Edoardo Monti
20016650@studenti.uniupo.it

Marco Sirianni
20016240@studenti.uniupo.it

Relazione sul progetto di Reti II Sperimentazioni

Indice

1. Specifiche funzionali che devono essere soddisfatte

Il modulo a noi commissionato si occupa di gestire il sistema di irrigazione di una serra con più colture aventi necessita` differenti. Nello specifico, il software deve leggere i valori rilevati dai sensori ubicati nelle serre, che inviano dati riguardanti l'umidità delle singole colture, della temperatura nella serra e dell'irraggiamento. Una volta raccolti i dati, il modulo si occupa di calcolare la quantità d'acqua dispersa e utilizzata dalla pianta in modo da poterla reintegrare in maniera ottimale. L'irrigazione verrà gestita automaticamente dal sistema, che andrà ad innaffiare in orari specifici. E` anche possibile gestire manualmente l'irrigazione attraverso un'interfaccia web, che esegue controlli sullo stato della serra ed eventualmente, prima di permettere all'utente di eseguire l'azione, lo avverte delle possibili controindicazioni. L'intera piattaforma dev'essere visibile anche all'esterno della rete privata in modo da poter controllare la propria serra comodamente anche quando non si è a casa.

2. Analisi delle scelte tecnologiche disponibili per realizzare le funzionalità richieste

Uno dei protocolli maggiormente utilizzati per la connessione dei device al Cloud è il protocollo HTTP attraverso l'architettura REST (REpresentation State Transfer). L'architettura REST si basa su HTTP; il funzionamento prevede una struttura degli URL ben definita (atta a identificare univocamente una risorsa o un insieme di risorse) e l'utilizzo dei verbi HTTP specifici per il recupero di informazioni (GET), per la modifica (POST, PUT, PATCH, DELETE) e per altri scopi (OPTIONS, ecc.).

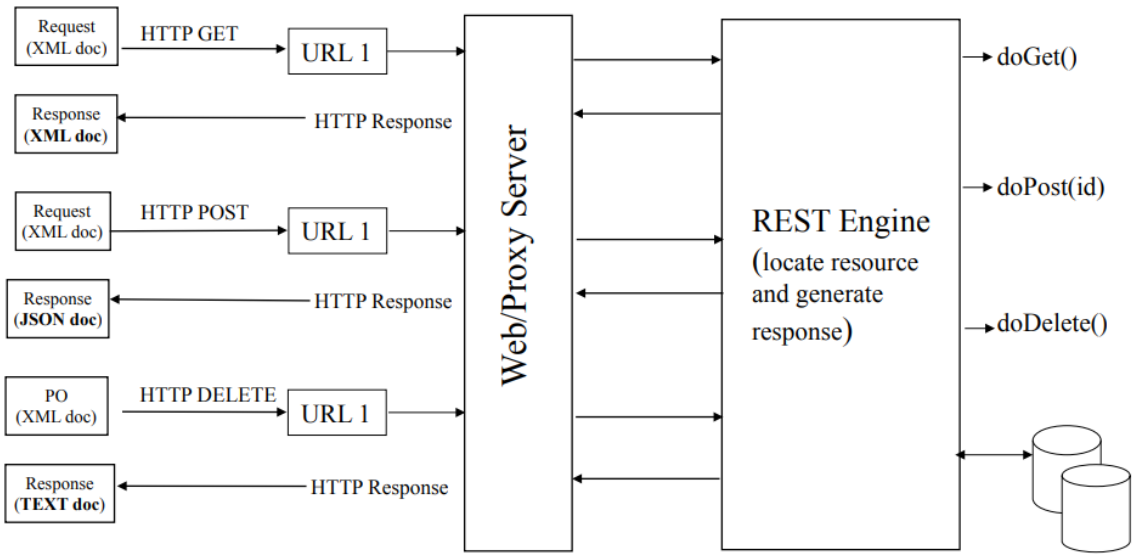
L'approccio architetturale REST è definito dai seguenti vincoli più altri applicati ad una architettura, mentre si lascia libera l'implementazione dei singoli componenti.

Il vincolo Client-Server richiede che il server offra una o più funzionalità e ascolti le richieste di possibili client. Un client invoca il servizio messo a disposizione dal server inviando il corrispondente messaggio di richiesta e il servizio lato server respinge la richiesta o esegue l'attività richiesta prima di inviare un messaggio di risposta al client. La gestione delle eccezioni è delegata al client.

La comunicazione client–server è ulteriormente vincolata in modo che nessun contesto client venga memorizzato sul server tra le richieste. Ogni richiesta da ogni client contiene tutte le informazioni necessarie per richiedere il servizio, e lo stato della sessione è contenuto sul client. Lo stato della sessione può anche essere trasferito al server attraverso un altro servizio posto a persistere, ad esempio un database.

Come nel World Wide Web, i client possono fare caching delle risposte. Le risposte devono in ogni modo definirsi implicitamente o esplicitamente cacheable o no, in modo da prevenire che i client possano riusare stati vecchi o dati errati. Una gestione ben fatta della cache può ridurre o parzialmente eliminare le comunicazioni client server, migliorando scalabilità e performance.

Uniform interface. Per avere un caching efficiente in una rete, i componenti devono essere in grado di comunicare tramite un'interfaccia uniforme. Con un'interfaccia uniforme, il carico utile può essere trasferito in un formato standard.

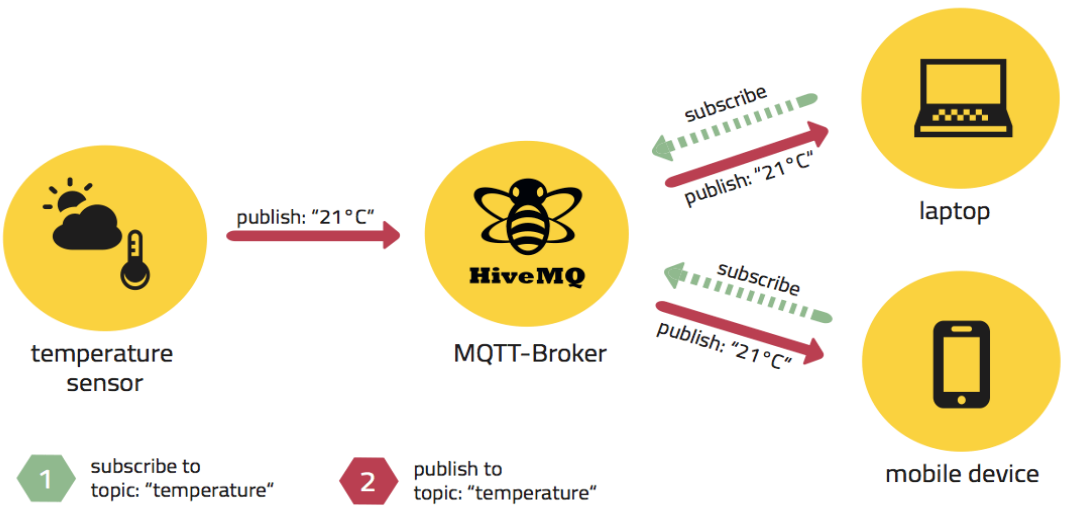


Un'altra architettura utilizzabile è quella del message broker. Il broker funge da intermediario tra i vari oggetti, che si collegano a lui in maniera asincrona mediante il design pattern publisher/subscriber.

Secondo questo pattern, il mittente di un messaggio (detto publisher) si rivolge al broker per "pubblicare" il proprio messaggio. I destinatari (subscriber) si rivolgono a loro volta al broker per "abbonarsi" alla ricezione di messaggi. Il broker inoltra ogni messaggio inviato da un publisher verso tutti i subscribers interessati a quel "tipo" (Topic) di messaggio. Come è facile intuire, quindi, non è necessario che un oggetto IoT interroghi ripetutamente il broker per sapere se ci sono nuovi messaggi da leggere (come avviene nella modalità pool): questi viene notificato automaticamente dal broker qualora ci sia un nuovo messaggio da consegnargli (cioè è una modalità push).

Il broker gestisce opportunamente un numero indefinito di code di messaggi per conto dei client remoti a lui collegato. Ciascuna coda di messaggi è contraddistinta da un nome, chiamato Topic cioè argomento, che possiamo considerare un po' come una categoria/sottocategoria di ciascun messaggio in quanto ogni Topic è identificato da una chiave gerarchica, volendo multilivello.

Grazie al message broker, comunque, i messaggi non si perdono mai per strada: si occupa lui di riceverli e di notificarli agli interessati, tramite il meccanismo delle code ed il funzionamento logico del pattern sopra descritto. Questo schema, davvero molto efficace, implica il fatto che ai publisher non sia noto quanti e quali siano i subscriber e viceversa, caratteristica che contribuisce alla scalabilità del sistema.



3. Scelta dell’approccio tecnologico

Basando questo progetto sull'IoT abbiamo la necessità che il traffico dati sia ridotto al minimo in maniera da non sovraccaricare le memorie dalla ridotta capacità dei device utilizzati, l'altra necessità che ci si pone di fronte è quella di ridurre al minimo il consumo energetico dei nostri device. Utilizzando device spesso con batteria integrata oppure posizionati in luoghi non facilmente accessibili è nostro interesse preservare la loro batteria da inutili sprechi, dato che la durata della batteria, spesso, è dettata dal tempo in cui il dispositivo risulta operativo, maggiore è il tempo in cui riusciamo a far rimanere dormiente il dispositivo maggiore sarà la longevità della nostra batteria.

Questo obiettivo viene raggiunto utilizzando un message broker, un'architettura che non richiede la costante connessione dei subscriber per verificare la ricezione di nuovi messaggi, ma essi vengono notificati dal broker riguardo a nuove pubblicazioni sui topic a cui loro sono iscritti, permettendo così ai subscriber di risparmiare energia tra una pubblicazione e l'altra.

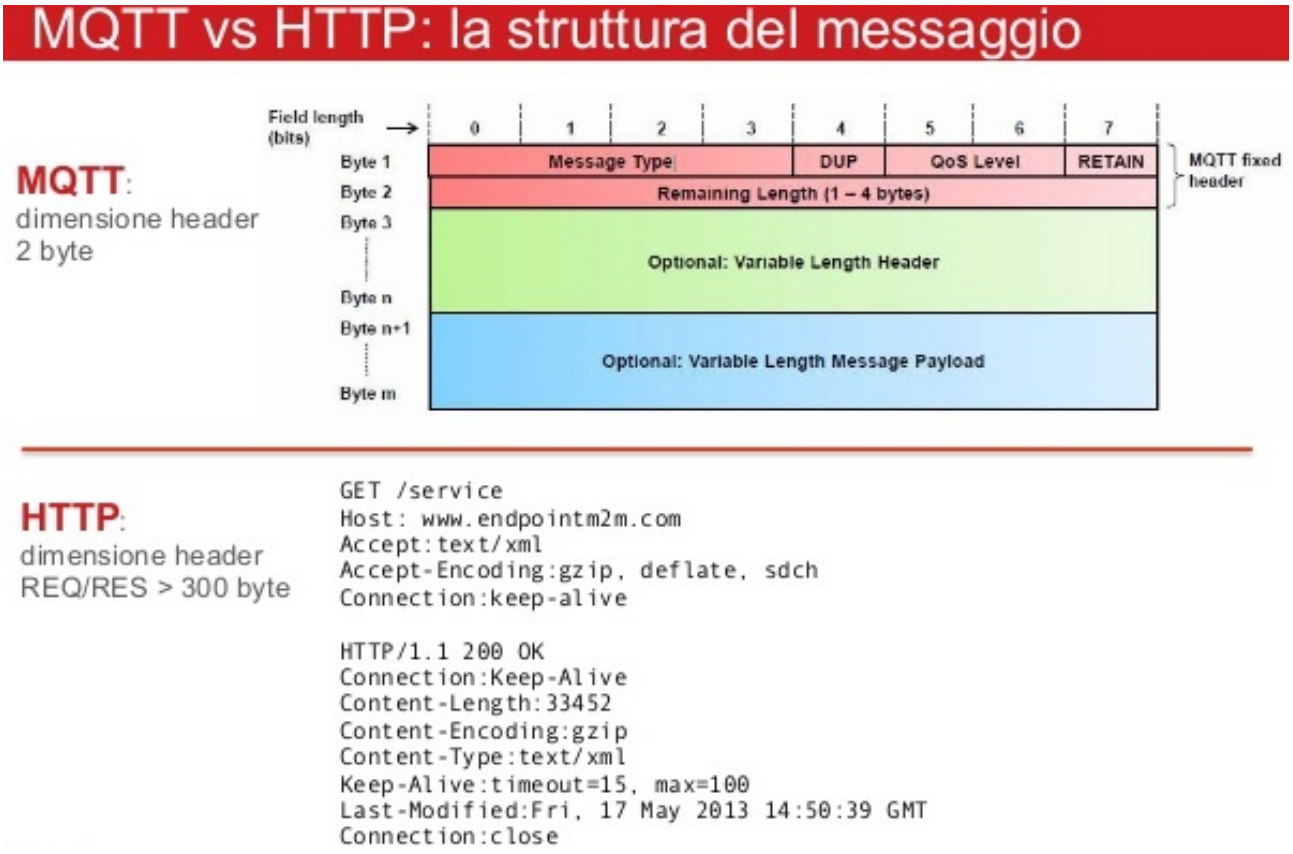
L’architettura del message broker utilizza, nel nostro caso, il protocollo MQTT che viene descritto come un protocollo di messaggistica estremamente leggero progettato per dispositivi limitati e reti a bassa larghezza di banda, alta latenza o sostanzialmente inaffidabili. I principi su cui si basa sono quelli di abbassare al minimo le esigenze in termini di ampiezza di banda e risorse mantenendo nel contempo una certa affidabilità e grado di certezza di invio e ricezione dei dati.

Caratteristica fondamentale è la possibilità di pubblicare/sottoscrivere pattern di messaggi per mettere a disposizione una distribuzione da uno a molti e un disaccoppiamento delle applicazioni. Quindi una tecnologia di trasporto dei messaggi agnostica rispetto al contenuto del payload, con l’uso nel contempo del protocollo TCP/IP, alla base di internet così come la conosciamo oggi, per fornire connettività di rete di base. Altre caratteristiche evidenziate di MQTT sono un overhead di trasporto ridotto, grazie a un header a lunghezza fissa da 2 byte, e scambi minimizzati per ridurre il traffico di rete e ridurre la quantità di dati da elaborare.

MQTT vs HTTP (long polling): prestazioni

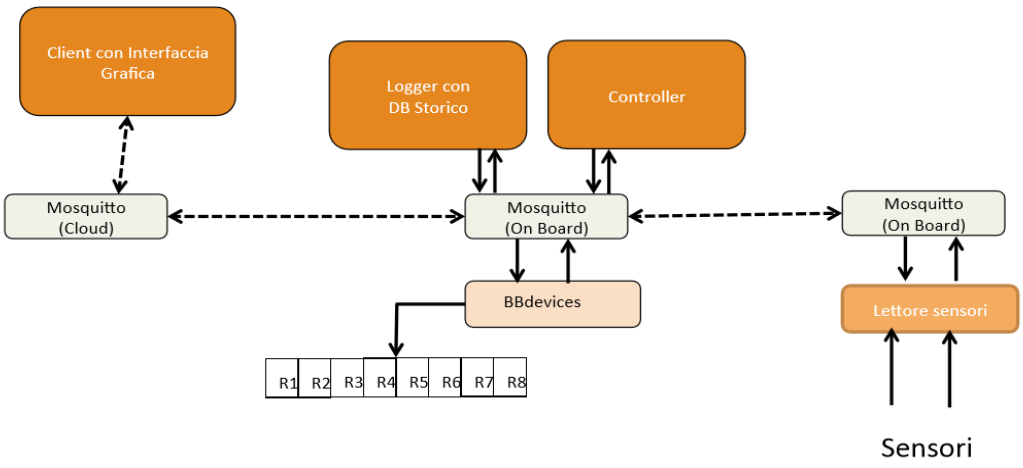
Test di spedizione (1024 messaggi con payload da 1 byte)

	3G		Wifi	
	HTTPS	MQTT	HTTPS	MQTT
% Battery / Hour	18.43%	16.13%	3.45%	4.23%
Messages / Hour	1708	160278	3628	263314
% Battery / Message *	0.01709	0.00010	0.00095	0.00002
Messages Received	240 / 1024	1024 / 1024	524 / 1024	1024 / 1024



4. Architettura del software derivante dalle scelte effettuate

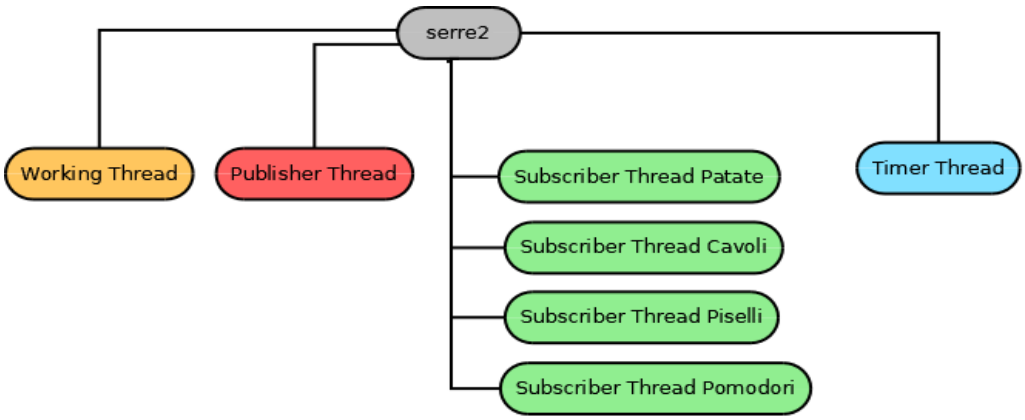
L'organizzazione dei moduli del nostro software può essere schematizzata nel modo seguente:



Il modulo del client con la Gui permette all'utente di interfacciarsi direttamente con Mosquitto e gestire manualmente l'irrigazione, avvisandolo sulle possibili controindicazioni generate dalle sue azioni, ma dando comunque la possibilità all'utente di forzare le sue scelte prevaricando il controller. Il modulo del logger si interfaccia anch'esso direttamente con Mosquitto iscrivendosi ai topic di interesse e registrando tutti i cambiamenti riguardanti gli stati degli irrigatori salvandoli su un file.

Si è scelto di rendere logger e controller con due moduli separati per evitare che problemi legati al logger, come per esempio la saturazione dello spazio disponibile, causassero l'arresto di tutto il modulo provocando malfunzionamenti.

Il modulo del controller si interfaccia direttamente con Mosquitto e si occupa di gestire la serra in modo automatico. Il controller è composto da varie thread che comunicano scambiandosi messaggi Json tramite la Subject Table secondo il design pattern già visto in precedenza del Publisher/Subscriber. La Subject Table può essere anche vista in maniera più comprensibile come un albero:



In questa Subject Table abbiamo la Subscriber Thread che si interfaccia direttamente con Mosquitto e si occupa di ricevere gli input dei sensori riguardanti ogni coltura, in particolare: radiazione, temperatura, e umidità. Questi dati ed altri da essi derivati verranno inviati alla Working Thread, la quale si occuperà di calcolare la quantità d'acqua necessaria da reintegrare alle piante, stimando il tempo per cui i nostri irrigatori dovranno rimanere aperti. Quest'ultimo dato verrà inviato alla Timer Thread che si occupa di gestire la temporizzazione degli irrigatori inviando messaggi alla Publisher Thread, l'ultimo anello della catena, che si interfaccia direttamente con Mosquitto ed invia i comandi di accensione e spegnimento degli irrigatori.

5. Descrizione dell’implementazione effettuata

Passando ad una descrizione più specifica della piattaforma software ideata, risulta opportuno partire descrivendo la Subject Table, per molti versi il cuore dell’applicazione. La Subject Table è utilizzata dalle diverse thread tra loro cooperanti per il passaggio di dati elaborati e/o da elaborare.

Strutturalmente è un vero e proprio albero di MessageQueue, strutture dati che contengono code di Message, in cui ogni URL è un percorso sull’albero. Dovendo la stessa istanza di Subject Table essere utilizzata da più thread in contemporanea, la feature “synchronized” fornita da Java è stata fondamentale per gestire in maniera corretta la funzionalità di questa componente in modo da evitare race condition e altri problemi noti tipici della programmazione concorrente.

La componente principale dell’applicazione è la thread Main, la thread principale che si occupa principalmente di due funzioni: avviare il server HTTP, unica componente fissa dell’intera piattaforma, e coordinare le thread secondarie gestendo anche la connessione alla rete MQTT.

Il server HTTP risulta essere una componente “fissa” in quanto si occupa di gestire la configurazione della piattaforma (mediante il caricamento di un file di configurazione in formato JSON) e del reset (che riavvia le strutture dati comuni e le varie thread, riportando le principali variabili condivise a valori di default). Entrambe le funzioni sono gestite attraverso un’interfaccia utente da browser. Dovendo quindi effettuare due funzioni così importanti, è necessario che questa componente debba essere sempre in funzione. Presenta quattro handler di richieste associati a percorsi differenti:

- ConfigRoot (associato al percorso “/”): ritorna una pagina HTML contenente l’indice della pagina di configurazione/reset.

- ConfigLoader (associato al percorso “/loader”): contiene un handler che gestisce la richiesta AJAX dell’interfaccia web. La richiesta è semplicemente una stringa JSON contenente le variabili globali da definire al momento dell’avvio. L’handler parsa il JSON, assegna alle variabili globali e avvia il main_task. Finché non viene caricato il file di configurazione, la componente main_task non si avvia e conseguenza tutto il resto dell’applicazione.
- ResetHandler (associato al percorso “/reset”): riceve la richiesta di reset tramite il bottone dell’interfaccia web. Dopodiché si occupa di effettuare la reset come descritta precedentemente.
- ResourcesHandler (associato al percorso “/resources”): si occupa semplicemente di gestire le richieste di caricamento che richiedono le immagini e i fogli di stile della pagina.

La seconda parte della thread Main, denominata main_task, è invece una componente dinamica avviata al momento del caricamento del file di configurazione. Essa si occupa di connettersi alla rete MQTT per richiedere la DynamicPage, un file JSON che contiene la struttura della serra. Dopo aver letto tale pagina, il Main è in grado di creare dinamicamente una Subscriber thread per ogni coltura rilevata. Si occupa anche di avviare le altre thread secondarie. Gestendo la componente dinamica del Main come task è possibile fermarla e riavviarla in maniera comoda al momento della reset. La lista delle thread secondarie è mantenuta in un ArrayList che ne contiene i riferimenti. Nella reset si ha un’iterazione su questa lista per fermare ogni singola thread, dopodiché viene fermata la thread principale. Dopo aver messo a default le variabili condivise, il main_task viene avviato, avviando a sua volta anche le thread secondarie.

Passando alle thread secondarie, si hanno:

- Publisher: si occupa di ricevere i comandi di accensione/spegnimento da inviare alla Beagleboard dalla Timer thread. Una volta ricevuto e parsato il comando (in formato JSON), viene inoltrato il comando ricevuto.
- Timer: si occupa di ricevere direttive dalla Working thread contenenti la coltura e il tempo di accensione degli irrigatori (calcolato in base alla temperatura media e alla radiazione media rilevata durante la giornata). Viene quindi inoltrato un comando di accensione alla Publisher. Il comando di spegnimento viene gestito con ScheduledExecutorService, una classe di Java che permette di eseguire una task dopo un certo quantitativo di tempo. Dopo aver atteso il tempo stabilito, viene inviato il comando di spegnimento alla Publisher.
- Working: si occupa di ricevere la temperatura media e la radiazione media della giornata, in modo da poter calcolare con una funzione (*calculateTime()*) il tempo di accensione degli irrigatori cosicché da poter arrivare alla temperatura ottimale per una certa coltura. Ottenuti i tempi di accensione, li invia tramite la Subject Table alla Timer thread, che si comporta come trattato nel precedente punto.
- Subscriber: esiste un’istanza della thread per ogni coltura rilevata all’avvio dell’applicazione. Si occupa di rilevare periodicamente temperatura e radiazione calcolandone la media. Dopodiché le invia alla Working thread.

Un’altra componente fondamentale per l’applicazione è il modulo di logging, che si occupa di registrare su file ogni azione compiuta dal controllore della serra, inserendo opportunamente l’ora di ogni azione. Inizialmente doveva essere gestito come thread da avviare insieme alle altre thread secondarie, ma si è ritenuto di renderla una componente indipendente dall’applicazione principale per motivi di solidità e resilienza della piattaforma.

Per quanto riguarda l'interfaccia utente, si è pensato di creare un'interfaccia web che utilizza JavaScript. Pensata per essere utilizzata in maniera "scollegata" dal server, risulta essere pressoché indipendente dall'applicazione principale e mediante le librerie MQTT permette di interfacciarsi alla rete MQTT utilizzata dalla serra. Principalmente permette di effettuare controlli sulla temperatura e radiazione correnti, informazioni sulle colture e umidità. La funzionalità più interessante permette di gestire manualmente gli irrigatori, dando la possibilità di accenderli e spegnerli.

Come è facile immaginare, questa possibilità dà molto controllo all'utente, che talvolta potrebbe sbagliare. Questo potere è mitigato da controlli effettuati dall'applicazione JavaScript che eventualmente avvisa l'utente di possibili controindicazioni causate dalle sue azioni (ad esempio: orario inadatto per innaffiare, umidità delle colture sufficientemente alta, etc.). L'interfaccia si costruisce dinamicamente dopo aver richiesto la DynamicPage, all'incirca come fa l'applicazione principale in Java. Così facendo può gestire variazioni nelle colture senza dover effettuare ritocchi al codice.

6. Validazione del software sviluppato

Il software è stato testato simulando un utilizzo reale dell'applicazione: la piattaforma è stata lasciata in esecuzione su una macchina dedicata insieme allo script di logging per un'intera giornata in modo da verificare anche gli aspetti legati all'orario reale. Durante l'arco della giornata il software ha periodicamente registrato temperatura e radiazione calcolando le medie da utilizzare per il calcolo dell'acqua da conferire al terreno ad un'ora prestabilita della sera (nel nostro caso alle 18).

Questo testing ci ha permesso di verificare non solo il corretto funzionamento del programma, ma anche la sua stabilità durante la giornata.

Durante lo sviluppo di ogni singola unità si è effettuato un testing ricreando ad hoc situazioni di utilizzo reali (ad esempio inviando messaggi di un certo tipo ad alcune thread simulando l'interazione tra componenti differenti). Questo ha permesso di rilevare errori talvolta anche subdoli se lasciati in un contesto di esecuzione globale e non unità per unità.

Il testing dell'interfaccia web è stato molto più interattivo in quanto è pensata per essere utilizzata attivamente da un utente (al contrario della piattaforma generale, pensate per funzionare in maniera autonoma come un demone).

7. Bibliografia

[1] https://it.wikipedia.org/wiki/Representational_State_Transfer

[2] https://italiancoders.it/

[3] https://www.slideshare.net/omnys_keynotes/mqtt-il-protocollo-che-rende-possibile-linternet-of-things

[4] http://www.html.it/

[5] Slide del corso disponibili sul DIR