Computer Engineering degree course

Master Degree Thesis

Design and Implementation of WebSocket API for the Dog Gateway

Supervisors: Fulvio Corno, Luigi De Russis **Candidate**: Teodoro Montanaro

March, 2014

Home automation refers to the use of computers and information technologies to manage home appliances and features (such as windows or lighting) with the aim of improving the quality of people's life.

As the number of controllable devices in the home rises, interconnection and communication become useful and desirable requirements. Nevertheless, the popularity of home automation has been increased only in recent years due to some critical issues, that nowadays continue to affect the home automation world, even if with less impact than before.

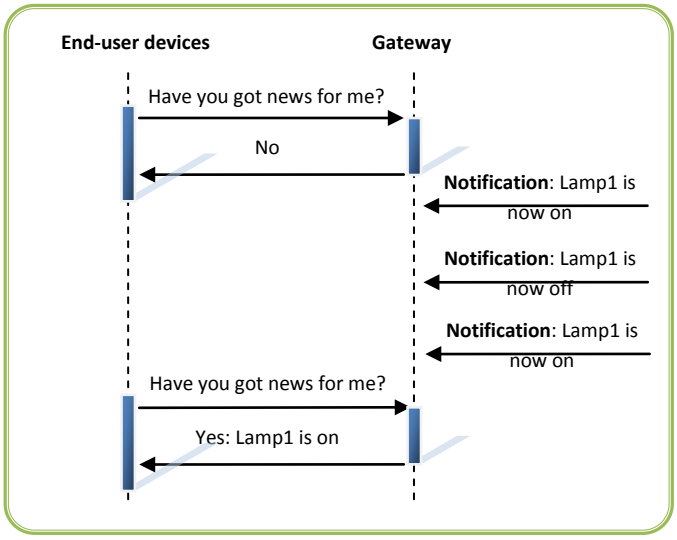

Fig. 1: How does polling work

The most important problem, actually felt by end-users and developers, is that does not exist a standard protocol for communications between devices and their main control system (called gateway). The gateway is responsible for retrieving information about the environment and interacting with any end-user device that uses it to get data and/or to send command requests.
In addition, most of existing gateways provide the so-called "polling" as method to exchange information with end-user devices connected to them.
As shown in Figure 1, this technique consists of actively and repeatedly sampling updated information from the gateway to know if something has changed after the last acquisition.

Let us analyze the drawbacks of this technique:

- considering that updates are taken every a well defined number of seconds, we cannot realize if something happens between two samples (e.g., as shown in Fig. 1, a rapid state change from on to off and then, again to on), causing a possible information loss;
- the connection is synchronous, so, considering that information are sent only when they are required and in order, it is not possible to send data (e.g., a notification) to an hypothetical end-user device when needed;
- the possibility of an overhead exists and may affect response time of the whole system; in fact, every time we need an information, a new connection must be established, so the time for starting it has to be added to the time actually needed for the request.

Goals

Considering the requirements and problems just described, this thesis aims at designing and implementing a solution to provide an asynchronous full duplex connection to existing domotics environment.

This solution will improve the exchange of information, reducing the time needed to get/put information from/to the gateway, and adding the possibility to send asynchronous data from the gateway to the end-user devices.
Specifically, we aim at providing an Application Programming Interface (API) implementing such a solution on the gateway developed at the Politecnico di Torino by the e-Lite research group [2]: Dog, a software gateway based on the OSGi framework, able to control smart environments in a vendor-independent way.

Entering in the details of specific goals pursued in this thesis, the first objective concerns the research of a cutting-edge and standard Web technology to provide an asynchronous ongoing and full duplex connection over the Internet. It should be a standard protocol letting any gateway implement a bidirectional communication to exchange information with all possible end-user devices.
Secondly, this thesis aims at designing all the features needed to implement an API that supports the chosen protocol.
After all, we will actually create the API by implementing the designed features so that other developers can use the Dog gateway through the chosen protocol.

Finally, a simple application to test and monitor the performances of such API will be developed. Moreover, a comparison with the existing REST API in terms of delay time and efficiency will be performed.

Design

The first step of this work concerned the choice of a protocol that is able to provide an asynchronous and full duplex communication between end-user devices and a gateway. After a depth analysis of existing technologies and protocols, we have chosen the WebSocket protocol for several reasons:

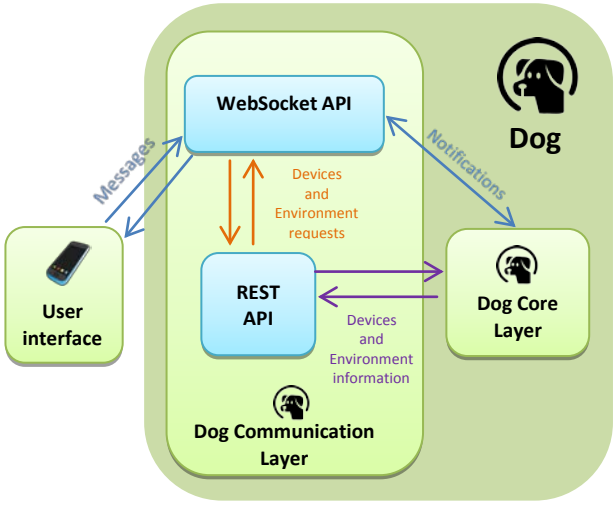

Fig. 2: How WebSocket API interacts with other system components

1. it is easily integrable in an OSGi execution environment, the framework used by Dog;
2. it lets the gateway send information (e.g., notifications) as soon as they are available, providing a full duplex communication;
3. it seems to be faster and more supported by browsers than its competitors.

After this phase, we designed the message structure for all the requests and the responses that the WebSocket endpoint could generate and/or accept.

Then, in order to apply the *Don't Repeat Yourself* (D.R.Y.) principle, aiming at reducing repetition of information and code, we have designed the architecture shown in Fig. 2.

As we can see the WebSocket API was designed to:

- wrap all the existing REST API methods for all the devices interactions (get their information or status, execute commands …); it was realized by the use of metaprogramming, that is the ability of a program to inspect existing external code in order to invoke its methods or functions;
- communicate directly with other Dog components (e.g., the Dog Core Layer) for the actions related to notifications (register/unregister on a notification topic or send a notification).

Through metaprogramming, we use the existing REST API methods thanks to Jersey (a framework for developing RESTful Web Services in Java) annotations that uniquely identify them. Such annotations are the "Path" annotation, that represents the URL needed to call a specific method, and the "HTTP method" annotation that specifies the HTTP verb by which a method should be called.

Consequently, looking at the best format for our structure, we choose to implement a JSON formatted structure (a lightweight data-interchange format), since it is a widely diffuse and a standard “de facto” format.

Implementation

To implement the WebSocket API in Dog, we started modifying the existing REST API to let their methods reply with relevant HTTP status codes to each received request: it was necessary because when WebSocket API calls a method, we must receive significant information about the status of the request.

Then, we have implemented the method that actually does the invocation: using the information received from the user (the URL path and the HTTP verb needed to invoke the method) we get the class in which the needed method is located, and we call it.

In turn, we have implemented the methods responsible for the notification management: they handle the subscription and unsubscription of notifications, intercept events, get the information contained in them, create the proper JSON message and send the notification to the right subscribed users.

Afterward, we have written detailed documentation about the usage of the API, providing a Java example that shows how an application could use it.
To supply a reusable example, we have at first created a client API library that provides a method for each possible request accepted by the WebSocket endpoint. These methods are responsible for retrieving information received as arguments, create the corresponding JSON message and then send it to the endpoint.
Finally, we have implemented the graphical application using JavaFX (see Fig. 3).

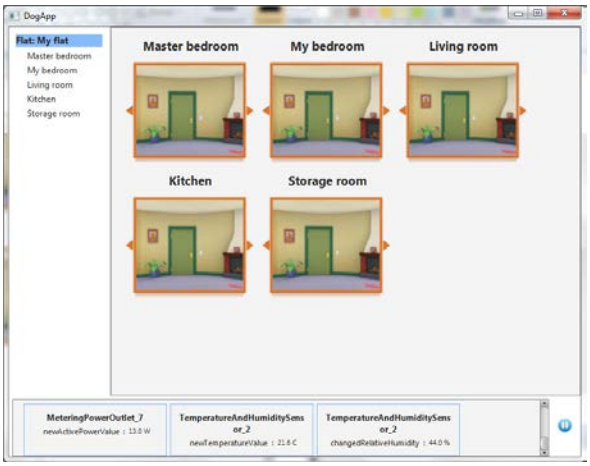

Fig. 3: Main view of the sample client application

Tests and conclusions

To test the performances of the implemented WebSocket API and to compare them with the performances achieved by the REST API, we have created some Java and JavaScript test applications that send all the possible requests in sequence to each endpoint and calculate the time required to return a response.

Request	REST average	WebSocket average
GET /api/v1/devices	33.38 ms	22.88 ms
PUT /api/v1/devices/MainsPowerOutlet_2/location	32.88 ms	14.25 ms
GET /api/v1/environment	83.13 ms	7.25 ms
POST /api/v1/environment/flats	30.38 ms	15.00 ms

Tab. 1: Comparison between some average time spent by the two implementation to reply to reported requests

Tab. 1 reports some of the requests sent to the two endpoint with the mean time needed to reply to them: as shown, the tests have demonstrated that the realized WebSocket endpoint is actually faster than the REST one. Moreover, they have demonstrated that the work done has actually respected all the requirements and so that we have reached all the goals of this thesis.