A Distributed Fog-based Access Control Architecture for IoT

Seham Alnefaie*, Asma Cherif, and Suhair Alshehri
Department of Information Technology, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia;
salnufaii0002@stu.kau.edu.sa, acherif@kau.edu.sa, sdalshehri@kau.edu.sa
*Corresponding author: Seham Alnefaie

Received March 31, 2021; revised August 17, 2021; revised October 11, 2021; accepted December 2, 2021; published December 31, 2021

Abstract

The evolution of IoT technology is having a significant impact on people's lives. Almost all areas of people's lives are benefiting from increased productivity and simplification made possible by this trending technology. On the downside, however, the application of IoT technology is posing some security challenges, among them, unauthorized access to IoT devices. This paper presents an Attribute-based Access Control Fog architecture that aims to achieve effective distribution, increase availability and decrease latency. In the proposed architecture, the main functional points of the Attribute-based Access Control are distributed to provide policy decision and policy information mechanisms in fog nodes, locating these functions near end nodes. To evaluate the proposed architecture, an access control engine based on the Attribute-based Access Control was built using the Balana library and simulated using EdgeCloudSim to compare it to the traditional cloud-based architecture. The experiments show that the fog-based architecture provides robust results in terms of reducing latency in making access decisions.

Keywords: ABAC, Access Control, Fog Computing, IoT, Synchronization.

A preliminary version of this paper appeared in IEEE ICCAIS 2019, May 01 - 03, Riyadh, Kingdom of Saudi Arabia. This version includes the comprehensive description of the design, implementation, and analysis of the proposed work. This research project was funded by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under grant No. (DG-9-612-1441). The authors, therefore, gratefully acknowledge the DSR technical and financial support.

http://doi.org/10.3837/tiis.2021.12.016 **ISSN : 1976-7277**

1. Introduction

A growing number of IoT-connected devices are making their way into our everyday lives. This technology is gradually changing human life towards smarter lifestyles. Numerous smart applications that rely on IoT have been implemented in many areas that affect us, including smart cities, healthcare monitoring, building management, and smart homes. This technology allows us to shape and simplify our environment.

IoT technology brings several benefits and plays an essential role in solving many problems in many areas of our lives, but it still also faces a range of challenges and limitations, including scalability, mobility, computational limitations, energy limitations, and memory limitations.

As a result of the continuous increase in the number of connected devices, new security challenges and risks have emerged. Protecting IoT devices from unauthorized access is one of the most crucial issues that face the IoT, because any kind of data leak may lead to severe risk given that these devices often deal with valuable and sensitive data. The first line of defense in guaranteeing the security of IoT data is guaranteeing access control, which is a security technique that controls access to resources by preventing unauthorized users from accessing data as well as preventing authorized users from using the data in an unauthorized manner. Such challenges require tailor-made designs to control access to IoT devices. Therefore, a comprehensive list of requirements has to be considered before developing IoT access control models. These requirements include dynamicity, fine granularity, scalability, revocability, decentralization, availability, low latency, efficiency, flexibility, and standardization.

Several studies have been published to address the access control issue in general [1]–[3]. These studies introduced some well-known and traditional access control models, including: the Discretionary Access Control (DAC) model, which is not suited for implementation in IoT environments due to the difficulty of defining an Access Control List (ACL) for every single device in the IoT; the Mandatory Access Control (MAC) model, which relies on a single administration authority that makes it prone to a single point of failure; the Role-based Access Control (RBAC) model that is based on defining access rights according to the role (job) of the subject, which is not granularly expressive enough to represent fine-grained access control, is not scalable and has centralised administration [2], [4]. ; the Capability-based access Control (Cap-BAC) model, which refers to a self-contained key or token that refers to a specific object, resource or information with a corresponding access rights and which leads to undesirable computation overhead, scalability and policy management problems in the process of granting and validating the capability token [5], [6]; and, the Attribute-based Access Control (ABAC) model, which is based on defining attributes for the subject, object, resources, and environment. ABAC is the most suitable AC model for IoT since it achieves most of the requirements of IoT except for decentralization and availability given that it is centralized and vulnerable to the single point of failure, thus negatively affecting availability [7].

This work is motivated by the need to overcome the shortcomings of centralized architectures by exploiting the emergent fog paradigm. As a matter of fact, fog computing allows data processing near data sources. This reduces data transfer between end nodes and the cloud. Fog computing is employed on one node or several nodes. This improves scalability, provides redundancy, thus increasing availability. More fog nodes can be added when additional computing power is needed [8]–[11]. Combing access control and fog computing is essential

for enforcing real-time access control decisions which is required in IoT applications such as healthcare systems mainly remote health monitoring, smart home management, etc.

In this paper, we improve on the ABAC model by exploiting the fog architecture to provide distribution, enhance availability, and reduce latency. In the proposed architecture, access control components are distributed to deploy policy information and policy decision mechanisms on fog nodes making it near end devices. To validate our solution, we implemented an access control engine based on the ABAC model. This engine performs the access control functions using the Balana library [12]. We simulated the proposed solution using EdgeCloudSim to test whether distributing the access control components in a fog architecture will reduce latency when compared with cloud architecture. The results demonstrated that the fog-based architecture obtained the most robust results in terms of reducing latency. The contribution of this research is twofold. First, we suggest a distributed ABAC access control model, prove its efficiency and compare it with its counterparts. Second, we propose an improved version of vector clocks to ensure the synchronization between the distributed components of the access control model.

This paper is structured as follows. Section 2 provides some propositions regarding the deployment of ABAC in different architectures. Section 3 explains the proposed architecture. Section 4 provides a detailed explanation about the implementation and testing methods and Section 5 concludes the paper.

2. Related Works

The ABAC model was widely used in the literature for controlling access in the IoT environments [13].

In [14], an access control framework called Healthcare Plane (H-Plane) was suggested. It is based on Remote Healthcare Monitoring. Its main component is the Global Data Plane that allows monitoring of patients' heart abnormalities. H-Plane collects the generated data through wearable sensors. The patient's smartphone represents the IoT gateway that tags the data according to its criticality and stores it in a log file. The log file is then moved out to the backend cloud nodes or other edge nodes based on its significance. The model uses the NIST Next Generation Access Control (NIST NGAC) standard. The authors claimed that the framework overcomes most of the limitations faced by current IoT architectures such as high latency, large bandwidth utilization, and low scalability. However, the distribution of ABAC components over the suggested architecture was not discussed. Besides, the efficiency of this framework in terms of policy management was not tested.

In [15], the authors proposed an Intelligent Transportation System (ITS) and an ABAC deployment in a fog-based architecture. Fog computing is used to achieve low latency and location awareness. In the proposed system, the Policy Information Point (PIP) resides in the cloud. This point stores policy information and attributes for one or various domains and the Policy Decision Points (PDPs) interact with the PIP to evaluate access decisions in the fog layer. The Policy Enforcement Points (PEPs) are hosted in the device layer and allow for enforcing access decisions provided by the PDP. This solution is centralized since it is based on a unique PIP hosted in the cloud which leads to a single point of failure problem.

Another deployment for ABAC was proposed in the context of the Industrial Internet of Things (IIoT) in [16] where the eXtended Access Control Markup Language (XACML) standard is used. The XAML functional points are distributed in the IIoT architecture. This model is based on four layers namely, the cloud layer, fog layer, mist layer, and device layer. PIP is placed in the cloud layer. Since PIP requires visibility on a certain domain, a local PIP along with PDP are hosted in the fog layer to provide the cloud functionalities near to the users. Moreover, the PEP, which is responsible for the decision enforcement, is deployed in the mist layer, which normally contains the micro-controllers. However, the model was not implemented, and the synchronization between the cloud PIP and the fog PIP was not discussed. Also, the mist layer is based on some specific hardware devices that require certain types of implementation.

Another distributed access control architecture using XACML was proposed in [17] based on a five-layers architecture (the cloud, the fog, the server, the gateway, and the things). The fog nodes represent intermediaries between the cloud layer and the IoT layer. All communications within the same layer or between layers have been described. The suggested model is based on the XACML standard. However, the authors did not explain the placement of the XAML functional points or how are they synchronized between the top-level layers. Furthermore, they did not provide any performance analysis to evaluate the proposed architecture.

3. Proposed Model

In this section, we propose a distributed access control model that combines ABAC and fog edge schemes (see **Fig. 1**). ABAC components are specified and their placement in the suggested architecture is discussed.

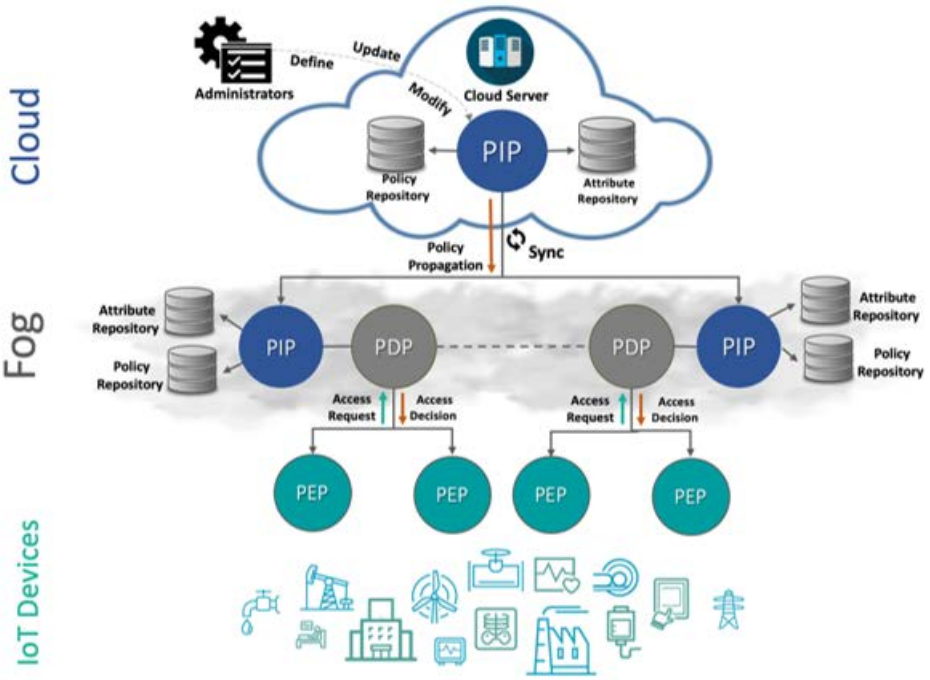


Fig. 1. The proposed architecture.

3.1 Access Control Specification

Based on the requirements mentioned in Section 1, we deduced that ABAC is the most suitable access control model to be employed in IoT environments. This is because it achieves most of the specified requirements, except for decentralization and availability.

ABAC is composed of a set of functional points that collaborate with one another to perform the access control process. These components are:

- Policy Decision Point (PDP): This plays the role of evaluating access requests against access policies and rules for making access decisions.
- Policy Enforcement Point (PEP): This enforces access decisions provided by the PDP.
- Policy Information Point (PIP): This provides the attributes required by the PDP to make access decisions.
- Policy Administration Point (PAP): This provides management of rules and policies [7], [15].

ABAC is a logical access control method where authorization is determined based on a set of attributes. These attributes can be associated with the following components:

- Subject: What or who demands access to an information asset. In healthcare, the subject attributes could be the subject role (a doctor, nurse, or staff member), the department (cardiology, neurology, or pediatric), user ID (Doctor-id, Patient-id, Employee-id).
- Resource: The object or information asset that is impacted by the requested action. In our context, the resource could be the patient data that is collected by the IoT sensors.
- Action: The requested operation or what the subject demands to perform with the resource, e.g., read or edit.
- Environment: The environmental conditions when the access is requested, such as client type (smartphone or PC), current time, or location of the requester [18].

All the aforementioned attributes are evaluated against access policies that describe which operations are allowed and which are not.

3.2 Architectural Design

To better achieve all the requirements, we suggest to adapt ABAC to a fog-cloud environment. Our proposed model is composed of three layers: (i) the cloud layer; (ii) the fog layer, and (iii) the device layer, as shown in **Fig. 1**.

The first layer is the cloud layer which includes the data centers and servers. The second layer is the fog layer and represents the network devices, such as gateways, routers, switches, and access points. The fog nodes in this layer can cooperate with one another to offer storage and computing facilities. The last and bottom layer is the device layer which comprises IoT-enabled sensors and devices, including smart ones [19].

The novelty of the proposed model is in the distribution of the access control components, namely the deploying of the PDP and PIP functions in the fog layer to be near to the users which positively helps to reduce the delays as well as improve availability.

The following points illustrate how ABAC main functional points are distributed in our architecture and briefly define the functionality of each component.

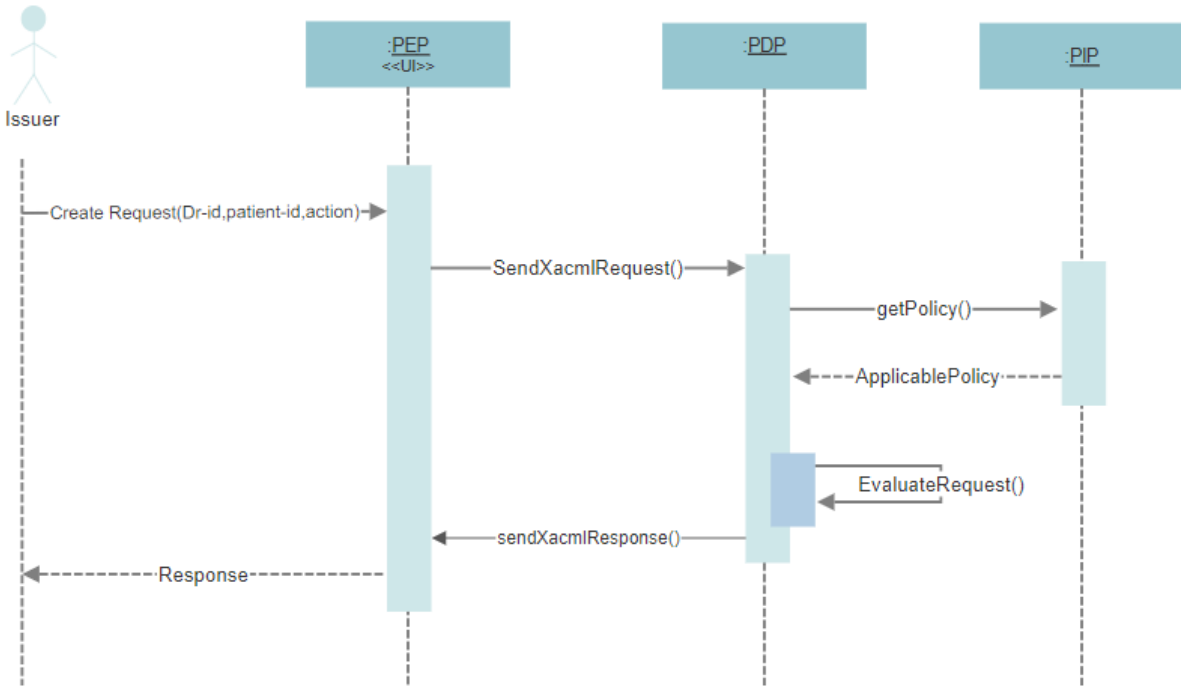


Fig. 2. Components communication.

Cloud Layer. This is the top layer, which includes the PIP and PAP. PIP involves the policies and attributes. PAP is the administration part of the system which is responsible for determining, updating, and modifying the access control policies, rules and attributes through any web service or application.

Fog Layer. This is the middle layer which contains multiple PDPs and PIPs with replicated copies of the PIP in the cloud. PDPs are responsible for evaluating access requests and making access decisions. Replicating PIPs in the fog layer brings part of the cloud functionality to the fog nodes to reduce delay, provide distribution, and minimize cloud server overhead.

Device Layer. This is the lower layer which includes the PEPs that receive access requests from users and enforce access decisions.

As illustrated in Figure **Fig. 1**, in the cloud layer the PAP manages policies and attributes and stores them in the original PIP in the cloud. This PIP replicates policies and attributes at all fog nodes. In the fog layer, the PIPs provide the appropriate attributes and policies to the PDPs to make access decisions. PDPs receive access requests from PEPs. They evaluate access requests based on the appropriate policies and attributes provided by the PIPs in the fog layer. The PEPs then enforce the access decisions made by the PDPs on the requesting IoT devices. Figure **Fig. 2** provides a sequence diagram that illustrates the communication protocols among the main access control functional points in the proposed architecture.

The main goal of having PIPs in the fog layer alongside the PDPs is to decrease the processing time of the decision-making process by reducing the distance between PIPS and PDPs and to reduce potential connection issues between the cloud layer and the fog layer. This, however, inevitably raises new concerns about the synchronization between the different replicas of the PIPs that are discussed below.

3.3 Synchronization between cloud PIP and fog PIP

In the proposed architecture, the cloud regularly synchronizes the set of policies and attributes stored in the cloud PIP with the copies stored in all the PIPs in the fog layer. There are a number of problems that could arise as a result of synchronising the cloud PIP with the fog PIP. The usual consequence is receiving an access request after the fog PIP has been updated, as shown in **Fig. 3**.

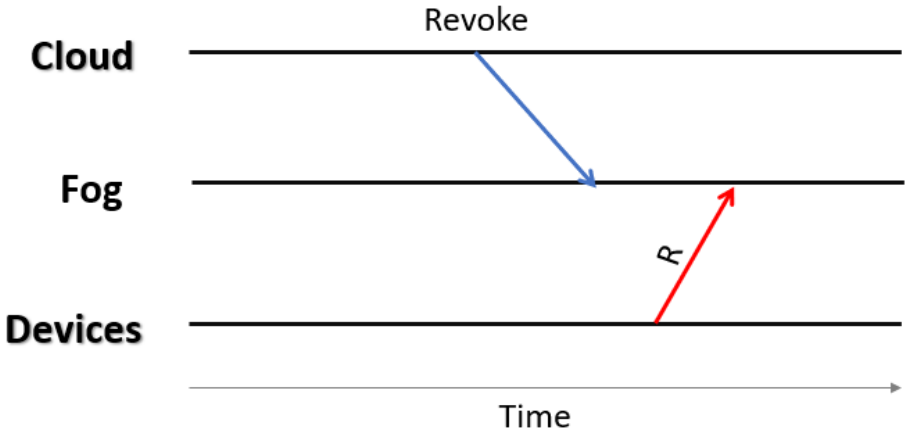


Fig. 3. Out of order reception of an access request after a fog PIP update.

However, the simultaneous reception of policy updates and access requests may bring up some issues. To clarify, the local PIP in the fog may receive an access request simultaneously with a policy update from the original PIP in the cloud as shown in **Fig. 4**.

Besides, concurrent PIP updates (coming from the cloud) and access requests (coming from the device) can be received in the wrong order as shown in **Fig. 5**. To handle this case, we propose to give the highest priority to cloud events. So, for example, if the administrator revokes the privilege of a user in the cloud PIP and sends it to the fog at the same time as the fog receives a concurrent access request from the revoked user, the policy update (cloud event) will have the highest priority so it will be enforced in the fog first, thus leading to the rejection of the user request. Note that requests may be delayed by a malicious user or network latency.

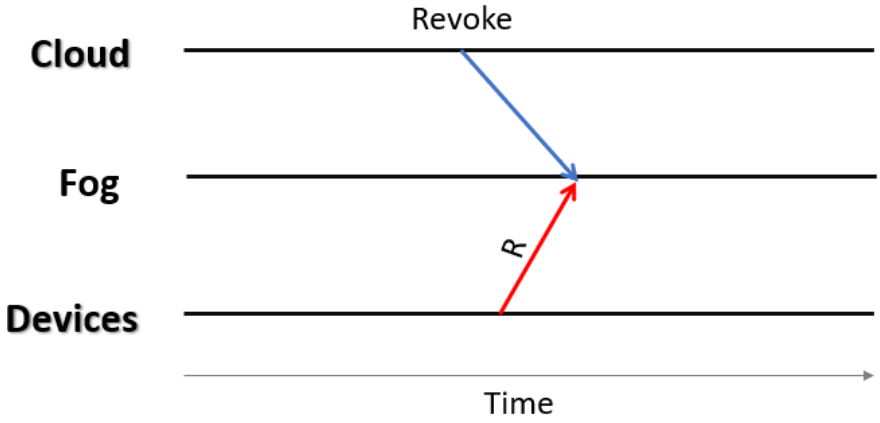


Fig. 4. Simultaneous reception of cloud update and device request.

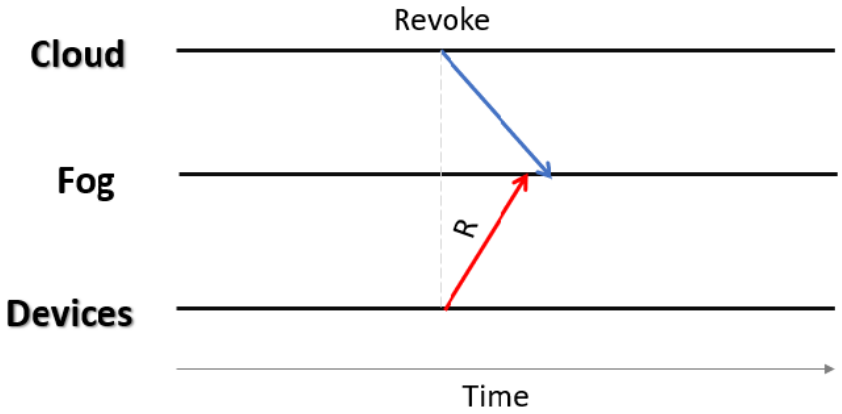


Fig. 5. Concurrent generation of cloud update and device request

Many algorithms can be used to detect the concurrency and ensure causality of events in distributed systems, including the vector clock algorithm. Vector clocks are a more robust extension of Lamport's scalar logical clocks. Vector clocks are generally used to overcome the limitations of scalar logical clocks. Instead of the single scalar timestamp in scalar logical clocks, vector clocks utilize several timestamps (combined to a vector) to grasp the causal dependencies between events in a more comprehensive manner than scalar clocks.

When using a vector clock to determine if two events are concurrent, we make an element-by-element comparison of the corresponding vector. If all elements of vector A are less than or equal to the corresponding element of vector B, A and B are not concurrent and A causally precedes B. If all elements of vector A are greater than or equal to the corresponding element of vector B, A and B are also not concurrent and B causally precedes A. On the other hand, if neither of the two cases above applies and some elements in vector A are greater than while others are less than the corresponding elements in B, then the two events are concurrent [20]. Inspired by the vector clocks algorithm, a more simplified version of vectors is suggested.

Definition 1 [Vector Clock] A synchronization vector is defined as a triple (C, F, R) where C is a counter used to track the cloud events, F is used to track the fog, and R is used to track the requests issued by the devices.

The vector is initially set to (0,0,0) for all processes. Each layer will increase its counter in case of generation or reception of events. Then, applying vector clocks algorithms will allow detecting the causality between events. For instance, consider the events of **Fig. 6**. Cv_1 and Cv_2 represent cloud events. Fv_1, Fv_2 and F_3 are fog events. Dv_1 and Dv_2 reflect the user requests. According to vector clocks, Cv_1 precedes Dv_1, since all Cv_1 counters are less than or equal the corresponding counters in Dv_1.

Similarly, concurrency between operations can be detected by comparing the counters of each vector with the corresponding ones in the other vector. For instance in **Fig. 6**, Dv_2 is concurrent with Cv_2 because the $Dv_2 = (1,2,3)$ and $Cv_2 = (3,4,2)$ cannot be ordered.

To clarify, the cloud and fog counters (1 and 2 respectively) in Dv_2 are less than the cloud and fog counters (3 and 4 respectively) in Cv_2, however, the request counter (3) in Dv_2 is greater than the request counter (2) in Cv_2 which means that Dv_2 and Cv_2 are not ordered. As a result, this requires an action to handle this concurrency.

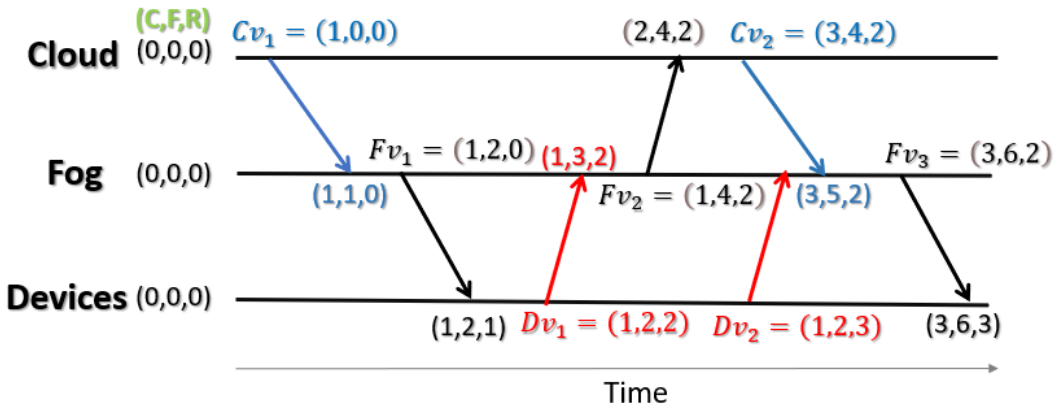


Fig. 6. Synchronization using Vector Clocks.

3.3.1 Improved Vector Clock

Although applying vector clocks allow ensuring the synchronization between the different layers, only two counters are required since the fog will not generate events. Thus we, propose to reduce the size of the vector and use only two counters (C, R), where C allows to track the policy in the cloud and R for device requests.

Definition 2 [Improved Vector Clock] Each event is attached with a couple (C, R) where C tracks the policy version and R tracks the device request.

As shown in Definition 2, the improved vector clock keeps track only of the cloud events and device events. Initially, all counters are set at zero in both Cloud vector $Cv = (0,0)$ and Device vector $Dv = (0,0)$.

Moreover, applying vector clocks naively highly increments the number of exchanged messages since every event in all three layers has to be tracked. To clarify, in the normal vector clocks, each time the fog layer receives an event from the device layer it increases the F counter in the fog vector and forwards it to the cloud layer such as Fv_2 in **Fig. 6**.

To reduce the number of messages being exchanged between the fog and the cloud, and to better detect the concurrency between layers, we suggest using two types of requests: policy updates denoted by U (see Definition 3) and access requests denoted by A (see Definition 4).

Definition 3 [Policy Update] A policy Update $U = (Cv, policy)$ is a request generated by the cloud and forwarded to the fog nodes. Cv is the latest cloud vector and $policy$ is the updated policy.

For example, before the cloud generates the event Cv_2, it gets the value of the device request counter from the fog. Then, the cloud updates its counters accordingly and sends the request (U) to the fog, which contains the latest cloud vector appended to the new policy as represented in **Algorithm 1**.

Definition 4 [Access Request] An access request $A = (Dv, Req)$ is a request generated by the device and forwarded to the fog to check whether the request req is allowed or not. Dv is the latest device vector and req is the requested access.

To clarify, When the fog receives an access request A, this type of request includes the device vector for example Dv_2 appended with a file that contains the access request in the XACML language.

In the case of concurrency detection, only concurrent requests are rolled back. As **Fig. 7** shows, $Cv_2 = (2,1)$ is concurrent to $Dv_2 = (1,2)$ because their counters cannot be organized. As a result, as soon as Cv_2 is received by the fog, all concurrent requests such as Dv_2 are undone.

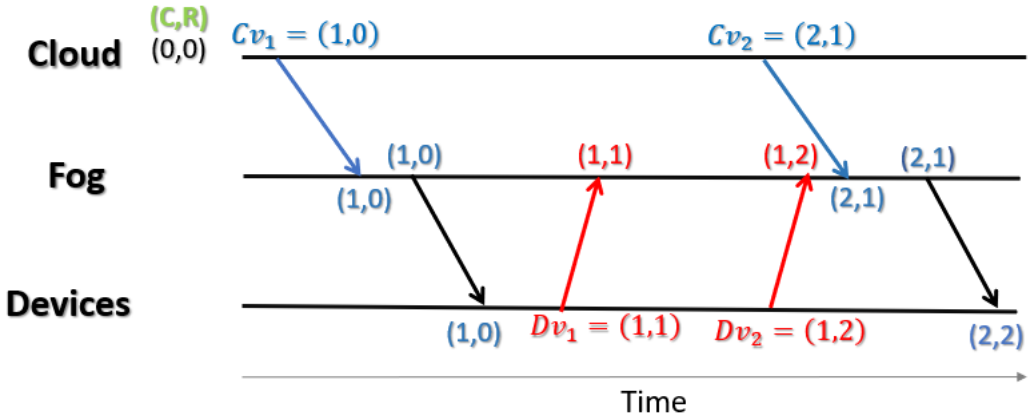


Fig. 7. Improved Vector Clock.

3.4 Algorithm elaboration

Algorithms 1-3 illustrate the functioning of each layer of the proposed model.

Cloud Layer. As shown in algorithm **Algorithm 1**, the cloud vector Cv involves two counter elements, the first to track the policy updates and the second to track the device requests. This vector initially starts at zero for both counters. Then, in case of policy update, PIP calls *Update_CloudVector()* function, to increment the policy counter by one and communicates with the fog to ask for the request counter of the latest device vector Dv (see **Algorithm 1** line **13**). Subsequently, the cloud sends a request
U to the fog layer that includes the current value of Cv along with the updated policy $U = (Cv, policy)$ (see **Algorithm 1** lines **9**,**10**).

Fog Layer. As mentioned earlier, in case of a policy update, the fog forwards the latest device vector Dv to the cloud as a response to the cloud request. Then, once the fog receives an incoming access request (A) from PEP in the device layer, which includes the latest Dv along with the XACML request. Afterward, PDP calls *Concurrent()* to determine if the device event and cloud event are concurrent based on the values of Dv and Cv (see **Algorithm 2** line **25** and **29**).

Algorithm 1. Cloud Layer

```
1: Initialization():
2:     Cv ← (0,0) //cloud vector
3:     Initialize PIP

4: main():
5:     Initialization()
6:     while not aborted do
7:         if there is a policy update (U) then
8:             Update_CloudVector()
9:             PIP sends Cv to fog
10:            PIP sends Updated Policy to fog
11:        end if
12:    end while

13: Update_CloudVector():
14:    Cv.C ← Cv.C+1  // increment the policy counter in Cv
15:    Cv.R ← Get_Request_Counter() // update the value of request counter in Cv

16: Get_Request_Counter():
17:    // Ask fog for latest request
```

If the returned value is True, then the two events are concurrent, and the device request should be reevaluated again after the policy update is performed (see **Algorithm 2** line **16**). Otherwise, the two events are not concurrent, and the request is checked against the applicable policy (see **Algorithm 2** line **18**). Finally, PDP sends the access decision to PEP in the device layer.

Device Layer. Similar to the cloud vector, the device vector consists of two elements, the first to track the policy updates and the second to track the device requests. Initially, both elements start at zero. Once the PEP receives an access request, the PEP gets the request and translates

it from the natural language into the XACML language. Then, PEP calls *Update_Device_vector()* function to increment the value of *R* by one (see **Algorithm 3** line **18**). Next, PEP sends a request *A* to the fog, which includes the translated request alongside the latest device vector $A = (Dv, Request)$. Finally, when PEP receives the response from PDP in the fog, it calls the *Enforce_Access_Decision()* function to enforce the suitable access decision (see **Algorithm 3** line **22**).

4. Experiments and Evaluations

In this part, we introduce the implementation and testing phases. Section 4.1 presents how our solution is implemented, including defining policies using the XACML language, building the access control components, creating an access request, evaluating the request, and receiving a response. Section 4.2 describes a series of experiments conducted using our solution. The objectives of these tests were to find the most appropriate combining algorithm that returns the results in the shortest time and demonstrates that edge-based architecture will positively affect results in terms of reducing the PDP processing time over the cloud-based architecture. Section 4.3 compares our solution with the existing solutions discussed in the related work.

Algorithm 2. Fog Layer

```
1: Initialization():
2:      Initiliaze PDP
3:      Concurrent ← false
4:      Cv ← (0,0)
5:      Dv ← (0,0)

6: main():
7:      Initialization
8:      while not aborted do
9:          if there is a device access request (A) then
10:             Dv ← A.Dv
11:         end if
12:         if there is a policy update (U) then
13:             Cv ← U.Cv
14:         end if
15:         if concurrent(Cv,Dv) then // the vectors are concurrent
16:             // reevaluate the request after the policy update
17:         else // the vectors are not concurrent
18:             PDP evaluates the request based on the local copy
19:         end if
20:         send access decision
21:     end while

22: Concurrent(Cv, Dv): //This function to check the concurrency of cloud event and
device event
23:     greater=false
24:     less=false
25:     if (Cv.C > Dv.C) or (Cv.R> Dv.R) then
26:         greater= true
27:     else
28:         less = true
```

```
29:     end if
30:     if greater and less then
31:             concurrent ← true // the vectors are concurrent
32:     else
33:             concurrent ← false // the vectors are not concurrent
34 :    end if
35 : return concurrent
```

4.1. Implementation

This part provides a detailed explanation for our work. Section 4.1.1 defines the policy language and the XACML elements, and Section 4.1.2 defines some standard policy combing algorithms. Then, Section 4.1.3 illustrates how we built the access control components.

4.1.1. XACML Policy Language

XACML stands for eXtensible Access Control Markup Language and was created by OASIS standard. This language is the most popular policy language that offers two-way communication for sending requests and receiving responses [21]. XACML is considered to be the most suitable policy language to use to perform access control policies in an IoT environment because it has many advantages over other access control policy languages such as standardization, distribution, generalization, powerfulness, and flexibility [22].

There are three top-level policy elements defined by XACML: <Rule>, <Policy> and <PolicySet> [21]. The <Rule> is the basic building block of an XACML policy. It consists of a Boolean expression that can be evaluated to either Deny or Permit. The <Policy> element involves a set of <Rule> elements along with a specified combining algorithm for combining the evaluation results as discussed in section 4.1.3. The <PolicySet> element consists of a set of <Policy> or even <PolicySet> elements and a specified combining algorithm that specify the procedure used to combine the evaluation results.

Algorithm 3. Device Layer

```
1: Initialization():
2:      Initiliaze PEP
3:      Cv ← (0,0)
4:      Dv ← (0,0)

5: main():
6:      Initialization
7:      while not aborted do
8:              Receive_Access_Request()
9:              Update_Device_Vector()
10:             PEP send request and Dv to PDP in the fog
11:             if respons isreceived from the fog then
12:                     Enforce_Access_Decision()
13:             end if
14:     end while

15: Receive_Access_Request():
16:     PEP gets the request
```

```
17:     PEP tanslates the request into XACML

18: Update_Device_Vector():
19:     //increment the request counter of deveice vector
20:     Dv.R ← Dv.R+ 1
21:     Dv.C ← Get_Policy_Counter()

22: Enforce_Access_Decision():
23:     if decision = Permit then
24:         PEP allows device access
25:     else
26:         PEP prevents device access
27:     end if

28: Get_Policy_Counter()
29:     get the policy counter from rCveceived from the fog
```

4.1.2. Combining Algorithms

OASIS standard defines a set of combining algorithms that can be identified either through a RuleCombiningAlgId attribute for the <Policy> element or by PolicyCombiningAlgId attribute for <PolicySet> elements.

The rule-combining algorithm specifies a method to arrive at the authorization decision given the individual evaluation results of a combination of rules. Likewise, the policy combining algorithm specifies a procedure to arrive at the access decision based on the individual evaluation results of a group of policies.

XACML defines four standard combining algorithms including:

- Deny-overrides: according to this algorithm, when any single <Rule> or <Policy> element is evaluated to "Deny", the access decision is Deny, while ignoring the result of evaluating the other <Policy> or <Rule> elements in this policy.
- Permit-overrides: this algorithm is similar to Deny-overrides. The combined authorization result is "Permit" if only a single "Permit" result is encountered.
- First-applicable: the evaluation result of the first applicable <Rule>, <Policy> or <PolicySet> will be the combined authorization decision.
- Only-one-applicable: this algorithm is a policy-combining algorithm to ensure that only one policy or policy set is applicable [21].

4.1.3. Building ABAC Engine

The first stage validating our proposition is to build an ABAC Engine that presents a smooth communication path between the previously mentioned access control components (PIP, PDP, and PEP). In this step, the PEP takes the request from the user and passes it to the PDP, which evaluates this request based on the appropriate policy and returns the response to the PEP to present it to the user.

To perform this, we have used Java over Eclipse IDE combined with the WSO2 BALANA library which is an open-source implementation of the XACML based on the sunxacml [12].

The PEP component proceeds as follows:

- Receive requests from the user as represented in **Fig. 7**.
- Create the request in XACML language through the Create Request function, as illustrated in **Fig. 8**, and send it to PDP.
- Receive the response from PDP and presents it to the user, as shown in **Fig. 9**.

Fig. 7. User request example.

```
public static String createXACMLRequest(String Dr_id, String P_id, String
    action) throws IOException{
return"<Request xmlns=\"urn:oasis:names:tc:xacml:3.0:core:schema:wd-17\"
    CombinedDecision=\"false\" ReturnPolicyIdList=\"false\">\n" +
                "<Attributes Category=\"urn:oasis:names:tc:xacml:3.0:
    attribute-category:action\">\n" +
                "<Attribute AttributeId=\"urn:oasis:names:tc:xacml:1.0:action
    :action-id\" IncludeInResult=\"false\">\n" +
                "<AttributeValue DataType=\"http://www.w3.org/2001/XMLSchema#
    string\">" + action + "</AttributeValue>\n" +
                "</Attribute>\n" +
                "</Attributes>\n" +
                "<Attributes Category=\"urn:oasis:names:tc:xacml:1.0:subject-
    category:access-subject\">\n" +
                "<Attribute AttributeId=\"urn:oasis:names:tc:xacml:1.0:
    subject:subject-id\" IncludeInResult=\"false\">\n" +
                "<AttributeValue DataType=\"http://www.w3.org/2001/XMLSchema#
    string\">" + Dr_id +"</AttributeValue>\n" +
                "</Attribute>\n" +
                "</Attributes>\n" +
                "<Attributes Category=\"urn:oasis:names:tc:xacml:3.0:
    attribute-category:resource\">\n" +
                "<Attribute AttributeId=\"urn:oasis:names:tc:xacml:1.0:
    resource:resource-id\" IncludeInResult=\"false\">\n" +
                "<AttributeValue DataType=\"http://www.w3.org/2001/XMLSchema#
    string\">" + P_id + "</AttributeValue>\n" +
                "</Attribute>\n" +
                "</Attributes>\n" +
                "</Request>";  }
```

Fig. 8. Request creation snippet.

The PDP component proceeds as follows:

- Initialise BALANA instance and sets the policy.
- Receive the request from PEP.
- Evaluate the request based on XACML policy.
- Send the response to PEP.

Fig. 10 shows the main part of PDP code.

```
Enter Doctor Id :
Dr1
Enter Patient Id :
1
Enter action (view - edit):
edit


 ======================== XACML Request ====================
<Request xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17" CombinedDecision="false" ReturnPolicyIdList="fals
<Attributes Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action">
<Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id" IncludeInResult="false">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">edit</AttributeValue>
</Attribute>
</Attributes>
<Attributes Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject">
<Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id" IncludeInResult="false">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Dr1</AttributeValue>
</Attribute>
</Attributes>
<Attributes Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource">
<Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id" IncludeInResult="false">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">1</AttributeValue>
</Attribute>
</Attributes>
</Request>
=============================================================

 ======================== XACML Response ===================
<?xml version="1.0" encoding="UTF-8" standalone="no"?><Response xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:w
=============================================================

Dr1 is authorized to edit  Patient 1 data
```

Fig. 9. Response evaluation snippet.

```
1  public static void pdpjob(String request) throws IOException, SAXException,
     ParserConfigurationException, TransformerException {
2    initBalana();
3   pdp = new PDP(balana.getPdpConfig());
4    System.out.println("\n ======= XACML Request =========");
5    System.out.println(request);
6    System.out.println("======================================");
7    String response = pdp.evaluate(request);
8    strToXML(response,"../response/response.xml");
9   }
```

Fig. 10. PDP job.

4.2. Testing

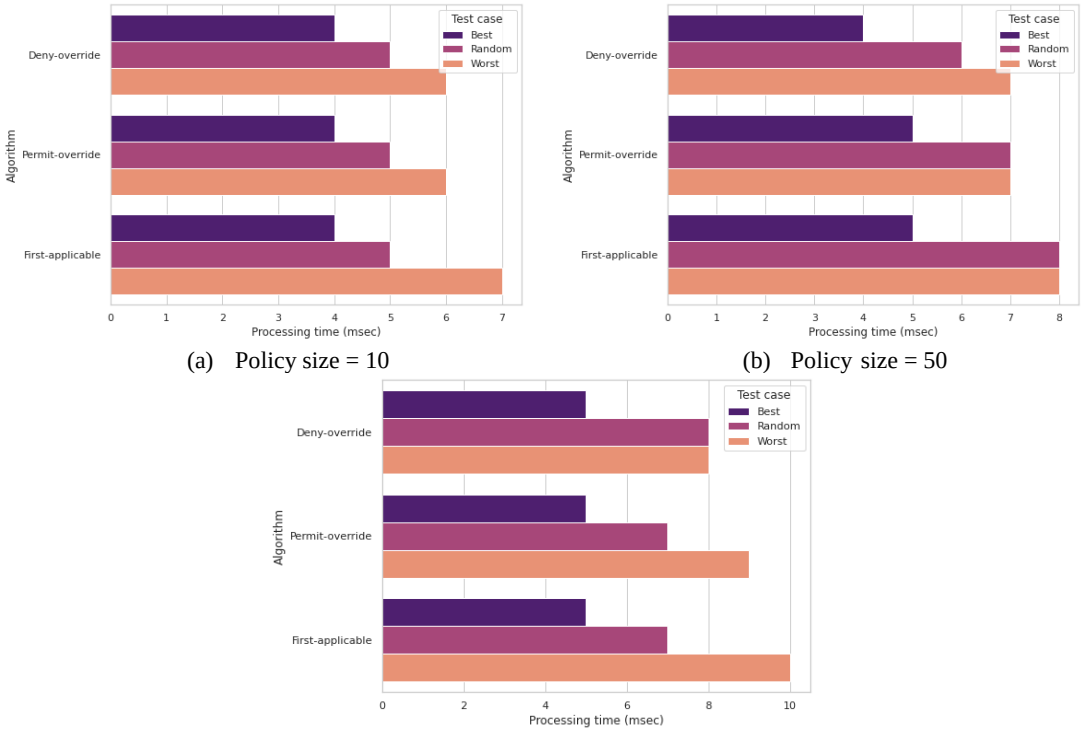
As discussed previously, one of our main goals is to reduce latency and provide the access decision in a fast manner. This part presents our testing and evaluation phase. Section 4.2.1 explains the testing environment and setup. Section 4.2.2 describes the first stage, which is to find the best combining algorithm that provides the lowest processing time. Then, in section 4.2.3, we test if distributing PDP components in the fog layer rather than the cloud layer will provide the lower latency.

4.2.1. Testing Environment and Setup

The experiments conducted in this study aimed to test whether our algorithm reduced latency and provided a fast response. A TP300L ASUS laptop running on a Linux operating system with Intel Core i5-5200U processor and 20 GB memory was used in this experiment. We used EdgeCloudSim to simulate and validate our solution and compare the processing time of the PDP in the edge layer and the cloud layer. EdgeCloudSim is an open-source simulator that targets the Edge Computing environments based on CloudSim [23]. Configurability is one of the main advantages of EdgeCloudSim since it uses three configuration files that can be tuned to reflect user needs. Using these files has a positive effect on reducing the programming effort. The first file includes the simulation parameters, such as the warm-up period, the simulation time, and the number of mobile devices used in the simulation scenario. The second file is the application file which contains the XML specification of the applications that will generate the tasks in the simulation, including the task length, the upload/download data size, and the active/idle period of the applications. The third file is an XML file that defines the edge server characteristics such as the specification of the edge devices and location of the edge access points [23].

4.2.2. Finding the best Combining Algorithm

In this section, we will test which combining algorithm behaves better than the others in terms of returning the access response in the shortest processing time. To achieve this, we followed these steps: creating 10 random policies with 10 different sizes, then sending 10 random access requests, and calculating the average processing time to evaluate all these requests. These steps were repeated for three combing algorithms, including Deny-overrides, Permit-overrides, and Firstapplicable. Our results demonstrated that, as the policy size increases, the Deny-overrides algorithm shows the lowest processing time. **Fig. 11** and **Fig. 12** illustrate how the three algorithms behave in three policy sizes.



(a) Policy size = 10 (b) Policy size = 50

(c) Policy size = 100

Fig. 11. The processing time vs policy size.

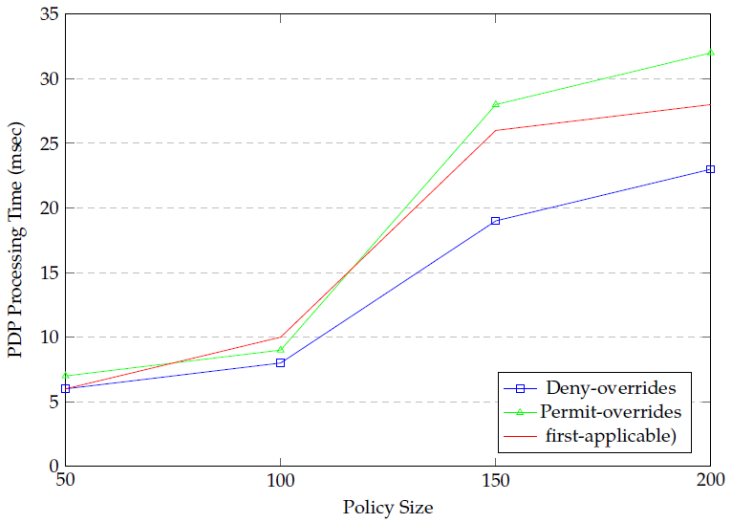


Fig. 12. PDP processing time in different policy sizes.

4.2.3. Comparing Edge and Cloud Architectures

The deployment of our solution in EdgeCloudSim showed that superior results are achieved with our method. As can be seen from **Fig. 13**, the PDP processing time in the edge layer provided evidence that deploying the PDP in the edge layer leads to lower latency than deploying the PDP in the cloud layer. Also, as shown in **Fig. 14**, as the number of devices increases, the delay slightly increases, which means that our solution is not heavy and will not affect the network when the number of devices increases.

We concluded that the Deny-overrides algorithm is the most appropriate combing algorithm to use. We also noted that increasing the policy sizes affects the PDP processing time. Overall, the simulation provided evidence that our method was the one that obtained the most robust results since it provides access responses in a very short time when compared with the cloud.

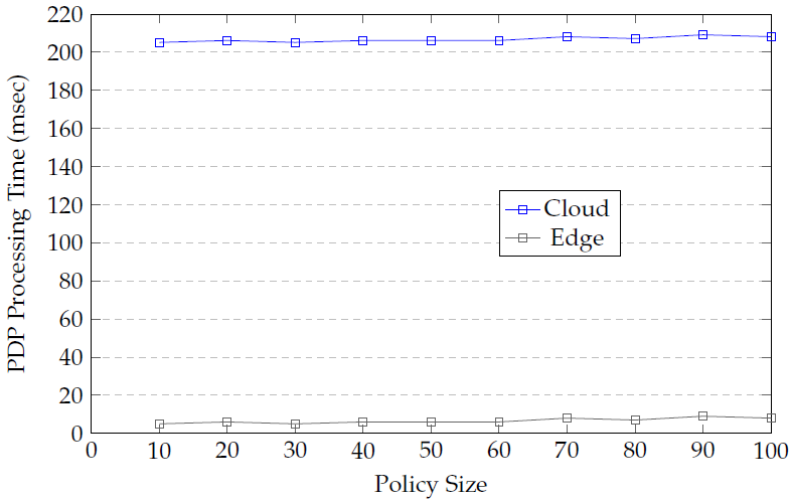


Fig. 13. PDP processing time in cloud and edge for the best combining algorithm (Deny-overrides algorithm).

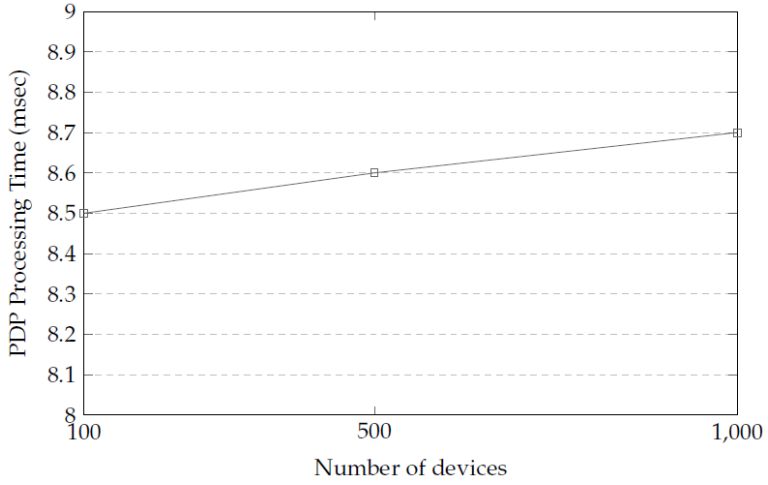


Fig. 14. PDP processing time in the edge for different device number.

4.3. Comparison with the related work

To compare our work with the existing solutions discussed in the related work section, we used the requirements mentioned in Section 1. As it can be seen from **Table 1**, [14], [15], [16], and [17] achieve all the requirements, except decentralization and availability. However, our model fulfills all requirements, including decentralization and availability. This is because our model distributes the access control components, the PDP and PIP functions, via deploying them in the fog layer near the end-users. This distribution reduces latency and improves availability.

Table 1. Comparing the proposed model with the related work

Access Control Requiremnts	[14]	[15]	[16]	[17]	**Our Model**
Dynamicity	✓	✓	✓	✓	✓
Fine granularity	✓	✓	✓	✓	✓
Scalability	✓	✓	✓	✓	✓
Revocability	✓	✓	✓	✓	✓
Decentralisation					✓
Availability					✓
Low latency	✓	✓	✓	✓	✓
Efficiency	✓	✓	✓	✓	✓
Flexibility	✓	✓	✓	✓	✓
Standardisation	✓	✓	✓	✓	✓

5. Conclusion

In this paper, we proposed a new distribution of the ABAC model over fog computing along with the appropriate synchronization mechanism to correctly enforce policy decisions in case of out-of-order policy updates and user requests. This introduces numerous benefits, namely, bringing the policy decision operations to the edges closer to the devices layer which minimizes latency and increasing availability. As a result, end nodes do not have to wait for the access decision from the remote data center but can quickly get access from the closest edges. To validate our solution, we implemented an access control engine based on ABAC. Our engine performs the access control process based on the Balana library. We simulated our model using EdgeCloudSim to test whether distributing the access control components in a fog architecture reduces the latency and provides distribution when compared with cloud architecture. The results demonstrated that fog-based architecture reduces the latency by 95.7 % when compared with cloud-based architecture. Future research, motivated by the promising results obtained by this study, could focus on validating the model through real-life implementation.

Acknowledgment

This project was funded by the Deanship of Scientific Research (DSR) at King Abdulaziz University, Jeddah, under the grant No. (DG-9-612-1441). The authors, therefore, gratefully acknowledge the DSR technical and financial support.

References

[1] S. Alnefaie, A. Cherif, and S. Alshehri, "Towards a Distributed Access Control Model for IoT in Healthcare," in *Proc. of 2019 2nd International Conference on Computer Applications Information Security (ICCAIS)*, pp. 1–6, 2019. Article (CrossRef Link).

[2] C. S. & V. F. Emmanuel Bertin Dina Hussein, "Access control in the Internet of Things: a survey of existing approaches and open research questions," *Ann. Telecommun.*, vol. 74, pp. 375–388, 2019. Article (CrossRef Link).

[3] S. Ravidas, A. Lekidis, F. Paci, and N. Zannone, "Access control in Internet-of-Things: A survey," *J. Netw. Comput. Appl.*, vol. 144, pp. 79–101, 2019. Article (CrossRef Link).

[4] A. Ouaddah, H. Mousannif, A. Abou Elkalam, and A. Ait Ouahman, "Access control in the Internet of Things: Big challenges and new opportunities," *Comput. Netw.*, vol. 112, pp. 237–262, Jan. 2017. Article (CrossRef Link).

[5] S. Pal, M. Hitchens, V. Varadharajan, and T. Rabehaja, "Fine-Grained Access Control for Smart Healthcare Systems in the Internet of Things," *EAI Endorsed Trans. Ind. Netw. Intell. Syst.*, vol. 4, no. 13, p. 154370, Mar. 2018. Article (CrossRef Link).

[6] D. Hussein, E. Bertin, and V. Frey, "A Community-Driven Access Control Approach in Distributed IoT Environments," *IEEE Commun. Mag.*, vol. 55, no. 3, pp. 146–153, Mar. 2017. Article (CrossRef Link).

[7] V. C. Hu et al., "Guide to Attribute Based Access Control (ABAC) Definition and Considerations," *National Institute of Standards and Technology*, NIST SP 800-162, Jan. 2014. Article (CrossRef Link).

[8] M. A. Aleisa, A. Abuhussein, and F. T. Sheldon, "Access Control in Fog Computing: Challenges and Research Agenda," *IEEE Access*, vol. 8, pp. 83986–83999, 2020. Article (CrossRef Link).

[9] I. Martinez, A. S. Hafid, and A. Jarray, "Design, Resource Management and Evaluation of Fog Computing Systems: A Survey," *IEEE Internet Things J.*, vol. 8, no. 4, pp. 2494-2516, 2021. Article (CrossRef Link).

[10] F. A. Kraemer, A. E. Braten, N. Tamkittikhun, and D. Palma, "Fog Computing in Healthcare–A Review and Discussion," *IEEE Access*, vol. 5, pp. 9206–9222, 2017. Article (CrossRef Link).

[11] M. Maksimović, "Implementation of Fog computing in IoT-based healthcare system," *JITA - J. Inf. Technol. Appl. Banja Luka - APEIRON*, vol. 14, no. 2, Jan. 2018. Article (CrossRef Link).

[12] WSO2, "WSO2 Balana Implementation," 2021. Article (CrossRef Link).

[13] S. Alnefaie, S. Alshehri, and A. Cherif, "A survey on access control in IoT: models, architectures and research opportunities," *Int. J. Secur. Netw.*, vol. 16, 2021. Article (CrossRef Link).

[14] I. Ray, B. Alangot, S. Nair, and K. Achuthan, "Using Attribute-Based Access Control for Remote Healthcare Monitoring," in *Proc. of 2017 Fourth International Conference on Software Defined Systems (SDS)*, Valencia, Spain, pp. 137–142, May 2017. Article (CrossRef Link).

[15] S. Salonikias, I. Mavridis, and D. Gritzalis, "Access Control Issues in Utilizing Fog Computing for Transport Infrastructure," in *Proc. of Critical Information Infrastructures Security*, vol. 9578, E. Rome, M. Theocharidou, and S. Wolthusen, Eds. Cham: Springer International Publishing, pp. 15–26, 2016. Article (CrossRef Link).

[16] S. Salonikias, A. Gouglidis, I. Mavridis, and D. Gritzalis, "Access Control in Industrial Internet of Things," in *Proc. of Security and Privacy Trends in the Industrial Internet of Things*, Springer, pp. 95-114, 2018. Article (CrossRef Link).

[17] L. A. Charaf, I. Alihamidi, A. Deroussi, M. Saber, A. Ait Madi and A. Addaim, "Proposed Access Control Architecture Based on Fog Computing for IoT Environments," in *Proc. of the 7th International Conference on Optimization and Applications (ICOA)*, IEEE, 2021. Article (CrossRef Link).

[18] S. Sakr and A. Y. Zomaya, Eds., "Attribute-Based Access Control (ABAC)," *Encyclopedia of Big Data Technologies*, Cham: Springer International Publishing, pp. 117–117, 2019,. Article (CrossRef Link).

[19] M. Mukherjee *et al.*, "Security and Privacy in Fog Computing: Challenges," *IEEE Access*, vol. 5, pp. 19293–19304, 2017. Article (CrossRef Link).

[20] T. Landes, "Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications," in *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, vol. 1, Las Vegas, Nevada, pp. 31-37, 2006. Article (CrossRef Link).
[21] eXtensible Access Control Markup Language (XACML) Version 3.0, XACML-V3.0. 2013. Article (CrossRef Link).
[22] H. Atlam, M. Alassafi, A. Alenezi, R. Walters, and G. Wills, "XACML for Building Access Control Policies in Internet of Things," in *Proc. of the 3rd International Conference on Internet of Things, Big Data and Security*, pp. 253-260, 2018. Article (CrossRef Link).
[23] C. Sonmez, A. Ozgovde, and C. Ersoy, "EdgeCloudSim: An environment for performance evaluation of edge computing systems," in *Proc. of 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, Valencia, Spain, pp. 39-44, 2017. Article (CrossRef Link).

Seham Alnefaie received her M.S. in Network Security at Department of Information Technology in the Faculty of Computing and Information Technology at King Abdulaziz University (Saudi Arabia) in 2020. She received her Bachelor's degree in Information Technology from King Abdulaziz University in 2016. Her main research interests include access control models, Internet of Things and fog computing.

Asma Cherif received MS. and Ph.D. degrees in computer science from Lorraine University (France) in 2008 and 2012 respectively. She conducted her research in the French research laboratory Loria. She is currently associate professor in the Faculty of Computing and Information Technology at King Abdulaziz University (Saudi Arabia). Her current research interests include access control in distributed systems and collaborative applications, cloud/edge computing, smart systems and Internet of Things.

Suhair Alshehri received the Ph.D. degree in Computing and Information Sciences from Golisano College of Computing and Information Sciences, Rochester Institute of Technology, in 2014. She is currently an Assistant Professor with the Information Technology Department, Faculty of Computing and Information Technology, King Abdulaziz University. Her main research interests include security and privacy in computer and information systems and applied cryptography, with an emphasis on the development of models for secure distributed access, and the viability of these models for healthcare and health information exchanges in the Internet of Things and distributed environments.