UNIVERSITÀ DEGLI STUDI DI PARMA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA, ELETTRONICA E DELLE TELECOMUNICAZIONI

PROGETTAZIONE E IMPLENTAZIONE DI UN FRAMEWORK DI SICUREZZA PER SMART OBJECTS IN SCENARI IOT

DESIGN AND IMPLEMENTATION OF A SECURITY FRAMEWORK FOR SMART OBJECTS IN IOT SCENARIOS

Relatore:

Chiar.mo Prof. MARCO PICONE

Correlatore:

Dott. Ing. SIMONE CIRANI

Tesi di Laurea di:

NICO MONTALI

ANNO ACCADEMICO 2014/2015

Vorrei dedicare questa tesi ad Aldo, Livia, Maria e Franco, i miei nonni. Grazie per essere stati sempre presenti e avermi guidato e consigliato durante tutta la mia vita.

Ringraziamenti

Vorrei ringraziare, primi su tutti, i prof. Marco Picone e Simone Cirani, Relatore e Correlatore, che hanno reso possibile il lavoro di questa tesi e mi hanno guidato, formato e supportato (o sopportato). Grazie inoltre per avermi proposto questo progetto coinvolgente in cui sono riuscito a dare tutto me stesso. Vorrei anche ringraziare tutti i ragazzi del WASNLab, in particolare Luca, che si sono rivelati sempre disponibili a dare una mano. Infine vorrei ringraziare il mio grafico di fiducia, mio fratello Simone, che ha creato tutte le immagini presenti in questa tesi.

Contents

Introduction

The Internet of Things and the interest about it, is rapidly growing nowadays. More and more objects become connected together in various way. But, with just such a fragmented situation, where every producer implements his own technology, the most important thing to focus on, in order to continue this rapid growth in a correct way, is to define standards that must be used by everyone. These standards must answer the question on how they communicate with each other and with the outer world. If a standard of this type is defined, every connected object, even if different in hardware and software, will be capable of communicating and gathering additional informations about its environment. The key feature of this standard protocols must be the lightness. This is because the majority of connected devices can not implement a powerful hardware to handle onerous computations and must focus on energy consumptions. Some of the most used standard to implement these type of protocol are CoAP and MQTT: CoAP will be used in this thesis, because it is more light-weight and implements a one-to-one communication. But with the increasing number of connected devices, more and more difficulties rise up, and must be taken in account before a real expansion of the IoT to the consumers. One of the main problem of connected devices is security: it is difficult to know who owns an object on the internet and who is authorized to use it. This thesis focuses on this problem, and provides a general solution using an external centralized web service.

Chapter 1

Internet of Things - State of the Art

1.1 The IoT world

The Internet of Things (IoT) can be defined as a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies. [1] All these interconnected things, from a physical point of view, are the so called smart objects. A smart object can connect to different entities:

- Another smart object. This is the M2M (Machine to machine) [2] communication. An example could be a watering system that asks to some humidity sensor in a garden to decide whether or not to water it.
- A user that wants to interact with the object, like to turn on a light or open a door lock.
- A cloud web service that gathers data for statistics (like big data).

This capability to connect together of the smart objects creates an enormous number of new scenarios, where common objects will "evolve" to new functionalities.

Connecting more and more things to the internet has also some disadvantages that must be avoided in order to let the IoT spread. First of all, the increasing number of devices, as shown in the graphic 1.1, requires a new structure for the web, partially resolved with the introduction of IPv6, the improved version of the standard IPv4, that allows an increase from 10^9 maximum devices to 10^{38} devices(using 128 bits instead of 32).

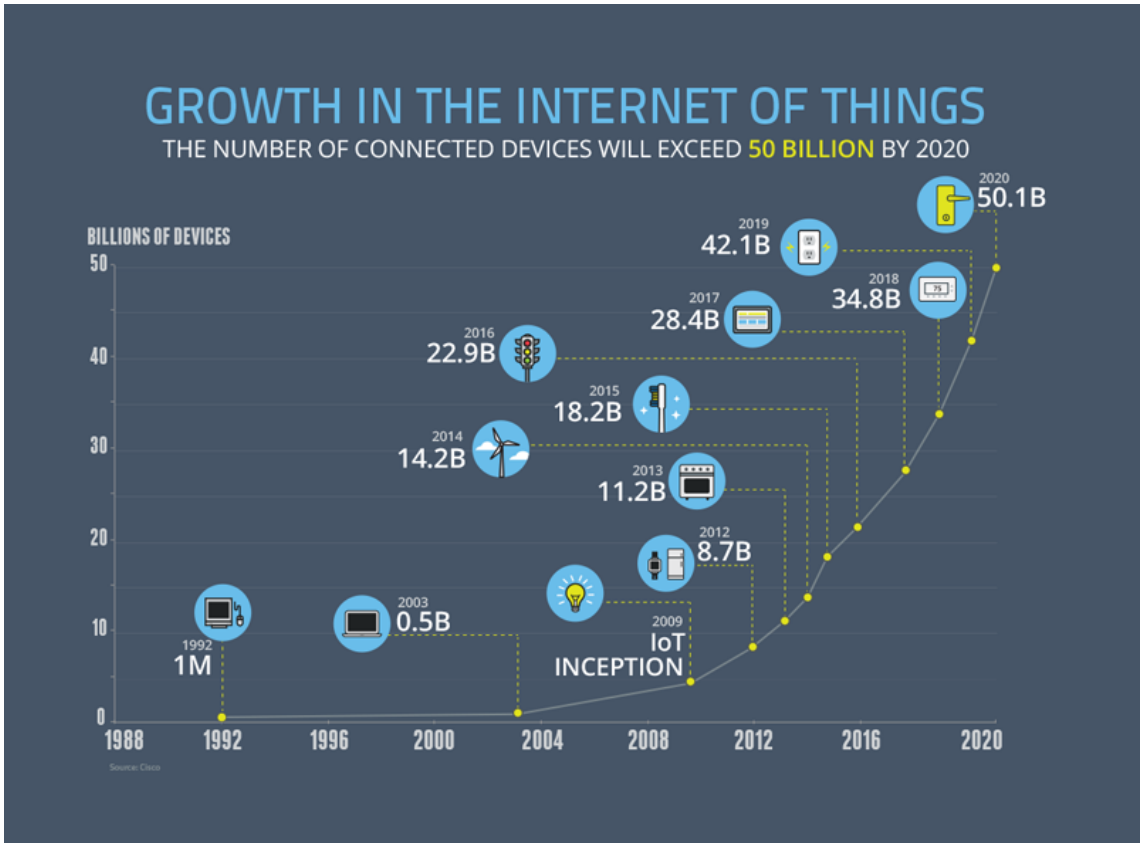


Figure 1.1: Growth in the Internet of Things, National Cable & Telecommunications Association.

The second real problem is the cost of the device; to produce and sell a smart device, the cost of the hardware used to connect it should be very low, at least not relevant compared to the cost of the object itself. This can be only achieved scaling down

the device, in size and computing power; also power consumption is important to hold low prices. All these considerations lead to a device that is reduced under every aspect: these are the so called Constrained Devices. A constrained device, then, has not all the possibilities of a "normal" electronic device, and this fact leads to a different approach in the implementation of an IoT environment. For example, the common communication protocol, HTTP, is just too complicated to handle when communicating with a constrained device, therefore a different, lightweight, alternative is used. One of the most famous protocols that realize this is CoAP (Constrained Application Protocol) [3], which will be described in detail in paragraph 1.3.

1.2 IoT visions: some scenarios

A significant example of what IoT can achieve is given by the concept and scenarios of a smart city. Smarter cities are based on smarter infrastructures, in order to optimize services like transportation, traffic or energy consumptions [4]. A large number of optimizations can be made using an IoT infrastructures, where data around the city are collected and analyzed. An example could be the management of the traffic, where traffic sensors can reveal congestions in real time so to redirect traffic. The city of Toronto has implemented this system and reduced delays by 40%. Another example is the management of parking: with the usage of sensor to determine if a parking spot is occupied or not, the city system can show the driver a free spot, reducing the time used to search for a spot, and so, reducing emissions and traffic. A study by Streetline, a company interested in smart parking, has marked that in Los Angeles, in a 15-block area, drivers drove 1.500.000 kilometers in one year searching for a spot. There are a lot of others possible scenarios for IoT in smart cities, and for sure great changes are expected. Another environment in which IoT

could bring changes and optimizations is the home. A smart home [5] could be made by some smart appliances, like air conditioners, refrigerators and other. With the data provided by sensors on these objects, the house could optimize energy consumption or notify the user failures or particular events.

1.3 CoAP: Constrained Application Protocol

As said before, CoAP is a lightweight protocol that substitutes HTTP in constrained environments. CoAP takes all the important and useful things from HTTP, but is based on a different protocol stack. The main difference in this protocol stack (ISO-OSI model) is at the transport layer where instead of TCP [6], CoAP uses UDP [7], which has some main advantages:

- Connectionless: TCP requires a previous connection before the real communication, UDP is simpler.
- No error management: it does not reorder packets and there is no retransmission.
- Fast: because of the previous specifications, UDP focus more on speed than reliability.

Because of these properties, UDP has been used to realize the lightweight CoAP protocol. In addition, the CoAP message packet is reduced to the essential in terms of bytes, so a constrained device can handle it better. CoAP has been developed mostly by the CoRE (Constrained RESTful environments), a IETF group. CoAP is specified in RFC 7252. Other differences from the standard internet protocol stack are the usage of 6LoWPAN for the network layer [8], an optimized IPv6 for constrained devices, and the usage of IEEE 802.15.4e, an optimized physical layer

protocol.

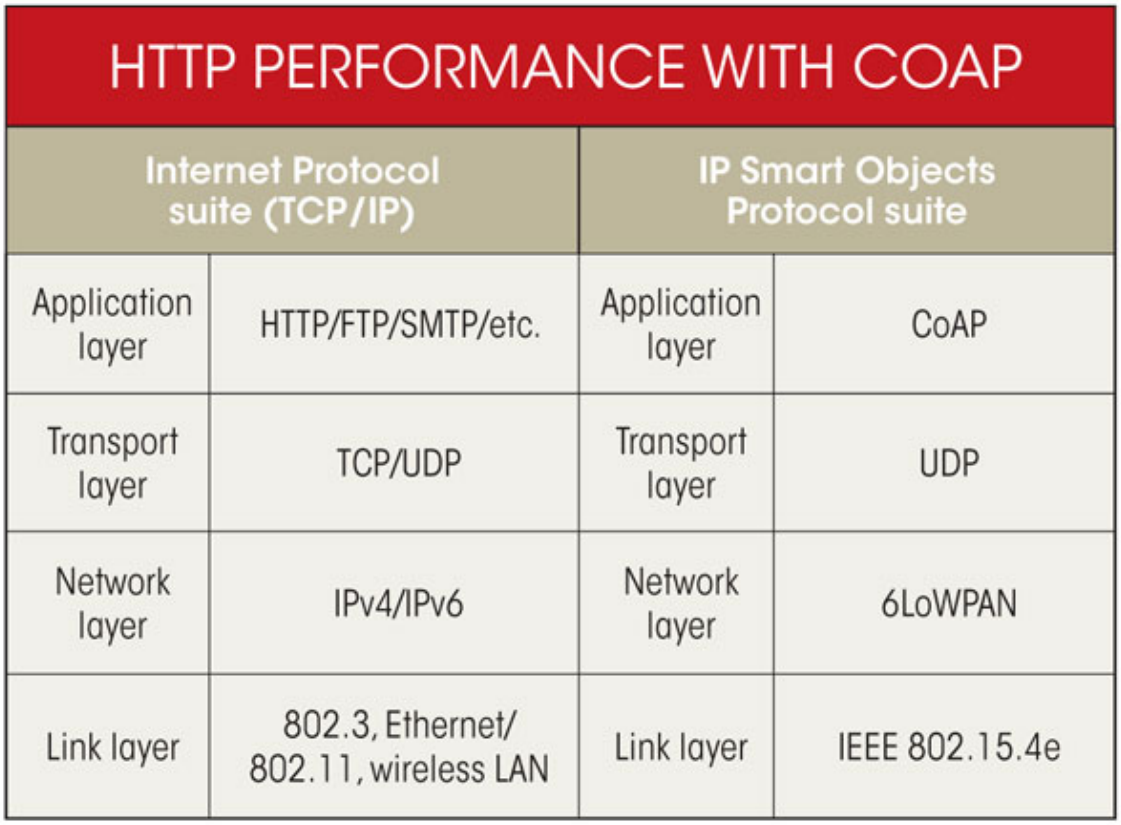


Figure 1.2: HTTP protocol stack and CoAP protocol stack

To best explain CoAP some words on RESTful [9] are needed. REST stands for REpresentational State Transfer and it is one of the most used architecture of the web. It is based on a few simple principles:

- Application state and functionalities are resources.
- Every resource has a uniform interface (limited functions, like http methods).
- Every connection is state-less and cacheable.

So, it means that everything on the internet is a resource, that has a certain state, and that can be used by a limited set of methods. In order to use this architecture

and join to HTTP in a better way, CoAP was built to simplify things, however it remained similar to HTTP: it uses the same method and the same logic in the requests and responses. But it is also an extension of it, providing new functionalities specific for IoT scenarios, like the OBSERVE method, that allows to monitor a certain resource automatically.

1.4 Security problem in IoT

When every common object around us becomes connected, one of the main problem to rise is security. If an object is connected, it means that someone, remotely, can obtain data or even control something. For some objects it may be good like public sensors around a city, because their data are not private. However if we think of a smart car, that can be used by someone that has access to it, the security of this access becomes the main problem to focus on. Then, particular attention must be paid to the security problem to make every object around become smart. The solution of this problem should use another IETF specification, the UMA architecture(User Managed Access) [10]. This architecture is based on OAuth (explained in section 1.9), and it represents a system where the users themselves manage access to web resources, with concepts of ownership, protection and authorization. The security problem described here is also described in an IETF draft [11], where some scenarios are also presented.

1.5 Thesis objectives and summary

The main goal of this thesis is to provide a working system that implements a solution to the security problem described above. The system is composed of different

parts:

- Smart objects network, using CoAP as protocol.
- A proxy to intercept all communication to the devices.
- A cloud service, called the OAS (OAuth Authorization Service), that manages all the permissions.
- An android app to manage permissions on the service and also perform requests to smart objects.

All these components have been developed for this thesis and are described later in detail. The remaining Chapter 1 is a brief view of the state of art and a little presentation of the main used technologies. In Chapter 2, the detailed architecture of the system is presented, along with usage scenarios for the end user. Chapter 3 shows the implementation of the system, with significant code snippets. Finally, in Chapter 4, a working demo of the system is presented, as well as some conclusion about this thesis.

1.6 IETF draft

The main document that shows an overview of the security problem in IoT applications is the IETF draft "Authentication and Authorization for Constrained Environments Using OAuth and UMA". In this document some scenarios are presented, as well as a generic structure built using UMA and OAuth, described later in chapter 1.9. There are two main scenarios presented, one is the usage of a smart scale, the second of a smart car. In the first scenario, a smart scale is bought by the user, and by some method of authentication using OAuth, like a QRcode or a pin inside the

box, the user is given an authorization token that proves the possession of the smart scale. If the user uses the scale now, all the data (weight, BMI etc) are saved in a remote service, and can be read in the user account. Also, using UMA, the user can authorize someone else to use the smart scale (e.g. family member), or to read the data gathered (e.g. the doctor). In the second scenario, a smart car is bought by the user, and at the moment of the purchase, the possession of the car is given to the user. The user now can use the car and view data about it. If the user wants to authorize someone (e.g. a valet parking) to use the car, he can provide access (also limited in time) to the car simply by using the connected application.

1.7 IoT-OAS

Another important document that represents the basis of this thesis is the document "IoT-OAS: An OAuth-Based Authorization Service Architecture for Secure Services in IoT Scenarios" [12] where the base structure of the project is presented. This document begins with some considerations about constrained devices, which lead to a remote authentication and authorization service called IoT-OAS. Another approach could be the simplification of the authorization process, in order to be executed also on a constrained device. The remote service IoT-OAS, however, gives some big advantages:

- Reduces the time for developers to create new code, because it uses standard authorization and authentication technologies like OAuth.
- Reduces the load of work on the constrained device, because all the security logic (that is complex) is implemented by a remote server that is not constrained.

- Manages all the authorization information dynamically and remotely.

The second part of the document presents the architecture of this system and the most common use cases of the system. The entities given in the use cases are very similar to the OAuth entities (User, Service Provider, Service Consumer, Authentication Service, Request or Access Tokens). In the last part of the document, a test environment is used to analyze performance of the system in terms of power consumption and time of execution. The power consumption results increased (by the doubled number of packets), but the advantages of this architecture are more important anyway.

1.8 Transport Layer Security

The communications in the system use 2 main protocols: CoAP for the communications between smart objects and HTTP for the communications between more complex nodes (like the server and the android application presented in chapter 2). The security at the transport layer for these two protocols is given by extending it with TLS [13] for TCP, or its UDP equivalent DTLS [14].

The TCP (Transmission Control Protocol) itself, does not provide any type of security on the transport layer. It means that in every basic TCP communication none of two entities that connect has evidence of the identity of the other (this means it could be a fake server or client) and, given that the communication is not encrypted, anyone could intercept sensitive data without permission. The solution to this problem is an extension on the transport layer, called TLS (Transport Layer Security, called previously SSL, Secure Sockets Layer). The latest TLS version is 1.2 and it is defined in RFC 5246. The primary goal of the TLS protocol is to provide privacy and data integrity, and it is composed of two layers: TLS Record Protocol

and TLS Handshake Protocol. The TLS Record Protocol guarantees:

- Private connection, using a symmetric cryptography to encrypt data. The keys for the encryption are unique and are generated by the Handshake layer privately.
- Reliable connection, implemented by message integrity checks, using keyed MAC (Message Authentication Code), like HMAC [15].

The TLS Handshake Protocol, instead, performs some operations before the real communication:

- Peer's identity can be authenticated using asymmetric or public key cryptography (e.g. RSA, DSA).
- The negotiation of a shared secret (the keys used in the message encryption) is secure and reliable.

TLS protocol is an extension of the transport layer protocol TCP, so it can work with every application layer protocol that uses TCP (like HTTP). The use of TLS in the HTTP stack protocol gives it a secure connection, mostly used on the internet, the so called HTTPS.
The same approach used to secure the TCP protocol on the transport layer can be used to secure UDP protocol. This is done by DTLS (Datagram Transport Layer Security), which solves two main problem that the use of TLS on UDP would involve:

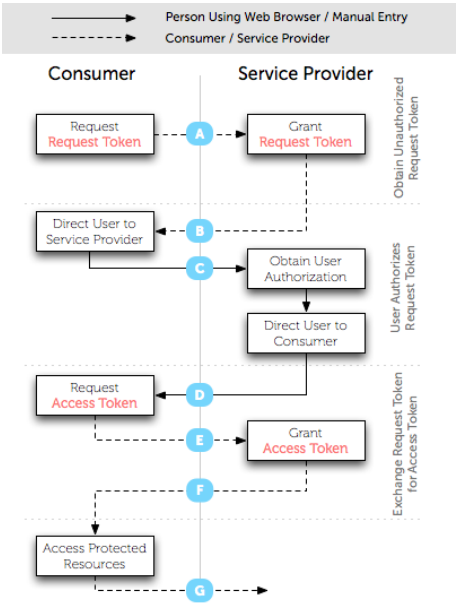
- TLS does not allow independent decryption of individual packets, because the integrity check depends on the sequence number.
- The TLS Handshake assumes that the connection is reliable, with retransmission if packets are lost.

The use of DTLS on top of UDP in the CoAP protocol stack generates the so called CoAPS.

1.9 Application Layer Security

The security at the application level of this project is based mostly on the OAuth protocol, used to provide access to a resource. In this section, is also given a little description of the UMA protocol, that provides methods for users to control access to their protected resources.

The traditional client-server authentication model, with the increasing use of web services and cloud computing, does not provide a delegated access. OAuth is designed specifically for these web services, introducing a third role: the resource owner (User). The client (called Consumer), that is not the User, requests access to the resource hosted by the server (Service Provider) but controlled by the User. In order to retrieve access to the resource, the User must first authorize the Consumer. The OAuth protocol is based on the exchange of tokens, generated temporary keys used to verify the authorization to access a resource. The Figure 1.3 represents the generic OAuth (version 1.0) flow of communication and data attached to the messages: we see Request Tokens (RT) in the first phase, and Access Tokens (AT) in the second. RTs are generated to guarantee the identity of the Consumer, where ATs provide access to the resource for the Consumer. The transition between Request and Access is always made with the interaction of the Resource Owner. In the HTTP protocol, all the OAuth informations (consumer key, signature, etcetera) are carried by the Authorization header. The version of OAuth used by this project is 1.0. There is also a OAuth 2.0, that introduces some advantages like support for non-browser applications or simplified cryptography and signatures.



The UMA architecture, that stands for User-Managed Access, is an extension of OAuth 2.0, providing methods for the users to control access in a delegated way: the User (Resource owner) delegates an external service, called Authorization Manager (AM). In this way, the user does not have to authorize every request directly, but is the AM that manages requests.

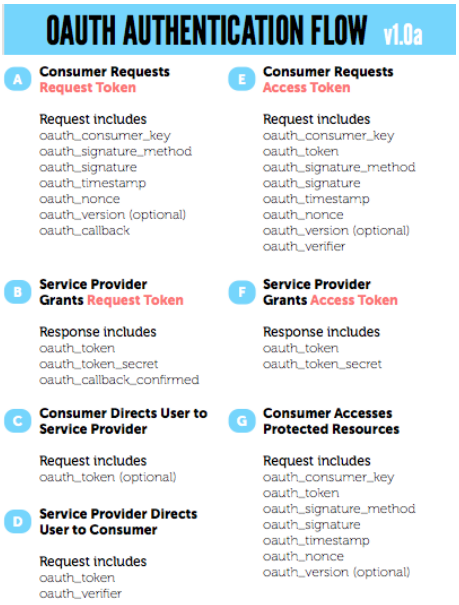


Figure 1.3: OAuth 1.0 Authentication flow.

Chapter 2

System Design

In this Chapter, the general architecture of the system is presented. Every component will be described in detail, providing scenarios in which the component is involved. Also, the interaction between the components is described here.

2.1 System architecture

The system architecture of this project is composed of various parts, which cooperate together. The most important part is the remote web service used to authenticate and authorize communications (described in detail later in Section 2.1.2). This web service realizes the IoT-OAS presented before. The architecture also comprehends the main protagonists of the IoT concept: smart objects. These smart objects are managed by the IoT-OAS, granting or denying requests from clients. In these set of smart objects, there must be at least one device, called proxy, which is unconstrained and intercepts all the communications directed to the smart objects associated to it. The reasons of this design choice are described later in Section 2.1.1. The last component of the structure is a mobile application, which gives to the end user a

simple and usable interface to interact with smart objects but also with the IoT-OAS. Through the application, the user can manage objects and the relative permissions. The ensure that all the communications between these entities are secure, all the HTTP communications are secured with TLS and all the CoAP communications are secured with DTLS.

2.1.1 The IoT network

This network is composed by an arbitrary number of constrained devices, called Servers, which uses CoAPS to communicate with each other and with the external world. Because of the constrained nature of these devices and the result given in the document of specification of IoT-OAS, a particular approach has been chosen for this project: instead of implementing the logic to make requests to the OAS directly on the servers, an unconstrained device, called Proxy, is present in the network. All the communications to the network must pass through this proxy, that redirects the message to the specific node requested. The direct communications to the devices are blocked. In this way, a generic server must not implement any additional code or make additional requests, which gives it 2 advantages:

- The power consumption denoted by the previous document is not localized anymore on the constrained device, but is moved to the unconstrained proxy.
- Given that the server must not implement any logic, it could be any generic smart object.

The most significant component of the network, the proxy, holds all the logic used to implement secure communications to the devices and communicates with the IoT-OAS to gather authorization informations. The proxy, in addition, because of

its unconstrained nature, can handle CoAPS and HTTPS, giving the client another advantage: the client has not to implement a CoAPS communication, but can use HTTPS instead (that is more diffused). So, the proxy must implement at least these functionalities:

- Receive HTTPS request from clients, containing Authorization header with OAuth informations and CoAP request information (method, resource, content).
- Make a request to the IoT-OAS for the specific client request, to know wether or not the client is authorized to use the resource. In this request it must specify informations about the resource and the request.
- Given the response from the IoT-OAS, the proxy must decide wether or not to forward the request to the device or respond with an Unauthorized error to the client.
- Redirect the response of the device to the client.

All these operations compose the flow of actions in a generic request from the client to the proxy. The proxy also implements a Resource Directory, a list of the attached devices used for service discovery. When a new device connects to the network, an instance of the object is added to the Resource Directory. The unique identification of the device is made using UUIDs (Universally Unique Identifier): the proxy includes this information in the requests made to the IoT-OAS.

2.1.2 The IoT-OAS

The IoT-OAS is a web service used to manage all the authorizations of the system. It is designed to work with multiple proxies, in a centralized way. The IoT-OAS

must implement these functionalities:

- Receive HTTPS request from client to authorize them to own a resource. The client is the resource owner (if he provides all the informations contained in the QRcode attached to the device) and the OAS gives the client all the authorizations.
- Receive HTTPS request from client to authorize someone else to use a resource. The client that makes the request is the resource owner, and provides informations of the other client that he wants to authorize.
- Receive HTTPS requests from the proxy and specify to it if the client is authorized or not to use the specific resource.
- Act as the Service provider of the OAuth, giving the client an Access Token to use in the requests.

All these functionalities compose a set of APIs that the IoT-OAS should provide, to interact with it and with the relative database structure where all the informations are saved.

2.1.3 The mobile application

The mobile application, developed for Android, must provide to the user all the functionalities to manage owned resources or shared resources. At the beginning of the execution, the user is requested to log in with typical providers (Facebook or Google+) to verify his identity in a unique way. When login is performed, using OAuth 2.0 services, the user is given (from the IoT-OAS) credentials to use. These credentials includes Consumer info and Tokens (described later). Using these credentials the application can perform requests to the OAS (e.g. authorize someone)

or to the proxy (for a specific device). The application also implements a mechanism of service discovery, where all the devices found in the network are listed.

2.2 Application scenarios

2.2.1 Authorize a new device through the application scenario

In this scenario, the user bought a new smart device and wants to use it. To start communicating with it, the user must provide to the IoT-OAS, that manages all the interactions, a proof of the ownership of the device. This could be a QRcode printed on the device (like in this case) or other form of physical verification. This process is done by the user through the mobile application. The process involved in this scenario is:

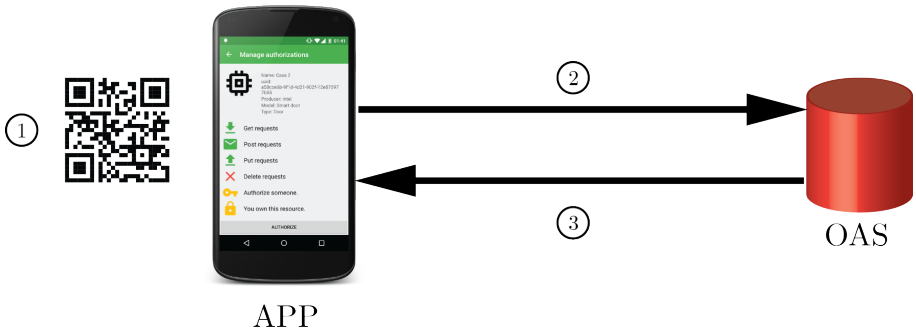


Figure 2.1: Authorize a new device through the application.

1. The user scans a QRcode from the device, containing informations about it (such as device uuid, producer, model, type of device) and decides a name for the device.

2. The application sends a request of authorization to the OAS, containing device informations and consumer access keys and tokens.

3. The OAS verifies if the access keys are correct and updates his database. It returns all the informations of the client, including the newly added device.

4. The user now has access to the resource and can perform all types of request to it.

These operations only involve the application and the IoT-OAS, so they could be performed remotely.

2.2.2 Authorize someone through application scenario

In this scenario, USER1 has bought a smart device that wants to share with USER2. USER1 has already completed the ownership process described before, and the IoT-OAS associates the device to him. USER2 also wants to use the same device, and asks (personally) to USER1 to share it. USER1, with the following process, can authorize USER2, providing him the permissions he wants.

1. APP2 wants to access a resource owned by USER1, so it generates a QRcode that represent it.

2. APP1 chooses the resource to share, reads the QRcode generated by APP2 and chooses the permissions to give to it.

3. APP1 sends a request of authorization to the OAS, containing all of his informations, the chosen resource and also the informations of APP2.

4. The OAS verifies the credentials of APP1 and updates the database to grant APP2 access to the specified resource. It returns to APP1 a positive response.

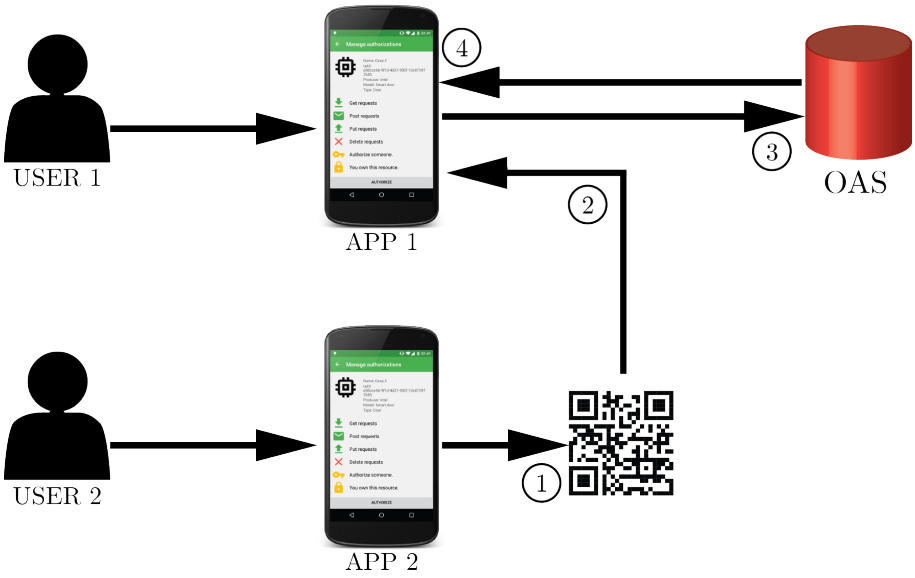


Figure 2.2: Authorize someone else through application scenario.

5. Now APP2 can access the resource, limited to the permissions USER1 has given to it.

These operations only involve the application and the IoT-OAS so they could be performed remotely. But, in the current version of the application, APP1 and APP2 must be physically near in order to scan the QRcode.

2.2.3 Perform a request to the device scenario

In this scenario, a user wants to use a smart device. The user could be authorized or unauthorized for the IoT-OAS, depending on the informations it has about it. The process described later, used to perform a request on a device, involves all the components of the project: the user sends the request through the application and can view results on it, the proxy intercepts the request and the IoT-OAS decides if the user is authorized.

1. The user, through the app, wants to access resource R1 in the proxy network.

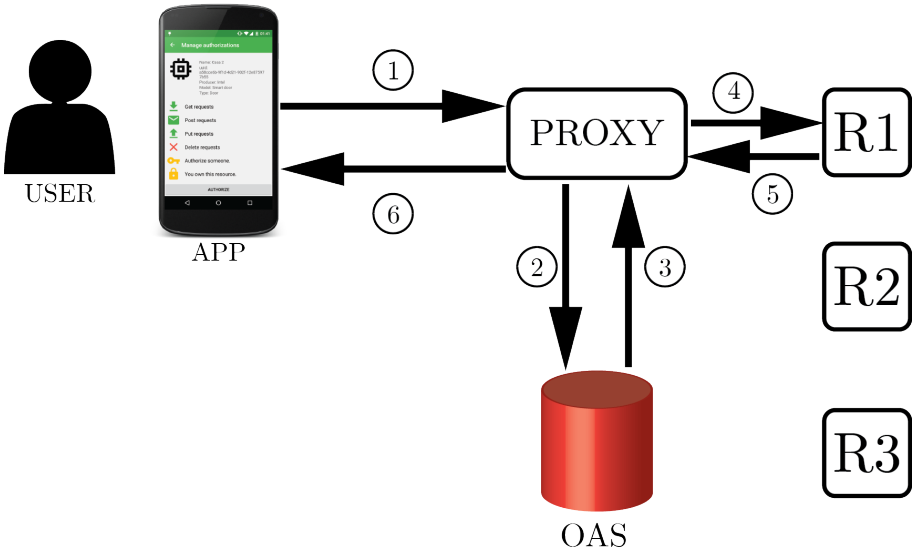


Figure 2.3: Perform a request to the device scenario.

So, he sends a request to the proxy, containing the resource and the OAuth authorization header via HTTPS.

2. The proxy now decodes the request to the resource, and sends a request to the OAS, containing consumer informations.

3. The OAS verifies if the request is authorized, and responds via HTTPS to the proxy with a OK code (HTTP - 200) or an unauthorized code (HTTP - 401).

4. If the consumer is unauthorized the proxy immediately answers to it with an unauthorized code. If the consumer is authorized instead, the proxy sends the decoded request to the resource R1.

5. Resource R1, without knowing anything about authorization and the OAS, answers to the proxy.

6. The proxy forwards the answer to the client.

These operations, instead, requires the interaction of the application, the IoT-OAS, the proxy and the specific resource. This involves that the user should be in the same network of the proxy (and use service discovery to find it) or at least know the global IP address of the proxy.

Chapter 3

System Implementation

In this Chapter, the implementation of the system is presented: for every component, it will describe how it was created, what are its main functionalities and how it works. Some code snippets will be provided to understand better the behavior of the system.

3.1 CoAP Server

The CoAP server developed for this project is realized in Java, using the library Californium [16]. Californium is a library that implements all the standard communications of the CoAP protocol, and gives the developer a set of high level objects to use. The main object used to implement the Server in this thesis is CoAPServer, that acts like a simple server. Every server can instantiates an arbitrary number of CoAPResources. Every resource can specify the behavior related to a request, overriding specific methods (handleGet, handlePost etcetera). But the default CoAPServer was not suitable for this project, because the communications, to be secure, must implement CoAPS. To implement CoAPS in the CoAPServer another library of

the californium project has been used: Scandium, a library that implements DTLS communications and can integrate with Californium to create CoAPS entities. The main object to implement DTLS security is DTLSconnector as seen in the following code (Source Code 3.1). Using Scandium, the Server now uses certificates and public/private keys to secure the connection.

```
/**
 * Public constructor, PSK and keystore capable
 * @param trustStoreLocation
 * @param trustStorePassword
 */
public SecureServer(String trustStoreLocation, String
    trustStorePassword){
   try{
      //Load trust store
      KeyStore trustStore = KeyStore.getInstance("JKS");
      InputStream inTrust = new FileInputStream(trustStoreLocation);
      trustStore.load(inTrust, trustStorePassword.toCharArray());
      //Load certificate
      Certificate[] trustedCertificates = new Certificate[1];
      trustedCertificates[0] =
          trustStore.getCertificate(CERTIFICATE_IDENTITY);
      connector = new DTLSConnector(new InetSocketAddress(DTLS_PORT),
          trustedCertificates);
      certificateCapable = true;
   }catch(Exception e){
      e.printStackTrace();
   }
```

```
}
```

Listing 3.1: DTLS implementation

Every server (a device) has a set of informations, shared in the resource directory to identify it. These properties are: a UUID, the Producer, the Model, and a Type.

3.2 CoAP Proxy

The CoAP proxy was built using other classes of the Californium framework, and it is composed of this main objects:

- HTTP server to receive requests from clients.
- COAP server to receive CoAP requests from clients.
- 2 ForwardingResource (included in Californium), used to translate requests.

The first thing to implement in addition of these objects, was to secure every communication: both the HTTP side and the CoAP side. The CoAP side was implemented in the same manner of the previous Server, using Scandium (and specifically the class DTLSconnector). The HTTP side security was implemented extending the apache HTTP server used by the standard proxy with JavaX SSL library. With these modifications the proxy can now use secure connections with clients and resources. But the most important thing to change in the default proxy is the logic of redirections: the request must be first analyzed and sent to the IoT-OAS. Depending on the response, the proxy should respond to the client. This behavior is implemented in code by changing the class responsible of request handling: HttpStack. Every request, is first analyzed, decoded and sent to the IoT-OAS. The specific request contains this basic informations:

- The method of the request.
- The UUID of the device requested.
- The Authorization header extracted from the HTTP request.

```
Gson gson = new Gson();
HttpClient client = verifiedClient();
HttpPost clientRequest = new HttpPost(OASLauncherSSL.OASurl);
clientRequest.addHeader("Content-Type", IoTOASRequest.HTTP_CONTENT_TYPE);
clientRequest.setEntity(new StringEntity(gson.toJson(oasRequest)));
HttpResponse rx = client.execute(clientRequest);
LOGGER.info("Sent request to OAS");
if (rx.getStatusLine().getStatusCode() == HttpStatus.SC_OK){
    //Right request
    LOGGER.info("Client is authorized to perfom request");
    handleVerified(proxiedRequest, httpExchange, httpContext);
}else{
    sendSimpleHttpResponse(httpExchange,rx.getStatusLine().getStatusCode())
        ;
}
```

Listing 3.2: Proxy redirection to IoT-OAS.

The request (in json format) is POSTed to the IoT-OAS and the proxy wait for a reply. When it receives a reply, it continue the execution in case of a positive response (Code 200 - OK), redirecting the request to the resource, or send a simple error response to the client directly.

```
{
```

```
  "method":"GET",
  "uuid":"09fc3fd0-c79a-4fc6-983f-a7e56e6161d1",
  "authorizationHeader":"OAUth oauth_consumer_key=***, oauth_token_key=***"
}
```

Listing 3.3: Json content of the POST request to the IoT-OAS.

3.3 The IoT-OAS Server

The IoT-OAS web service is composed of 2 parts: an API interface realized with CodeIgniter [17], and a verification side, realized in Java. CodeIgniter is a PHP framework to realize Model-View-Controller web applications. In this case, through CodeIgniter, various PHP functions had been implemented, creating a simple API interface for all the basic operations that need the database. To understand better the structure of the APIs and of the project, the scheme of the database is presented in the image below.

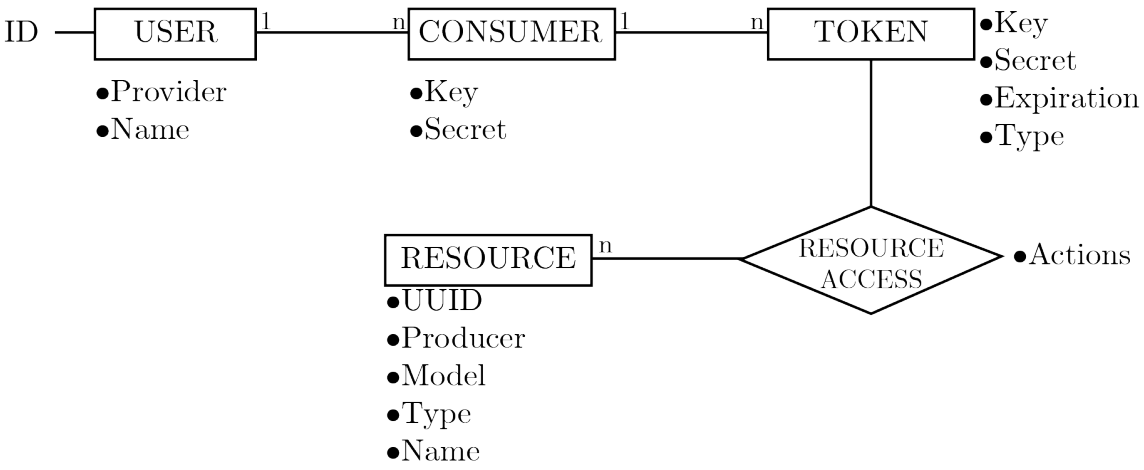


Figure 3.1: Database structure.

The User table represents a logged user from Facebook or Google+. Each User

can have multiple Consumers, alias the application used to access the resource. Every consumer has a key and a secret (public and private). Every Consumer has at least a token, and each token has a key, a secret and an expiration date. The resource table memorizes all the registered resource of the system. The association between resources and tokens defines the policy of the access to the resource. To interact in a simple way with the database, 2 main PHP functions were built:

- Fetch: this function, called specifying a Consumer and an associated Token, answers with all the informations about them. This informations contain all the resource access given to them. See Source 3.4.
- AuthorizeToken: this function, is called by POSTing different information. First of all, the identity of the caller, giving Consumer key and Token key. The Content of the request must contain the token to authorize, the device to give access to and the operations permitted to it.

```
{ "consumer":
  {
    "key":"-OWEj7xtzd1JCb2E1Nc2Vq394",
    "secret":"TTqux9Im6zvdLD2Ofn2hEi92k",
    "name":"Generic client"
  },
  "token":{
    "token":"bz04YUzRlHpJ00qPLuA-L05YS",
    "secret":"BDblgln4AH26gvGO91qZHfKbF",
    "type":"AT",
    "expires":"0000-00-00 00:00:00"},
    "resources":[
```

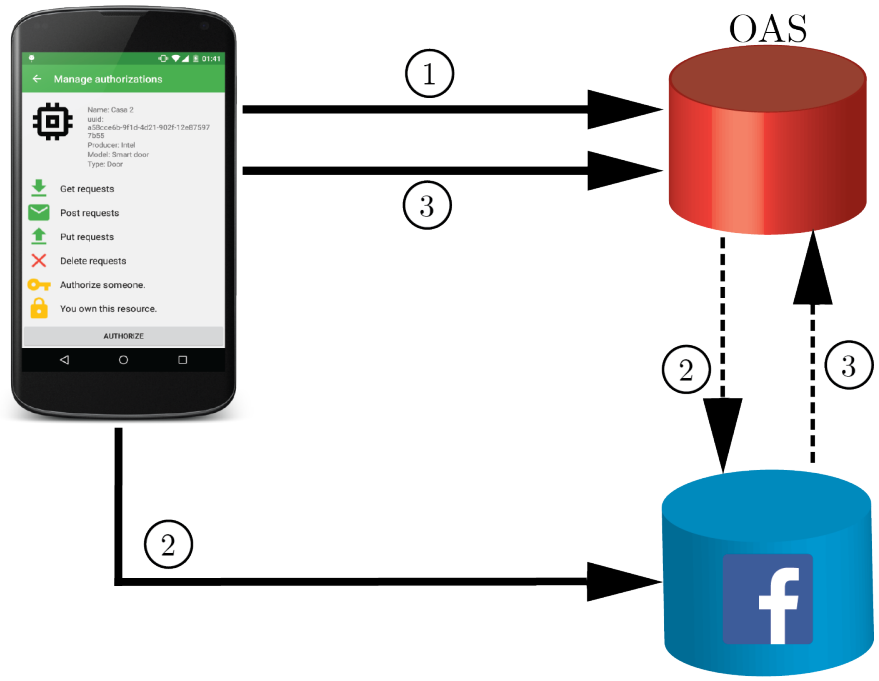
```
    { "uuid":"hoj329edhkjsdksl9292",
      "producer":"Unipr",
      "model":"IoT HUB",
      "name":"Lab hub 2",
      "type":"Hub",
      "actions":"GET,POST,PUT,DELETE,AUTH,OWN"
    }
  ]
}
```

Listing 3.4: Response to a Fetch function call.

To provide all the needed functionalities, another API has been created to handle the login with other providers. (Figure 3.2) This function implements the OAuth 2.0 flow request by the specific providers (Facebook and Google+). This function, basically, makes a redirection of the user to the providers OAuth url, and get the provider response, through another redirection, containing all the informations about the authenticated user.

The Java part of the server, instead, has only a specific task: to analyze requests from proxies and to decide if the Consumer has access to the resource. This server is realized using an HTTPS Jetty Server, which receives the POST request from the proxies as seen above in Source 3.3. The Java server also calls the Fetch API on the PHP side, and receives, for a specific Consumer and Token, all the resources permitted. When it has this information, it verifies the authorization header provided with the request in 2 steps:

- Verifies if consumer has access to the resource with the specified method.
- Verifies the signature of the OAuth header.



1 https://oasserver/auth/facebook

2 https://graph.facebook.com/...

3 https://oasserver/oauthcallback

Figure 3.2: OAuth 2.0 implementation.

The signature is a verification code, computed using a specific algorithm (in our case, HMAC-SHA1). This computation requires both the public and private keys of consumer and token; in this way, only the consumer can encrypt the data and only the server can decrypt them. When the server has verified if the consumer has access or not, it responds to the proxy.

3.4 Android Application

The android application was developed using Android studio and tested on various devices. The android application is intended to be used to manage all the permissions to smart objects a user has and to perform request to them (through the proxy). The application general structure uses the MVC (Model-View-Controller) paradigm, where the objects of the system are divided in the 3 categories. The first objects to analyze are those ones that belong to the model, in order to understand better the structure of the problem. The classes of the model category represent all the basic ”objects” of the system itself; the main class is ConsumerInfo, which contains all the information about the consumer (keys, tokens, managed resources) and provide methods to simplify the implementation, like the OAuth header generator (including the signature). All the model classes are managed by a singleton (an object with a unique instance), InfoManager, that contains a ConsumerInfo but also informations about proxies and discovered resources. The majority of classes in the model section are POJOs (Plain Old Java Objects), that contains informations (variables) and their getter/setter. This objects are often used to decode JSON messages. Now the other important category, the Controller, that implements the interaction with the user, will be described presenting a usage scenario, as the user would see. When the application is launched, the user is presented a login page (view Figure 3.3). Using the 2 buttons in the page, the user starts the authentication process, using one of two provider: Facebook or Google+. The authentication process is handled by the IoT-OAS that redirects the user through various pages to access. This navigation is made using a WebView inside a Dialog.

When the user has finished the authentication process, the application shows him the principal page, the Home Activity (see Figure 3.4). Here the user has a view of

Figure 3.3: Login activity.

the overall situation. In this activity, by swiping from the left, a drawer is shown. In this drawer (shown in Figure 3.5) the user can choose different options. The first two options simply change the context of the main activity: this is composed, mostly, by a list of devices (Recycler View). The first option ("Home") shows all the devices found in the resource directory of a specified proxy. The request made to the proxy to gather informations about these devices is shown in Figure 3.6 . The second option of the drawer, instead, shows all the devices that are associated to the user (owned and shared). The third option of the drawer shows the application QRcode used to authorize it (see the subsection 2.2.2). The QRcode is generated using the ZXing library by Google, using the user token as content. The last option logs out the user and returns to the login page.

The list of managed devices, instead, is filled by calling the API fetch on the

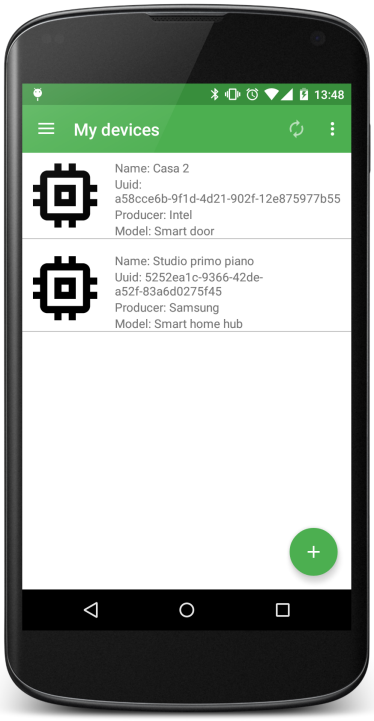


Figure 3.4: Home activity.

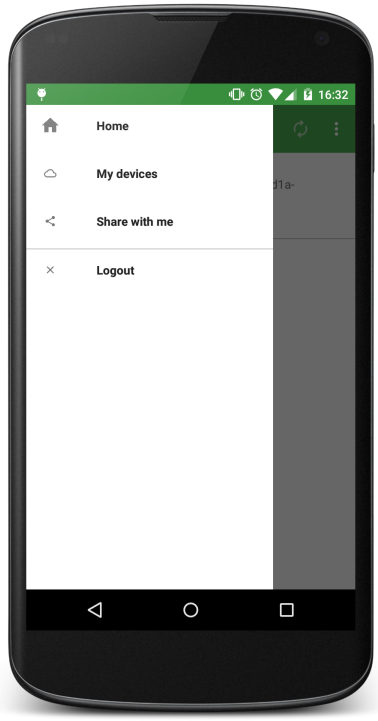


Figure 3.5: Drawer options.

Figure 3.6: Resource Directory request.

server, which returns the list of all the resource associated with the token given.

When the user clicks on a device item in the first list (the resource directory), the application opens a new page, the Client Activity (see Figure 3.7). In this page, the user can perform simple REST request to the selected device, also deciding the specific resource on the device. If the user is authorized from the IoT-OAS to perform request, then the application shows the response of the device (that has passed through the proxy). If the user is unauthorized, an error is prompted to the

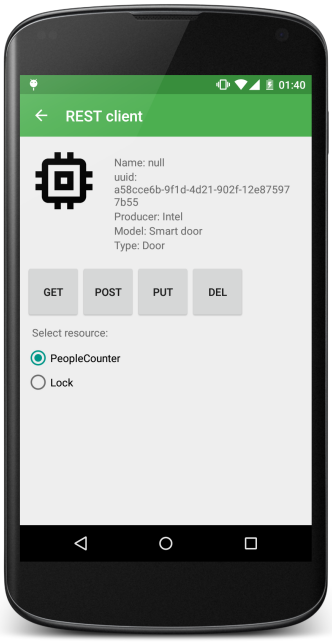


Figure 3.7: REST client.

user to inform him.

By clicking on an item of the second list (the managed resources), the application opens a new page, the Authorization Activity (see Figure 3.8). Here the user can view all the permissions he has on the device, listed as images (greyed if permission is not owned, colored if owned). The permissions are the below:

- The 4 HTTP method implemented (GET,POST,PUT,DELETE).
- Authorization: it means that the user can authorize someone else (only limited to the permissions he has).
- Own: show if the user is the owner of the resource.

By clicking the Authorize button, the authorization process is started. First, the user chooses the permissions to give to other through a Dialog with checkboxes.

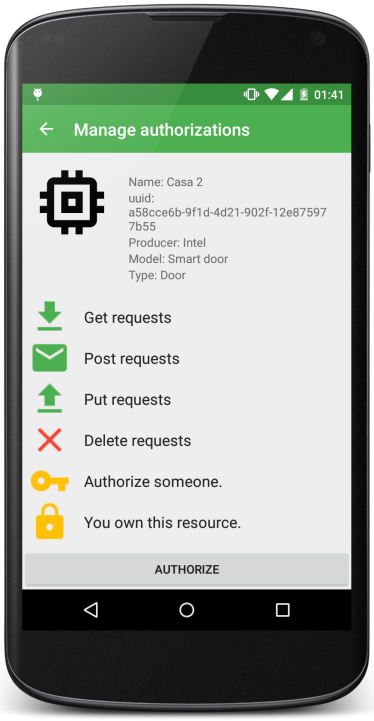


Figure 3.8: Authorization management activity.

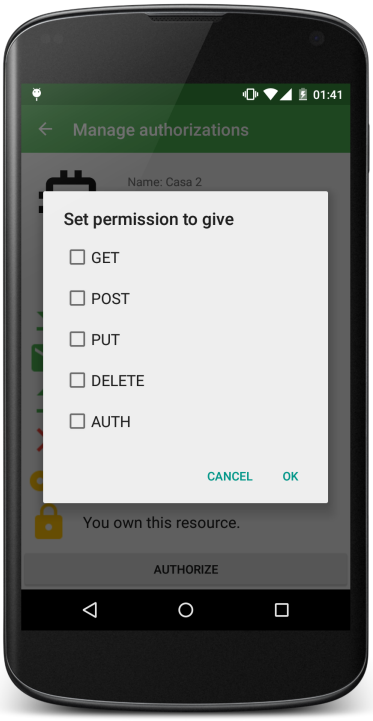


Figure 3.9: Authorization process: choose permissions.

Every checkbox represents the permission to a specific operation (one of the set seen above). If checked, the other user will be given this permission. When the user validates the checkboxes by pressing ”OK”, the QRcode scan application is opened. Now the user that owns the device scans the other one QRcode (that has opened with the third option of the drawer, ”Share with me”). Then the operation completes and the application returns to the authorization page. All the QRcodes generated in the application and the QRcode reader used were implemented using Google ZXing library.

Chapter 4

Experimental analysis and conclusions

In this Chapter, the real implementation on a testbed is presented. It describes also the test phases, used to verify the proper operation of the entire system. The second part includes a conclusion about this thesis and possible future extensions of the system.

4.1 WoTT Testbed

To test in a real environment the system presented in this thesis, an already existing testbed was used, the WoTT (Web of Things Testbed) hosted by WASNLab. This testbed is composed by a vast number of smart objects, different in software, computational power and physical network interfaces. This test environment tries to simulate the heterogeneous composition of devices that are used today as smart objects: there are two main categories of devices, the CIoT (Constrained IoT) and SBC (Single Board Computer). CIoT are classified as Class 1 devices using the nota-

tion used in [18] and mainly run ContikiOS, a lightweight operating system. Some of the CIoT devices are: TelosB, Zolertia Z1 ad OpenMote, all running Contiki. The SBC devices, instead, are classified as Class 2 devices, and mainly run Linux distributions. Some of them are: Intel Galileo (Debian), Raspberry Pi (raspbian) and Arduino Yun (OpenWRT).

The test on this project were done using 2 SBC (Raspberry PI) as CoAP servers (S1 and S2). The Java code that uses the Californium library was installed using SSH on these two devices. Every server was built with some generic resources, capable of all the HTTP methods. Also, every server was assigned the basic informations needed for the resource directory: uuid, producer, model, name, type. The uuid were generated randomly by a uuid generator tool. The Proxy was installed on a computer in the WoTT network. The direct connections to the devices of the WoTT were not really blocked, but were not used. The address of the proxy was assigned to the Servers as their resource directory. First tests verified the functionalities of this system: first, the resource discovery on the proxy, using a HTTP REST client (DHC). Then sending requests to the servers, that was blocked because the IoT-OAS was still not present (all the communications were unauthorized). The second implementation of the testbed was the IoT-OAS: the Java part was loaded on a computer in the WoTT network (not necessary, could be a remote server). The PHP part using CodeIgniter was loaded via FTP on a remote server (Altervista). The database of the system was initially loaded manually with resources and consumers info to test it. In this test, using again the REST client, the same requests as above were done, but these included the Authorization Header (built manually). Both authorized and unauthorized requests were tried. In this phase also the IoT-OAS APIs have been tested: the login using a provider (through a standard browser) and the requests to authorize a token. The last phase was to implement the mobile

application. There were two devices running this application: an Android emulator (Genymotion) using version 4.4.4 and logged in with Facebook (U1) and a real device (LG Nexus 4) running version 5.1.1 and logged in with Google+ (U2). With this last implementation, the system was complete and all the functionalities tested. All the QRcodes that were scanned from the application were generated manually using an online QRcode generator. This is the checklist of the tested functionalities:

1. Login using the application through the IoT-OAS APIs.
2. Authorization of S1 as a new device for U1.
3. Request on the proxy for S1 from U1. Authorized.
4. Request on the proxy for S1 from U2. Unauthorized.
5. Requests on the proxy for S2 from U1 and U2. Unauthorized.
6. Authorization of S1 (limited to GET) from U1 to U2.
7. Request on the proxy for S1 from U2. GET authorized, other unauthorized.
8. Authorization of S2 as a new device for U2.
9. Requests on the proxy for S2 from U2. Authorized.
10. Requests on the proxy for S2 from U1. Unauthorized.

4.2 Conclusion and future work

In conclusion, all the goals defined in Section 1.5 had been achieved by the system built. The CoAP servers work with the secure CoAPS and the proxy can communicate in a secure way with both CoAPS and HTTPS. The proxy, in addition,

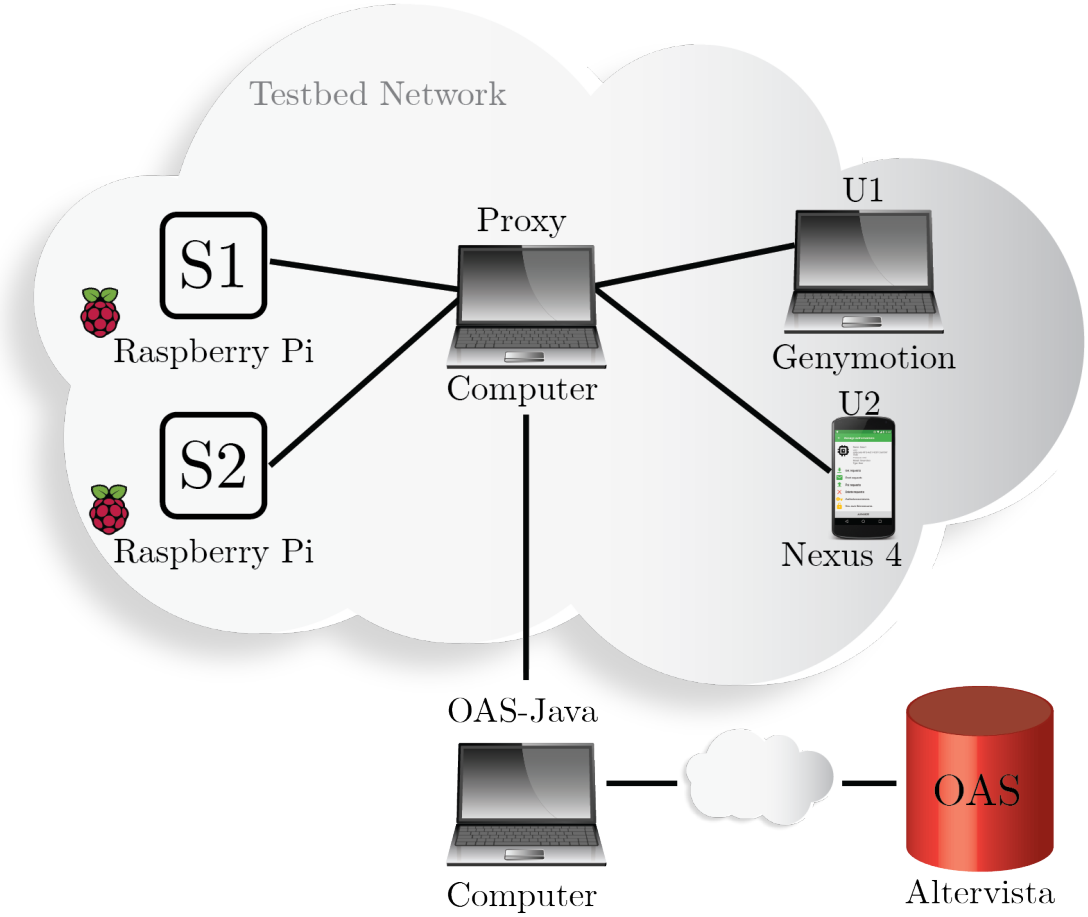


Figure 4.1: Testbed structure.

delegates to the IoT-OAS the handling of the authorization. The IoT-OAS can handle this requests from the proxy and also gives the programmer simple APIs to manage the permissions. The mobile application can manage permissions using this APIs and can perform requests to the nodes through the proxy. So, the system provides a working security framework to integrate in IoT scenarios, giving the end user simple and usable tools to manipulate it. This is an implementation of [12] and also, with the usage of an intercepting unconstrained proxy, the power consumption considerations are not issues anymore. The only issue of the system is the increased time in the requests: being composed by more elements and not using

a direct communication, the time for a client to receive a response increases. Then, the overall performance of the system are worsened. This could be a relevant factor in choosing wether or not to implement this system: if the system needs to be very responsive and gather data fast, another system should be taken in account. If the time of response is not an issue, this system provides a full set of functionalities. There are a lot of things that could be done to extend this project: first of all, in the application, the proxy is selected in a static way. The first thing to do is to implement a service discovery to find the proxy dynamically. A further extension could be the connection to a generic proxy, not in the same network of the device, in order to use devices remotely. This approach requires an external server that tracks all the proxies available and all the resources of every proxy. In this way, a moving object that changes network, could easily be found by the client. Another thing to implement could be the integration of third-party application instead of the generic REST client. For example, if the discovered device is a smart light, the click on the device should open the dedicated application and control the device from it. But all the authorization needed for it to work would still be managed from the project application. A last thing to implement, still in the application, could be a dynamic request of authorization not using QRcode read physically: the user that wants to use a device he does not own, sends a request from the application. The Resource Owner receives a notification and decides to authorize or not the other user. In this way there is no need for the users to be together, and all the management could be done remotely.

Bibliography

[1] ITU, "Overview of the internet of things - itu-t y.2060," 2012. [Online]. Available: http://www.itu.int/rec/T-REC-Y.2060-201206-I

[2] ETSI, "Machine-to-Machine communications," ETSI, Tech. Rep., 2013. [Online]. Available: http://www.etsi.org/deliver/etsi_tr/102700_102799/102725/01.01.01_60/tr_102725v010101p.pdf

[3] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252 (Proposed Standard), Internet Engineering Task Force, Jun. 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7252.txt

[4] L. R. L. Cisco, "Smart cities are built on the internet of things." [Online]. Available: http://www.cisco.com/web/solutions/trends/iot/docs/smart_cities_are_built_on_iot_lopez_research.pdf

[5] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswamia, "Internet of things (iot): A vision, architectural elements, and future directions." [Online]. Available: http://arxiv.org/pdf/1207.0203.pdf

[6] J. Postel, "Transmission control protocol," RFC 793 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168, 6093, 6528. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[7] ——, "User datagram protocol," RFC 768 (INTERNET STANDARD), Internet Engineering Task Force, Aug. 1980. [Online]. Available: http://www.ietf.org/rfc/rfc768.txt

[8] J. Hui and P. Thubert, "Compression format for ipv6 datagrams over ieee 802.15.4-based networks," RFC 6282 (Proposed Standard), Internet Engineering Task Force, Sep. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6282.txt

[9] R. Fielding, "Architectural styles and the design of network-based software architectures," 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[10] T. Hardjono, "User-managed access (uma) profile of oauth 2.0," IETF draft, April 2015. [Online]. Available: https://tools.ietf.org/html/draft-hardjono-oauth-umacore-13

[11] H. Tschofenig and E. Maler, "Authentication and authorization for constrained environments using oauth and uma," IETF draft, Mar 2015. [Online]. Available: https://tools.ietf.org/html/draft-maler-ace-oauth-uma-00

[12] S. Cirani, M. Picone, P. Gonizzi, L. Veltri, and G. Ferrari, "Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios," Feb 2015.

[13] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[14] E. Rescorla and N. Modadugu, "Datagram transport layer security version 1.2," RFC 6347 (Proposed Standard), Internet Engineering Task Force, Jan. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6347.txt

[15] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication," RFC 2104 (Informational), Internet Engineering Task Force, Feb. 1997, updated by RFC 6151. [Online]. Available: http://www.ietf.org/rfc/rfc2104.txt

[16] M. Kovatsch, M. Lanter, and Z. Shelby, "Californium: Scalable cloud services for the internet of things with coap," 2014. [Online]. Available: http://www.vs.inf.ethz.ch/publ/papers/mkovatsc-2014-iot-californium.pdf

[17] *CodeIgniter User Guide*, 2015. [Online]. Available: http://www.codeigniter.com/user_guide/

[18] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks," RFC 7228 (Informational), Internet Engineering Task Force, May 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7228.txt