Gallo Massimiliano, #10031370

Jhonathan Jaramillo Munoz, #10032452

Reti 2, Sperimentazioni

Implementazione di un agente “antifurto” RESTfull

Si è scelto di realizzare un primo semplice prototipo di antifurto in rete.

Sensori ed attuatori sono forniti dal sistema HAT, installato sulla macchina “OSSO-4”. HAT fornisce un’interfaccia di tipo REST per il loro utilizzo in rete. Questa macchina utilizza l’hardware “beagle bone” per l’agente software HAT, ed una cape OSSO per interfacciare sensori ed attuatori.

Dunque, avendo OSSO4 a disposizione, il nostro sistema antifurto sarà composto da un agente software (a sua volta RESTfull) scritto in JAVA, che implementi la logica di un’antifurto sfruttando attuatori e sensori messi a disposizione dalla OSSO4 .

La comunicazione fra questi due agenti, essendo di tipo RESTfull, avverrà in rete, senza bisogno di bus o collegamenti particolari.

In particolare sfrutteremo messaggi di tipo http per comunicare i nostri “input” alla osso4, mentre per ascoltarne gli “output” sfrutteremo una web socket messa a disposizione dall’agente HAT stesso .

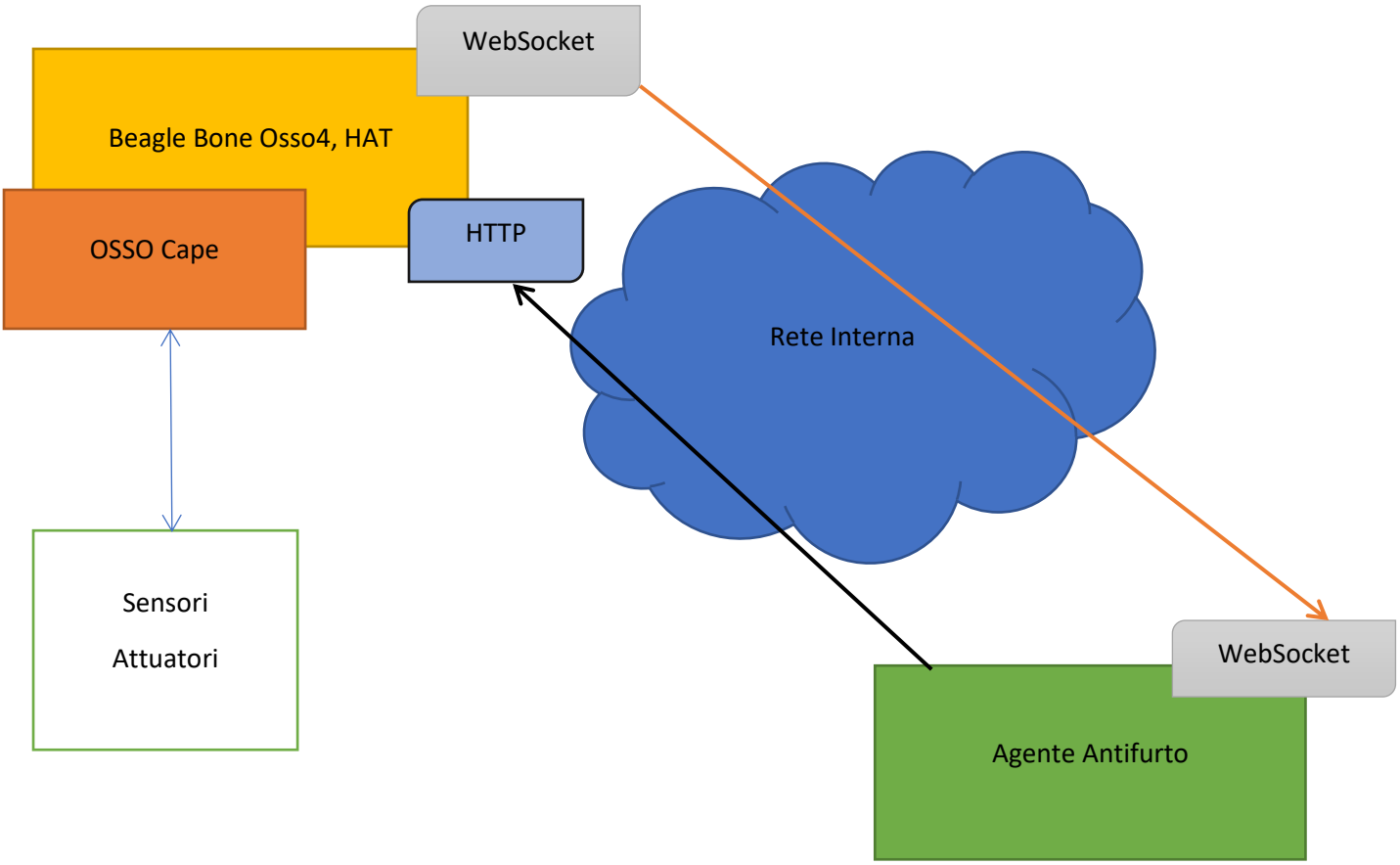


Figura 1: Architettura agente pt1

Tuttavia, come rendiamo possibile l’interazione del nostro agente con l’utente? In prima istanza abbiamo abbozzato una semplice interfaccia di tipo web, sfruttando un piccolo server http minimale “forgiato” sulle nostre esigenze. Tuttavia, in seguito abbiamo optato per l’uso del protocollo MQTT, tramite il brocker Mosquitto installato sulla macchina 193.206.55.23.

L'uso di MQTT ha diversi vantaggi rispetto all'uso di altri protocolli tipo http:

1) MQTT è estremamente leggero, molto più leggero rispetto ad http, e ben si adatta anche a reti lente
2) Offre il modello subscribe-notify, lo stesso che useremo nel nostro agente.
3) Offre messaggi M2M in tempo reale, non è più necessario fare polling della pagina! (a differenza delle classiche pagine html)
4) Offre un servizio di QOS
5) Risolve in maniera elegante il problema della comunicazione verso reti esterne, utilizzando MQTT over websocket. alcuni broker, come mosquitto offrono anche un servizio di web server.

   Il normale http server dell'agente è raggiungibile solamente dalla rete interna, una soluzione per raggiungerlo dall'esterno potrebbe esser stato un tunnel ssh, soluzione decisamente meno elegante e più macchinosa dal punto di vista dell'utente finale.

Il nostro agente fornisce un client web scritto in javascript in grado di dialogare con l'agente tramite il protocollo MQTT over websocket, scritto con la libreria Eclipse PAHO. L'agente invece possiede un thread "MyMQTTClient Thread" che funge da "gateway" fra il modello subscribe-notify proprietario del nostro agente ed MQTT .

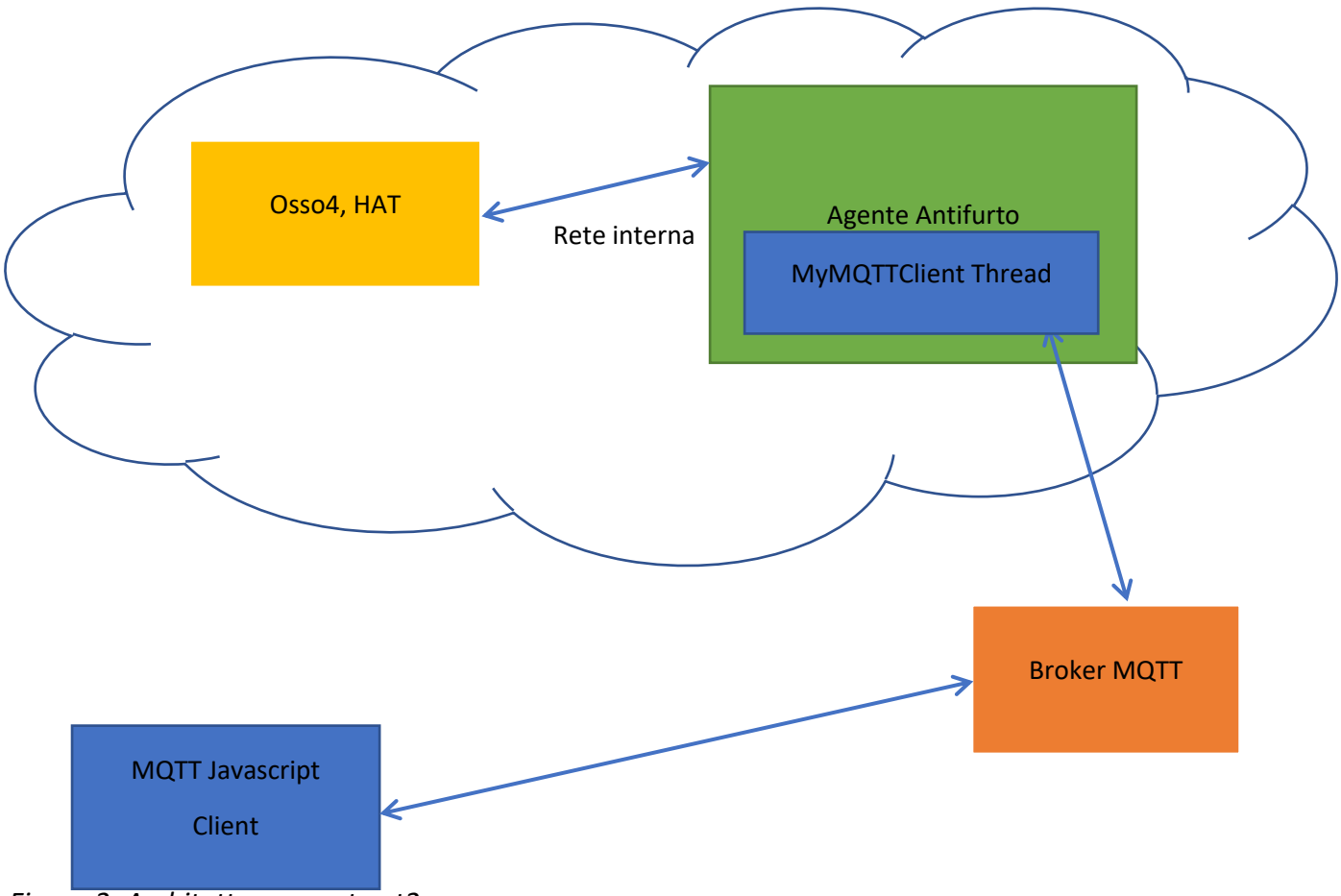


Figura 2: Architettura agente pt2

Nel seguito vedremo una rapida descrizione delle classi che compongono il nostro agente software.

Il cuore dell'agente, la subject table

Consente lo scambio di messaggi REST tra le varie thread e daemons che compongono il nostro agente, con un modello del tipo subscribe-notify. Si noti che la sottoscrizione ad un certo servizio, esempio "/devices" sottointende la sottoscrizione automatica a tutti i figli, esattamente come la wildcard "#" di mosquitto.

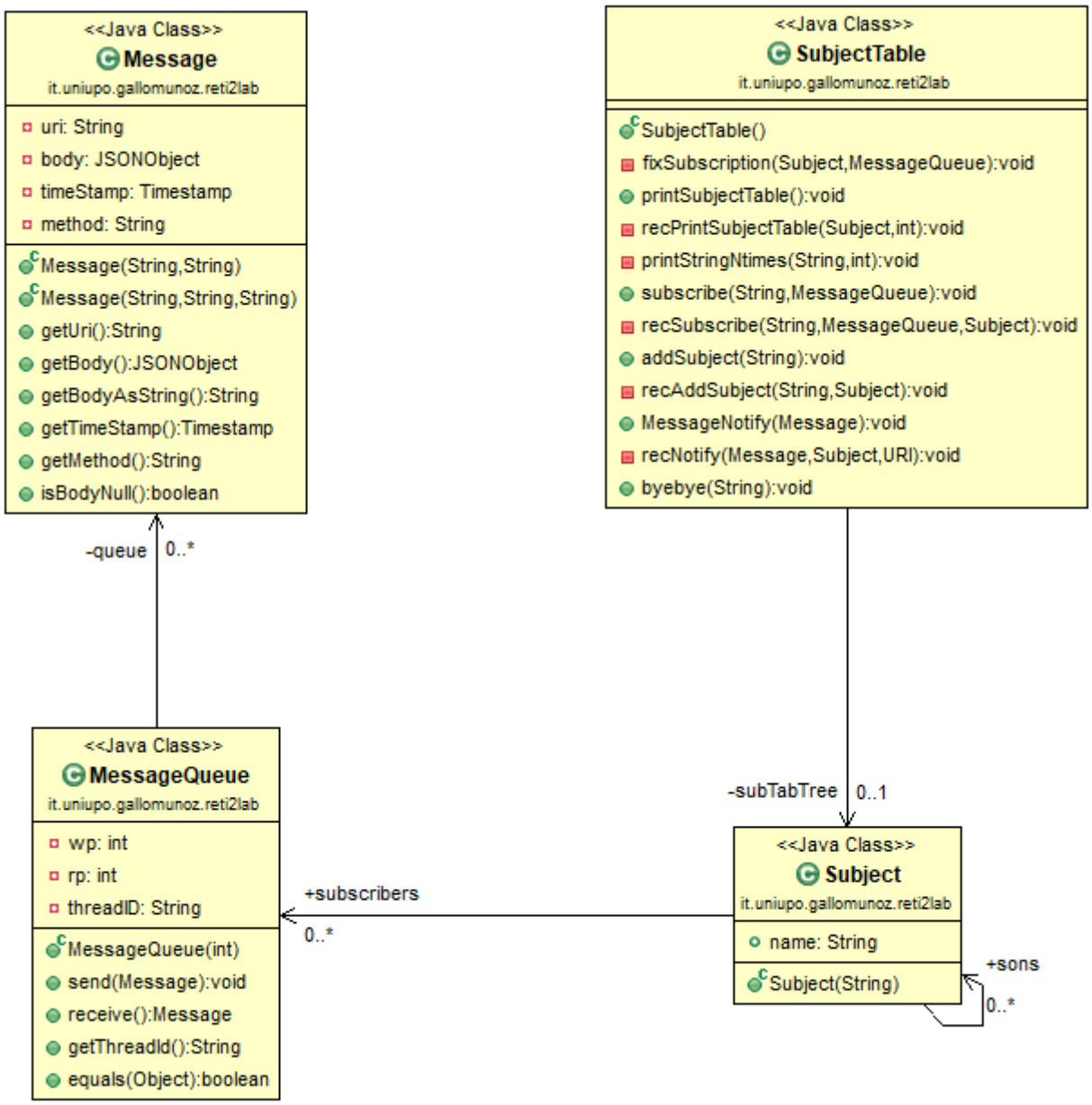


Figura 3: Classi Subject Table

Message rappresenta il messaggio scambiato.

MessageQueue è una coda circolare di messaggi, rappresenta la "mailbox" di un certo thread.

Subject rappresenta un soggetto della SubjectTable, la relativa classe implementa la tavola dei soggetti con una struttura dati ad albero "n-ario" dove i vari nodi sono soggetti. I vari metodi che esplorano l'albero posseggono la relativa routine ricorsiva per l'esplorazione dell'albero. Ogni subject contiene una "lista di mailbox", quando avviene una notify su un certo subject, il messaggio viene pubblicato su ogni mailbox della lista. Quando una thread si vuole sottoscrivere sarà sufficiente aggiungere la propria mailbox alla lista di tale subject.

La comunicazione dell'agente, WebServer

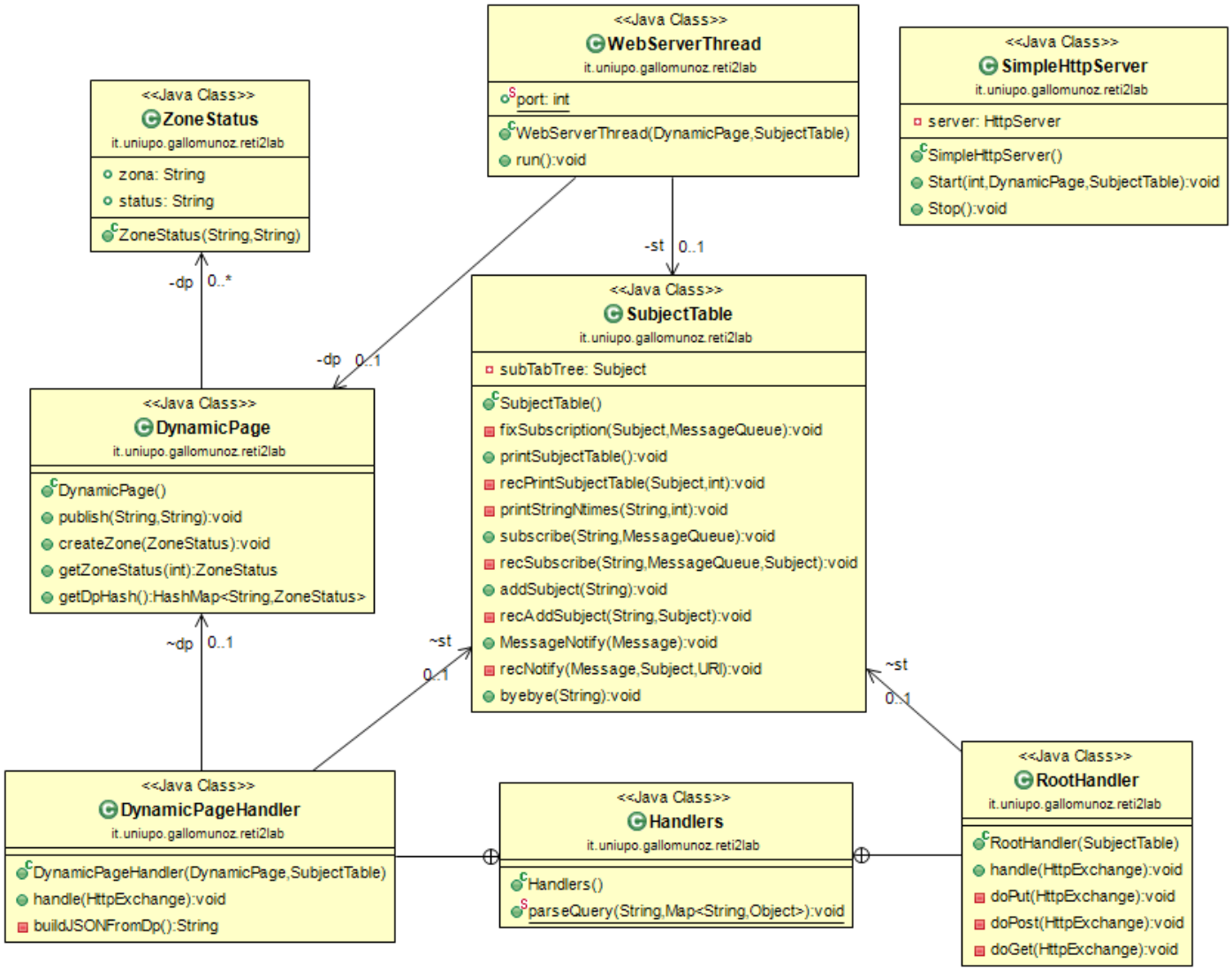


Figura 4: Classi web server

E possibile vedere la figura nel file fig4.png visto le dimensioni.

La classe Handlers contiene delle classi di tipo “HttpHandler”, sono gli handler del server http.

La RootHandler si comporta come semplice server http: una GET http su una certa risorsa otterrà come risposta la risorsa stessa. Attenzione: Questa GET lavora solo su file testuali, non c’è l’ handler per le immagini . Avendo ripiegato su MQTT non abbiamo continuato ulteriormente il lavoro sul server web.

La DynamicPageHandler ci costruisce un json con lo stato delle zone nel nostro sistema . Accetta, inoltre le chiamate PUT sulle zone (si veda più avanti l’interfaccia REST dell’agente) .

La DynamicPage è una struttura dati che rappresenta l’insieme delle zone disponibili in un certo momento nel nostro sistema, utile per generare dinamicamente pagine contenenti lo stato delle zone. Viene opportunamente aggiornata dai thread dell’agente, come mostra la fig. 5

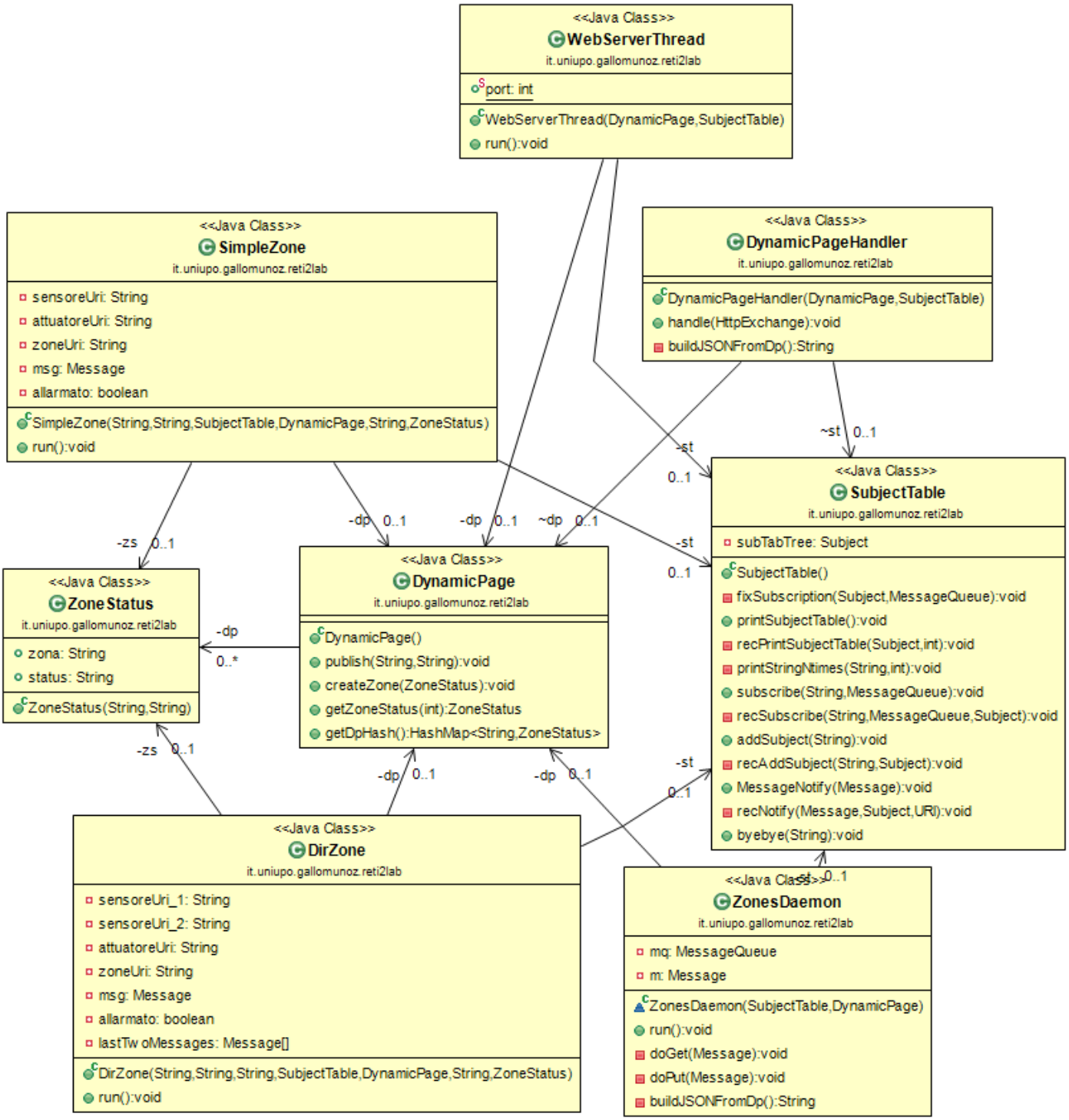


Figura 5: Classi Dynamic Page

Ad esempio la classe SimpleZone, che come vedremo descrive una zona semplice con singolo sensore e singolo attuatore, si appoggia a DynPage direttamente. (Opera tramite la struttura dati stessa, volendo fare un modello più RESTfull si potrebbe creare una thread per la dyn page e gestire la cosa tramite messaggi).

Dunque, dall'interno della rete è possibile fare una chiamata HTPP di tipo GET su "/zones" ed ottenere una risposta JSON di questo genere :

```
{"zones":[{"/zones/z1":"off"},{"/zones/z2":"off"},{"/zones/z3":"off"},{"/zones/z4":"off"}]}
```

La chiave "zones" identifica un array di zone. Come vedremo per fare la stessa cosa col nostro modello subscribe-notify ed/o MQTT avremo un demone preposto a tale scopo ….

La comunicazione dell'agente, MQTT

Per gestire la comunicazione tramite MQTT, useremo un daemon apposito, che iscritto a "/" nel nostro modello subscribe-notify proprietario, riceverà qualsiasi messaggio, lo tradurrà in un messaggio MQTT e lo spedirà al broker. Inoltre questa thread è adibita a ricevere qualsiasi messaggio in arrivo dai client (via broker), quando ne riceverà uno scatterà la callback, assemblerà un nuovo messaggio del nostro modello proprietario a partire dal messaggio MQTT e notificherà la nostra SubjectTable.

Inoltre, per evitare "cicli" è stato necessario usare 2 topic distinti sul broker :

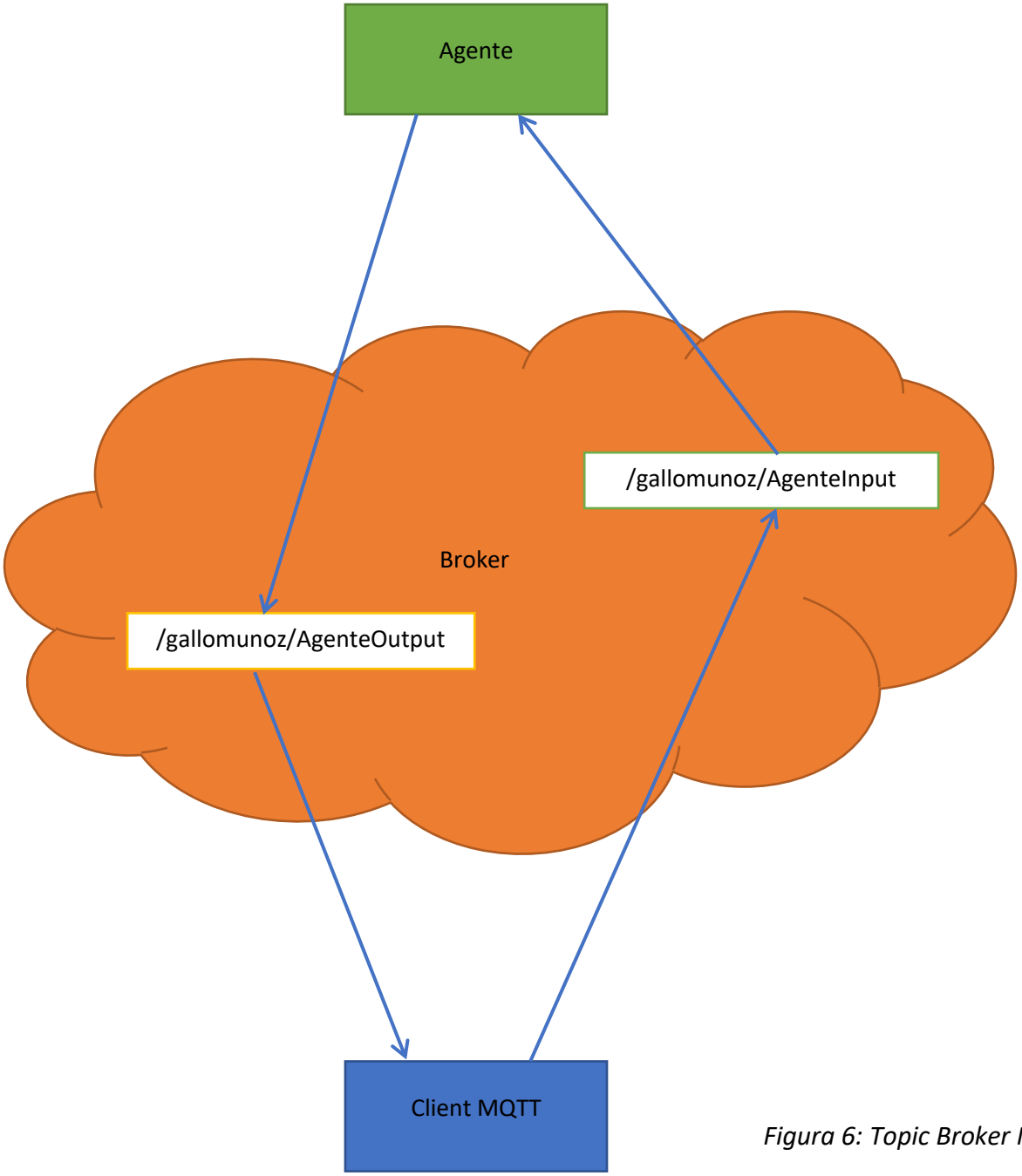


Figura 6: Topic Broker MQTT

Un altro problema da risolvere è stato quello di più client consecutivi: Problema in realtà di facile soluzione, è stato sufficiente estrarre a sorte una parte del client ID. Con 16 caratteri estratti a caso, la possibilità di collisione è estremamente bassa.

```
// Create a client instance
client = new Paho.MQTT.Client("193.206.55.23",
                              Number(9001),
                              "AntifurtoClientID" + randomString(16,
                              '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'));
```

Dal lato agente, come descritto sopra, la comunicazione tramite MQTT è gestita da un'apposito demone:

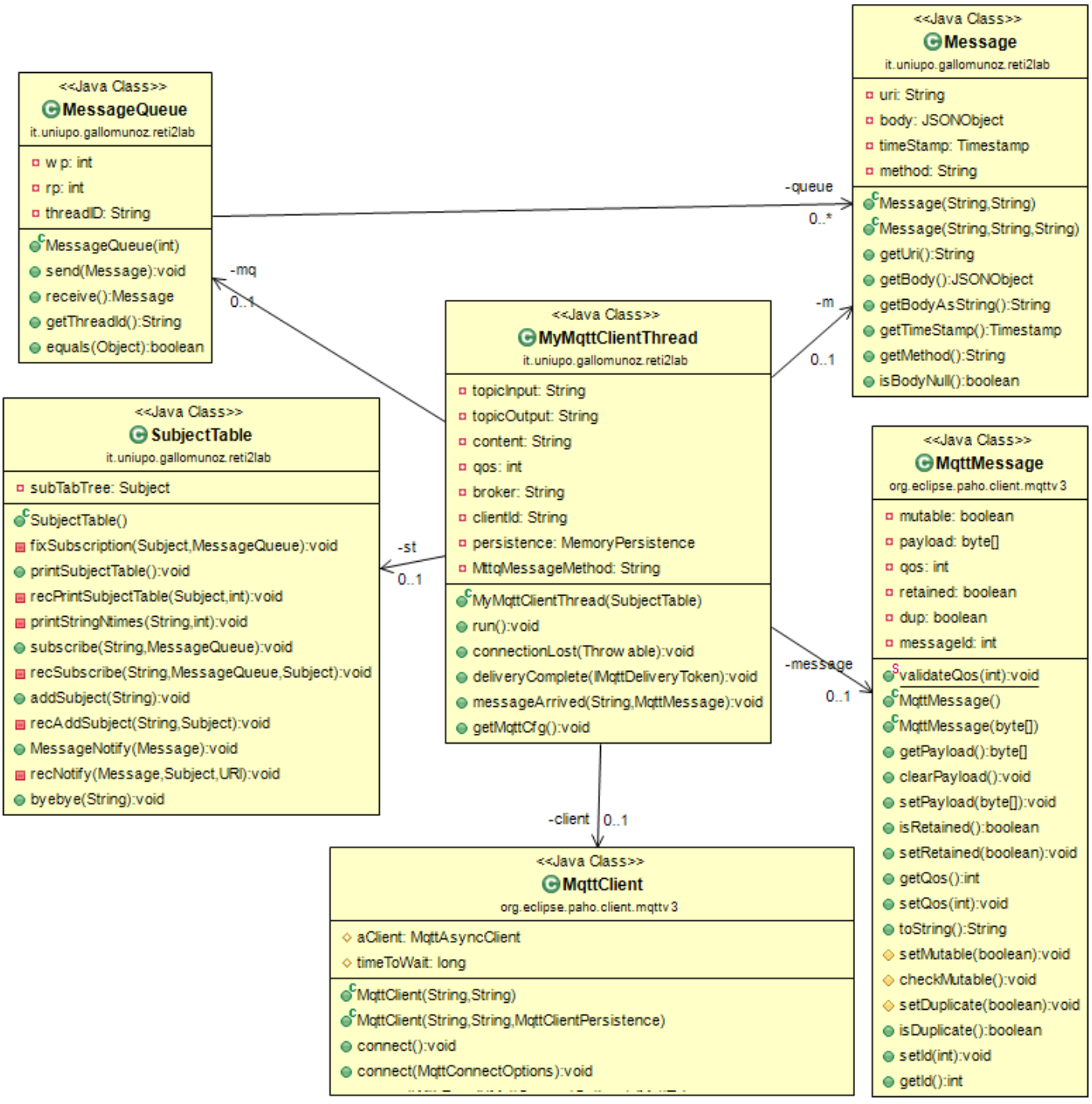


Figura 7: Classi MQTT

NB: abbiamo "tagliato" qualche metodo della classe MqttClient per ragioni di spazio .

Di sicuro interesse è vedere il codice della classe MyMqttClientThread che esegue l'interfacciamento tra MQTT ed il nostro modello subscribe-notify:

Ascolto SubjectTable ed assemblaggio messaggi MQTT:

```
/*
 * mi sottoscrivo a / nella subjectable del nostro agente
 * quando ricevo un messaggio lo inoltro al broker mqtt
 */

 mq = new MessageQueue(100);
 try {
       st.subscribe("/", mq);
       } catch (Exception e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
       }

      while(true){
              m = mq.receive();
              message = new MqttMessage(m.getBodyAsString().getBytes());
              try {
                     client.publish(topicOutput + m.getUri(), message);
                     System.out.println("MQTT: publish sul topic " + topicOutput + m.getUri() +
                     " del messaggio" + m.getBodyAsString());
              } catch (MqttPersistenceException e) {
                     // TODO Auto-generated catch block
                     e.printStackTrace();
              } catch (MqttException e) {
                     // TODO Auto-generated catch block
                     e.printStackTrace();
              }
     }
```

Ascolto messaggi MQTT ed assemblaggio messaggi verso la nostra subject table:

```
public void messageArrived(String topic, MqttMessage message) throws Exception {
        System.out.println("[ " + Thread.currentThread().getName() + " ]"  + "topic: " +
topic);
        System.out.println("[ " + Thread.currentThread().getName() + " ]" + "message: " + new
String(message.getPayload()));

        /*
         * quando arriva un messaggio da mqtt dobbiamo assemblarlo in un nostro messaggio
         * della classe Message e mandarlo alla subject table, col giusto metodo
         * i messaggi che arrivano da mqtt devono  possedere il campo "method", e quindi
         * scegliere il giusto metodo REST
         */

        JSONObject payloadMqtt= new JSONObject(new String(message.getPayload()));
        MttqMessageMethod = payloadMqtt.getString("method");

        //a questo punto non ci serve piu il campo method nel json
        payloadMqtt.remove("method");



        Message m = new Message(("/" +
URI.create(topicInput).relativize(URI.create(topic))).toString(),
                                              MttqMessageMethod,
                                              payloadMqtt.toString());
        st.MessageNotify(m);
    }
```

La comunicazione dell'agente, WebSocket Client

L'agente utilizza un client WebSocket per ascoltare i messaggi in arrivo dal sistema HAT, abbiamo utilizzato il codice fornito dal docente a lezione (WSClient) opportunamente "threaddizzato":

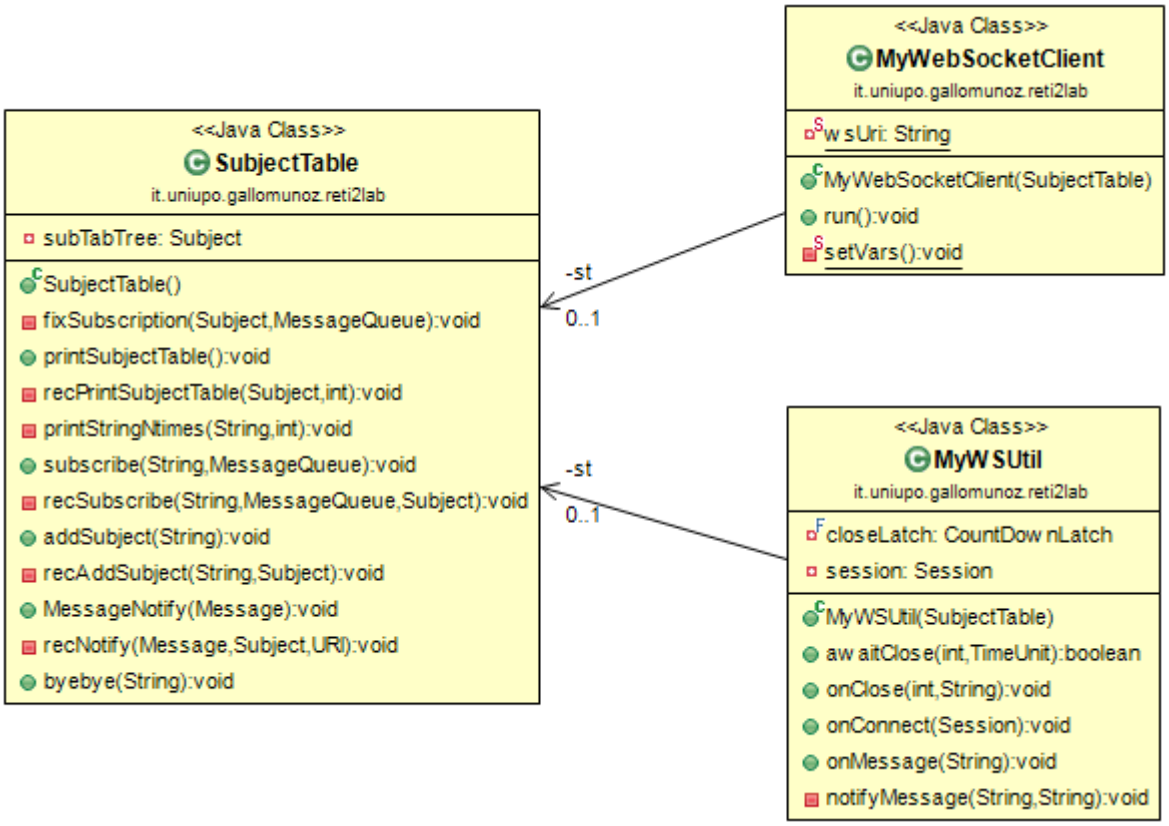


Figura 8: Classi WSClient

In particolare ecco la sezione di codice che preleva i messaggi dalla websocket e li pubblica sulla subject table:

```
@OnWebSocketMessage
	public void onMessage(String msg) {
		System.out.println("[" + Thread.currentThread().getName() + "]" + " received: " +
msg);

		//preparo un messaggio e lo notifico alla subject table
		JSONObject msgJson =  new JSONObject(msg);

		switch((String) msgJson.get("name")){
			case "SE_Pir1":
				notifyMessage("/sensors/s1", msg);
				break;

			case "SE_Pir2":
				notifyMessage("/sensors/s2", msg);
				break;

			case "SE_Pir3":
				notifyMessage("/sensors/s3", msg);
				break;

			case "SO_Lampeggiatore":
				notifyMessage("/attuators/lampeggiatore", msg);
				break;
```

```
                case "SO_Sirena":
                        notifyMessage("/attuators/sirena", msg);
                        break;

                default:
                        //nulla
                        break;
        }
}

private void notifyMessage(String uri, String msg){
        Message m = new Message(uri, "PUT", msg);
        try {
                st.MessageNotify(m);
        } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
}
```

Il sistema di allarme dell'Agente, Classi delle zone

La logica del nostro agente antifurto e racchiusa nelle threads di tipo "zone". Ogni zona è gestita da una singola thread in esecuzione sull'agente, con una propria interfaccia REST. In quanto Thread (o demoni) queste classi implementano l'interfaccia Runnable.

In particolare abbiamo

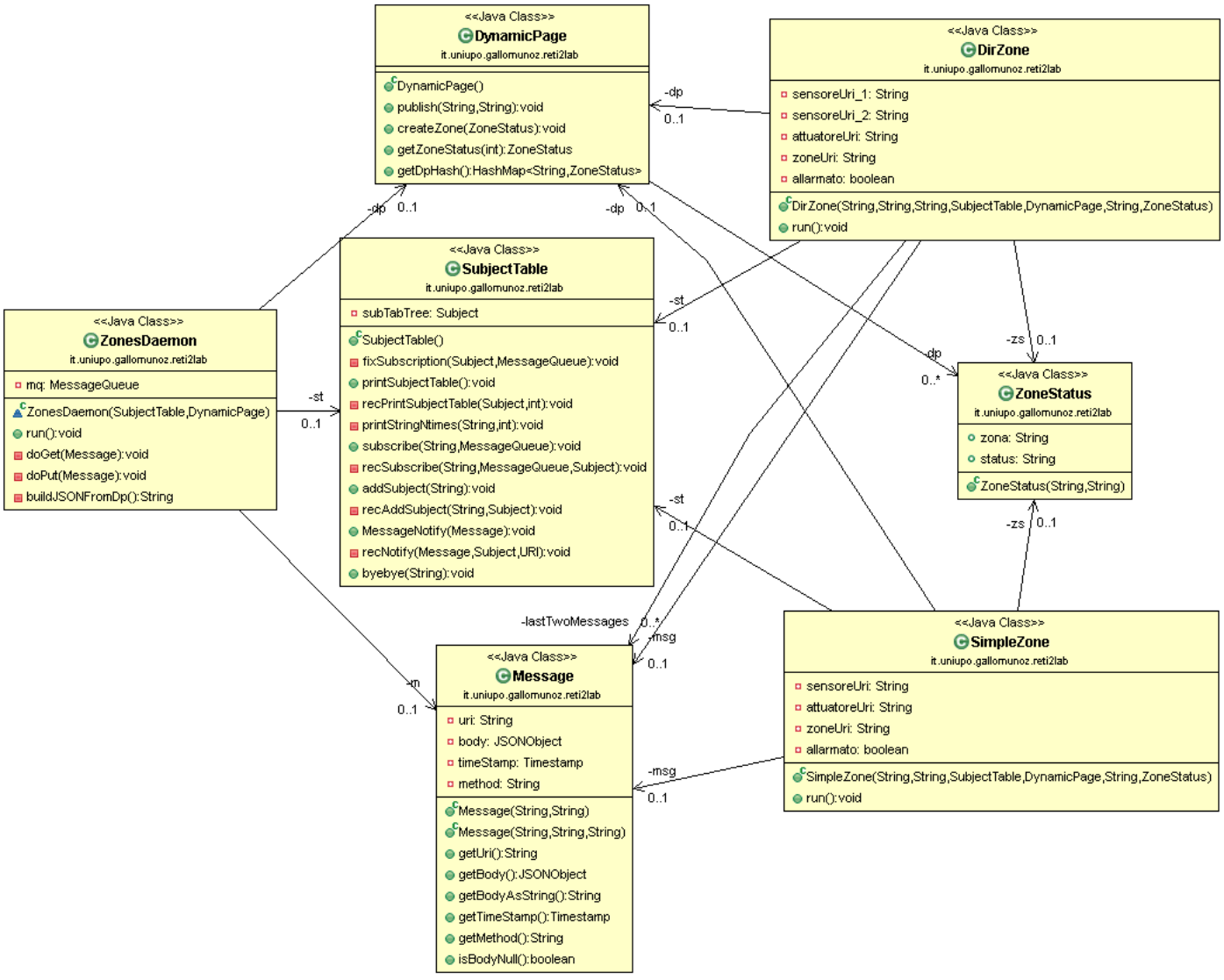


Figura 9: Classi delle zone

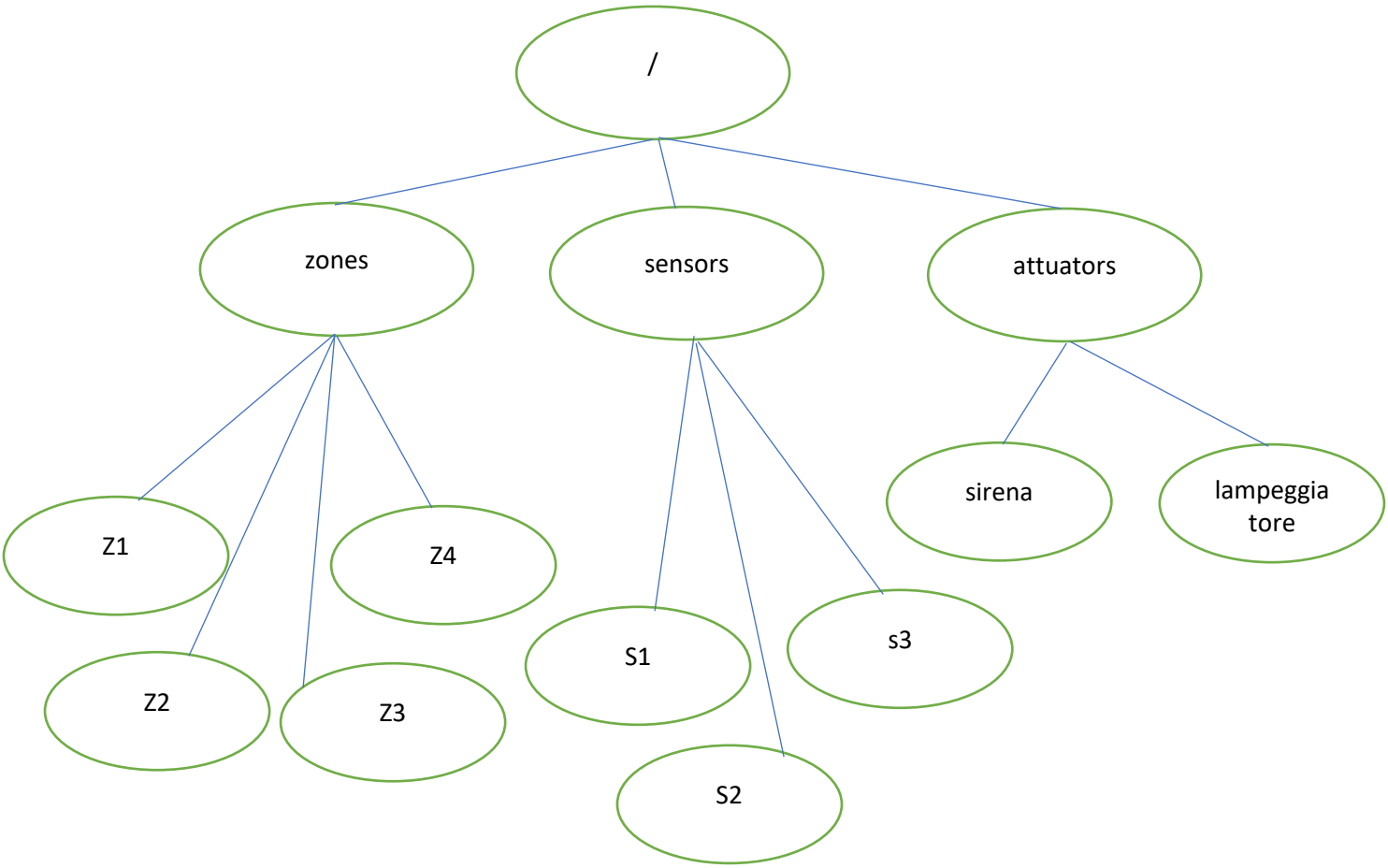
Classe ZonesDaemon : questo demone ascolta i messaggi indirizzati a tutte le zone, e li propaga sulle thread delle zone: in particolare è utile per allarmare e disallarmare tutte le zone.

Classe SimpleZone: Questa classe gestisce una semplice zona di antifurto con annessi un singolo sensore ed un singolo attuatore, specificabili nel costruttore. L'attuatore viene fatto scattare quando arriva il messaggio di "attivazione" da parte di un sensore.

Classe DirZone: Questa classe è una demo per dimostrare la possibilità di realizzare zone di antifurto con logiche più complesse ed interessanti: nel merito, a questa zona sono annessi 2 sensori ed un attuatore (sempre specificabili da costruttore). Questo tipo di zona fa scattare l'attuatore se e solo se il sensore 1 scatta prima del sensore 2 e non viceversa. Un ulteriore aggiunta alla funzionalità di questa classe potrebbe essere verificare anche il tempo trascorso tra l'attuazione dei 2 sensori (tramite timestamp), per abbassare i falsi positivi.

Il sistema di allarme dell'Agente, Interfaccia REST

Il nostro agente si presenta con la seguente gerarchia:



Sensori ed attuatori di OSSO4 sono stati "rimappati" sull'interfaccia REST del nostro agente (e di cui, come vedremo sono responsabili le apposite thread).

Per quanto riguarda le zone l'interfaccia REST è la seguente:

➔ Allarma/Disallarma una zona specifica:

risorsa: /zones/x	Metodo: PUT	Corpo richiesta: {“allarmato”:”on/off”}

Questa chiamata REST allarma o disallarma una zona specifica oppure tutte le zone

Risposta: ottiene una risposta con corpo {“allarmato”:”on/off”,”risorsa”:”/zones/x”}, nel caso di chiamata alla risorsa /zones si riceverà una risposta per ogni singola zona.

➔ Attivazione di un’attuatore

risorsa: /attuators/y	Metodo: PUT	Corpo richiesta: {“accensione”:”true”}

Questa chiamata REST attiva un’attuatore (non è implementata su /attuators)

Risposta: ottiene una risposta direttamente dal sistema HAT, propagata dal nostro agente. Si veda la documentazione di HAT in tal proposito.

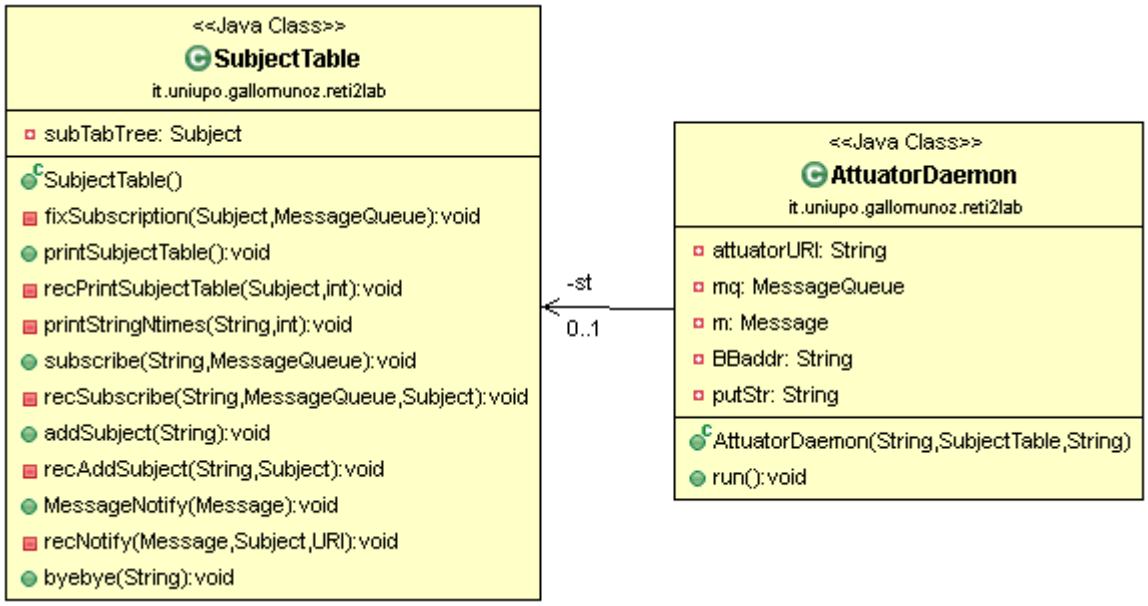
➔ Ottenere un riepilogo delle zone dalla dynamic page:

risorsa: /zones	Metodo: GET	Corpo richiesta: nessuno

Questa chiamata REST si comporta esattamente come la GET su /zones tramite HTTP Server

Risposta: La risposta è la stessa della GET su /zones tramite HTTP Server (una chiave zones con un array di zone)

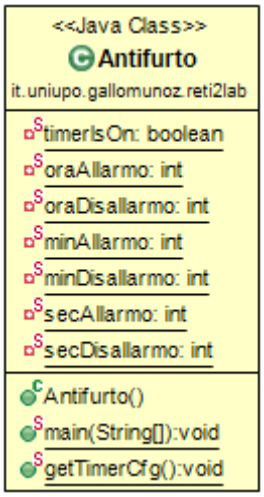
Il sistema di allarme dell’Agente, Classe per gli attuatori



AttuatorDaemon descrive una thread che sia responsabile per uno specifico attuatore. (Da definire nel costruttore).

Il sistema di allarme dell'Agente, La classe Principale

La classe Antifurto è responsabile per l'avviamento e la direzione di tutte le componenti prima descritte.



Questa classe, inoltre, gestisce la funzione di timer: ovvero l'allarmo e disallarmo di tutte le zone a precisati orari del giorno (configurabile dal file timer.cfg)

Come demo vengono istanziati 3 sensori (s1, s2, s3), e due attuatori (sirena e lampeggiatore)

Vengono avviate tutte le thread ed i demoni, in particolare:

3 zone di tipo SimpleZone (z1, z2, z3)

1 zona di tipo DirZone (z4)

HTTP server, WS Client, MQTT Client, Demone delle zone, Demoni per gli attuatori.

Viene infine stampata la subject table a video, utile per visualizzare la configurazione:

```
>  : [ MqttDaemon ]
> > updates :
> > sensors :
> > > s1 : [ zone4 ][ zone1 ]
> > > s2 : [ zone2 ][ zone4 ]
> > > s3 : [ zone3 ]
> > attuators :
> > > lampeggiatore : [ lampeggiatoreDaemon ]
> > > sirena : [ sirenaDaemon ]
> > zones : [ zones ]
> > > z3 : [ zone3 ]
> > > z1 : [ zone1 ]
> > > z2 : [ zone2 ]
> > > z4 : [ zone4 ]
```

Questa stampa è realizzata dal metodo printSubjectTable della classe SubjectTable, e permette di visualizzare quali thread sono in ascolto su una data risorsa.

Il sistema di allarme dell'Agente, Client MQTT JavaScript

Il client MQTT funziona con un sistema AJAX, che permette di visualizzare i messaggi in tempo reale, senza dover fare polling. Il javascript ha come punto di partenza la funzione start().

- Come prima cosa creiamo il client MQTT ed impostiamo le callback

```
// Create a client instance
	client = new Paho.MQTT.Client("193.206.55.23",
                                                        Number(9001),
                                                        "AntifurtoClientID" +
randomString(16, '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'));

	// set callback handlers
	client.onConnectionLost = onConnectionLost;
	client.onMessageArrived = onMessageArrived;

	// connect the client
	client.connect({onSuccess:onConnect});
```

- Poi eseguiamo una subscribe sul topic di output dell'agente:

```
client.subscribe("/GalloMunoz/Antifurto/AgenteOutput/#");
```

- Successivamente l'elenco delle zone è costruito dinamicamente, consultando l'elenco di queste ultime tramite una chiama REST di tipo GET su /zones.

```
/*
   * per prima cosa ci serve un elenco delle zone
   */
  message = new Paho.MQTT.Message("{\"method\":\"GET\"}");
  message.destinationName = "/GalloMunoz/Antifurto/AgenteInput/zones";
  client.send(message);
```

Quindi da questo momento l'apposita callback scatterà nel momento in cui arriverà un nuovo messaggio.

Ai pulsanti invece viene assegnata la giusta funzione.

Il sistema di allarme dell'Agente, Istruzioni di Utilizzo

E' sufficiente avviare il jar:

>java -jar antifurto.jar

E configurare i files di configurazione nel modo opportuno:

bbUrl.cfg : url del sistema HAT per comandi rest (porta 14990)

cfg.txt: url del sistema HAT per ascoltare la websocket (porta 14991)

mqttBroker.cfg:

prima riga : topic input

seconda riga: topic output

terza riga: indirizzo broker

quarta riga: ID dell'agente

timer.cfg:

orari di allarmo e disallarmo, si vedano i commenti per le istruzioni

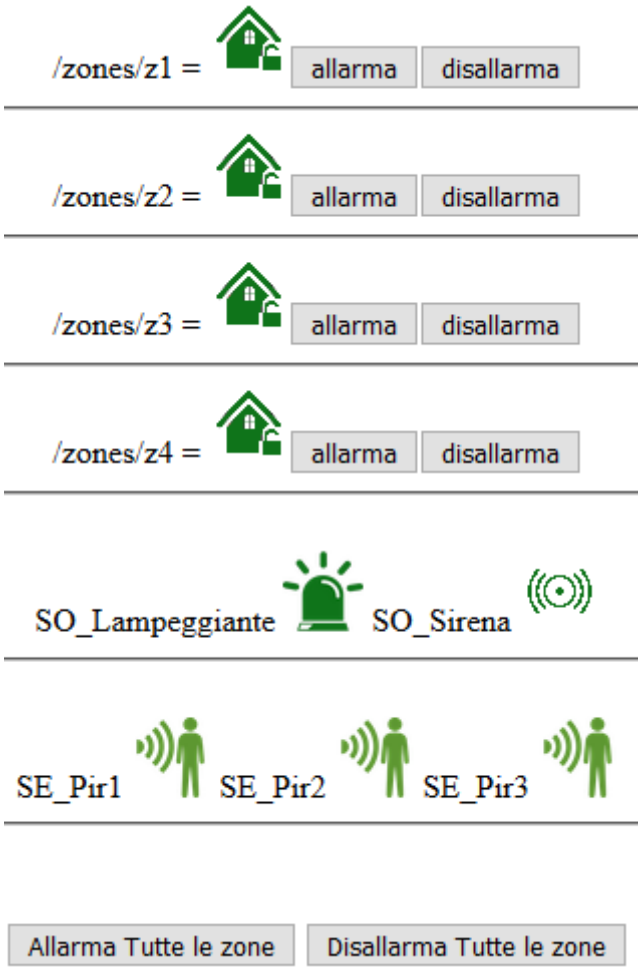
Per la parte client (JavaScript) è sufficiente caricare in un qualsiasi server web (o su file system) la directory "www" ed aprire col browser il file "/gallomunoz/antifurto.html".

Attenzione: il server web fornito con l'agente non serve le immagini, quindi allo stato attuale non può essere utilizzato per servire questa interfaccia, sarebbe però possibile con poco lavoro adattare il server web dell'agente per servirle e quindi usarlo per il client javascript.

Per comodità, il client è stato caricato ed è reperibile sul server del broker mosquitto:

http://193.206.55.23:9001/gallomunoz/antifurto.html

L'interfaccia è piuttosto intuitiva e non dovrebbe presentare difficoltà del suo utilizzo.



I sensori diventano rossi quando attivati, le zone diventano rosse quando allarmate (e con lucchetto chiuso).

Al presente progetto vengono allegati i seguenti file:

- src/ : contiene i sorgenti. In particolare l'antifurto è nel package it.uniupo.gallomunoz.reti2lab
- jar/ : contiene tutte le dipendenze per il progetto sotto forma di file *.jar
- relazione/ : questa relazione ed alcune figure
- www/ : il client web javascript
- bbUrl.cfg, cfg.txt, mqttBroker.cfg e timer.cfg : files di configurazione dell'agente
- agenteAntifurto.jar : l'agente sotto forma di jar eseguibile
- Ulteriori files prodotti da Eclipse lasciati per comodità

Conclusioni

Questo primo prototipo di agente REST ci ha permesso di comprendere più a fondo questo modello di programmazione, oltre a topic importanti come la programmazione in tempo reale e distribuita.

Ovviamente molte cose sono migliorabili tra cui:

- Estendere l'interfaccia REST ad altri servizi (es: dynamic page)
- Estendere le funzionalità del server http embedded
- Permettere la definizione e cancellazione di zone tramite chiamata REST
- Offrire nuove logiche di zona
- Migliorare i files di configurazione (un po sparpagliati)
- Servizio di timer assegnabile a singole zone tramite REST
- Un JavaDoc ben fatto
- Sicurezza: Sfruttare https, wss e certificati. Fornire login ed utenze.
- Un demone di "Logging".
- Ecc ...

Features non implementate in questo primo prototipo per esigenze di tempo.