Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 170 (2020) 145–152

Procedia Computer Science

www.elsevier.com/locate/procedia

The 11th International Conference on Ambient Systems, Networks and Technologies (ANT)
April 6 - 9, 2020, Warsaw, Poland

iXen: context-driven service oriented architecture for the internet of things in the cloud

Xenofon Koundourakis, Euripides G.M. Petrakis*

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 computing; IoT architecture; service oriented architecture; micro-services; context management;

1. Introduction

Cloud is the ideal environment for IoT applications deployment due to reasons related to its affordability (no upfront 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.

* Corresponding author. Tel.: +030-28210-37229 ; fax: +030-28210-37542.
E-mail address: petrakis@intelligence.tuc.gr

10.1016/j.procs.2020.03.019

Securing IoT infrastructures is a challenging task. The principles of Security by Design and, Security and Privacy by Default [2] 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 [3]. This creates new challenges for dealing with the cause and point of system failure if security fails. The Industrial Internet Consortium (IIC) [4] 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. 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.

iXen's ambition is to overcome the limits of existing IoT architectures in the cloud and deal with challenges of security, openess and interoperability. iXen architecture is highly configurable and modular and supports generation of fully customizable applications by re-using services and devices. Leveraging flow-based programming 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[1] 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 roles 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 (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[2] for context information management and IoT systems design. iXen prototype is implemented in OpenStack and Fiware[3], the open-source distributed cloud infrastructure of

[1] https://oauth.net/2/
[2] https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/EU+Standards
[3] https://www.fiware.org

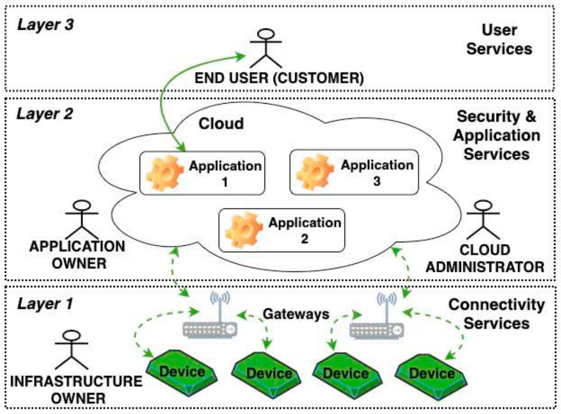


Fig. 1. iXen 3-tier design.

the EU. iXen is implemented as a composition of modular cloud micro-services implementing fundamental functionalities and communicating with each other using REST. More services can be added on demand or, any service can be replaced or moved to a different VM (on the same or different cloud) with minimum overhead (i.e. only the IP of the service will change).

In regards to similar work in the literature, iXen concept resembles DIAT model [5] for IoT architectures addressing the challenges of interoperability and scalability. It encompasses a usage control policy model to support security and privacy in a distributed environment. DIAT is a layered model architecture for three different component layers (referred to as Virtual Object, Composite Virtual Object and Service layer) similar to SOA. The work 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.

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 show that iXen is capable of responding in real-time under heavy work loads (i.e. many users applying several requests per second).

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.

2. Design and architecture

We followed a valid design approach 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 analysed, (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 [6].

2.1. User groups

Each user belongs to a user class. Each user class is assigned a role encoding authorization to access other services. The following user groups and functional requirements associated with each group are identified:

Systems 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.

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.

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.

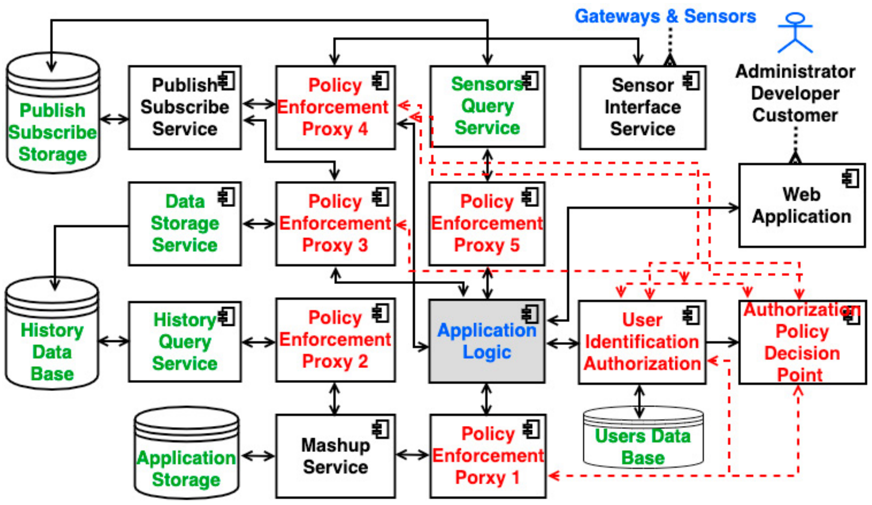

Gateways & Sensors
Administrator
Developer
Customer
Publish Subscribe Storage
Publish Subscribe Service
Policy Enforcement Proxy 4
Sensors Query Service
Sensor Interface Service
Web Application
Data Storage Service
Policy Enforcement Proxy 3
Policy Enforcement Proxy 5
History Data Base
History Query Service
Policy Enforcement Proxy 2
Application Logic
User Identification Authorization
Authorization Policy Decision Point
Application Storage
Mashup Service
Policy Enforcement Porxy 1
Users Data Base

Fig. 2. iXen architecture.

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 rights to applications.

2.2. *iXen 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. 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. 2 illustrates iXen architecture. In the following, groups of services implementing the same functionality are discussed together.

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 management*[4] 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[5], a data exchange format based on JSON. It is the standard of the EU for 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. Only devices registered to this service can publish data to iXen. *ORION Context Broker*[6] is a reference implementation 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) 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. 2 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 (i.e. it suits better than a relational database to the semi-structured nature

[4] https://catalogue-server.fiware.org/enablers/backend-device-management-idas
[5] https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/EU+Standards
[6] https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Orion+Context+Broker

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[7].

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")

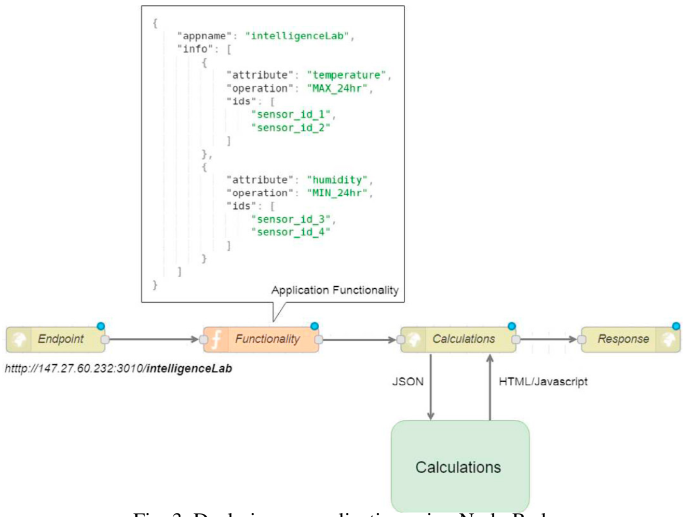
Data storage service collects data flows (history values) from *Publish-Subscribe* service. The time series created from the history of data are stored in *History database* as (a) raw (unprocessed) values as received from devices and, (b) aggregated (processed) values (i.e. statistics). More specifically, 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[8], 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.

User database is a relational (MySQL) database which holds users 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 Paragraph 6).

3) Mashup services: application owners are entitled to create new applications. The service is realized with the aid of Node-Red[9], 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

[7] https://www.npmjs.com/package/mongo-query-generator
[8] https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Cygnus
[9] https://nodered.org



```
{
    "appname": "intelligenceLab",
    "info": [
        {
            "attribute": "temperature",
            "operation": "MAX_24hr",
            "ids": [
                "sensor_id_1",
                "sensor_id_2"
            ]
        },
        {
            "attribute": "humidity",
            "operation": "MIN_24hr",
            "ids": [
                "sensor_id_3",
                "sensor_id_4"
            ]
        }
    ]
}
```

Fig. 3. Declaring an application using Node-Red.

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. 3 declares *Functionality* of *IntelligenceLab* application 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 queries 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. 3 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[10].

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. User requests are issued on the Web interface. 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 (Paragraph 6).

5) Web application: the users access the system using a Web interface. Application owners can issue requests (queries) for available devices and subscribe to selected devices and customers can issue queries to select applications available for subscriptions.

6) 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. An 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[11] (i.e. a vendor neutral declarative access control policy language based on XML). *Keyrock identity manager*[12] is an implementation of this service. For each user, a XACML file is stored in *Authorization Policy Decision Point (PDP)*[13] service.

[10] https://developers.google.com/chart
[11] https://fiware-tutorials.readthedocs.io/en/latest/administrating-xacml/index.html
[12] https://keyrock.docs.apiary.io/#reference/keyrock-api/role
[13] https://authzforce-ce-fiware.readthedocs.io/en/release-5.1.2/

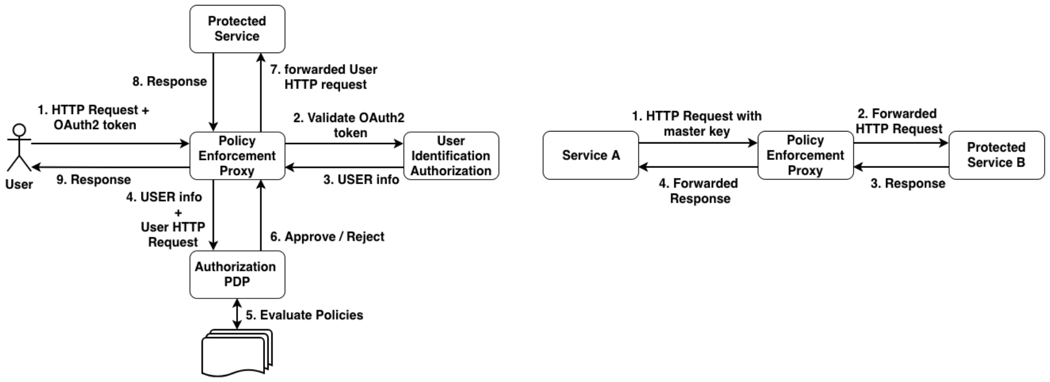


Fig. 4. Protecting a service with an *OAuth2* token (left) and an *OAuth2* master key (right.)

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. 2 illustrates five protected services and their corresponding security services (in red color). This security mechanism is realized by means of *Policy Enforcement Proxy (PEP)*[14] 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 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. 4 (left). 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.

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. 4 (right). In Fig. 2, *Sensor interface*, *Mashup* and *Data Storage* services are protected using a *master key*.

3. Performance Evaluation

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.

There are 10 BLE Estimote[15] 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.

[14] https://fiware-pep-proxy.readthedocs.io/en/latest/
[15] https://estimote.com

Table 2 summarizes the performance of the most representative operations. ApacheBench[16] 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.

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

Request	Time (ms)	RAM (GB)
Get temperature and humidity sensors	12.5	1.32
Get current temperature of a sensor	7.12	0.67
Get sensors in a specific location	6.80	0.61
Get maximum temperature of a sensor	6.10	0.68
Get user subscription to applications	6.80	0.60
Create subscription to application	6.63	0.61
Get application information	3.86	0.49

User requests are issued on *Web application* service 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 and 7 address *Mashup* and Application storage services. 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. 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).

iXen may produce big amounts of data and receive many requests, requiring large processing capabilities surpassing the capacities of this implementation. In order to guarantee uniform performance, more instances of the application (or for groups of services) can be spawned in additional VMs.

4. Future Work

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 [3] is underway. Securing the IoT network for handing risks due to malicious behavior of IoT devices is still an open issue.

Acknowledgment

This research has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code: T1EDK-04759, MARS).

References

[1] E. G. Petrakis, S. Sotiriadis, T. Soultanopoulos, P. T. Renta, R. Buyya, N. Bessis, Internet of things as a service (itaas): Challenges and solutions for management of sensor data on the cloud and the fog, Internet of Things 3–4 (9) (2018) 156–174.
[2] A. Cavoukian, M. Dixon, Privacy and Security by Design: An Enterprise Architecture Approach (Sep. 2013).
[3] T. Wang, S. Zhang, A. Liu, Z. A. Bhuiyan, Q. Jin, A secure iot service architecture with an efficient balance dynamics based on cloud and edge computing, IEEE Internet of Things Journal 6 (3) (2018) 4831–4843.
[4] I. I. C. (ICC), Industrial Internet of Things Volume G4: Security Framework (Sep. 2016).
[5] C. Sarkar, A. U. Nambi, R. V. Prasad, A. Rahim, R. Neisse, G. Baldini, Diat: A scalable distributed architecture for iot, IEEE Internet of Things Journal 2 (3) (2015) 230–239.
[6] X. Koundourakis, Design and Implementation of Service Oriented Architecture for Deploying IoT Applications in the Cloud, Tech. Rep. TR-TUC-ISL-02-2019, Diploma Thesis, School of ECE, Technical University of Crete (TUC), Chania, Crete (Feb. 2019).

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