Journal of Ubiquitous Systems & Pervasive Networks
Volume 14, No. 2 (2021) pp. 01-10

iXen: Secure Service Oriented Architecture and Context Information Management in the Cloud

Euripides G.M. Petrakis*, Xenofon Koundourakis

School of Electrical and Computer Engineering, Technical University of Crete (TUC), Chania, Crete, Greece

Abstract

iXen's ambition is to overcome the limits of existing IoT platforms in the cloud and deal with challenges of security and interoperability. Therefore, iXen is interoperable and expandable (i.e. services can be added or removed) while being secure by design: access to services is granted only to authorized users (or other services) based on user roles and access policies. Leveraging principles of Service Oriented Architectures (SOA) and the most recent EU standards for context information management, iXen is implemented as a composition of RESTful micro-services in the cloud. iXen adopts a 3-tier architecture design model. The first layer supports connectivity of the vast diversity of IoT devices with the cloud. The second (middle) layer implements IoT data functionality including, database, security and context management services allowing devices to publish information and, users (or other services) subscribed to devices to get notified about the availability of this information. Flow-based programming services in the middle layer allow fast development of new applications by wiring together IoT devices and services. The third layer makes IoT applications available to customers based on subscriptions. The experimental analysis shows that iXen is responding in real-time to complex service requests under heavy workloads.

Keywords: *Cloud, IoT, Service Oriented Architecture, Micro-Services, Context Management, Flow-Based Programming*

1. Introduction

Cloud is the ideal environment for IoT applications deployment due to reasons related to its affordability (no up-front investment, low operation costs), scalability, easy maintenance and accessibility. Cloud platforms facilitating IoT application development are known to exist and many are available as commercial products [1]. These are highly configurable solutions and capable of making strong commitments (by means of SLAs) for meeting the needs of Quality of Service (QoS) critical applications. However, these solutions are fully proprietary and vendor specific and as such, they do not support interoperability with third party systems and services. Research should go beyond these limits and towards more open, secure and re-configurable IoT platforms.

Securing IoT infrastructures is a challenging task. Potential risks and counter-measures for dealing with security have been identified by the industry, by regulatory entities and the literature [2]. The principles of Security by Design and, Security and Privacy by Default [3] must be applied since the design phase of a system. The cloud infrastructure is exposed to risks due unauthorized attempts to access services. These are handled successfully with the aid of traditional security methods (e.g. encryption, authorization, auditing). However, an IoT system is also vulnerable due to malicious devices operating at the edge of the network.

The security mechanisms for the cloud must be complemented with trust evaluation methods for dealing with these risks [4]. This creates new challenges for dealing with the cause and point of system failure if security fails [5]. The Industrial Internet Consortium (IIC) [6] emphasizes the need for monitoring devices, networks, applications and the cloud. Solutions to malicious behavior or malfunction detection, suggest continuous monitoring of, the state of IoT nodes, of the cloud components or, periodically monitoring system logs or, all of the above. The General Data Protection Regulation (GDPR)[1] of the EU has a significant impact on IoT systems design. Data protection is also crucial, as potential intrusion may not only lead to vulnerable personal data theft but may risk system operation overall.

iXen focuses on securing the cloud infrastructure from unauthorized access to services and data. Securing the IoT network is outside the scope of this paper. Building upon our previous work on IoT architecture design and implementation [2][7] based on Service Oriented Architectures (SOA) [12] and cloud micro-services[2], iXen's ambition is to overcome the limits of existing IoT architectures in the cloud and deal with challenges of security, openness and interoperability. iXen architecture is highly configurable and modular and supports generation of fully customizable applications by re-using

[1] https://eugdpr.org

[2] https://aws.amazon.com/microservices/

* Corresponding author. Tel.: +302821037229
Fax: +302821037542; E-mail: petrakis@intelligence.tuc.gr

DOI: 10.5383/JUSPN.14.02.001

services and devices. Leveraging flow-based programming [8], new applications can be generated with the aid of user-friendly interfaces. The interest of a developer in sensors and services for composing a new application is expressed by means of queries specifying the desired device and service properties. iXen services are re-usable, implement fundamental functionality and offer a public interface allowing secure connections with other services (even third-party ones). Therefore, iXen is interoperable and expandable (i.e. services can be added or removed) while being secure by design: all services are protected by an *OAuth2.0*[3] mechanism. Access to services is granted only to authorized users (or authorized services) based on user roles and access policies. This mechanism is realized as a synthesis of security micro-services which are both, generic and re-usable (i.e. the same mechanism is applied for securing all services offered by the platform).

iXen features an elaborate 3-tier architecture design model. Each layer implements functionality addressing the needs of different users' type, namely infrastructure owners (i.e. device owners), application owners (i.e. applications developers) and customers (who subscribe to applications). The same user may have more than one role in iXen. Infrastructure owners are entitled to install and make devices available to application owners which, in turn, subscribe to devices in order to create applications; finally, customers (i.e. end-users) subscribe to applications.

The first level of the architecture allows devices to connect to iXen in the cloud. Captured data from devices are encrypted and streamed to the cloud. iXen is capable of handling large collections of devices of any type. This is the only part of the system which is affected by the property of a device (e.g. a sensor) to use a specific IoT protocol (e.g. Bluetooth, Zigbee). The rest of the system is sensor agnostic (i.e. data are processed in JSON which is a sensor agnostic format). The second (middle) layer implements advanced data processing functionality including, database, security, flow-based programming for creating applications and, event-driven publish-subscribe (i.e. context) services allowing devices to publish information and users (or other services) to be notified when this information becomes available (i.e. only subscribed users or services get notified). The third layer makes applications available to customers based on subscriptions. Devices and applications are easily discoverable by means of user-friendly query mechanisms (a feature which is of particular interest for large scale IoT systems).

Fig. 1 illustrates an example 3-tier system structure, the physical entities and their interaction: (a) four devices in layer 1 connect to gateways and from there to the cloud. The application owner in layer 2 makes three applications available to customers in layer 3. The customer in layer 3 subscribes to one application. Leveraging this 3-layer design, iXen is ready to incorporate a business logic (e.g. billing policies) for different types of users and become self-sustainable. All users may benefit from their participation in iXen based on their offerings or be charged based on a pay-per-use cost model or, based on their subscriptions to cloud services (left as future work).

iXen is a research prototype and as such, it is not intended to compete with existing commercial platforms in terms of services offering or performance, but rather, to show how a cost effective and self-sustainable IoT eco-system can be designed based on principles of SOA design and cloud micro-services, using well established, open-source technologies. iXen design relies on the most recent EU standards[4] for context information management and IoT systems design. iXen prototype is implemented in OpenStack and Fiware[5], the open-source distributed cloud infrastructure of the EU.

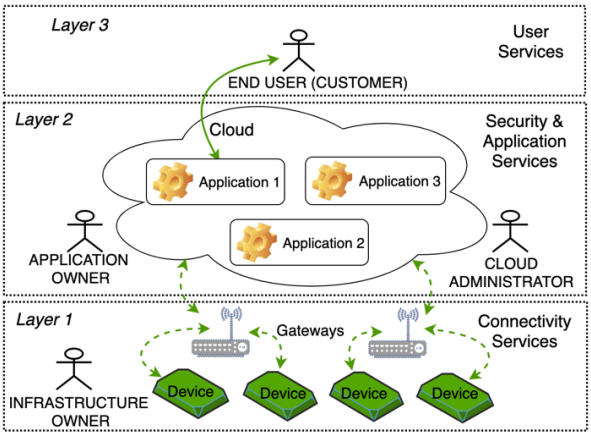

Fig. 1. iXen 3-tier design.

iXen is application agnostic allowing a wide range of IoT devices (e.g. sensors and gateways) to connect to the cloud. iXen is implemented as a composition of modular cloud micro-services implementing fundamental functionalities and communicating with each other using REST [11]. More services can be added on demand or, any service can be replaced or moved to a different Virtual Machine (VM) (on the same or on a different cloud) with minimum overhead (i.e. only the IP of the service will change).

In regards to similar work, iXen concept resembles DIAT model [10], an IoT architecture for addressing the challenges of device and service interoperability, extendibility and security. DIAT layered architecture comprises of three layers (referred to as Virtual Object, Composite Virtual Object and Service layer). Similar to iXen and SOA, DIAT decouples services into independent modules and binds similar functionalities together in different layers. DIAT applies a powerful yet very expensive (in terms of complexity and speed) security policy and defines a new model for expressing users privacy and authorization using ontologies. In contrast, iXen is opted for the state-of-art-approach for expressing role-based user authorization based on XAMC[6]. DIAT is positively evaluated as a model but it is neither a cloud nor a Service-Oriented Architecture. It is not accompanied by implementation and its performance has not been assessed in a real setting. In contrast, iXen, relies on a Service Oriented Architecture in order to handle the issues of modularity, extendibility, re-usability and security.

We run an exhaustive set of experiments using real and simulated (but realistic) data aiming to evaluate both, iXen response time and scalability. We stressed iXen with high data streams and many simultaneous requests. The experimental analysis shows that iXen is capable of responding in real-time under heavy workloads (i.e. many users applying several requests per second).

The rest of this paper is structured as follows: issues related to iXen design and implementation are discussed in Sec. 2 followed by an analysis of performance in Sec. 3. Conclusions and issues for future work are discussed in Sec. 4.

[3] https://oauth.net/2/

[4] https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/EU+Standards

[5] https://www.fiware.org

[6] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

2. Design and Architecture

iXen is a novel Future Internet (FI) cloud service for data collection from IoT devices in an automatic, generalized and modular way. Building upon principles of Service Oriented Architectures (SOA) design [12] and driven by the key requirements of today's IoT systems for adaptability, low-cost and scalability, iXen's architecture is modular and expandable. iXen encompasses IoT-A [13][14] design principles in an attempt to develop an innovative IoT platform that supports generic services and IoT devices (i.e. independent of connectivity and not coupled to specific IoT protocols). IoT-A proposes an Architecture Reference Model (ARM) defining the principles and guidelines for generating IoT architectures, providing the means to connect vertically closed systems in the communication layer (i.e. where IoT devices interact with the system) and service layer (i.e. where services are provided). IoT-A and FIWARE compliant architectures may assure that generated knowledge will be modular and reusable across domain or use-case specific boundaries.

We followed a state-of-the-art design approach [16] that identified functional and non-functional system requirements and specifically, (a) functional components and their interaction, (b) information that is managed and how it is acquired, transmitted, stored and analyzed, (c) different types of users and how they interact with the system, (d) requirements for assuring data, network and user security and privacy. Detail on system design (including a full set of use case, activity and deployment UML diagrams) can be found in [15].

2.1. User Groups

Each user belongs to a user class. Each user class is assigned a role encoding authorization to access other services. iXen users have an identifier, an email and a name being displayed. The following user groups and functional requirements associated with each group are identified:

System administrators: they configure, maintain and monitor the cloud. Except their competence to providing cloud services, they are responsible for performing Create, Read, Update, Delete (CRUD) operations on (a) users (e.g. they can register new users to the system and define their access rights) and, (b) devices (e.g. they can register new devices to the system). They are responsible for monitoring system operations at all times (e.g. monitoring users' activities).

Infrastructure owners: they subscribe to the cloud for a fee and are granted permission (by the cloud administrator) to register, configure, monitor or remove devices in their possession. iXen provides functionality for connecting and controlling devices over the Web.

Application owners: they subscribe to the cloud and to a set of devices for a fee. Once subscribed to devices they can create applications by means of flow-based programming. iXen provides query mechanisms for selecting devices of interest using device properties such as, device type, location, purpose etc. An application is defined by wiring together the outputs of selected devices.

Customers: they subscribe to applications for a fee. Once subscribed to an application they are granted access to the application over the Web. iXen provides query mechanisms for selecting applications available for subscriptions based on criteria such as, location, functionality etc. Customers are granted only access right to applications. They issue requests to application owners (e.g. for reporting events, issuing requests for assistance).

2.1.1. Information (Class) Diagram

The class diagram describes the structure of the main entities (classes) of the application domain (i.e. the system), their properties, operations allowed on these entities and their interrelationships. Typically, it is represented as a class (IS_A) hierarchy with the most general classes at the top and more specific classes (i.e. specializations of general classes) lower in the hierarchy. Other types of relationships between classes (referred to as "object properties") can be also defined (e.g. "part of", "have" property relationships). A class is also described by a set of attributes (referred to as "data properties") together with a set of operations that can be executed on entities of this class. Entities lower in a class hierarchy inherit all properties and operations of entities higher in the hierarchy. Fig. 2 is the class diagram of iXen in UML. There are different types (classes) of users that interact with the system. Each type of user is associated with different services that handle different types (classes) of information:

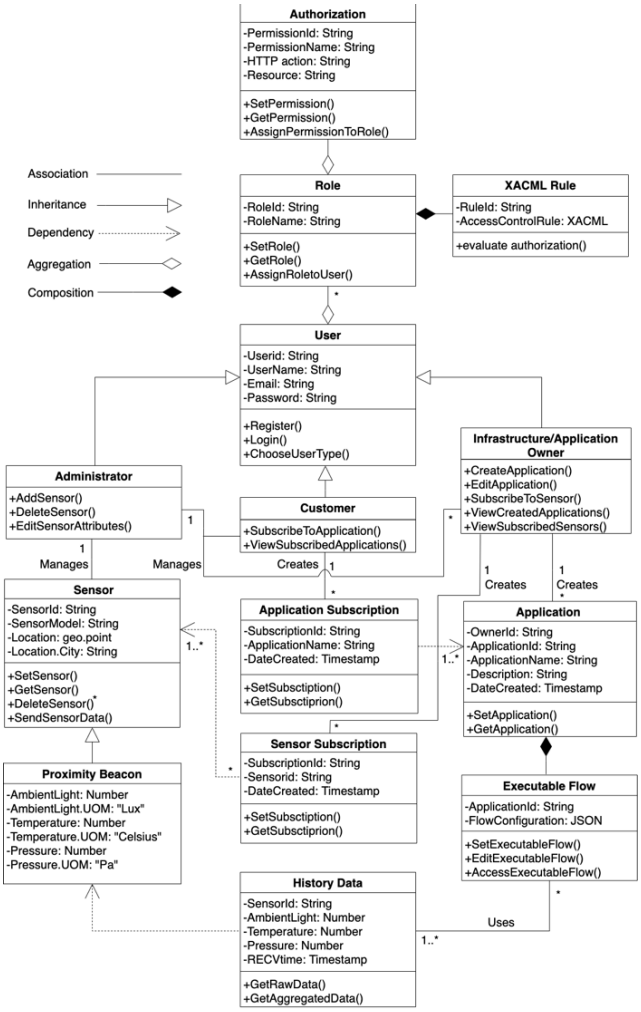

Fig. 2. Class Diagram.

User Class: describes users interacting with the system. Each user is described by an identifier (ID), email, and a name

being displayed. The class is specialized in three user sub-categories (sub-classes) namely, administrator, customer and infrastructure/application owners. Each user is assigned a role (by the administrator) at login. Each user class corresponds to a role (represented by a class). A user may have more than one role (e.g. can be both customer and application owner at the same time).

Authorization class: represents the permission of end-uses (customers) to access the applications they have subscribed. Methods of this class relate to granting authorization (e.g. access to services) to users and to assigning them a role. All users have a role and access rights which correspond to their role (i.e. a customer can create subscriptions to application but is not authorized to add sensors. A snapshot of this class describes a service request that a user is authorized to execute (e.g. for HTTP request GET: "http://host/example").

Role class: the system automatically assigns a role to a user when he/she subscribes to an application in order to obtain permission to access its endpoint. Methods of this class relate to creating and assigning a role to users. A snapshot of this class corresponds to an authorization, so when a user is assigned a role she/he becomes the holder of the corresponding authorization.

XACML Rule class: represents an access control policy which defines the access rights of a user. A snapshot of this class is an XACML[7] file (i.e. a vendor neutral declarative access control policy language based on XML) which defines the rules for granting users access to services based on their role.

Application Class: Represents applications that have been created by application developers. A snapshot of this class contains information about an application (i.e. the unique identifier, a description of its functionality, date of creation and owner). An application can belong to exactly one application developer.

Executable flow class: represents applications which are defined by wiring together IoT devices and services. A snapshot of this class is a description of the behavior of an application, as a synthesis of predefined processes that exchange data using predefined connections. A snapshot of this class always corresponds to a snapshot of application class.

Application subscription class: A user who needs access to an application must first subscribe to it. A snapshot of this class is associated with exactly one user and declares applications that the user has subscribed. A user may also subscribe to additional applications in which case new applications are added to her/his subscription.

Sensor subscription class: an application owner (developer) is entitled to create new applications by selecting IoT devices from the list of devices that she/he has subscribed. A snapshot of this class is associated with exactly one application user (i.e. application owner) and contains the sensors that the user has subscribed. A user may also subscribe to additional devices in which case new devices are added to her/his sensor subscription snapshot.

Sensor class: represents the models of all sensors. A snapshot of this class provides information about sensors installed in iXen and are available for subscription. This includes sensor name and identifier and the location where the sensor is installed.

Proximity Beacon class: it represents the special model (proximity beacon) of the sensors used in iXen. We use off-the-shelf Estimote[8] beacon sensors that use Bluetooth Low Energy [9] (BLE) wireless protocol to communicate measurements of ambient temperature, lighting and atmospheric pressure.

History Data class: represents data acquired by sensors and stored in a database. The time series created are stored either as (a) raw (unprocessed) values as received from devices and, (b) aggregated (processed) values (i.e. maximum, minimum and average values over predefined time intervals). A snapshot of this class corresponds to one sensor and can be used in applications. Methods of the class implement functionality for storage and retrieval of raw and aggregate data.

2.2. Activity Diagrams

A series of activity diagrams are defined for responding to different use case scenarios for different types of users [15]. In the following scenario, an application developer logs into iXen and subscribes to sensors. After successful login the user searches for installed sensors (e.g. using keywords or by browsing). The system displays the JSON-LD descriptions of sensors matching the selection criteria and the user is opted to create subscriptions to one or more sensors. Confirmed subscriptions are recorded and stored an XACML file. Fig. 3 illustrates this sequence of actions.

[7] https://fiware-tutorials.readthedocs.io/en/latest/administrating-xacml/index.html
[8] https://estimote.com
[9] https://www.bluetooth.com/learn-about-bluetooth/bluetooth-technology/radio-versions/

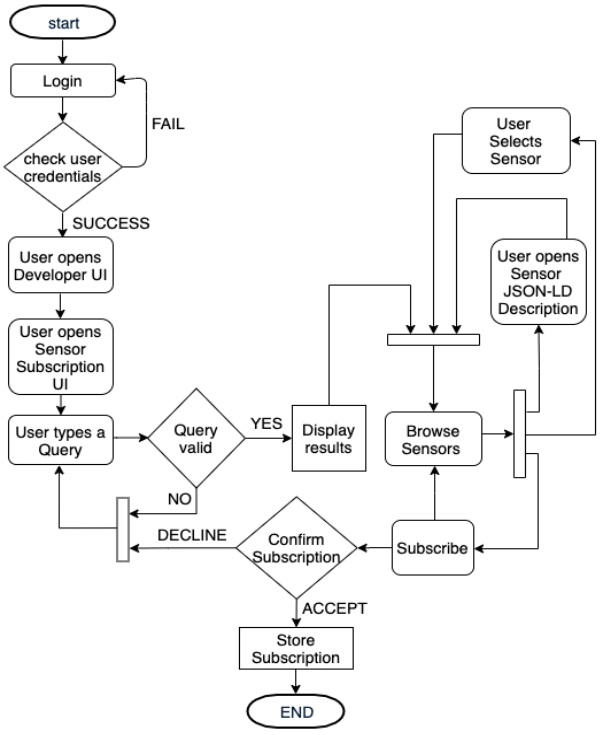

Fig. 3. Subscribing to sensors.

In the next scenario, an application developer creates an application using sensors following the sequence of actions illustrated by Fig. 4. After successful login, the user selects sensors to be included in the application mashup and specifies the type of actions to be applied on the supplied measurements (e.g. average temperature value over 24 hours intervals). The query selection process is similar to that of Fig. 3. However, the query addresses only the list of sensors on which the user has subscribed. The application is then stored in the applications database.

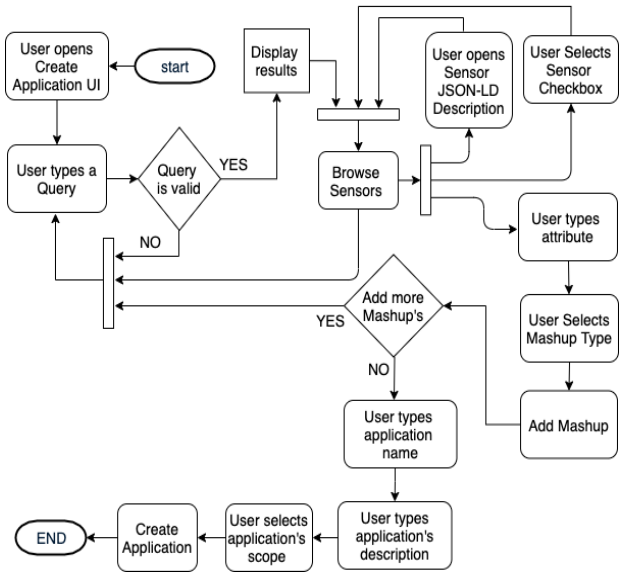

Fig. 4. Creating an application.

In the next scenario, a customer subscribes to an existing application. After login, the user is prompted to input a query using keywords. The applications can be searched by category or by functionality (e.g. indoors or environmental monitoring) or by browsing. Fig. 5 illustrates the above sequence of actions.

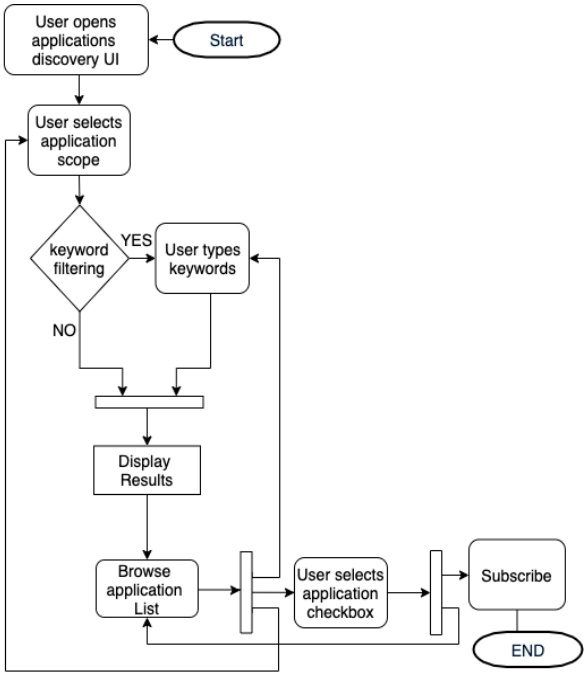

Fig. 5. Searching applications for creating subscriptions.

2.3 Architecture

iXen is designed as a composition of autonomous RESTful micro-services communicating with each other over HTTP. They are organized in groups of services. The services within a group are implemented in the same VM. Network delays are expected due to the nature of this design. However, as shown in Sec. 3, iXen is capable of responding in real time under heavy workloads. Fig 6 illustrates iXen architecture. In the following, groups of services implementing the same functionality are discussed together.

2.3.1 Sensor Services

IoT devices are connected to iXen using *Sensor interface* service. It collects data from gateways (where sensors are connected) using an IoT IP protocol (e.g. MQTT, CoAP). It is implemented using the IDAS backend device maagement[10] service of Fiware. It is the only service which is affected by the property of devices to use a specific protocol. Following *Sensor interface* service, data are communicated in NGSI[11] a data exchange format based on JSON. It is the standard of the EU or handling context information. It describes information being exchanged and entities involved (e.g. sensors that publish measurements and users or services
that subscribe to this information).

The *Sensor interface* service publishes IoT context information to *Publish-Subscribe* service in NGSI format.

[10] https://catalogue-server.fiware.org/enablers/backend-device-management-idas

[11] https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/EU+Standards

Only devices registered to this service can publish data to iXen. *ORION Context Broker* [12] is a reference implementation

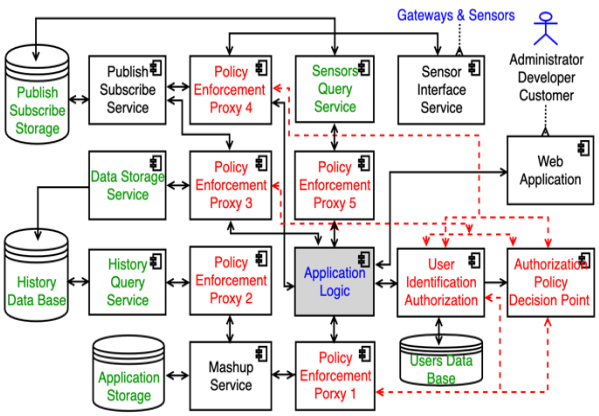

Fig. 6. iXen architecture.

of this service and the service standard of the EU for handling context information. *Publish-Subscribe* service receives measurements from devices registered to *Sensor interface* service and makes this information available to other services and users based on subscriptions. Sensors register to *Publish-Subscribe* service as NGSI "public entities" and users or other services can subscribe to these entities to get notified on value changes or, when new values become available. Each time a new sensor registers to iXen, a new entity is created in *Publish-Subscribe* service. Each time a new sensor value becomes available, this component is updated and a notification is sent to entities subscribed to the sensor. The service holds the most recent values from all registered sensors (i.e. current values are stored in a non-SQL database). History (past) measurements are forwarded to *Data storage* service and from there to *History database*.

2.3.2 Database Services

iXen implements databases for devices, device data, users and applications. Access is facilitated by database interface services. Database and database interface services in Fig 6 are illustrated in green color.

Publish-Subscribe storage holds (in NGSI format) published context and subscription information (e.g. devices that publish data, active subscriptions to devices) and, descriptions of IoT devices along with their most recent measurements. It is implemented using MongoDB[13] (i.e. it suits better than a relational database to the semi-structured nature of this information). Requests addressing this information are issued by the *Sensors query* service using a (close to) natural language syntax involving custom data types (defined in iXen), attribute values and conditional operators (i.e. "*and*", "*or*", "*not*", "*equals*", "*less*", "greater than" etc.). Alternatively, query formulation is facilitated by a graphical user interface providing query forms and the user is prompted to select properties and query operators. Before submitted to *Publish-Subscribe* storage, queries are parsed and are translated to equivalent MongoDB queries involving iXen data types using *Mongo Query Generator*[14].

Table 1 shows data types (i.e. for devices and their properties) to be used by *Sensors query* service for hiding the complexity of MongoDB queries. The following query will retrieve temperature and humidity measurements acquired by weather sensors installed in the city of Chania:

(observes:temperature || observes:humidity) && isModel == ``Estimote beacon" $$ isInCity == ``Chania".

The equivalent MongoDB query is:

'$and' : ['$or' : ['attribute.temperature': '$exists':true, 'attribute.humidity': '$exists' : true], attribute.Model.value: '$eq' : "Estimote beacons", attribute.Location.metadata.City.value: '$eq' : 'Chania'].

Table 1. Data types and properties to be used in user queries.

Data type	Property
isModel	Device type (e.g. "proximity beacon")
Observes	Value type (e.g. "temperature, humidity")
isInCity	Where (e.g. "Chania")
Owner	Infrastructure owner (e.g. "Estimote")
When	Date, time or time interval (e.g. "15/4/2019")

Data storage service collects data flows (history values) from Publish-Subscribe service. The time series created from the history of data are stored in *History database* as (a) raw (unprocessed) values as received from devices and, (b) aggregated (processed) values (i.e. statistics). More specifically, maximum, minimum and average values over predefined time intervals (e.g. every hour, day, week etc.) are stored. The *Data storage* service is implemented using Cygnus[15], the EU standard for handling history of context data in NGSI format.

The *History database* is implemented using MongoDB. The *History query* service provides a query interface to the *History database*: query requests are expressed using the syntax explained earlier and are translated to MongoDB queries.

Application storage is a non-relational database that holds information for applications available to customers for subscriptions. They are created by application owners using Mashup service. Applications are stored in JSON in a non-relational database (i.e. MongoDB). Similar to *Sensors query* service, the database can be searched by properties (i.e. using the data types of Table 1), by name or by owner. Alternatively, a list with all applications can be displayed (together with their descriptions) and the user is prompted to select applications for subscription.

The *User database* is a relational (MySQL) database which holds user's login and authorization information (i.e. users profile data, roles, session information and session history). For each user, ownership and subscription information is also stored (i.e. customers subscribing to applications, application owners subscribing to sensors, infrastructure owners providing sensors for subscription). Before a user submits a service request, his/her role (i.e. a token corresponding to a role) is retrieved and attached in the header of the request. Subsequently, the token will be checked by the target service to verify that the user has the right to access the service (the mechanism is described in Sec. 2.3.5).

12 https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Orion+Context+Broker

13 https://www.mongodb.com

14 https://www.npmjs.com/package/mongo-query-generator

15 https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Cygnus

2.3.3 Mashup services

Application owners are entitled to create new applications. The service is realized with the aid of Node-Red[16], an open-source flow-based programming tool for the IoT allowing for defining applications as a sequence of customizable templates selected from a list. Applications are defined as a sequence of four steps namely, *Endpoint*, *Functionality*, *Calculations* and *Response*. The name and IP address of the service being created, as well as the REST methods (notably *GET*, *PUT*, *POST*) for accessing the service are declared in *Endpoint*. The application is defined as a composition of methods (i.e. functions) receiving inputs from specific devices which are declared in *Functionality*. *Calculations* contains the implementation of the methods (i.e. the software) declared in *Functionality*. The methods implemented in *Calculations* provide current values and value statistics (i.e. average, minimum and maximum values over 1 hour, 24 hours, week and month). Finally, *Response* specifies a URL where the output will be forwarded (typically the address of an application on the Web). Each step forwards information to the next. The application is stored as a JSON entity in *Application storage* (i.e. a MongoDb). Fig 7 declares *Functionality* of *IntelligenceLab* application [15] which computes the maximum (over 24 hours) temperature values from sensors 1, 2 and minimum (over 24 hours) humidity values from sensors 3 and 4.

In order to select sensors to be used in an application, the user issues query to *Sensors query* service. The query is translated to MongoDB syntax and is forwarded to *Publish-Subscribe* storage. Typically, an application will operate on history data by the selected sensors. The application of Fig. 7 will retrieve maximum and minimum values of temperature and humidity over the last 24 hours from *History database*. The output will be generated in HTML/Javascript and will be displayed on a Web interface using Google Charts[17]. Fig. 8 illustrates the output of this query.

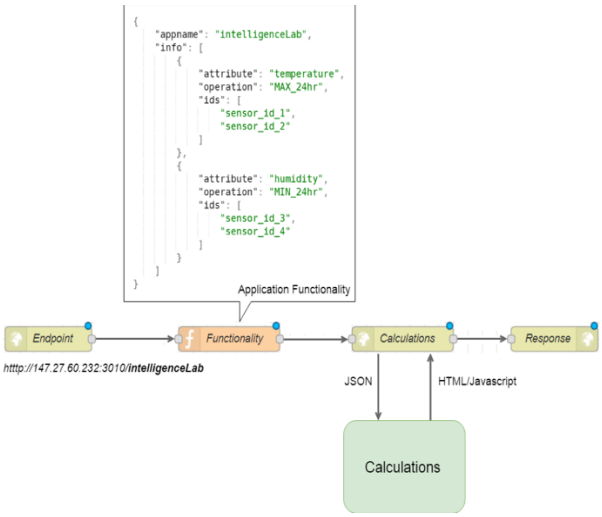

Fig. 7. Declaring an application using Node-Red.

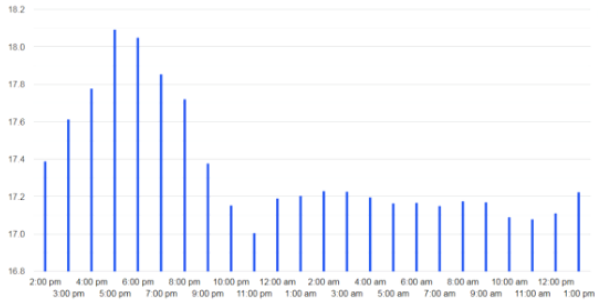

Fig. 8. Web interface showing average temperature over 24 hours.

2.3.4 Application logic

Its purpose is to orchestrate, control and execute services running in the cloud. When a request is received (from a user or service), it is dispatched to the appropriate service. First, a user logs in to iXen using a login name and password. The user is then assigned a role (by the cloud administrator) and receives a token encoding his/her access rights (i.e. authorization to access iXen services). This is a responsibility of the *User identification and authorization* service. Each time application logic dispatches the request to another service, the token is attached to the header of the request. It is a responsibility of the security mechanism to approve (or reject) the request. In iXen all public services are protected by a security mechanism (Sec. 2.3.5).

User requests are issued on the Web interface (e.g. application owners can issue queries for available devices and subscribe to selected devices, customers can issue queries to select applications available for subscriptions).

2.3.5 Security services

They implement access control to services based on user roles and access policies. Initially, users register to iXen to receive a login name, a password and a role (i.e. customer, application owner, or infrastructure owner) encoding user's access rights. This is a responsibility of the cloud administrator. Once a user is logged-in, he/she is assigned an *OAuth2* token encoding his/her identity. The token remains active during a session. A session is initialized at login and remains active during a time interval which is also specified in advance. A new token is issued every time a new session is initiated (e.g. at next user login). User respective user access rights are described by means of XACML (i.e. a vendor neutral declarative access control policy language based on XML). Keyrock identity manager [18] is an implementation of this service.

For each user, a XACML file is stored in *Authorization Policy Decision Point (PDP)*[19] service. Services offering a public interface (i.e. typically SOA services) are protected by a security mechanism (i.e. they do not expose their interface to the Web without protection). Fig. 6 illustrates five protected services and their corresponding security services (in red color). This security mechanism is realized by means of *Policy Enforcement Proxy (PEP)*[20] service. Each public service is protected by a separate *PEP* service (stored in the same VM with the service). It is a responsibility of this service to approve or reject a request to the protected service. Each user request is

[16] https://nodered.org
[17] https://developers.google.com/chart
[18] https://keyrock.docs.apiary.io/#reference/keyrock-api/role
[19] https://authzforce-ce-fiware.readthedocs.io/en/release-5.1.2/
[20] https://fiware-pep-proxy.readthedocs.io/en/latest/

forwarded to *Application logic* service which dispatches the request to the appropriate service. The security process is carried-out by applying the sequence of steps illustrated in Fig. 9. The request comes with a token in its header. The *PEP* service will check if the token is valid by sending a request to User *Identification and authorization* service. If the token is valid (and the session is active), *User Identification and authorization* will respond with user's role. PEP service will forward user's role to *Authorization PDP* service which stores the XACML files for all users. The decision whether the user is authorized to access the protected service will be determined by evaluating the XACML file. This process is carried-out by *Authorization PDP* service which will respond to *PEP* service with a decision. If the request is approved, it is forwarded to the protected service.

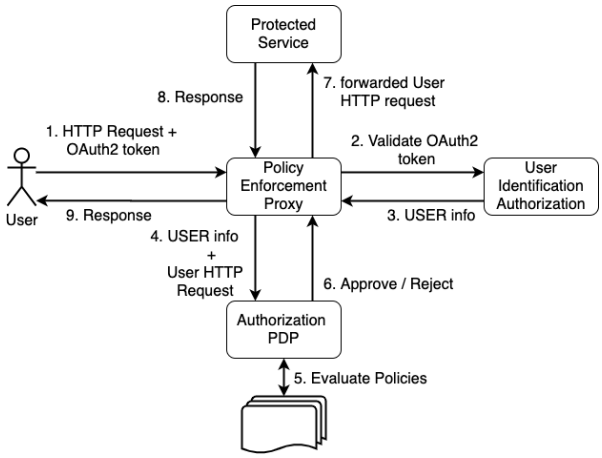

Fig. 9. Protecting a service with an *OAuth2* token.

Not all services accept requests by users. There are also services which are accessible by other services only. They are distinguished from other protected services because they are not directly connected with *Application logic*. These services are protected by a security key, referred to as *master key*. In this case, *PEP* service stores the *master key*. Only requests with the correct key in their header can access the protected service. The mechanism is illustrated in Fig. 10. In Fig. 6, *Sensor interface*, *Mashup* and *Data Storage* services are protected using a *master key*.

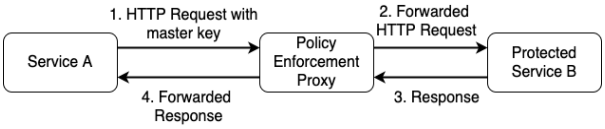

Fig. 10. Protecting a service with an *OAuth2* master key.

2.3.6 Access to Services

In relation to the layered architecture (Fig. 1) and user roles (classes), system administrators can access all of services in any layer. Administrators are also entitled to insert, update or remove users (infrastructure owners, application owners or customers) provided that they do not have subscriptions to devices or applications (otherwise, prior to deleting a user, they are prompted to remove their subscriptions).

Sensor services are available to infrastructure owners and administrators for adding, removing devices and for monitoring the operation of connected devices. However, infrastructure owners can only control devices in their possession. No such restriction applies to administrators (they control all devices connected to iXen).

Database services are available to all user classes. In particular, application owners can subscribe to existing devices for creating new applications. Customers can only subscribe to applications (and not directly to sensors) through subscriptions.

Mashup services are available to application owners and to customers (for subscribing to mashups). Finally, *security services* (i.e. assigning users a role) are available to system administrators only. *Application logic* services are not directly associated to any user role (class).

2.3.7 Implementation

The implementation is in line with the layered design of Fig. 1. All user services are projected on the Web interface in layer 3 and can be selected from a pull-down menu by the users depending on their authorization [15]. IoT data processing and analysis services (e.g. applications), as well as mashup and security services are implemented in layer 2, in the cloud. Sensor connectivity with gateways (fog layer) and with the cloud is implemented in layer 2, except from the sensor interface services (Sec. 2.3.1) which are implemented in layer 1.

iXen is deployed in 5 (small flavor) VMs. Each one has one processor (x86_64 processor architecture, 2,800MHz), 2,048MB RAM, 20GB hard drive, runs Ubuntu 14.04 and an Apache HTTP server. The first VM runs *Publish-Subscribe*, *Sensor query* and *Sensor interface* services. The second VM runs *Mashup*, *Application storage* and *User Identification and Authorization* services. The third VM runs *History database* and *History query* services. The fourth VM runs *Data storage*, *Application logic* services and the *Web application*. The fifth VM runs *Authorization PDP* service. Each service is protected by a dedicated *PEP* service installed in the same VM.

3. Performance Evaluation

A high-level smart-home user case scenario and evaluation of an earlier version of the architecture is described in [17]. Because a large infrastructure (e.g. a smart city) with thousands of sensors installed (e.g. at homes or the environment) is not available to us, we had to rely on simulations for the evaluation. The purpose of the following experiment is to evaluate the performance if iXen under stress (i.e. up to 2,000 requests out of which, 100 are executed in parallel). In [15] we run a series of experiments simulating different system conditions and user needs (i.e. 2,000 requests, and up to 250 users issuing requests concurrently).

There are 10 BLE Estimote beacon sensors transmitting (each one) 100 temperature and humidity measurements per hour (24,000 per day). The sensors connect to a gateway (i.e. a mobile device) and from there, sensor measurements are transferred to *Sensor interface* service in the cloud. The sensors are registered to *Publish-Subscribe* service of iXen. The *History database* consists of two data sets, one with raw (i.e. unprocessed) measurements and one with statistical values (i.e. minimum, maximum, values) taken every hour.

In order to run a more realistic experiment we created a much bigger dataset with measurements from 2,000 simulated sensors. Each simulated sensor produces pseudo-random measurements in the same value range and form as a real sensor. In this set-up (with all actual and simulated sensors in

place), the *History database* contains more than 50 Million measurements.

Table 2 summarizes the performance of the most representative operations. ApacheBench [21] is used to stress iXen with 2,000 simultaneous requests (for each operation), 100 of which are executed in parallel (simulating the case of 100 concurrent users). All measurements of time reported below are averages over 2,000 requests. CPU utilization is almost 100% for all requests. Resource usage metrics are taken using the Linux *htop* command. All measurements of time reported below account also for the time spent for the communication between VMs or between services within the same VM (i.e. network delays are expected due to the nature of SOA design).

User requests are issued on the Web interface and are forwarded to *Application logic*. From there, they are dispatched to the appropriate iXen services. All operations address storage services: operations 1, 2 and 3 address *Publish-Subscribe* or *Publish-Subscribe* storage services; operation 4 address *History query* and *History database* services; operations 5, 6, 7 and 8 address *Mashup* and *Application storage* services. The responses follow the same path to the Web interface in reverse order. More extensive results are reported in [15]. For each request in Table 2, response times improve with the simultaneous execution of requests (i.e. the Apache HTTP server switches to multitasking) reaching their lowest values for concurrency between 50 and 150. Even with concurrency = 250 the average execution time per request is close to real-time in most cases.

Table 2. Performance of basic iXen operations for 2,000 requests and concurrency = 100.

No.	Request	Time (ms)	RAM (GB)
1	Get temperature and humidity sensors	12.5	1.32
2	Get current temperature of a sensor	7.12	0.67
3	Get sensors in a specific location	6.80	0.61
4	Get maximum temperature of a sensor	6.10	0.68
5	Get user subscription to applications	6.80	0.60
6	Create subscription to application	6.63	0.61
7	Get application information	3.86	0.49
8	Create a new application	49.8	1.32

An important observation is that almost 15% of the time reported in Table 2 accounts for security checks (i.e. for validating user authorization credentials).

4. Conclusions and Future Work

iXen exhibits all desirable features of platforms design and implementation such as openness and expandability (i.e. new services can be added to iXen, even third-party ones), advanced connectivity for the vast diversity of devices, security (i.e. access to data and services is granted only to authorized users or services based on access policies) and, smart application development using flow-based programming. iXen design relies on the most recent EU standards for context information (i.e. information related to user's interaction with an application) management and IoT systems design. Besides being open, modular and expandable, iXen can support a business logic and become self-sustainable.

iXen is currently being extended to support billing policies and functionality for dealing with complex events. Incorporating scalability features for dealing with increased workloads is an important direction for future work. A possible solution would be deploying iXen in Kubernetes and a serverless environment. Transforming iXen to multi-edge cloud (MEC) architecture for dealing with distributed IoT deployments at the edges of the network and incorporating trust evaluation mechanisms for dealing with internal risks [4] is underway. HTTPS protocol will eventually replace HTTP as a secure solution for the transmission of confidential information. Securing the IoT network for handing risks due to malicious behavior of IoT devices is still an open issue.

[21] https://httpd.apache.org/docs/2.4/programs/ab.html

References

[1] Petrakis EGM, Sotiriadis S Soultanopoulos T, Tsiachri Renta P, Buyya R, Bessis N. Internet of Things as a Service (iTaaS): Challenges and Solutions for Management of Sensor Data on the Cloud and the Fog. Internet of Things Journal. September 2018; pp. 156-174. https://www.sciencedirect.com/science/article/pii/S2542660518300350,, https://doi.org/10.1016/j.iot.2018.09.009

[2] Radoglou Grammatikis PI, Sarigiannidis PG, Moscholios ID. Securing the internet of things. Securing the Internet of Things: Challenges, Threats and Solutions. Internet of Things Journal. March 2019; pp. 41-70. https://www.sciencedirect.com/science/article/pii/S2542660518301161 https://doi.org/10.1016/j.iot.2018.11.003

[3] Cavoukian A, M. Dixon M. Privacy and Security by Design: An Enterprise Architecture Approach. September 2013. https://www.ipc.on.ca/wp-content/uploads/Resources/pbd-privacy-and-security-by-design-oracle.pdf

[4] Wang T, Zhang S, Liu A, and Z. A. Bhuiyan ZA, Jin Q. A Secure IoT Service Architecture with an Efficient Balance Dynamics Based on Cloud and Edge Computing. September 2018; pp. 4831-4843. https://ieeexplore.ieee.org/document/8464241 https://doi.org/10.1109/JIOT.2018.2870288

[5] Garfinkel T, Rosenblum M. A Virtual Machine Introspection-Based Architecture for Intrusion Detection. Proceedings of Network and Distributed Systems Security Symposium (NDSS). February 2003; pp. 191-206. https://www.ndss-symposium.org/ndss2003/virtual-machine-introspection-based-architecture-intrusion-detection

[6] Industrial Internet Consortium (ICC). Industrial Internet of Things Volume G4: Security Framework. September 2016. https://www.iiconsortium.org/IISF.htm

[7] Douzis K, and Sotiriadis S, Petrakis EGM, Amza C. Modular and Generic IoT Management on the Cloud. Future Generation Computer Systems

(FGCS). January 2018; pp. 369-378. https://www.sciencedirect.com/science/article/pii/S0167739X16301662 https://doi.org/10.1016/j.future.2016.05.041

[8] Azzola F. IoT Visual Programming Tools. DZone, IoT Zone, White paper. https://dzone.com/articles/iot-and-the-iot-visual-programming-tools

[9] IoT Analytics. IoT Platforms: The Central Backbone for the Internet of Things. November 2015. https://iot-analytics.com/product/iot-platforms-white-paper/

[10] Sarkar C, Uttama Nambi A, VeVenkatesha Prasad R, Rahim A, Neisse R, Baldini G. DIAT: A Scalable Distributed Architecture for IoT. IEEE Internet of Things Journal. June 2015; pp. 230-239. https://ieeexplore.ieee.org/document/7000513 , https://doi.org/10.1109/JIOT.2014.2387155

[11] Schreier S. Modelling Restful Applications. Proceedings of ACM International Workshop on RESTful Design (WS-REST'11). March 2011; pp. 15-21. https://dl.acm.org/doi/10.1145/1967428.1967434, https://doi.org/10.1145/1967428.1967434

[12] Erl T. SOA Principles of Service Design. Upper Saddle River, NJ, USA: Prentice Hall. 2007. https://dl.acm.org/citation.cfm?id=1296147

[13] Bassi A, Bauer M, Fiedler M, Kramp T, Kranenburg R, Lange S, Meissner S. Enabling Things to Talk: Designing IoT Solutions with the IoT Architectural Reference Model. Springer, Heidelberg. 2013. https://www.springer.com/gp/book/9783642404023 https://doi.org/10.1007/978-3-642-40403-0

[14] Preventis A, Stravoskoufos K, Sotiriadis S, Petrakis EGM. IoT-A and FIWARE: Bridging the Barriers Between the Cloud and IoT Systems Design and Implementation. Proceeding of Cloud Computing and Services Science (CLOSER), 2016; pp. 146-153. http://www.gbv.de/dms/tib-ub-hannover/871309920.pdf https://doi.org/10.5220/0005912001460153

[15] Koundourakis X. Design and Implementation of Service Oriented Architecture for Deploying IoT Applications in the Cloud. Diploma Thesis, School of ECE, Technical University of Crete (TUC), Chania, Greece. February 2019. http://www.intelligence.tuc.gr/index.php?module=view&class=publication_file&id=572

[16] Rozanski N, Woods E. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2nd Edition, 2012. https://www.viewpoints-and-perspectives.info/home/book/

[17] Euripides G.M. Petrakis, George Myrizakis: iHome: Secure Smart Home Management in the Cloud and the Fog, in Edge Computing for Massive Parallel Processing of IoT Data Streams, Fatos Xhafa, Arun Kumar Sangaiah (Eds). Vol 35: Advances in Parallel Computing, IOS Press, pp. 237-263, 2020. http://www.intelligence.tuc.gr/~petrakis/publications/iHome-IOS.pdf