see Section 7.2). The derived notifications refer to a predefined event type, respectively characterized by fields corresponding to the subjects, objects and environment attributes embedded in $cp_{PB}{}^{*}$. The generated events' notifications are delivered to the CEP engine, whereupon the *execution manager* stays on hold waiting for a CEP engine notification. If the CEP engine notifies that no event has been detected, the *execution manager* immediately terminates. In contrast, on receipt of an event *ecet*, *execution manager* generates a MQTT publishing request $cp_{PB}{}^{CEP}$ on the basis of *ecet* content. The topic of $cp_{PB}{}^{CEP}$ is initialized to the result of the evaluation of the expression referred to by component *tp* of *ac* (see Section 5). [9] Similarly, the payload of $cp_{PB}{}^{CEP}$ is derived iterating over the initialization expressions referred to by component *pl* of *ac*, each targeting a different payload's attribute.

Finally, $cp_{PB}{}^{CEP}$ is delivered to the broker of the monitored MQTT environment by a MQTT publisher embedded in the CEP interface (see Section 6) and connected to the MQTT broker at system initialization time.

Example 13. Let us consider again the case introduced in Example 12, now assuming that the evolution which in Example 12 has caused the activation of es_1 refers to action *SevereBreathlessnessNotifier* (see Example 7). After the current stage of es_1 is updated, *update-notifier* instantiates an *execution manager* task *emt* providing as input a copy of $cp_{PB}{}^{*}$ derived from *ct*, and the action *SevereBreathlessnessNotifier* referred to by the evolution. *emt* first derives a primitive event notification from a built-in primitive event type that does not specify binding expressions, but simply maps as payload fields the subject, object and environment attributes in $cp_{PB}{}^{*}$. Then, *emt* delivers the derived notifications to the CEP engine. On receipt of these event notifications, the CEP engine notifies the detection of an event $\text{SevereBreathlessness}_{ce}$. As a consequence, *emt* generates an MQTT publishing request $cp_{PB}{}^{CEP}$ on topic *critical/severebreathlessness*, with a payload that maps the one of the just detected events. Finally, the generated packet is issued to the broker by the MQTT client administered by the CEP interface.

8. Experimental evaluation

In this section, we first present the application of our framework to the nursing home scenario introduced in Section 2, then we evaluate the framework performance with a set of experiments based on the same application scenario.

Our experiments rely on a prototype of the framework introduced in Section 6. Our prototype includes an enforcement monitor, defined as an extended version of the monitor presented in Colombo and Ferrari (2018), which here has been redesigned to enforce emergency policies. Metadata related to access control and emergencies are managed by an instance of Redis,[10] a popular in-memory key-value datastore. Event detection is carried out by the CEP engine Esper[11], using the Event Processing Language (EPL)

[9] We remind that *tp* is an initialization expression built referring to any attribute in the payload of events of type *cet*.

[10] https://redis.io.

[11] https://www.espertech.com/esper.

Table 3
Primitive event types specified for the case study.

pet	adc	bcr	adf	description
Result	{“pid”: string, “result”: boolean, “tDate”: date, “reqId”: long, “time”: long}	t.TopicName. includes(“result”)	{“pid”:o.patientID, “result”:t.Payload.result, “tDate”:t.Payload.testDate, “reqId”:t.Payload.reqId, “time”:e.time}	Shows the results of a COVID-19 test to which a patient has undergone.
Prescription	{“pid”: string, “tDate”: date, “reqId”: long, “time”: long}	t.TopicName. includes(“prescription”)	{“pid”:o.patientID, “tDate”:t.Payload.testDate, “reqId”:t.Payload.reqId, “time”:e.time}	Shows the prescription of a COVID-19 test for a patient.
Location	{“pid”: string, “pos”: string, “time”: long}	t.TopicName. includes(“location”)	{“pid”:o.patientID, “pos”:t.Payload.location, “time”:e.time}	Shows the room where a patient is located at specified time
Temperature	{“pid”: string, “temp”: float, “time”: long}	t.TopicName. includes(“temperature”)	{“pid”:o.patientID, “temp”:t.Payload.temperature, “time”:e.time }	Shows the body temperature of a patient at a specified time.
RespiratoryRate	{“pid”: string, “bpm”: float, “time”: long}	t.TopicName. includes(“respiratory”)	{“pid”:o.patientID, “bpm”:t.Payload.respiratory, “time”:e.time}	Shows the respiratory rate of a patient at a specified time.
EstimatedSpO2	{“pid”: string, “SpO2”: float, “time”: long}	t.TopicName. includes(“saturation”)	{“pid”:o.patientID, “SpO2”:t.Payload.saturation, “time”:e.time}	Shows the peripheral oxygen saturation of a patient at a specified time.
ReqAttSet	{“cid”: string, “uid”: string, “gid”: string, “pSet”: Set(string), “relativeOf”: Set(string), “pid”: string, “ts”: long }	$\perp$	{“cid”: s.cid, “uid”: s.uid, “gid”: s.gid, “pSet”: s.pSet, “relativeOf”: s.relativeOf, “pid”: o.patientId, “ts”: e.time}	Maps the set of subject, object and environments attributes which characterize access requests in the considered application scenario.

for implementing queries able to detect events of complex types. The CEP interface has been developed in Java, and allows managing the evolution of emergency scenarios on the basis of MQTT control packets forwarded by the enforcement monitor and events detected by the CEP engine.

8.1. The case study

The considered MQTT-based IoT application scenario allows detecting early symptoms of COVID-19 in nursing home patients, and tracking their close contacts. Due to the high COVID-19 mortality in extended care units (European Centre for Disease Prevention and Control, 2020), in such environments, COVID-19 diffusion is contrasted through the preventive isolation of any identified possibly infected patient. The quarantine protocol, which is normally applied to confirmed COVID-19 cases, is here extended to any patient with early symptoms of COVID-19 who has not yet undergone a test or is still waiting for a result, and to any patient among his/her recent close contacts.

We assume that sensors worn by patients monitor physiological data, such as patients' temperature, respiratory rate, and peripheral oxygen saturation, whereas patients' movements are tracked through the interaction of patients' bracelets with proximity sensors deployed in any room of the nursing home. Additional exchanged data include the prescriptions of COVID-19 tests for nursing home patients, the related results, treatment options communicated to patients, and patients' consent to proceed. We assume that all devices and software modules that generate data are provided with an MQTT interface, and data are exchanged by means of the MQTT protocol. Table 3 exemplifies a selection of primitive event types specified for the above-mentioned data, each denoting a class of primitive events derived from MQTT messages on given topics exchanged in the nursing home environment. Column *pet* specifies the identifier of the considered event type, *adc* declares all fields that compose the payload of the represented event class, *bcr* defines the conditions to be met by an MQTT message for deriving an event of the represented class, and finally, *adf* specifies the expressions that allow the initialization of payload fields.

Different groups of subjects are involved in the considered application scenario. Physicians and nurses in the medical staff of the nursing home access patients' data and issue communications by means of a mobile app. Similarly, external specialists use an app to remotely check patients' conditions and to communicate possible treatments. Patients can use an app to check their own health status. The app can also be used by registered relatives of patients subject to COVID-19 quarantine, to be updated on their kin conditions.

We model the possible evolution of a COVID-19 case as an emergency development plan characterized by the following emergency situations:

- *Suspected COVID-19*, is an emergency situation with moderate severity (level 2), related to a patient who has had COVID-19 symptoms in the last days;
- *Possible COVID-19* is an emergency situation with mild severity (level 1) related to a patient who has been referred to as close contact of a suspected or confirmed COVID-19 patient;
- *COVID-19 asymptomatic* is an emergency situation with considerable severity (level 3), related to a confirmed COVID-19 patient with no symptom;
- *COVID-19 symptomatic* is an emergency situation with high severity (level 4), related to a confirmed symptomatic COVID-19 patient.
- *Severe COVID-19* is an emergency situation with critical severity (level 5), related to a symptomatic COVID-19 patient with severe symptoms.

The possible evolution of a COVID-19 case is represented by the state machine in Fig. 5, where emergency situations are represented as states, and evolutions as transitions.

Multiple emergency scenarios have then been defined (one scenario per patient), which refer *COVID-19 case* as emergency development plan, and a set of involved subjects that includes: a patient p, the medical staff of the nursing home that takes care of p, the external specialists who could be consulted, and the close relatives authorized by p to receive information about his/her health conditions.

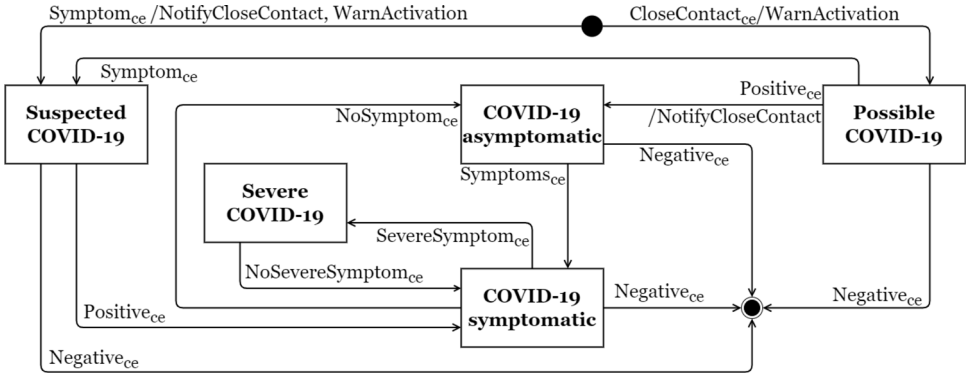

Fig. 5. State machine representing the possible evolution of a COVID-19 case.

A COVID-19 case scenario related to patient *p* can be activated if *p* shows a COVID-19 symptom or *p* is referred to as close contact of a suspected or confirmed COVID-19 patient. The former condition causes the entry into emergency situation *Suspected COVID-19*, whereas the latter into *Possible COVID-19*. Both emergency situations imply the need to isolate *p* and to let *p* take a COVID-19 test. If the test is negative, either emergency situations are resolved deactivating the emergency scenario, whereas, in case of a positive result with /without recent symptoms the emergency situation *COVID-19 symptomatic* /*COVID-19 asymptomatic* is entered. The passage from *COVID-19 symptomatic* to *COVID-19 asymptomatic* is only possible if, for some consecutive days, *p* does not show COVID-19 symptoms, whereas the opposite transition occurs as soon as a symptom is detected. If within *COVID-19 symptomatic* emergency situation *p* shows clear signs of aggravation, the emergency evolves into a *Severe COVID-19* case. The backward transition is only possible if, for some consecutive days, no severe symptom is observed. A COVID-19 case related to *p* is resolved in case of negative result to a COVID-19 test.

The evolution of a COVID-19 case is triggered by the occurrence of complex events. Table 4 exemplifies a selection of complex event types for the considered scenario, leveraging on primitive events reported in Table 3. Column *cet* specifies the name of the considered complex event type, *adc* specifies the set of fields composing the payload of the represented events, *ets* indicates the set of event types referred to in the specification, whereas *exp* models event specifications using the abstract event algebra introduced in Section 3.2. Event types reported in Table 4 consider classes of events denoting: the presence / absence of COVID-19 symptoms with various severities, the activation of a COVID-19 case, the result of the last COVID-19 test to which a patient has undergone, the rooms recently visited by a patient, and all contacts/close contacts of a patient in the last days.

The analysis of physiological and proximity data allows deriving all patients who recently have had symptoms of COVID-19, as well as those who have had close contact with a suspected or confirmed COVID-19 patient.

In order to promptly contrast COVID-19 diffusion, all suspected cases and their close contacts have to be immediately reported to the medical staff so that physicians could promptly isolate these patients and prescribe a test, and they can be promptly informed of their condition. This is obtained through the modeling of the actions *WarnActivation* and *NotifyCloseContact*, shown in Table 5. More precisely, at the early detected symptoms of COVID-19, *WarnActivation* publishes an MQTT message to inform the suspected COVID-19 patient and his/her attending physicians to be involved in an active emergency scenario. The action converts events of type *Activation* into MQTT messages, which, within the payload fields *pid* and *reqId*, simply denote the identifier of the suspected COVID-19 patient and the timestamp at which the case has been detected. In contrast, *NotifyCloseContact* publishes an MQTT message for any detected close contact of the patient who has just entered the emergency situations *Suspected COVID-19* and *COVID-19 asymptomatic*. The action converts events of type *CloseContact* into MQTT messages. The specification of *CloseContact* (cfr. Table 4) shows a possible way to derive the close contacts of a patient *p*. For the proposed calculation we assume that proximity sensors check the presence of patients in any room of the nursing home at a rate of one sampling per second, and these data are then published as MQTT messages. The presence of a patient in a room is notified by primitive events of type *Location*, which also report the room (specified by field *pos*), date, and time of the observation. Through the specification of complex event type *VisitedRoom*, we pick any primitive event of type *Location* observed in the last 10 days, which refers to the presence of *p*. In contrast, any complex event of type *Contact* notifies that a pair of patients, which includes *p*, have been in close contact for 1 s,[12] and reports all data related to the observation. A *Contact* event is derived from a primitive event of type *Location* which notifies that, at the time specified by an event of type *VisitedRoom*, another patient was in the same room. Finally, complex events of type *CloseContact* are derived by counting all events of type *PossibleContact* which refer to the same pair of patients and date. If the referred pair of patients have spent together more than 15 min (900 s) in a day, they are notified as close contacts.

Manifold privileges are granted to medical personnel operating in the nursing home, as well as to patients. A selection of the corresponding ordinary policies is reported in Table 6. For any policy *p*, column *s* refers to the subjects who can benefit from the privileges granted by *p, tf* shows the topic filter expression denoting the topics of the protected messages, *exp* shows the parametric predicate that specifies under which conditions *p* grants the access, whereas *pr* shows the read /write privilege granted by *p*.

According to these policies, patients can receive, through their mobile app, communications related to: i) test prescriptions and results, ii) warning messages informing them to be suspected of COVID-19 or to have been referred to as close contact of a COVID-19 case, iii) medical bulletins, and iv) treatment options. Patients can also use the app to give consent to undergo specific treatments.

Additionally, any physician, through his/her mobile app, is allowed to: i) monitor the physiological conditions of his/her patients, ii) prescribe a COVID-19 test for his/her patients and receive the results, iii) receive notifications issued by the monitoring application, reporting the activation of new COVID-19 cases, iv) illus-

[12] This corresponds to the length of the interval between two consecutive samplings by proximity sensors.

Table 4
Complex event types specified for the case study.

cet	adc	ets	exp
Symptom	{"pid": string}	{Temperature, RespiratoryRate, EstimatedSpO2}	$\pi_{\text{T.pid}}(\sigma(\text{maxT}>=38 \vee \text{maxBpm}>=25 \vee \text{maxSpO2}<0.95))($ $_{\text{T.pid}}\mathcal{G}_{\text{ max(bpm) as maxBpm, max(temp) as maxT, max(SpO2) as maxSpO2}}($ $\sigma_{\text{T.pid=R.pid} \wedge \text{R.pid=S.pid}}(\text{RespiratoryRate}_{\text{pe}}$ as R $\wedge$ $\text{Temperature}_{\text{pe}}$ as T $\wedge$ $\text{EstimatedSpO2}_{\text{pe}}$ as S) $))^{\text{now}}_{\text{now - 2 days}}$
Shows any patient who has had COVID-19 symptoms in the last 2 days.			
NoSymptom	{"pid": string}	{Temperature, RespiratoryRate, EstimatedSpO2}	$\pi_{\text{T.pid}}(\sigma(\text{maxT}<38 \wedge \text{maxBPM}<25 \wedge \text{maxSpO2}\geq 95)($ $_{\text{R.pid}}\mathcal{G}_{\text{ max(bpm) as maxBpm, max(temp) as maxT, max(SpO2) as maxSpO2}}($ $\sigma_{\text{T.pid=R.pid} \wedge \text{R.pid=S.pid}}(\text{RespiratoryRate}_{\text{pe}}$ as R $\wedge$ $\text{Temperature}_{\text{pe}}$ as T $\wedge$ $\text{EstimatedSpO2}_{\text{pe}}$ as S) $))^{\text{now}}_{\text{now - 2 days}}$
Shows any patient who did not show COVID-19 symptoms in the last 2 days.			
SevereSymptom	{"pid": string, "bpm": float}	{RespiratoryRate}	$\pi_{\text{pid, bpm}}(\sigma_{\text{bpm}>30}(\text{RespiratoryRate}_{\text{pe}}))^{\text{now}}_{\text{now - 2 days}}$
Shows any patient who has had severe breathlessness episodes in the last two days, along with the observed respiratory rate.			
NoSevereSymptom	{"pid": string}	{RespiratoryRate}	$\pi_{\text{pid}}(\sigma_{\text{maxB}<30}({}_{\text{pid}}\mathcal{G}_{\text{max(bpm) as maxB}}(\text{RespiratoryRate}_{\text{pe}})))^{\text{now}}_{\text{now-2 days}}$
Shows any patient who did not show severe breathlessness episodes in the last two days.			
Activation	{"pid": string, "reqId": long}	{ReqAttSet}	$\pi_{\text{pid, ts as reqId}}(\text{RequestAttribute}_{\text{ce}})$
Denotes the need to isolate a patient and let him/her to undergo a COVID-19 test			
UnderTest	{"pid": string}	{LastTest, Activation, Prescription}	$\pi_{\text{L.pid}}($ $\sigma_{\text{L.pid=P.pid} \wedge \text{L.reqId=P.reqId} \wedge \text{LP.time}>_{\text{P.time}}(}$ $\text{Prescription}_{\text{ce}}$ as P $\wedge\neg\text{LastTest}_{\text{ce}}$ as L $\wedge\neg\text{Prescription}_{\text{ce}}$ as LP) $\vee\sigma_{\text{L.pid=A.pid} \wedge \text{A.reqId=L.reqId} \wedge \text{LA.reqId}>_{\text{A.reqId}}(}$ $\text{Activation}_{\text{ce}}$ as A $\wedge\neg\text{LastTest}_{\text{ce}}$ as L $\wedge\neg\text{Activation}_{\text{ce}}$ as LA) $)$
Shows any patient who is waiting for the results of a test or who is going to undergo a COVID-19 test.			
LastTest	{"pid": string, testDate: date, reqId: long, result: boolean}	{Result}	$\pi_{\text{LR.pid, LR.testDate, R.reqId, R.result}}($ $\sigma_{(\text{LR.pid=R.pid})\wedge(\text{LR.tDate=R.tDate})}($ $({}_{\text{pid}}\mathcal{G}_{\text{ max(tDate) as testDate}}(\text{Result}_{\text{pe}}))$ as LR $\wedge$ $\text{Result}_{\text{pe}}$ as R $)$ $)$
Specifies the results of the last COVID-19 test of a patient.			
Positive	{"pid": string }	{LastTest, UnderTest}	$\pi_{\text{L.pid}}(\sigma_{\text{L.pid=U.pid} \wedge \text{ L.result}}(\neg\text{UnderTest}_{\text{ce}}$ as U $\wedge$ $\text{LastTest}_{\text{ce}}$ as L))
Shows any patient whose last COVID-19 test is positive, for whom no new test has been reserved.			
Negative	{"pid": string }	{LastTest, UnderTest}	$\pi_{\text{L.pid}}(\sigma_{\text{L.pid=U.pid} \wedge\neg \text{ L.result}}(\neg\text{UnderTest}_{\text{ce}}$ as U $\wedge$ $\text{LastTest}_{\text{ce}}$ as L))
Shows any patient whose last COVID-19 test is negative, for whom no new test has been reserved.			
VisitedRoom	{"pid": string, "pos": string, "time": datetime, "date": date, "ts": long}	{Location, ReqAttSet}	$\pi_{\text{L.pid, pos, time, getDate(time) as date, ts}}($ $\sigma_{\text{R.pid=L.pid}}(\text{Location}$ as L $\wedge$ $\text{ReqAttSet}_{\text{pe}}$ as R) $)^{\text{now}}_{\text{now - 10 days}}$
Shows any room visited by a patient in the last 10 days, along with the time at which he/she was in the room.			
Contact	{"pid": string, "rpid": string, "pos": string, "time": datetime, "date": date, "ts": long}	{Location, VisitedRoom, ReqAttSet}	$\pi_{\text{L.pid, V.pid AS rpid, V.pos, V.time, V.date, V.ts}}($ $\sigma_{(\text{V.pos=L.pos})\wedge(\text{V.time=L.time})\wedge(\text{V.pid}\neq\text{L.pid})\wedge(\text{V.pid=R.pid})\wedge_{(\text{V.ts=R.ts})}(}$ $\text{Location}_{\text{pe}}$ as L $\wedge$ $\text{VisitedRoom}_{\text{ce}}$ as V $\wedge$ $\text{ReqAttSet}_{\text{pe}}$ as R $)$ $)$
Shows any patient who has met patient rpid in the last 10 days, as well as the room and time at which the meeting occurred.			
CloseContact	{"pid": string, "rpid": string, "date": datetime, "duration": float, "ts": long}	{Contact, ReqAttSet}	$\pi_{\text{C.pid, C.rpid, C.date, num_of_1sec_intervals as duration, P.ts}}($ $\sigma_{\text{num\_of\_1sec\_intervals}>900}($ $_{\text{C.pid, C.rpid, C.date, C.ts}}\mathcal{G}_{\text{count(*) as num\_of\_1sec\_intervals}}($ $\sigma_{(\text{C.rpid=R.pid})\wedge(\text{C.ts=R.ts})}(\text{Contact}_{\text{ce}}$ as P $\wedge$ $\text{ReqAttSet}_{\text{pe}}$ as R) $)$ $)$ $)$
Shows any patient who, cumulatively, in a day, has stayed close to patient rpid for at least 15 min, along with the cumulative duration of these meetings, and the date when they occurred.			

Table 5
Actions involved in the COVID-19 case study.

aid	cet	tp	pl
NotifyCloseContact	CloseContact	closecontact	{"pid": CloseContact_{ce}.pid}
WarnActivation	Activation	warning	{"pid": Activation_{ce}.pid, "time": Activation_{ce}.reqId}

Table 6
Ordinary policies for the nursing home application.

s	tf	exp	pr	description
patient	prescription	o.patientId==s.uid	r	Allows patients to be informed of COVID-19 tests they must undergo.
patient	result	o.patientId==s.uid	r	Allows patients to get the results of COVID-19 test they underwent.
patient	warning	o.patientId==s.uid	r	Allows patients to be warned of having activated a COVID-19 case.
patient	closecontact	o.patientId==s.uid	r	Allows patients to be warned of being close contacts of suspected / confirmed COVID-19 cases.
patient	treatment	o.patientId==s.uid	r	Allows patients to be informed of treatment options.
patient	consent	o.patientId==s.uid	w	Allows a patient to consent to undergo a treatment.
medical_personnel	physiological/#	o.patientId $\in$ s.pSet	r	Allows physicians to access physiological data of their patients.
medical_personnel	prescription	o.patientId $\in$ s.pSet	w	Allows physicians to prescribe COVID-19 tests for their patients.
medical_personnel	result	o.patientId $\in$ s.pSet	r	Allows physicians to receive COVID-19 test results of their patients.
medical_personnel	warning	o.patientId $\in$ s.pSet	r	Allows physicians to be notified of patients' COVID case activations.
medical_personnel	treatment	o.patientId $\in$ s.pSet	w	Allows physicians to communicate treatment options to their patients.
medical_personnel	consent	o.patientId $\in$ s.pSet	r	Allows physicians to collect the consent from their patients.
medical_personnel	bulletin	o.patientId $\in$ s.pSet	w	Allows physicians to publish medical bulletins for their patients.
medical_personnel	closecontact	o.patientId $\in$ s.pSet	r	Allows physicians to be warned of patients identified as close contact of a suspected / confirmed COVID-19 case.

Table 7
Emergency policies for the COVID-19 case study.

s	tf	exp	pr	esf	stf
medical_personnel	location	o.patientId $\in$ s.pSet	r	edp="COVID-19 case"	All
Allows medical personnel to check the position of their patients.					
external specialist	physiological/#	true	r	edp="COVID-19 case"	{COVID-19 symptomatic, Severe COVID-19}
Allows external specialists to access physiological data of overt COVID-19 patients.					
relative	bulletin	o.patientId $\in$ s.relativeOf	r	edp="COVID-19 case"	All
Allows relatives to receive the medical bulletin of their kin.					
guardian	treatment	o.patientId $\in$ s.guardianOf	r	edp="COVID-19 case"	{Severe COVID-19}
Allows the guardian of a patient in severe conditions to access treatment options.					
guardian	consent	o.patientId $\in$ s.relativeOf	w	edp="COVID-19 case"	{Severe COVID-19}
Allows the guardian of a patient in severe conditions to give the consent to made his/her kin undergo specific treatments					
guardian	result	o.patientId $\in$ s.guardianOf	r	edp="COVID-19 case"	{Severe COVID-19}
Allows the guardian of a patient in severe conditions to access his/her test results.					

trate treatment options to any of his/her patient, and collect the consent to proceed with the treatment, v) issue medical bulletins to his/her patients, informing each of them about his/her conditions, and vi) access notifications issued by the monitoring application, denoting that one of his/her patients has had close contact with a suspected or confirmed COVID-19 case.

Subjects involved in an emergency situation can benefit from additional privileges. A selection of emergency policies for the considered application scenario is reported in Table 7. For any emergency policy *ep*, column *esf* and *stf* specify expressions respectively denoting the set of emergency scenarios and emergency situations to which *ep* is applied, whereas columns *s, tf, exp*, and *pr* maintain the same meaning as in Table 6.

Under any of the considered emergency situations, physicians can access the position of their patients, as the efficient localization of suspected or possible COVID-19 patients allows their prompt isolation, and prevent infection diffusion. To identify effective treatments for overt COVID-19 patients, or to identify patients who could require hospitalization, external specialists are also authorized to monitor physiological data of patients under the emergency situations *COVID-19 symptomatic* and *Severe COVID-19*.

In order to better bridge the gap between patients under quarantine protocol and their families, relatives are made aware of the conditions of their kin who cannot be visited during the quarantine. The access to the medical bulletin of a patient under any of the considered emergency situations is thus extended to a set of preregistered patient's relatives. Relatives can also play a fundamental role for patients in severe conditions, who, due to their health status, are unable to understand or take actions, acting as their guardians. Privileges granted to patients in ordinary situations are then applied to guardians of patients in critical conditions. For instance, in a Severe COVID-19 emergency situation, a guardian receives communication of the patient's treatment options and consents to specific treatments on his/her behalf.

8.2. Experiments

Let us now focus on the experiments we have carried out to evaluate the efficiency of the proposed access control approach, considering as a reference scenario the case study introduced in Section 8.1.

For our performance evaluations we focus on the following aspects: *transmission time*, which denotes the time a published message takes for being received by a rightful subscriber, *time overhead*, which quantifies the time requested by the enforcement monitor to take a decision related to an access request, and *throughput*, namely, the average number of MQTT control packets per seconds which are analyzed by our framework.

Transmission time provides a first indication of the framework's usability, as it allows quantifying the overall communication latency. However, it is a quite coarse-grained property, as it shows the total duration of multiple communication phases. A client to client (*c2c*) communication in an MQTT environment is articulated into an initial client to broker (*c2b*) communication phase, during which a publishing request is issued by a client to broker, followed by a broker to client (*b2c*) phase, within which the broker forwards a copy of the received message to any rightful subscriber. Each packet issued by a client or forwarded by a broker is intercepted by the enforcement monitor, which allows the transit only if applicable policies grant it. Therefore, to assess the impact of policy enforcement on the overall transmission time, for any *c2c* communication, we keep track of the *time overhead* introduced by the enforcement monitor during the phases *c2b* and *b2c*, and, comprehensively, during the whole *c2c* communication. Similarly, *throughput* is calculated with reference to the communication phases *c2b, b2c*, and *c2c*.

The assessment of our framework performance refers to a target setup of the monitoring application, which aims at supporting a realistic deployment tailored for a nursing home of big size. Our empirical evaluation is then complemented with a further setup, introduced to show the framework behavior in an extreme case configuration of the monitoring application.

Target setup considers a subject set of 300 patients, 60 healthcare workers among nurses and physicians, 60 relatives, and 6 specialists. In contrast, in the extreme case setup, any subject group's numerousness is multiplied by 5.

Subjects communication in both setups is regulated by a policy set that includes the ordinary and emergency policies presented in Section 8.1, and a few additional ones introduced to grant to all nursing home devices the privilege to publish sensed data.

Our experiments refer to a deployment that includes 3 enforcement monitors, each managing the connections of one-third of the subjects of each subject group. Time overhead, transmission time, and throughput are calculated at each enforcement monitor interface with the nursing home's MQTT environment. For our experiments, MQTT clients have been configured in such a way that, in the whole MQTT environment, on average, 60 publishing requests per second are generated, and the analyzed scenario refers to a period of 30 days of simulated executions of the monitoring application.

A detailed view of the computed performance measures is presented in Table 8, whereas Fig. 6 shows, at aggregate level, their trend in the considered setups. In the target setup, on average, a transmission time of ∼43 ms has been observed, of which, almost ∼31 ms is due to the enforcement overhead. Overall, in this setup, our framework analyzes ∼96 control packets per second. In contrast, the transmission time grows up to ∼71 ms in the extreme case setup, with an average time overhead of ∼69 ms, and a total throughput which decreases to ∼88 cp/s. As visible in Fig. 6, in each setup, the 3 enforcement monitors almost introduce the same time overhead, show similar transmission times, and comparable packet processing rates. For each monitor, time overhead related to phase *c2b* (shown in red) is significantly higher than in phase *b2c* (in blue), whereas, even though with a less marked difference, the opposite trend is observed with the throughput related to the *c2b* and *b2c* phases (respectively shown in yellow and green). This behavior is justified by the enforcement monitor activities in each communication phase. Indeed, in the *c2b* phase, on receipt of a control packet, in order to select and enforce the applicable policies, the enforcement monitor has to interact with the CEP system to check whether the intercepted packet causes the evolution of any emergency scenario, whereas, in the *b2c* phase no interaction with the CEP system is required. It is worth noting that the abovementioned differences are more accentuated in the extreme case setup, since, due to a higher number of patients to be monitored, the number of instances of COVID-19 case scenarios to be managed by the CEP interface is significantly higher.

Overall, our experiments have shown satisfactory results in both setups. The observed enforcement overhead is reasonably low even in the extreme case, where the size of the monitored environment is not negligible.

8.3. Results and discussion

The case study and related experimental evaluation have shown the feasibility of our approach, timely emergency identification, and reasonably efficient policy enforcement. However, the experience has also highlighted a few framework limitations that we discuss in the remainder of this section, along with possible strategies to handle them.

Preparedness The first shortcoming is the lack of services that could favor timely planning of countermeasures to potential aggravations of an emergency. The proposed approach has been designed considering the correctness of the enforcement mechanism as the primary requirement. Although our experimental evaluation has shown the ability of the system to operate with almost real-time data, the proposed emergency management mechanism has an inherently reactive nature. Indeed, emergency situations are faced only when these have occurred. To enhance subjects' preparedness for possible emergency worsening, our framework can be complemented with additional modules, that analyze the messages exchanged in a target environment to predict the possible evolutions of an emergency scenario before the CEP system observes them. Although not exploitable for access control purposes,[13] this strategy could make emergency management proactive, favoring the organization and enactment of timely countermeasures. For instance, referring to the COVID-19 case scenario (see Section 8.1), let us consider the case of a patient involved in the emergency situation *COVID-19 symptomatic*. Based on the emergency development plan represented in Fig. 5, such an emergency situation can evolve into *Severe COVID-19* if specific symptoms occur. According to the specification of the complex event *SevereSymptom* proposed in Table 4, such an evolution is triggered by severe breathlessness episodes, during which the respiratory rate goes over the threshold of 30 breaths per minute. Patients with severe symptoms could require particular treatments, such as supplemental oxygen, or it could even be necessary to move them to a hospital for more intensive care. However, it may happen that these measures could not be immediately applicable. For instance, transfers could be constrained by the availability of an ambulance, whereas oxygen administration by a limited stock of

[13] Access control decisions depend on the policies applicable to access requests, which in turn depend on the current stage of the emergency scenarios where the requesting subjects are involved. The evolution of emergency scenarios cannot rely on predictions. It has to depend on occurring events to ensure the correctness and completeness of granted authorizations.

Table 8
Observed performance measures.

		Phase	Monitor 1	Monitor 2	Monitor 3	Avg	Tot
Target setup	Time	c2b	25.25 ms	25.74 ms	25.35 ms	25.45 ms	–
	overhead	b2c	5.1 ms	5.17 ms	5.15 ms	5.14 ms	–
		c2c	30.36 ms	30.91 ms	30.50 ms	30.59 ms	–
	Transmission time	c2c	40.48 ms	44.49 ms	44.09 ms	43.02 ms	–
	Throughput	c2b	14.75 cp/s	14.69 cp/s	14.72 cp/s	14.72 cp/s	44.16 cp/s
		b2c	17.28 cp/s	17.26 cp/s	17.29 cp/s	17.28 cp/s	51.83 cp/s
		c2c	32.03 cp/s	31.94 cp/s	32.01 cp/s	32.00 cp/s	95.99 cp/s
Extreme case setup	Time	c2b	62.15 ms	62.24 ms	64.03 ms	62.80 ms	–
	overhead	b2c	5.83 ms	5.73 ms	6.16 ms	5.91 ms	–
		c2c	67.98 ms	67.97 ms	70.19 ms	68.71 ms	–
	Transmission time	c2c	70.86 ms	70.77 ms	72.27 ms	71.3 ms	–
	Throughput	c2b	13.93 cp/s	13.85 cp/s	13.85 cp/s	13.87 cp/s	41.63 cp/s
		b2c	15.50 cp/s	15.46 cp/s	15.48 cp/s	15.48 cp/s	46.44 cp/s
		c2c	29.43 cp/s	29.31 cp/s	29.33 cp/s	29.36 cp/s	88.07 cp/s

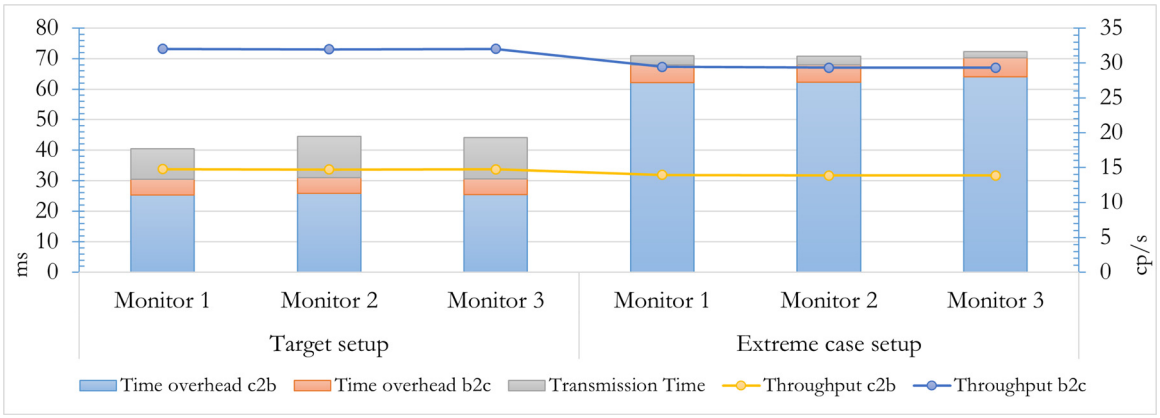

Fig. 6. Performance analysis results.

cylinders in the nursing home. Therefore, a service capable of predicting a worsening and informing the medical personnel could help to shorten these delays, favoring more timely treatments.

Machine learning (ML) / Deep Learning (DP) and CEP-based solutions are typically employed as alternative approaches to the analysis of data streams in IoT applications. Nonetheless, a few integrated solutions have been proposed to enable predictive analytics in CEP systems (e.g., see Akbar et al., 2017). Following the same idea, we believe our CEP system could benefit from ML/DP services able to predict the occurrence of complex events. However, the integration is far from being straightforward, as several challenging issues should be addressed, which we briefly discuss in what follows. Although some prediction algorithms have already been proposed for the same purpose (e.g., see Akbar et al., 2017), a thorough analysis and experimental evaluation are required to evaluate their applicability in our scenario, identifying the best algorithms to employ. Even hypothesizing the availability of an ML algorithm that fits our scenario, one should also consider further issues, namely: i) how ML/DP modules could be interfaced with the existing system, and ii) once predictions are derived, how they can be used, exploiting the native framework's communication and access control features. The former issue requires defining an efficient data analysis pipeline for MQTT messages to support real-time analytics. A promising approach to be considered for possible adoption in our framework requires bridging MQTT environments to Kafka[14] ecosystems (e.g., see Hugo et al., 2020; Štufi. and Bačlć., 2022). Data streams can then be analyzed by exploiting the Kafka Stream API (Seymour, 2021) and third-party libraries for ML / DP, such as, for instance, Tensor Flow.[15]. This technology has been already used in several use cases. For instance, Audi uses it in the back-end of a connected vehicle infrastructure to perform real-time traffic recommendations and prediction maintenance.[16]

As far as prediction usage is concerned, sinks could be necessary to collect predicted events, which could then be converted into MQTT messages and issued to authorized subjects. An approach similar to the one we adopt for the actions referred to by emergency evolutions (see Section 7.3) could be used for this purpose. More precisely, a new type of action can be defined and executed during the whole permanence into an emergency scenario stage, which converts predicted events into MQTT messages and delivers them to the rightful subjects.

Concurrency control and scalability Although our experiments have shown reasonably good results in both the considered setups, the observed growth of the transmission time in the extreme case setups could suggest the need for techniques to make the approach efficient even in very large-scale scenarios.

A possible reason for the observed behavior is the centralized approach that handles the evolution of emergency scenarios, which has been designed to be executed on a single CEP Interface.

A key role is played in our system by the handlers *update-notifier* and *no-change-notifier*, which are responsible for managing the evolution of an emergency scenario *es*. In our prototype, a unique component carries out both handlers. This component implements a single-threaded event loop model, employing a pool of internal threads for the interaction with an instance of Redis, which keeps track of metadata related to emergencies and access

[14] https://kafka.apache.org Shapira et al. (2021).

[15] https://www.tensorflow.org.

[16] https://www.confluent.io/kafka-summit-london18/keynote-fast-cars-in-a-streaming-world-reimagining-transportation-at-audi.

control. The incoming complex events generated by the CEP engine are added to an event queue. These events can either refer to a complex event type referred to by the evolutions of *es*, or a special one, hereafter denoted with *no-cet*, which denotes that the CEP engine does not detect any compatible event.

For any event *ce* that is picked up from this queue, if *ce*'s type is *no-cet*, it cannot trigger the evolution of *es*. Therefore, *ce*'s analysis is immediately terminated. Otherwise, a dedicated thread from the pool is selected, which selects the current stage of the emergency scenario from the Redis keyspace, and checks if *ce* causes the evolution of *es*. In such a case, this thread updates the scenario's current stage in the Redis keyspace. Our prototype has been designed in such a way that, at most, one internal thread is active at a time and can update the current stage of *es*. The events are analyzed in the same order the CEP engine has generated them, and thus, *es* evolves following their chronological detection order. Therefore, it has the benefit of avoiding possible conflicts which could arise with distributed implementations; however, as mentioned before, the presence of a single CEP Interface could be a system bottleneck for large-scale scenarios. Different solutions could be employed to address this issue. A straightforward one could require substituting the centralized CEP Interface with multiple instances of this component, each devoted to interacting with a different enforcement monitor. This change would allow splitting the emergency management workload into multiple parallel ones, requiring the management of concurrent updates to the scenario's current stage in the Redis keyspace. Despite conflicts can be avoided by specifying read and write accesses to the Redis keyspace within transactions,[17] the updating order cannot be guaranteed to comply with the event detection order. Although this problem could be addressed by configuring the system so that each emergency scenario is entirely handled by only one CEP interface, this strategy is unsuited for applications characterized by a single emergency scenario of big size.

A totally different solution to the issues mentioned above could be an approach allowing the integrated management of event detection and emergency evolution by a unique data management system. Recent data stream processing systems (DSPS), such as Apache Flink,[18] allow sharing state information among events. A few state-of-the-art systems (e.g., Botan et al., 2012; Wang et al., 2011; Zhang et al., 2020), allow supporting concurrent state access during stream processing with transactional semantics. The access to application states by multiple executors is achieved using state transactions,[19] whose execution is scheduled in such a way to respect the chronological order of their trigger events. It is expected that our framework could benefit from integrating a similar system. Indeed, in such a case, our CEP Interface could be relieved from the necessity to handle emergencies' evolutions, as this task would be in charge of the adopted DSPS. However, several issues should be addressed before this integration could be possible.

The first issue is related to modeling aspects. We believe emergency evolutions can be implemented as state transactions executing concurrent accesses to the current stage of an emergency scenario, which, along with all other emergency management metadata, represents a shared state. We also think the events that trigger these state transactions cannot refer to the same complex event type as those that trigger the implemented evolutions. Indeed, in our application scenario, a transaction implementing an evolution *ev* could only be executed if *ev* refers to the current stage of the emergency scenario within its source component *ev.src*. This additional precondition is not caught by the events of type *ev.cet*, as in our framework complex event types do not specify the emergency situations where related events can occur, and thus, event occurrence does not depend on the scenario's current stage. As a consequence, it is necessary to determine if a new class of event types is required to trigger transactions' execution.

Another issue is related to the management of actions possibly specified by the evolutions. Based on the previous reasoning, evolutions could be implemented as state transactions. Therefore, how actions can be executed when transactions are scheduled for execution should be defined.

Finally, a further significant issue is how policy retrieval can be achieved. To select the access control policies applicable to access requests, the enforcement monitor has to be aware of the current stage of the emergency scenarios where requesting subjects are involved. In the current prototype, based on the fact that emergency metadata and access control policies are both managed by the same instance of Redis, we have used a Redis transaction to achieve this selection. This transaction retrieves the current stage of the emergency scenarios where the subject is involved and derives the applicable policies for the resulting selection. A similar approach could also be used if a stateful DSPS which employs concurrent state access was employed. However, to define how this selection can be achieved, one should first evaluate if emergency and access control metadata can both be represented as states or how Redis could be jointly used with the DSPS for this selection.

Our early analysis has revealed that the DSPS presented in Zhang et al. (2020), denoted TStorm, distinguishes from the other proposals for efficient state transaction scheduling and processing mechanisms designed to exploit parallelism opportunities offered by multi-core architectures. Therefore, assuming that previously mentioned issues could be addressed, the integration of TStorm within our system could magnify scalability and efficiency benefits.

9. Related work

The great majority of approaches to handle access control during emergencies employ the break the glass (BtG) paradigm, according to which, during an emergency, a user requests and gains access to resources that in normal situations would not be permitted.

A seminal work by Brucker and Petritsch (2009) proposed an approach to integrate BtG policies into access control models. The proposed mechanism relies on emergency levels, namely BtG policies that extend the privileges granted by regular policies, allowing fine-grained control over protected resources. Enabling policies are employed to allow subjects to activate BtG policy at run-time. In case of break the glass requests during an emergency, the access decision is derived from the applicable active BtG policies. The same authors in Brucker et al. (2010) investigated the integration of BtG mechanisms with Attribute-based Encryption (ABE), which is a technique that uses public-key cryptography to enforce fine-grained access control based on user attributes. The approach proposed in Brucker et al. (2010) is based on a hierarchy of emergency attributes employed to encrypt and decrypt data resources. Emergency attributes denote emergency severity levels activated/inactivated by a central authority and are used to encode BtG policies. A BtG access is only possible when the emergency attribute required for decryption is active and the same attribute was active at encryption time.

The approaches by Brucker and Petritsch (2009); Brucker et al. (2010) and Rajput et al. (2021) have not been designed for IoT applications, which in contrast have been targeted by several more recent proposals. For instance, Yang et al. (2018) propose a password-based break-glass access control mechanism for IoT ecosystems. In Yang et al. (2019) the

[17] Redis employs an optimistic concurrency control, where the check-and-set mechanism is used to control transaction executions.

[18] Apache Flink, https://flink.apache.org/.

[19] A state transaction is a set of state accesses triggered by the processing of one input event at one operator (Zhang et al., 2020).

same approach has been used to protect the access to patients' medical files in a cloud-enabled IoT healthcare ecosystem.

Aski et al. (2021) proposed an access control mechanism based on BtG and ABE to regulate the access to healthcare data within a cloud-enabled IoT medical ecosystem during emergency situations. The proposed approach employs pre-shared passwords to extract BtG keys, however, implementation details of the proposed model are not discussed.

de Oliveira et al. (2020) proposed a cloud-enabled framework where a BtG mechanism is used to grant medical personnel access to encrypted medical data managed by a cloud-based application during emergency situations.

Belguith et al. (2018), proposed a BtG access control approach that leverages on: i) Shamir's secret sharing scheme, to derive secret shares from a secret access key, ii) ABE, used to encrypt the secret shares, and iii) QR encoding of the encrypted shares. In order to execute a BtG access users have to scan QR codes and recover individual keys with their attributes.

Van Bael et al. (2020) proposed a context-aware BtG access control framework for IoT environments. A key feature of Van Bael et al. (2020) is the ability to predict emergencies from contextual information generated by IoT sensors. On the prediction of an emergency, users are notified of the predicted situation, and contextually, the related break-glass policies are activated. The system then waits for possible break-glass requests.

Marinovic et al. (2014), proposed a BtG model that employs a logic programming language to reason about unknown and conflicting information in policy decisions, and a policy specification language that allows security administrators to rule break-glass accesses.

Several BtG extensions have also been proposed for RBAC (e.g., Ferreira et al., 2009; Maw et al., 2014; Maw et al., 2016; Nazerian et al., 2019).

Our approach and BtG based approaches have significant differences. In our framework subjects do not need to explicitly ask for exceptional access permissions, since they gain access privileges granted by emergency policies as soon as emergency situations are detected, favoring a more efficient control of the protected data. In addition, none of the above-mentioned proposals were designed to work with MQTT based IoT ecosystems, and except for Van Bael et al. (2020), none provide an emergency detection mechanism.

Padmashree et al. (2021) proposed an access control framework that employs Elliptic Curve Cryptography to enforce secure access to patient data over Healthcare IoT in both normal and emergency situations. The main contribution of Padmashree et al. (2021) is a lightweight cryptographic enforcement mechanism that can be used during emergencies. However, no support for emergency detection is provided, nor for the specification and management of emergency evolutions.

We are only aware of a few more approaches to emergency detection and data sharing regulation in emergency situations (i.e., Carminati et al., 2013; Dallel et al., 2021; Kabbani et al., 2014). More precisely, Kabbani et al. (2014) proposed an approach to enforce ABAC policies in ordinary and emergency situations. Like our model, ordinary and emergency situations are detected by employing a CEP-based approach. However, different from our work, no systematic approach to gathering events from event sources and binding events to ordinary and emergency situations are discussed, and no performance evaluation is proposed.

Carminati et al. (2013) proposed a framework to enforce controlled information sharing under emergency situations, which employs a CEP system for emergency detection. In Carminati et al. (2013), emergency policies regulate the generation of temporary access control policies that override ordinary privileges in emergency situations. Once an emergency is detected, the applicable temporary access control policies are generated, stored in local repositories, and kept active until either another emergency is detected or the current emergency is over. In contrast, our approach does not require generating and managing temporary policies. Once an emergency situation is detected, the applicable emergency policies are selected to grant to the involved subjects the exceptional privileges permitted in the considered situation. In addition, different from our work, in Carminati et al. (2013) no management support is given to the possible development of an emergency situation into an articulated emergency scenario.

Lastly, Dallel et al. (2021) proposed an XACML-based access control framework to manage emergencies within IoT smart buildings. In Dallel et al. (2021), a smart building includes several types of emergency detection sensors (e.g., smoke detectors) which employ an MQTT-based communication interface. A key feature of Dallel et al. (2021) is an emergency communication center (ECC), which alerts rescue agencies (e.g., the fire department) of the emergency notifications issued by the above-mentioned sensors. In addition, to favor a prompt intervention, the ECC delegates rescue teams access to relevant data such as damaged areas, safe exits, and evacuees' locations. Data sharing is controlled by delegation policies, embedded in capabilities tokens, and enforced by an ad-hoc designed module denoted delegation decision point. Despite both our framework and Dallel et al. (2021) target MQTT-based IoT ecosystems, these works have significant differences. In particular, in Dallel et al. (2021), no support is given for the modeling and management of emergency evolutions. In addition, it is not clear how ordinary access control is restored once an emergency is over.

10. Conclusions

In this paper, we have proposed an access control system to enforce controlled data sharing within MQTT-based IoT ecosystems during emergency and ordinary situations. The system analyzes the MQTT messages exchanged in a monitored ecosystem leveraging on Complex Event Processing for emergency detection. Emergency and ordinary ABAC policies are employed to regulate data sharing in emergency and ordinary situations respectively.

We have assessed the feasibility of the proposed approach with a case study related to a healthcare application that monitors nursing home patients during the COVID-19 pandemic. Early experimental performance evaluations show promising results and a quite acceptable policy enforcement overhead. We plan to enhance this preliminary assessment by testing the approach in further application scenarios.

In spite of this positive feedback, the experience has also allowed the identification of framework limitations, related to emergency preparedness, and efficiency when growing the scale of application scenarios. Although we have started reasoning on potential addressing strategies (see Section 8.3), we plan to further explore them in future work.

Preparedness can be favored by services that predict emergency aggravations. Indeed, such a functionality allows the preventive planning of countermeasures and thus shortens the reaction time to emergency occurrences. However, the implementation of these services requires addressing some methodological and technological issues that we have discussed in Section 8.3.

On the other hand, different approaches could favor a higher framework efficiency in large-scale scenarios. In particular, we have considered the use of multiple CEP interfaces, and the possible substitution of the CEP engine with a DSPS able to support concurrent state accesses. The latter option appears quite promising, even due to systems able to exploit the parallelism opportunities offered by modern multi-core architectures. However, at the same

time, it is constrained by issues related to the modeling and management of state transactions (see Section 8.3).

An additional strategy to optimize our framework could be the use of multi-query optimization techniques for CEP systems (e.g., see Zhang et al., 2017). These approaches aim at reducing redundant computation among pattern queries that work on the same data streams. We plan to design similar techniques for our framework as future work. Lastly, we are also planning to develop a tool that helps security administrators to perform administrative operations related to ordinary and emergency situations, and a monitoring tool to evaluate the effects of the specified policies.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Pietro Colombo: Conceptualization, Methodology, Software, Writing – original draft, Writing – review & editing. **Elena Ferrari:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing, Supervision, Project administration, Funding acquisition. **Engin Deniz Tümer:** Methodology, Software, Investigation, Writing – original draft.

Acknowledgments

This work has received funding from CONCORDIA, the Cybersecurity Competence Network supported by the European Union's Horizon 2020 Research and Innovation program under grant agreement no. 830927, and from RAIS (Real-time analytics for the Internet of Sports), Marie Skłodowska-Curie Innovative Training Networks (ITN), under grant agreement no. 813162. The content of this paper reflects only the authors' view and the Agency and the Commission are not responsible for any use that may be made of the information it contains.

Appendix A. Correctness

In this appendix, we discuss the key properties of the proposed framework, which are instrumental in correctly (i) managing the evolution of emergency scenarios and (ii) selecting emergency and ordinary policies applicable to an access request. The correctness of policy enforcement leverages on these properties, which represent the core contribution of the current paper. In contrast, we do not cover here the correctness of the enforcement mechanism, since these proofs would be heavily based on the message-altering approach presented in Colombo and Ferrari (2018), which here has only been used as a black-box external service.

In summary, the properties we cover in this section are the following:

1. the enforcement monitor intercepts any MQTT control packet exchanged in the monitored environment, and forwards any intercepted client's publishing request to the CEP interface;
2. any primitive event notification generated by the CEP interface on receipt of an intercepted publishing request refers to a complex event type that is bound to that packet;
3. any specified complex event type is referred to by at least one primitive event of the generated set;
4. the CEP engine detects any complex event of the specified types which occurs in the monitored environment, notifying it to the CEP interface;
5. the CEP engine correctly updates the current stage of the emergency scenarios based on the complex events notified by the CEP engine;
6. the enforcement monitor correctly selects the policies applicable to an access request issued by a subject based on the current stage of the emergency scenarios where that subject is involved.

Let us consider an application scenario where it is required to regulate data sharing in an MQTT environment *tenv*, in the presence of an emergency scenario Es that involves a set S of subjects. Let Ps be the set of ordinary and emergency policies specified for Es, and let PET and CET be the set of primitive and complex event types specified for the considered scenario.

Let *C* be the set of MQTT clients of *tenv*, let *b* be the message broker of *tenv*, and let us assume a framework deployment that includes an enforcement monitor *m*, a NoSQL datastore that keeps track of metadata related to emergency management and access control, and a CEP interface which manages the evolution of Es relying on the detection abilities of a CEP engine.

We assume that all clients of *tenv* are configured to connect with m rather than directly with b.

Statement 1 (System interface)**.** Any control packet issued by an MQTT client c_i or the MQTT broker b of *tenv* is intercepted by m.

Discussion. Based on the above-mentioned assumptions, any client c_i is configured to connect with m rather than directly with b, and m to connect with b on behalf of c_i. Since control packets can only be delivered through the established connections, all packets issued by c_i, or b, are intercepted by m.

To reason on the behavior of the adopted CEP system, we rely on some obvious assumptions: i) all primitive event notifications observed by the CEP engine are generated by the CEP interface, ii) the CEP engine observes all primitive event notifications generated by the CEP interface, and iii) all complex event notifications generated by the CEP engine are received by the CEP interface.

Let cp_{PB} be a client publishing a request intercepted by m and notified to the CEP interface within a packet $cp_{PB}{}^*$ along with the subject, object, and environment attributes associated with the related access request context. Let us denote with PEN_t the set of primitive event notifications generated by the CEP interface on receipt of $cp_{PB}{}^*$ at time t. In addition, let *isBound(et,p)* be a boolean function that receives as input a primitive event type *et* and a publishing request *p* and evaluates true iff *p* is bound to *et*, namely if the binding criteria referred to by *et* are satisfied for *p*.

Statement 2 (Compliancy of primitive event generation)**.** For any primitive event notification $pen \in PEN_t$, cp_{PB} is bound to the primitive event type $pet \in PET$ referred to as type of pen.

Discussion. As discussed in Section 7.1, the control task that is instantiated by the CEP interface on receipt of $cp_{PB}{}^*$ at time t iterates over the set of primitive event types PET to select those usable for primitive event generation. A primitive event type pt of PET is selected iff *isBound*(pt, cp_{PB}) returns true. Any selected primitive event type pt is then employed to derive a primitive event notification pe which refers to pt as type. The union of the generated notifications composes PEN_t. Therefore, since $pen \in PEN_t$, and PEN_t has been derived from cp_{PB}, then *pen* has been generated starting from a complex event type pet that is bound to cp_{PB}, which is also referred to as the type of *pen*.

Statement 3 (Completeness of primitive event generation)**.** For any primitive event type $pet \in PET$ such that cp_{PB} is bound to pet, there exists a primitive event notification $pen \in PEN_t$, which refers to pet as type.

Discussion. By construction, the control task instantiated by the CEP interface on receipt of $cp_{PB}{}^{*}$ generates a primitive event notification pen for any primitive event type pet of PET such that isBound(pet, cp_{PB}) is true, which specifies pet as type. As a consequence, there cannot exist a primitive event type *pet'* ∈ PET such that isBound(pet', cp_{PB})=true, which is not referred to as type by a primitive event notification pen ∈ PEN_t.

Statement 4 (Correctness of complex event generation)**.** Any complex event of type cet ∈ CET which occurs in the monitored environment *tenv* is detected by the CEP engine and notified to the CEP interface.

Discussion. Correctness of complex event generation entirely rely on the detection abilities of the CEP engine, and we assume the CEP engine's ability to correctly catch events of any specified complex event type cet ∈ CET, notifying all detected events to the CEP interface.

Let us now focus on the ability to correctly manage the evolution of emergency scenarios. At Es specification time, the CEP interface instantiates the event handlers *update-notifier* and *nochange-notifier*, which subscribe to the notification of events generated by the CEP engine and manage the evolution of the emergency scenario (see Section 7.2). In particular, update-notifier is notified of the detection of any complex event ce that refers as type the same complex event type referred to by at least one evolution of the Es' emergency development plan.

Let Edp be the emergency development plan of Es, and let Ev be the set of emergency evolutions referred to by Edp (see Section 5).

Let us denote with *ems* the current stage of Es, and let *v* be the last value to which *ems* has been set at time *t*, where *t* refers to the time annotation of the complex event type that has caused the update. Let $CE_t{}^{v}$ be the set of complex events notified by the CEP engine to *update-notifier* since the last update of ems to v at time *t*. A strict total order is defined on $CE_t{}^{v}$, based on the time annotation referred to by the collected events, and thus, complex event notifications are processed in the same order as they have been generated by the CEP engine. In addition, let us refer to the time annotation and type associated with a complex event notification ce ∈ $CE_t{}^{v}$ with notation ce.ts and ce.type, respectively.

To reason about the possible evolution of the current stage of Es, let us refer to any evolution ev in Ev which refers to v within component src as *evolution based on v*, and let us denote with *trigger of v evolution* any event ce ∈ $CE_t{}^{v}$ such that there exists an evolution ev based on v which refers to ce.type as complex event type (i.e., ce.type = ev.cet).

Statement 5 (Correctness of emergency scenario evolution)**.** On receipt of an event event ce that is a trigger of v evolution, the value referred to by *ems* is updated from v to v' iff ∃ ev ∈ Ev that is an evolution based on v, such that ev.trg=v' ∧ ev.cet=ce.type ∧ ∄ ce' ∈ $CE_t{}^{v}$ that is a trigger of v evolution, such that t<ce'.ts<ce.ts.

Discussion. By construction, update-notifier is the only process that can update ems. In addition, by construction, on receipt of ce, update-notifier looks for an evolution ev based on v in Ev which refers to ce.type as complex event type. Based on the well-formedness rules of emergency development plans (see Section 5), any pair of evolutions that refer to the same value within component src, have to specify events of different types within component cet. As a consequence, if such an evolution ev based on v exists, it is unique within Ev, and thus, denoting the value of ev.trg with v', update-notifier updates the current stage of Es setting ems to v'.

Let us now suppose by absurd that although ce has caused the update of ems from v to v', there exists an event ce' ∈ $CE_t{}^{v}$, such that t<ce'.ts<ce.ts, which is a trigger of v evolution. update-notifier has been defined in such a way to receive and process complex event notifications based on time annotation in ascending order, thus, ce' is handled before ce. Since ce' is a trigger of v evolution, there exists an evolution ev' based on v which specifies ce'.type as complex event type. Let us now denote with v^{*} the value referred to by ev'.trg, and with t^{*} the value of ce'.ts. On receipt of ce', update-notifier updates ems to v^{*}. By assumption ce, which is processed after ce', causes the update of ems from v to v'. Therefore, at the processing time of ce, ems would refer to a value v^{*} potentially different from v, and, as a consequence, ce could not be a trigger of v evolution, contradicting our initial statement.

Let us now consider how emergency and ordinary policies to be applied to subjects' access requests are determined. Let us start with some preliminary definitions. We say that a an emergency scenario Es involves a subject s iff s' attributes satisfy the subject filter expression sf of Es. Similarly, an emergency situation v of Es involves a subject s iff Es involves s Es, and Es refers to v as its current stage. In addition, s is said to be in an ordinary situation iff all emergency scenarios where s is involved are inactive (i.e., they refer to ⊥ as their current stage), or s is not involved in any scenario.

Let us now focus on policy applicability. Let ar be an access request issued by a subject s, which requires to read /write a message on topic t, and let Ps be the set of ordinary and emergency policies defined for the monitored environment tenv.

Let us first consider emergency policies. Based on Definition 6 in Section 5, an emergency policy ep of Ps is said to be applicable to the access request ar iff: i) ep.s matches s attributes, ii) s is involved in an emergency scenario Es which is among the emergency scenarios referred to by ep.esf, iii) the emergency situation v referred to as current stage of Es (where, v ≠⊥) is among the emergency situations referred to by ep.stf, iv) t is matched by ep.tf, and v) ep.pr matches the read/write privilege requested by ar.

In contrast, an ordinary policy p of Ps is said to applicable to the access request ar issued by s iff: i) s is not involved in any emergency scenario or all scenarios where s is involved are inactive, ii) p.s matches s attributes, v) t is matched by p.tf, and v) p.pr matches the read/write privilege requested by ar.

Statement 6 (Correctness of policy selection)**.** Any access request ar issued by a subject s is regulated by a set of policies APs, where APs ⊆ Ps, which is either entirely composed of emergency policies or ordinary policies. Any emergency/ordinary policy p in APs is applicable to ar, and does not exist an emergency/ordinary policy p' of Ps applicable to ar that does not belong to APs.

Discussion. Correctness of policy selection entirely rely on the selection abilities of the enforcement monitor. For ordinary policies we rely on the mechanism described in Colombo and Ferrari (2018). Emergency policy selection is achieved by employing a similar technique and therefore, in the current paper, its description has been omitted.

References

Akbar, A., Khan, A., Carrez, F., Moessner, K., 2017. Predictive analytics for complex iot data streams. IEEE Internet of Things Journal 4 (5), 1571–1582. doi:10.1109/JIOT.2017.2712672.

Akinbi, A., Forshaw, M., Blinkhorn, V., 2021. Contact tracing apps for the covid-19 pandemic: a systematic literature review of challenges and future directions for neo-liberal societies. Health Information Science and Systems 9 (18).

Aski, V., Dhaka, V.S., Parashar, A., 2021. An attribute-based break-glass access control framework for medical emergencies. In: Innovations in Computational Intelligence and Computer Vision. Springer, pp. 587–595.

Banks, A., Briggs, E., Borgendale, K., Gupta, R., 2019. Mqtt version 5.0. OASIS Standard.

Belguith, S., Gochhayat, S.P., Conti, M., Russello, G., 2018. Emergency access control management via attribute based encrypted qr codes. In: 2018 IEEE Conference on Communications and Network Security (CNS). IEEE, pp. 1–8.

Botan, I., Fischer, P.M., Kossmann, D., Tatbul, N., 2012. Transactional stream processing. In: Proceedings of the 15th International Conference on Extending Database Technology. Association for Computing Machinery, New York, NY, USA, pp. 204–215. doi:10.1145/2247596.2247622.

Bray, T., 2017. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. doi:10.17487/RFC8259.

Brucker, A.D., Petritsch, H., 2009. Extending access control models with break-glass. In: Proceedings of the 14th ACM symposium on Access control models and technologies, pp. 197–206.

Brucker, A.D., Petritsch, H., Weber, S.G., 2010. Attribute-based encryption with break-glass. In: IFIP International Workshop on Information Security Theory and Practices. Springer, pp. 237–244.

Carminati, B., Ferrari, E., Guglielmi, M., 2013. A system for timely and controlled information sharing in emergency situations. IEEE Transactions on Dependable and Secure Computing 10 (3), 129–142.

Colombo, P., Ferrari, E., 2018. Access control enforcement within mqtt-based internet of things ecosystems. In: Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies, pp. 223–234.

Colombo, P., Ferrari, E., Tümer, E.D., 2021. Regulating data sharing across MQTT environments. J. Netw. Comput. Appl. 174, 102907. doi:10.1016/j.jnca.2020.102907.

European Centre for Disease Prevention and Control, 2020. Increase in fatal cases of COVID-19 among long-term care facility residents in the EU/EEA and the UK.

Cugola, G., Margara, A., 2012. Processing flows of information: From data stream to complex event processing. ACM Computing Surveys (CSUR) 44 (3), 1–62.

Dallel, O., Ayed, S.B., Taher, J.B.H., 2021. Secure iot-based emergency management system for smart buildings. In: 2021 IEEE Wireless Communications and Networking Conference (WCNC). IEEE, pp. 1–7.

Ferreira, A., Chadwick, D., Farinha, P., Correia, R., Zao, G., Chilro, R., Antunes, L., 2009. How to securely break into rbac: the btg-rbac model. In: 2009 Annual Computer Security Applications Conference. IEEE, pp. 23–31.

G. Cugola and A. Margara, 2015. The complex event processing paradigm. In: Data Management in Pervasive Systems. Springer, pp. 113–133.

Giatrakos, N., Alevizos, E., Artikis, A., Deligiannakis, A., Garofalakis, M., 2020. Complex event recognition in the big data era: a survey. The VLDB Journal 29 (1), 313–352.

Hugo, A., Morin, B., Svantorp, K., 2020. Bridging mqtt and kafka to support c-its: a feasibility study. In: 2020 21st IEEE International Conference on Mobile Data Management (MDM), pp. 371–376. doi:10.1109/MDM48529.2020.00080.

Kabbani, B., Laborde, R., Barrere, F., Benzekri, A., 2014. Specification and enforcement of dynamic authorization policies oriented by situations. In: 2014 6th International Conference on New Technologies, Mobility and Security (NTMS). IEEE, pp. 1–6.

Marinovic, S., Dulay, N., Sloman, M., 2014. Rumpole: An introspective break-glass access control language. ACM Transactions on Information and System Security (TISSEC) 17 (1), 1–32.

Maw, H.A., Xiao, H., Christianson, B., Malcolm, J.A., 2014. An evaluation of break-the–glass access control model for medical data in wireless sensor networks. In: 2014 IEEE 16th international conference on e-health networking, applications and services (Healthcom). IEEE, pp. 130–135.

Maw, H.A., Xiao, H., Christianson, B., Malcolm, J.A., 2016. Btg-ac: Break-the-glass access control model for medical data in wireless sensor networks. IEEE Journal of Biomedical and Health Informatics 20 (3), 763–774. doi:10.1109/JBHI.2015.2510403.

Mishra, B., Kertesz, A., 2020. The use of mqtt in m2m and iot systems: A survey. IEEE Access 8, 201071–201086.

Nazerian, F., Motameni, H., Nematzadeh, H., 2019. Emergency role-based access control (e-rbac) and analysis of model specifications with alloy. Journal of information security and applications 45, 131–142.

de Oliveira, M.T., Bakas, A., Frimpong, E., Groot, A.E., Marquering, H.A., Michalas, A., Olabarriaga, S.D., 2020. A break-glass protocol based on ciphertext-policy attribute-based encryption to access medical records in the cloud. Annals of Telecommunications 1–17.

Ouslander, J.G., Grabowski, D.C., 2020. Covid-19 in nursing homes: calming the perfect storm. Journal of the American Geriatrics Society 68 (10), 2153–2162.

Padmashree, M., Khanum, S., Arunalatha, J., Venugopal, K., 2021. Etpac: Ecc based trauma plight access control for healthcare internet of things. International Journal of Information Technology 13 (4), 1481–1494.

Qiu, J., Tian, Z., Du, C., Zuo, Q., Su, S., Fang, B., 2020. A survey on access control in the age of internet of things. IEEE Internet of Things Journal 7 (6), 4682–4696.

Rajput, A.R., Li, Q., Ahvanooey, M.T., 2021. A blockchain-based secret-data sharing framework for personal health records in emergency condition. In: Healthcare, Vol. 9. Multidisciplinary Digital Publishing Institute, p. 206.

Schefer-Wenzl, S., Bukvova, H., Strembeck, M., 2014. A review of delegation and break-glass models for flexible access control management. In: International conference on business information systems. Springer, pp. 93–104.

Seymour, M., 2021. Mastering Kafka Streams and ksqlDB [electronic resource] / Seymour, Mitch, 1st edition O'Reilly Media, Inc..

Shapira, G., Palino, T., Sivaram, R., Petty, K., 2021. Kafka: the definitive guide. " O'Reilly Media, Inc.".

Van Bael, D., Kalantari, S., Put, A., De Decker, B., 2020. A context-aware break glass access control system for iot environments. In: 2020 7th International Conference on Internet of Things: Systems, Management and Security (IOTSMS). IEEE, pp. 1–8.

Wang, D., Rundensteiner, E.A., Ellison, R.T., 2011. Active complex event processing over event streams. Proc. VLDB Endow. 4 (10), 634–645. doi:10.14778/2021017.2021021.

Yang, Y., Liu, X., Deng, R.H., 2018. Lightweight break-glass access control system for healthcare internet-of-things. IEEE Transactions on Industrial Informatics 14 (8), 3610.

Yang, Y., Zheng, X., Guo, W., Liu, X., Chang, V., 2019. Privacy-preserving smart iot-based healthcare big data storage and self-adaptive access control system. Information Sciences 479, 567–592.

Zhang, S., Vo, H.T., Dahlmeier, D., He, B., 2017. Multi-query optimization for complex event processing in sap esp. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 1213–1224. doi:10.1109/ICDE.2017.166.

Zhang, S., Wu, Y., Zhang, F., He, B., 2020. Towards concurrent stateful stream processing on multicore processors. In: 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020. IEEE, pp. 1537–1548. doi:10.1109/ICDE48307.2020.00136.

Štufi, M., Bačlć, B., 2022. Designing a real-time iot data streaming testbed for horizontally scalable analytical platforms: Czech post case study. In: Proceedings of the 11th International Conference on Sensor Networks - SENSORNETS,. SciTePress, INSTICC, pp. 105–112. doi:10.5220/0010788300003118.

Pietro Colombo is an associate professor of Computer Science at the University of Insubria (Italy), where he works within the STRICT SociaLab of the Department of Theoretical and Applied Sciences. His most recent research activities are in the field of access control in modern data management systems, and Internet of Things applications. He has published more than 40 scientific papers in international journals and conference proceedings, and he is co-inventor of 2 US patents.

Elena Ferrari is a full professor of Computer Science at the University of Insubria, Italy, and scientific director of the K&SM Research Center. Her research activities are related to access control, privacy and trust. In 2009, she received the IEEE Computer Society's Technical Achievement Award for "outstanding and innovative contributions to secure data management". She received a Google Award in 2010, and an IBM Faculty Award in 2014. Since 2012 she has been an IEEE fellow, and in 2019 she has been named ACM Fellow.

Engin Deniz Tumer is a Ph.D. student in Computer Science and Computational Mathematics at the University of Insubria, Italy. He holds a Master degree (2018) in Computer Engineering at Ege University – Turkey. His main research interests are related to Machine Learning, Blockchain, Data Mining and Security. Before starting his Ph.D., he has been a Research Assistant in Computer Engineering at Celal Bayar University – Turkey.