

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Corso di Architetture e Sistemi Software Distribuiti

Integrazione di un ‘gate’ per il parcheggio in Proxima City

*Prof:*  *Studenti:*

Eugenio Zimeo Angelo Petraccaro

Antonio Rapuano

A.A. 2022/2023

Sommario

[1 Analisi del problema 3](#_Toc115433559)

[1.1 Obiettivo 3](#_Toc115433560)

[1.2 Introduzione 3](#_Toc115433561)

[Schema circuitale: Tinkercad 4](#_Toc115433562)

[1.3 Soluzioni adoperate 6](#_Toc115433563)

[2 Tecnologie abilitanti 9](#_Toc115433564)

[2.2 Paradigma REST 9](#_Toc115433565)

[2.3 MQTT – Message Queuing Telemetry Transport 9](#_Toc115433566)

[2.3.1 Mosquitto Broker 9](#_Toc115433567)

[2.3.2 Qualità del servizio MQTT 9](#_Toc115433568)

[3 Architettura dell’AI\_Emulator 12](#_Toc115433569)

[3.1 Domain Class Diagram 12](#_Toc115433570)

[4 Gestori di stato su Arduino 16](#_Toc115433571)

[4.1 Gestore MQTT 16](#_Toc115433572)

[4.2 Gestore REST 16](#_Toc115433573)

[4.3 Gestore WEBHOOK 17](#_Toc115433574)

[5 Risultati-Tempi 18](#_Toc115433575)

[5.1 Confronto di tempistiche tra le 3 soluzioni 18](#_Toc115433576)

[5.2 Testing della soluzione REST 19](#_Toc115433577)

[5.3 Testing della soluzione Webhook con Apache JMeter 19](#_Toc115433578)

[5.4 Test visuale 21](#_Toc115433579)

[6 Soluzione adottata 22](#_Toc115433580)

[6.1 Diagramma di allocazione 22](#_Toc115433581)

[7 ISTALLAZIONE SU PROXIMA CITY 23](#_Toc115433582)

[8 Sitografia 23](#_Toc115433583)

**Elenco delle figure**

[Figura 1- Modello Simulink della logica decisionale del sistema1 4](#_Toc115433584)

[Figura 2 - Diagramma di allocazione del sistema2 5](#_Toc115433585)

[Figura 3 - Diagramma di collaborazione della soluzione con gestore MQTT 6](#_Toc115433586)

[Figura 4 - Diagramma di collaborazione della soluzione con gestore REST 7](#_Toc115433587)

[Figura 5 - Diagramma di collaborazione della soluzione con gestore WEBHOOK 8](#_Toc115433588)

[Figura 12 – MQTT QoS 10](#_Toc115433589)

[Figura 6 - Domain class diagram dell’AI\_Emulator per la soluzione MQTT 12](#_Toc115433590)

[Figura 7 - Domain class diagram dell’AI\_Emulator per la soluzione REST 13](#_Toc115433591)

[Figura 8 - Domain class diagram dell’AI\_Emulator per la soluzione WEBHOOK 14](#_Toc115433592)

[Figura 9 – Codice che mostra parte della funzione di callback relativa al gestore MQTT 16](#_Toc115433593)

[Figura 10 - Codice relativo al polling con la soluzione REST 16](#_Toc115433594)

[Figura 11 - Codice relativo alla funzione di gestione di una richiesta ricevuta da un Web\_Client 17](#_Toc115433595)

[Figura 13 – Codice relativo all’avvio del timer 18](#_Toc115433596)

[Figura 14 – Codice relativo al calcolo dell’elapsed time 18](#_Toc115433597)

[Figura 15 – Scheda Arduino MKR 1000 19](#_Toc115433598)

[Figura 16 – Risultato del test effettuato con JMeter 19](#_Toc115433599)

[Figura 17 - Test visuale per il confronto delle soluzioni adottate 21](#_Toc115433600)

[Figura 18 - Diagramma di allocazione della soluzione finale 22](#_Toc115433601)

1. Analisi del problema

## Obiettivo

Integrazione di un ‘gate’ per l’ingresso/uscita automobili all’interno del parcheggio gestito da intelligenza artificiale presente in ProximaCity e gestione della comunicazione tra i vari end-point. Il gate da integrare è parte di un precedente progetto sviluppato per il corso di Sistemi Multiagente, il quale prevedeva la progettazione e implementazione della logica di controllo su scheda a microcontrollore Arduino UNO per il funzionamento di un parcheggio automatico che gestisce l’ingresso e l’uscita delle automobili.

Prima di realizzare l’integrazione del ‘gate’ (sistema1) si è emulato il comportamento dell’intelligenza artificiale (sistema2) evitando di interagire direttamente con ProximaCity. Sono state implementate 3 diverse soluzioni di integrazione tra i 2 sistemi ed è stata scelta la migliore.

* La prima soluzione fa uso del protocollo di messaggistica MQTT
* La seconda soluzione fa uso del paradigma REST
* La terza soluzione fa uso del meccanismo WEBHOOK

Nel prosieguo del documento, ci si riferisce al ‘gate’ come sistema1 ed al parcheggio con AI presente su ProximaCity come sistema2.

## Introduzione

Il sistema1 originariamente fu realizzato utilizzando la scheda Arduino UNO che integra il microcontrollore ATmega328P, grazie al quale si è in grado di programmare delle logiche di controllo da rilasciare sulla scheda.

In questo progetto, però, per consentire al sistema1 di comunicare in rete ci siamo serviti di una scheda dotata di modulo Wi-Fi integrato, ovvero ArduinoMKR1000. Inoltre, per triggerare l’arrivo di una auto che intende parcheggiare è stato aggiunto un sensore ad ultrasuoni.

Di seguito viene mostrata una tabella contenente tutta la componentistica utilizzata nella replicazione del sistema1, con l’aggiunta del sensore ad ultrasuoni:

|  |  |  |
| --- | --- | --- |
| Componente | Descrizione | Quantità |
| ArduinoMKR1000 | Scheda con microcontrollore e modulo Wi-Fi integrato. | 1 |
| SG90 | Micro servo motore. | 1 |
| HC – SR04 | Sensore ad ultrasuoni. | 1 |
| HV – 201 | Sensore ad ostacoli IR. | 2 |
| LCD 1602 | Display LCD. | 1 |

I due sensori IR sono stati impiegati per indicare l’avvicinamento al gate delle automobili. Il display è stato impiegato per vedere in tempo reale la disponibilità dei posti nel parcheggio. Il servomotore è stato utilizzato per l’apertura e la chiusura della sbarra di ingresso/uscita.

La logica decisionale del sistema1 è stata simulata con il seguente modello Simulink:

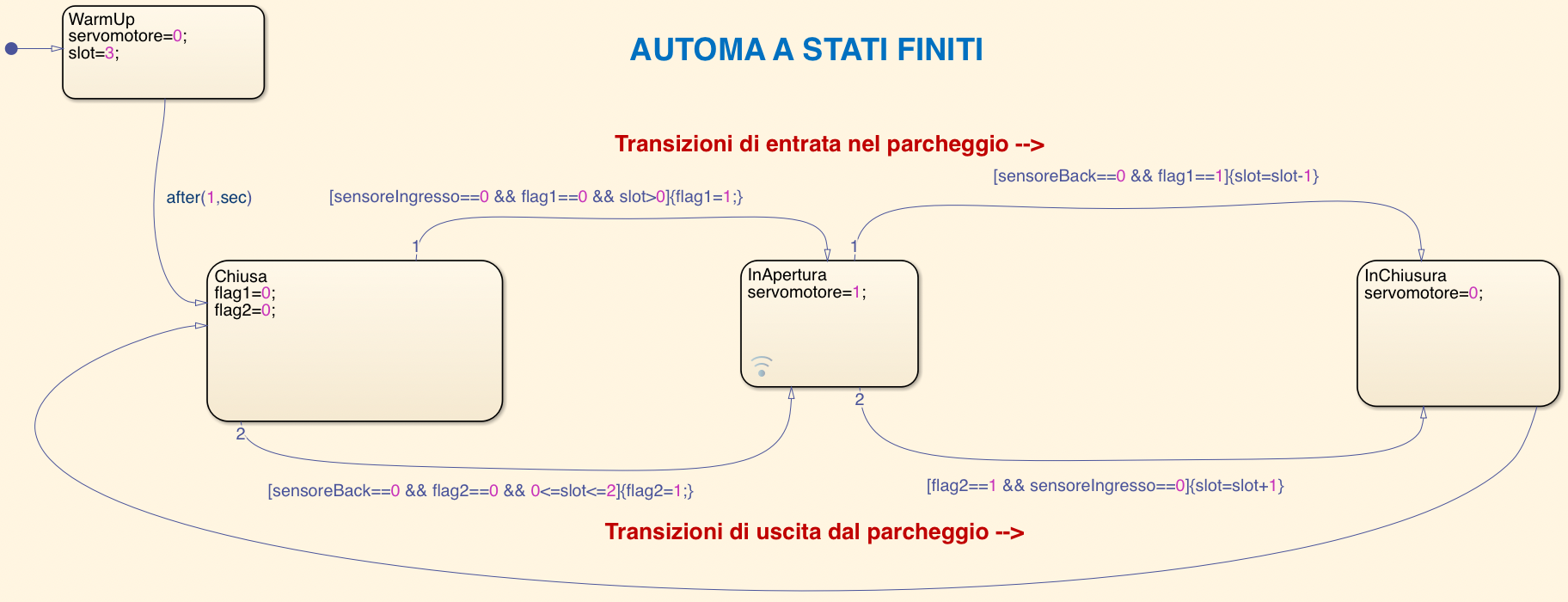


Figura 1- Modello Simulink della logica decisionale del sistema1

***WarmUp*** rappresenta lo stato iniziale della macchina necessario per il warm up del sistema, dove la sbarra viene abbassata e vengono settati i posti disponibili nel parcheggio.

***Chiusa*** è lo stato che indica la sbarra chiusa; all’avvicinarsi della macchina in ingresso al parcheggio fa un check sulla disponibilità dei posti e qualora non ce ne siano blocca l’apertura. Da questo stato si può andare nello stato successivo attraverso due transizioni. Sono state utilizzate due variabili ’flag1’ e ’flag2’ per indicare la direzione di ingresso (flag1=1) o di uscita di una macchina (flag2=1). La condizione per l’attivazione delle transizioni in uscita dallo stato ’Chiusa’ dipende da queste due variabili, dai posti disponibili e dall’attraversamento del primo sensore della sbarra (sensoreIngresso dal lato di ingresso e sensoreBack dal lato di uscita).

***InApertura*** è lo stato in cui la sbarra si sta aprendo a seguito dell’ingresso o dell’uscita di una macchina dal parcheggio. Arrivati in questo stato viene eseguita l’azione riportata nella sezione entry, ovvero l’attivazione del motorino per far alzare la sbarra. Da questo stato si può andare nello stato successivo attraverso due transizioni. La condizione per l’attivazione delle transizioni dipende dalle due variabili flag1 e flag2 e dall’attraversamento del secondo sensore della sbarra.

***InChiusura*** è lo stato in cui la sbarra si sta chiudendo a seguito del passaggio di una macchina davanti al secondo sensore della sbarra. All’attivazione della transizione di uscita viene attivato il motorino per far chiudere la sbarra e si passa allo stato successivo.

## Schema circuitale del sistema1

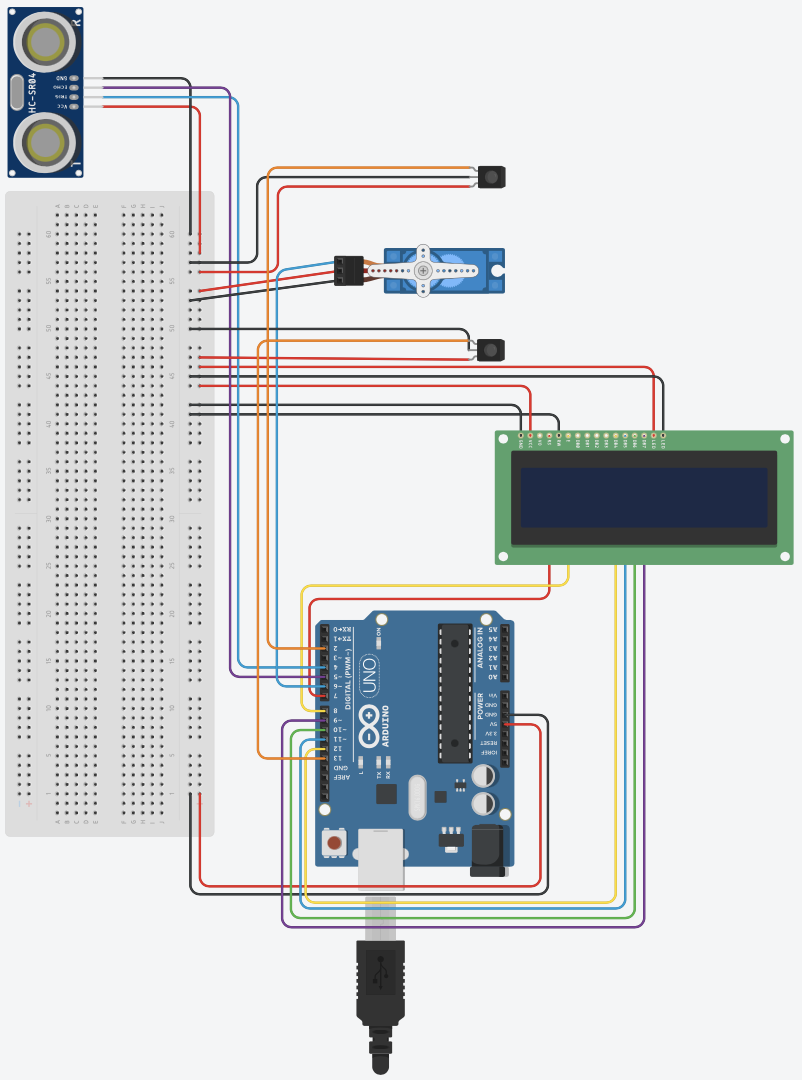


Figura 2 - Schema circuitale del sistema1

Immagine che contiene testo

Descrizione generata automaticamente

Figura 3 - Sistema1

Il sistema2, già presente su Proxima City, ha il compito di scattare una foto al parcheggio a seguito di un evento di trigger lanciato dal sensore ad ultrasuoni e poi farne una elaborazione tramite AI usando la libreria Tensor-Flow per il riconoscimento degli oggetti (automobili parcheggiate nella delimitazione dei posti auto). A seguito di tale elaborazione viene pubblicato lo stato del parcheggio su ‘topic/status’.

Le componenti appartenenti al sistema2 sono, quindi:

* una fotocamera che inquadra il parcheggio dall’alto.
* un sensore ad ultrasuono.
* un'intelligenza artificiale per il riconoscimento di immagini.
* un tappetino di lego con le delimitazioni dei posti auto.
* delle automobiline giocattolo



Figura 4 - Diagramma di allocazione del sistema2

Immagine che contiene elettronico, circuito

Descrizione generata automaticamente

Figura 5 - Sistema2

## Soluzioni adoperate

Le 3 soluzioni implementate sono:

1. La prima consiste di un subscriber MQTT, un AI\_Thread che emula il comportamento della fotocamera e di un publisher MQTT che pubblica il nuovo stato. Il Parking Agent è la componente che ingloba sia il subscriber che il publisher.

Sia in ingresso che in uscita di un’auto dal parcheggio il subscriber si sottoscrive al topic “test” ed alla ricezione dell’evento di trigger (dal sensore ad ultrasuoni o dal sensore IR) istanzia un AI\_Thread che crea uno stato fittizio e dopo 2 secondi lo aggiorna con le informazioni corrette.

Il clientMQTT, in esecuzione su arduino, si sottoscrive al topic “status” e quando riceve lo stato trasferito dal broker confronta quest’ultimo con lo stato interno; se i due coincidono abilita il gate, altrimenti stampa un errore sul monitor seriale.

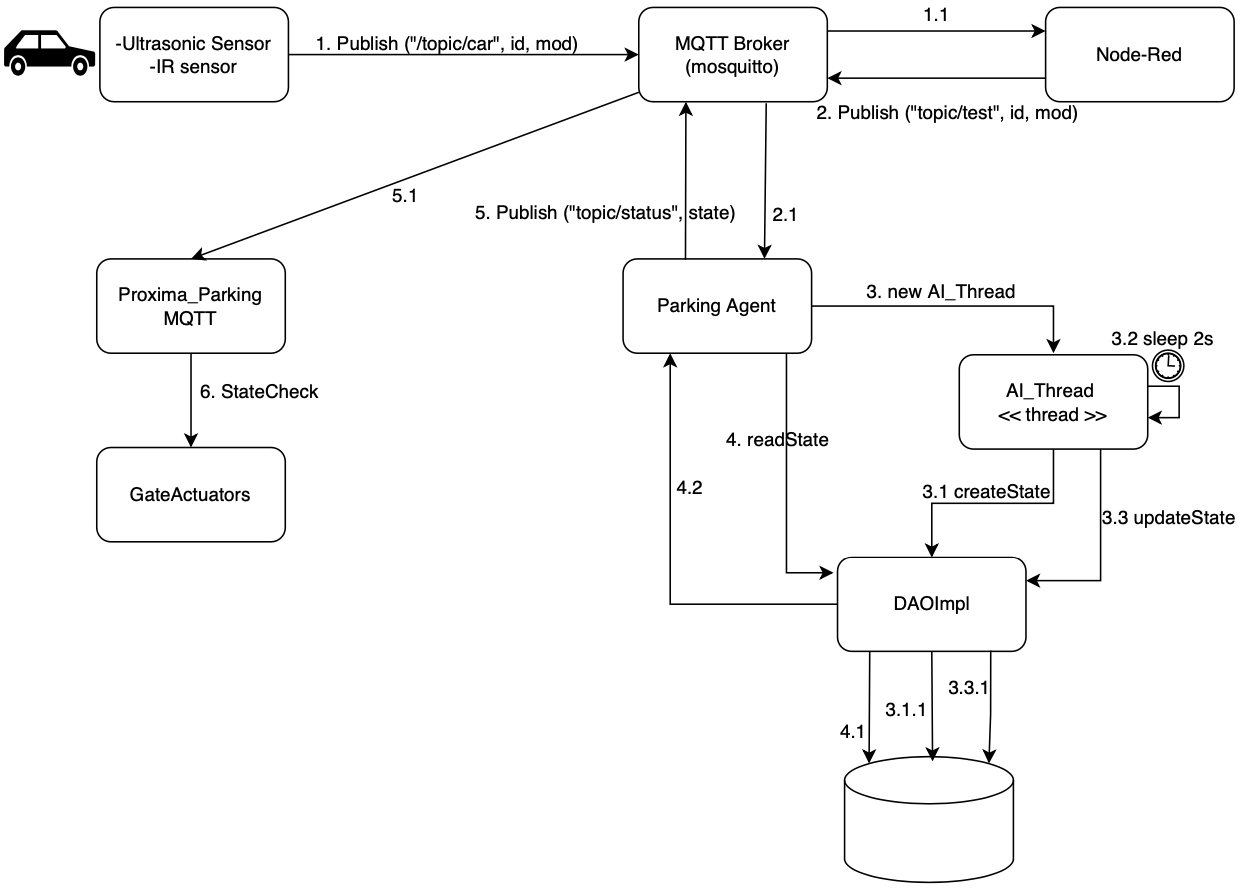


Figura 6 - Diagramma di collaborazione della soluzione con gestore MQTT

Per questa soluzione sono stati adottati 3 livelli di qualità del servizio del protocollo MQTT. Per i primi due livelli è stata possibile l’implementazione, mentre per il livello QoS 2 non è stato possibile realizzare un subscriber perché la libreria PubSubClient di arduino non lo supporta.

1. La seconda consiste di un subscriber MQTT, un AI\_Thread che emula il comportamento della fotocamera e di una interfaccia REST che espone il metodo GET /status/{id} il quale restituisce lo stato relativo a tale id. In questo caso il Parking Agent è la componente che ingloba soltanto il subscriber.

Sia in ingresso che in uscita di un’auto dal parcheggio il subscriber si sottoscrive al topic “test” ed alla ricezione dell’evento di trigger (dal sensore ad ultrasuoni o dal sensore IR) istanzia un AI\_Thread che crea uno stato fittizio e dopo 2 secondi lo aggiorna con le informazioni corrette.

Il clientREST, in esecuzione su arduino, continua ad effettuare delle richieste http GET con l’id finchè non riceve uno stato consistente (polling). Quando viene ricevuto uno stato non consistente il codice di stato del messaggio ricevuto è “404 Not Found”. Quando viene ricevuto uno stato consistente confronta quest’ultimo con lo stato interno; se i due coincidono abilita il gate, altrimenti stampa un errore sul monitor seriale.

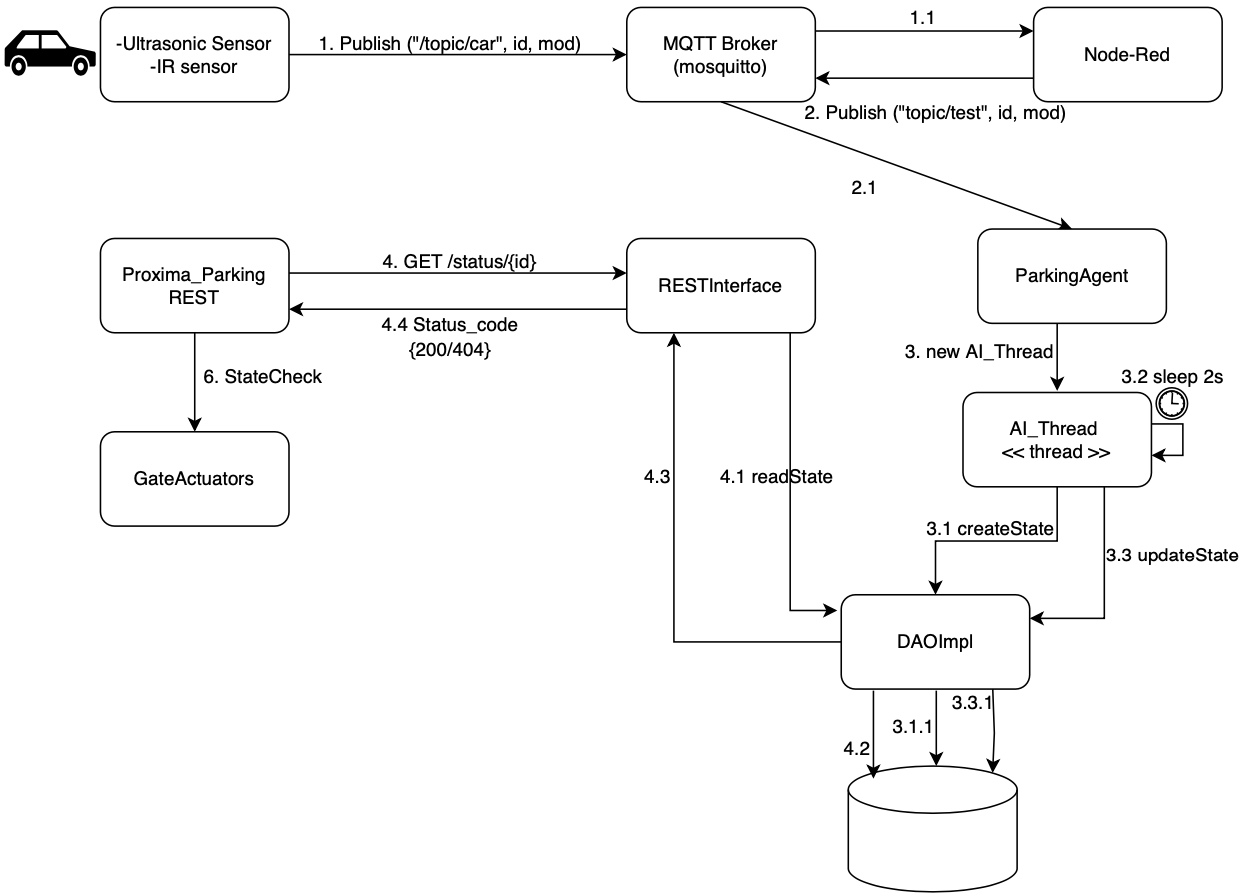


Figura 7 - Diagramma di collaborazione della soluzione con gestore REST

1. La terza consiste di un subscriber MQTT, un AI\_Thread che emula il comportamento della fotocamera, un Web\_client che effettua il POST dello stato e di un Web\_server in ascolto su arduino. Il Parking Agent è la componente che ingloba il subscriber ed il Web\_client.

Sia in ingresso che in uscita di un’auto dal parcheggio il subscriber si sottoscrive al topic “test” ed alla ricezione dell’evento di trigger (dal sensore ad ultrasuoni o dal sensore IR) istanzia un AI\_Thread che crea uno stato fittizio e dopo 2 secondi lo aggiorna con le informazioni corrette.

Successivamente il Web\_client effettua una POST con lo stato al Web\_server in ascolto su arduino. Il Web\_server riceve il messaggio di richiesta, se non si verificano errori estrae il body del messaggio (stato di AI) e confronta quest’ultimo con lo stato interno; se i due coincidono abilita il gate ed invia un messaggio di risposta con il codice di stato “201 created”; altrimenti stampa un errore sul display LCD ed invia un messaggio di risposta con il codice di stato “500 Internal Error”. Il Web\_client quando riceve il codice di errore “500 Internal Error”, per annullare la precedente operazione effettua il ripristino dello stato aggiornando la capienza.

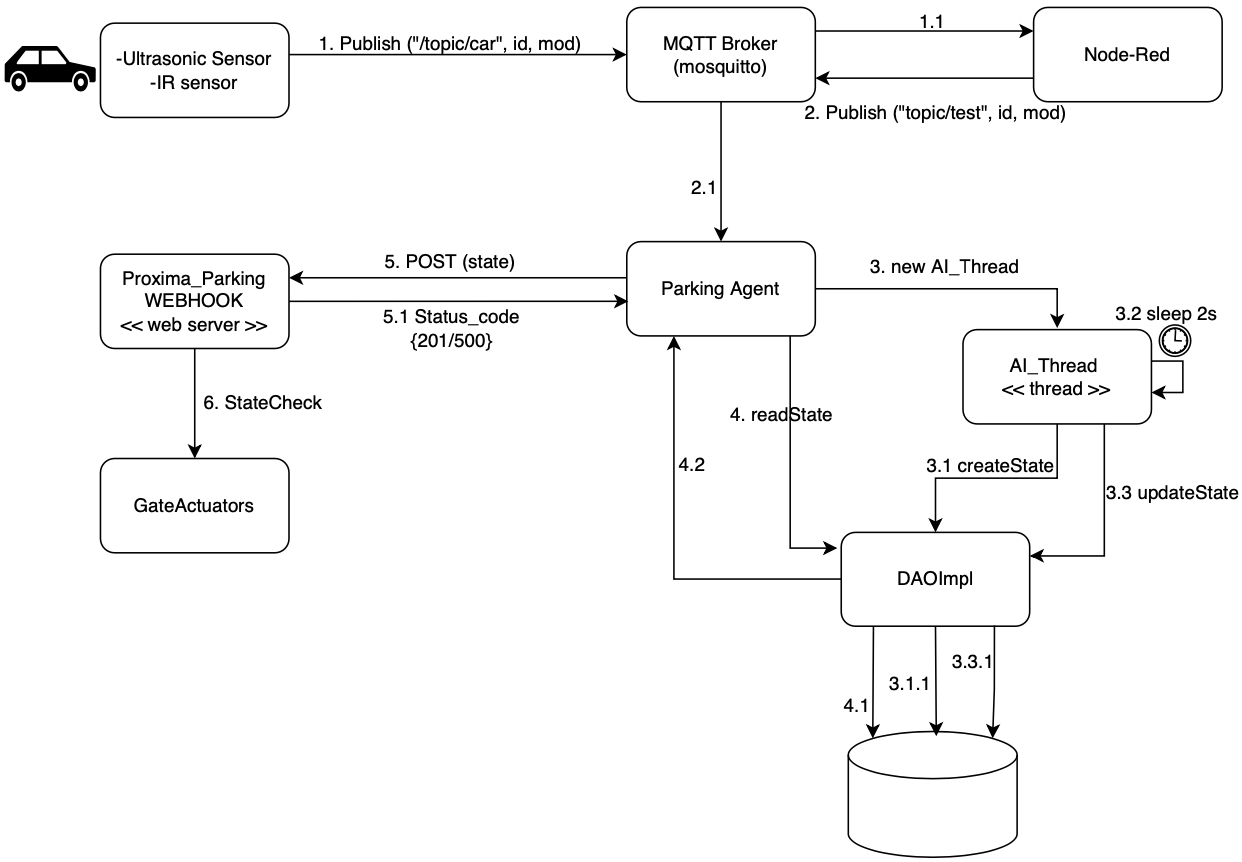


Figura 8 - Diagramma di collaborazione della soluzione con gestore WEBHOOK

1. Tecnologie abilitanti
   1. Arduino

Arduino è una piattaforma elettronica open source basata su hardware e software facili da usare. La piattaforma hardware di Arduino prevede un ambiente di sviluppo per la programmazione (Arduino IDE), il cui codice è derivato dai linguaggi C/C++ e i file sorgenti sono chiamati sketch.

## Paradigma REST

Lo stile REpresentational State Transfer (REST) è una specializzazione dello stile Client/Server. Nello stile REST le informazioni sono organizzate come risorse; ogni risorsa è referenziata tramite un identificatore globale, come un URL in HTTP. Le componenti di rete, user agent e origin server comunicano tramite un’interfaccia standardizzata (per esempio HTTP), e si scambiano le rappresentazioni delle risorse. Una risorsa può essere rappresentata in molti modi diversi, ad esempio HTML, XML, JSON o anche file JPEG. La rappresentazione più utilizzata nelle implementazioni REST è il formato JSON. Una API REST supporta i seguenti metodi HTTP per manipolare le risorse:

• POST(CREATE): crea una risorsa;  
• GET (lettura): restituisce una rappresentazione di una risorsa;

• PUT (UPDATE): aggiorna una risorsa esistente;  
• DELETE: elimina la risorsa.

## MQTT – Message Queuing Telemetry Transport

È un protocollo leggero la cui comunicazione è basata sul paradigma Publish/Subscribe. Il protocollo MQTT è uno degli standard più apprezzati nel mondo dell’IoT per le sue caratteristiche di leggerezza dal punto di vista dell’impiego delle risorse, efficacia, semplicità d’uso e versatilità. Tra i limiti abbiamo la dimensione del messaggio, la cui massima è 256 MB. Questa dimensione è anche molto grande se si pensa all’uso che tipicamente si fa di questo protocollo (messaggi che contengono sample legati alla misurazione di un parametro fisico o chimico). È un protocollo a bassa latenza perché si predilige la latenza piuttosto che il resto.

### Mosquitto Broker

Eclipse Mosquitto è un broker di messaggi open source che implementa le versioni 5.0, 3.1.1 e 3.1 del protocollo MQTT. Mosquitto è leggero ed è adatto all'uso su tutti i dispositivi, dai computer a scheda singola a bassa potenza ai server completi.

Il protocollo MQTT sostanzialmente si basa su un modello di brokering, vale a dire l’esistenza di un soggetto (il broker) al quale tutti i client si registrano e verso il quale quest’ultimi inoltrano e ricevono messaggi per e da altri client anch’essi connessi. Il progetto Mosquitto fornisce anche una libreria C per l'implementazione dei client MQTT usando il comando mosquitto\_pub e mosquitto\_sub.

### Qualità del servizio MQTT

La QoS rappresenta una modalità di funzionamento dell’infrastruttura, finalizzata ad assicurare varie proprietà come l’affidabilità. MQTT offre 3 livelli di qualità del servizio.

•QoS 0: AT MOST ONCE

Quando la QoS è impostata su 0, un messaggio viene recapitato al massimo una volta. Potrebbe non venire consegnato affatto. Non c’è conferma della consegna con questo livello di QoS. Il messaggio non viene archiviato e può essere perso se il client si disconnette o il server non funziona. Questo livello di QoS viene talvolta definito “fire and forget” ed è la modalità di trasferimento dati più veloce in un’implementazione MQTT. Questo QoS ha il minor sovraccarico del traffico di rete e il minor carico sul client e sul broker ed è spesso utile per i dati di telemetria in cui non importa se alcuni dei dati vengono persi.

• QoS 1: AT LEAST ONCE

QoS 1 indica che il messaggio viene sempre consegnato almeno una volta. La ricezione del messaggio deve essere confermata e la mancata ricezione di un riconoscimento comporta il reinvio del messaggio. Il risultato di questo QoS è che il destinatario può ricevere il messaggio più volte e aumenta anche il sovraccarico di rete rispetto a QoS 0. Inoltre, viene posto un carico maggiore sul mittente in quanto deve archiviare il messaggio e riprovare nel caso in cui non riesca a ricevere un ack in un tempo ragionevole.

• QoS 2: EXACTLY ONCE

È Il più costoso dei QoS (in termini di traffico di rete e onere per mittente e destinatario): garantisce che il messaggio venga ricevuto da un destinatario esattamente una volta. Ciò assicura che il destinatario non riceva mai copie duplicate del messaggio e alla fine lo otterrà sicuramente, ma al costo aggiuntivo del sovraccarico di rete e della complessità richiesta al mittente e al destinatario.

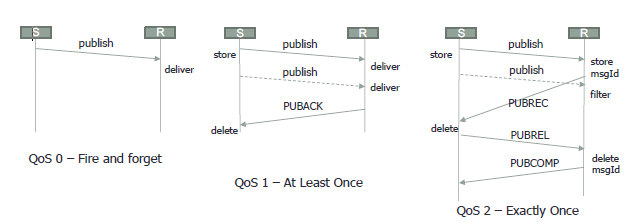


Figura 9 – MQTT QoS

* 1. Apache JMeter

L'applicazione Apache JMeterTM è un software open source, progettata per testare il comportamento funzionale e misurare le prestazioni. È stato originariamente progettato per testare applicazioni Web, ma da allora si è esteso ad altre funzioni di test.

Apache JMeter può essere utilizzato per testare le prestazioni sia su risorse statiche che dinamiche, applicazioni dinamiche Web.

Può essere utilizzato per simulare un carico pesante su un server, un gruppo di server, una rete o un oggetto per testarne la forza o per analizzare le prestazioni complessive sotto diversi tipi di carico.

Le funzionalità di Apache JMeter includono:

* Capacità di caricare e testare le prestazioni di molte applicazioni/server/tipi di protocollo diversi:
  + Web - HTTP, HTTPS (Java, NodeJS, PHP, ASP.NET, ...)
  + Servizi Web SOAP / REST
  + FTP
  + TCP
  + LDAP
  + Database tramite JDBC
  + Middleware orientato ai messaggi (MOM) tramite JMS
  + Mail - SMTP(S), POP3(S) e IMAP(S)
  + Comandi nativi o script di shell
  + Oggetti Java

1. Architettura dell’AI\_Emulator

L’AI\_Emulator è la componente che emula il sistema2. Grazie all’AI\_Emulator è stato possibile implementare le 3 soluzioni evitando di interagire direttamente con ProximaCity.

## Domain Class Diagram

I seguenti diagrammi descrivono l’architettura dell’AI\_Emulator nelle 3 soluzioni adottate.

Le tre architetture differiscono unicamente per la logica applicativa.

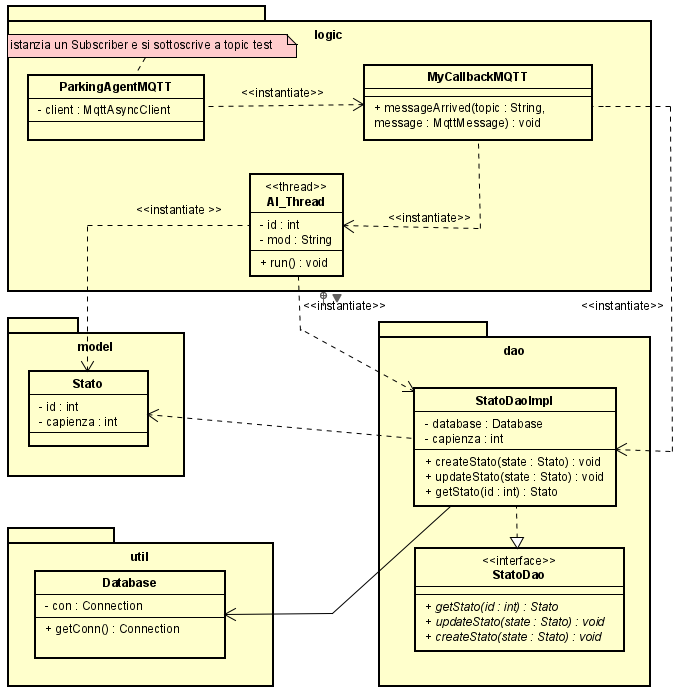


Figura 10 - Domain class diagram dell’AI\_Emulator per la soluzione MQTT

|  |  |
| --- | --- |
| **Classe** | **Descrizione** |
| ParkingAgent | Classe che istanzia un subscriber mqtt, si sottoscrive a ‘topic/test’ e crea un oggetto di tipo MyCallbackMQTT. |
| MyCallbackMQTT | Classe che implementa il metodo messageArrived, che viene chiamato ogni volta che arriva un messaggio su ‘topic/test’; crea un AI\_Thread, legge lo stato dal DAO e lo pubblica sulla coda mqtt associata a ‘topic/status’. |

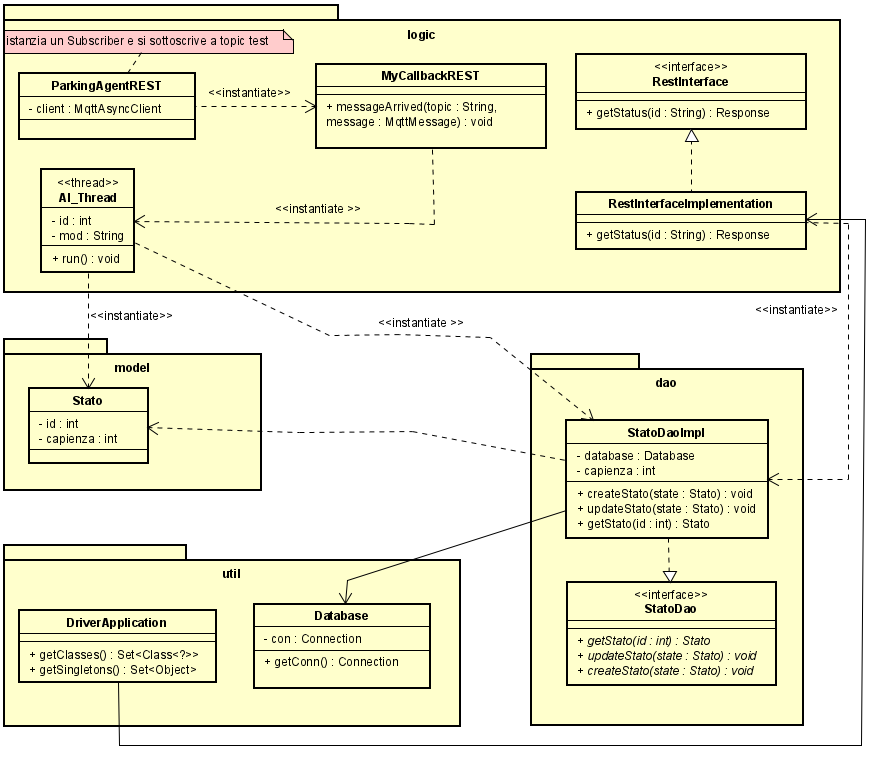


Figura 11 - Domain class diagram dell’AI\_Emulator per la soluzione REST

|  |  |
| --- | --- |
| **Classe** | **Descrizione** |
| ParkingAgent | Classe che istanzia un subscriber mqtt, si sottoscrive a ‘topic/test’ e crea un oggetto di tipo MyCallbackREST. |
| MyCallbackREST | Classe che implementa il metodo messageArrived, che viene chiamato ogni volta che arriva un messaggio su ‘topic/test’; crea un AI\_Thread. |
| RestInterface | Interfaccia che espone il metodo getStatus(id) per il recupero dello stato dato l’id. |
| RestInterfaceImpl | Classe che implementa l’interfaccia RestInterface, legge lo stato dal DAO, se è consistente lo inserisce nel body del messaggio http di risposta. Se lo stato non è consistente restituisce un messaggio di risposta caratterizzato dallo status code 404 Not Found. |
| DriverApplication | Classe che implementa il metodo getClasses, che crea un nuovo oggetto ad ogni invocazione ed il metodo getSingleton che mantiene lo stato mantenendo l’oggetto usato per dar vita al servizio. |

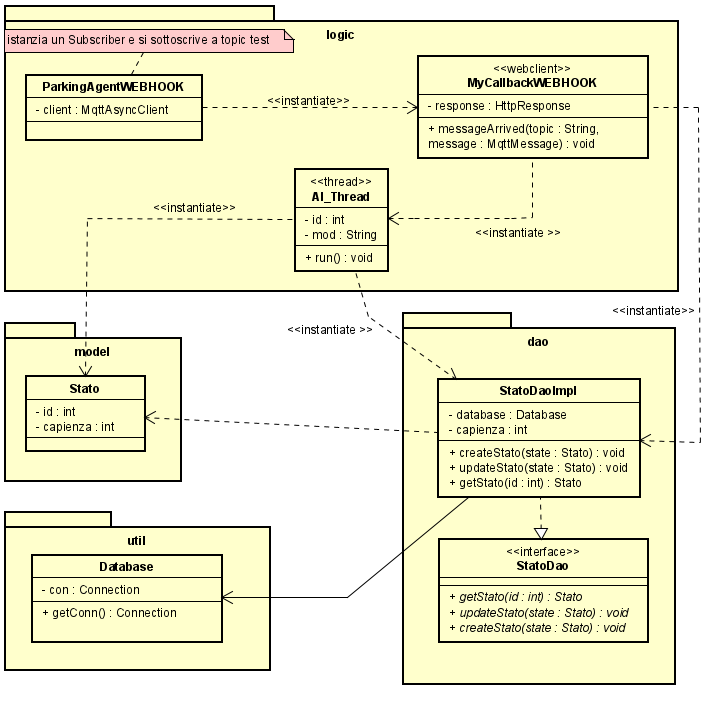


Figura 12 - Domain class diagram dell’AI\_Emulator per la soluzione WEBHOOK

|  |  |
| --- | --- |
| **Classe** | **Descrizione** |
| ParkingAgent | Classe che istanzia un subscriber mqtt, si sottoscrive a ‘topic/test’ e crea un oggetto di tipo MyCallbackWebhook. |
| MyCallbackWebhook | Classe che implementa il metodo messageArrived, che viene chiamato ogni volta che arriva un messaggio su ‘topic/test’; crea un AI\_Thread, legge lo stato dal DAO e crea un Web\_client che invia tramite POST lo stato letto. |

Di seguito è mostrata la tabella che raccoglie tutte le classi in comune delle 3 diverse architetture.

|  |  |
| --- | --- |
| **Classe** | **Descrizione** |
| Stato | Classe che modella lo stato del parcheggio. |
| Database | Classe che effettua la connessione al db MySQL. |
| StatoDAO | Interfaccia che espone le operazioni che si possono effettuare sul database, ovvero leggere, creare ed aggiornare oggetti di tipo Stato. |
| StatoDAOImpl | Classe che implementa l’interfaccia StatoDAO e quindi i relativi metodi getStato(), CreateStato() ed UpdateStato(). |
| AI\_Thread | Classe che crea uno stato fittizio, lo salva sul DAO, attende 2 secondi e poi ne fa un aggiornamento. |

1. Gestori di stato su Arduino

## Gestore MQTT

La parte saliente della soluzione MQTT è la funzione callback che viene chiamata ogni qual volta viene ricevuto un messaggio sul topic/status. La funzione prende in input:

* il puntatore al nome del topic
* il puntatore alla sequenza di byte che rappresenta il messaggio
* la lunghezza del payload

La funzione legge il contenuto del payload per ogni byte, lo converte in carattere e lo inserisce all’interno di una stringa. Infine tale stringa viene convertita in Json e viene estratto lo stato.

Immagine che contiene testo

Descrizione generata automaticamente

Figura 13 – Codice che mostra parte della funzione di callback relativa al gestore MQTT

## Gestore REST

La parte saliente della soluzione REST prevede l’utilizzo del polling. Nello specifico il gestore REST continua ad effettuare richieste http GET finché non riceve uno stato consistente. Invece, uno stato non consistente è caratterizzato da una risposta http con codice di stato 404. Il polling è realizzato con un do-while che cicla finchè non trova uno status\_code uguale a 200. Ad ogni iterazione viene effettuata la verifica di connessione del client e se è connesso effettua la GET. In seguito, vi è un ciclo di attesa della risposta sul canale, quando diventa disponibile si effettua la lettura della risposta che viene memorizzata in una stringa. Infine da tale stringa si estrae lo stato.

Immagine che contiene testo

Descrizione generata automaticamente

Figura 14 - Codice relativo al polling con la soluzione REST

## Gestore WEBHOOK

La parte saliente della soluzione WEBHOOK è la funzione ‘gestioneRichiesta’. Nello specifico il web\_server attivo su arduino è in attesa di una richiesta http POST effettuata da un web\_client. Quando la riceve ne estrae il contenuto (rappresentante lo stato) e se questo è stato letto correttamente invia una risposta al client caratterizzata dal codice di stato 201 Created, altrimenti segnala un errore interno e produce una risposta con codice di stato 500 Interal Server Error.

Nel caso di risposta corretta il codice di stato restituito è 201 Created in quanto viene creato un nuovo stato; in alternativa viene restituito 500 Internal Error per segnalare errori interni dovuti a limiti prestazionali della scheda arduino o ad una non corretta implementazione della libreria WiFi101. Tali limiti saranno visibili in seguito nei test.

Immagine che contiene testo

Descrizione generata automaticamente

Figura 15 - Codice relativo alla funzione di gestione di una richiesta ricevuta da un Web\_Client

1. Risultati

Per valutare quale fosse la soluzione migliore è stata misurata la latenza ed è stato effettuato testing di robustezza per capire l’affidabilità delle varie soluzioni.

## Confronto di tempistiche tra le 3 soluzioni

Per stabilire quale fosse la soluzione migliore sono state effettuate delle misurazioni della latenza a cui è stato sommato il tempo di elaborazione del messaggio perché la lettura viene effettuata seguendo diverse modalità. Il clock usato come riferimento è quello di arduino, ed il timer è stato avviato prima della pubblicazione dell’evento di trigger e stoppato a seguito della lettura dello stato.



Figura 16 – Codice relativo all’avvio del timer

Immagine che contiene testo

Descrizione generata automaticamente

Figura 17 – Codice relativo al calcolo dell’elapsed time

Sono state effettuate 30 misurazioni per ognuna delle soluzioni e ne è stata fatta la media. I risultati sono riportati nella seguente tabella.



Osservando il valore medio la scelta migliore è risultata essere la soluzione con gestore REST.

## Testing della soluzione REST

Per testare l’affidabilità della soluzione REST sono state effettuate un numero elevato di richieste GET ed è stato aumentato il tempo di elaborazione dello stato sull’AI\_Emulator in modo tale da far ritardare l’arrivo della risposta con uno stato consistente, provocando un numero maggiore di richieste GET (polling).

A seguito di tale test è stato riscontrato un lampeggiamento del led CHRG (indicatore di Temperature Fault, Timer Fault e Low Battery, che è stata esclusa in quanto la schedina era collegata all’alimentazione del pc). Ciò è dovuto ai limiti prestazionali della board ArduinoMKR1000, di conseguenza si è arrivati alla conclusione che il polling con questa scheda non è la soluzione ideale da adottare.

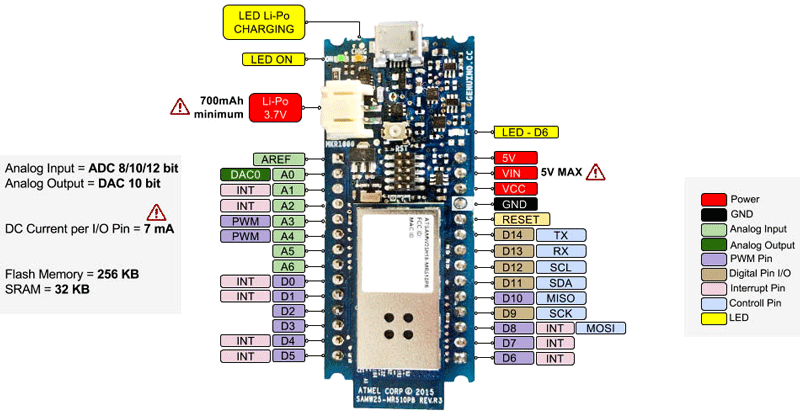


Figura 18 – Scheda Arduino MKR 1000

## Testing della soluzione Webhook con Apache JMeter

Come detto in precedenza, la terza soluzione presenta dei problemi. Per comprenderne la natura è stata isolata e testata, utilizzando il tool JMeter, la componente server in esecuzione su arduino.

Immagine che contiene testo

Descrizione generata automaticamente

Figura 19 – Risultato del test effettuato con JMeter

In particolare, si è simulato un carico di richieste POST sul server con diversi delay (1s, 2s, 500ms, 300ms…) ed è stata tratta la seguente conclusione:

come è possibile notare dalla figura in alto, non tutte le richieste vengono evase. Probabilmente gli errori sono dovuti a limiti prestazionali della scheda arduino o ad una non corretta implementazione della libreria WiFi101. Di conseguenza si è deciso di escludere questa soluzione per la scarsa affidabilità che propone.

* 1. Test visuale

In questo test la soluzione webhook non è stata presa in considerazione per le motivazioni sopracitate, ma anche perché le misurazioni dei tempi si discostavano molto dalle altre soluzioni.

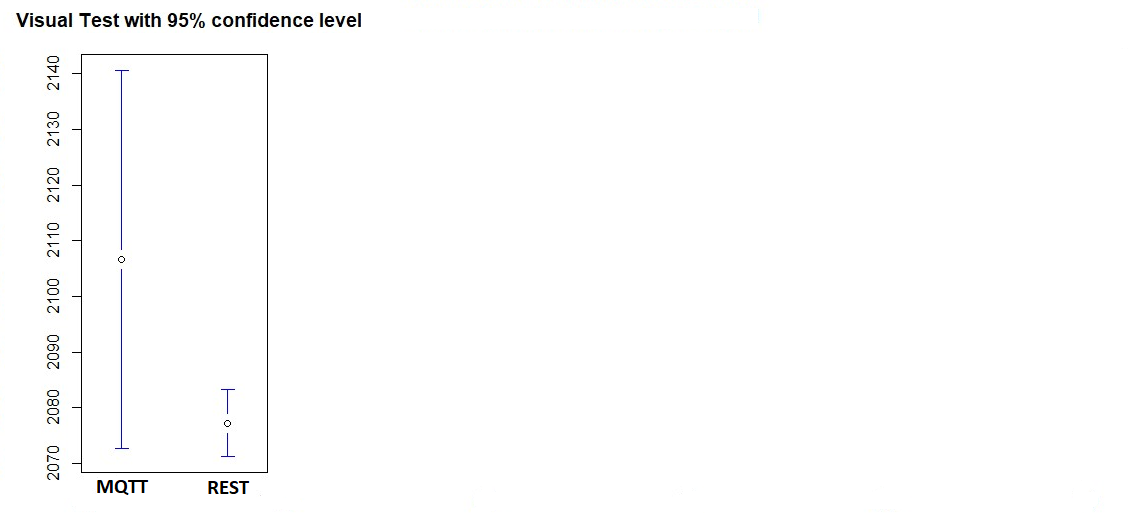


Figura 20 - Test visuale per il confronto delle soluzioni adottate

Siccome le osservazioni di MQTT e REST variavano di poco, per stabilire quale fosse la soluzione migliore è stato effettuato il test visuale. Per effettuare il test visuale bisogna calcolare media e l’intervallo di confidenza delle misurazioni ed infine plottarli. Come si evince dal plot, il CI di REST presenta minore variabilità e dal punto di vista statistico risulta essere la più stabile. Il test consente di confrontare n alternative e stabilire la migliore di esse nel seguente modo:

* Se i due intervalli non si sovrappongono e l’intervallo A ha una media più alta di B, allora il sistema A è migliore del sistema B.
* Se i due intervalli si sovrappongono e sovrappongono e la media dell’uno è contenuta nell’intervallo dell’altro, dal punto di vista statistico le due soluzioni non presentano differenze. (Questo è il caso in questione)
* Se gli intervalli si sovrappongono ma la media dell’uno non è contenuta nell’intervallo dell’altro non si può stabilire quale sistema sia il migliore, perché vi è necessità di fare il t-test.

Dato che ci si trova nel secondo caso, dove non è possibile stabilire dal punto di vista statistico la migliore soluzione, la scelta che abbiamo fatto è stata quella di considerare la soluzione più affidabile e meno pesante, quindi un gestore che comunica usando il protocollo MQTT.

1. Soluzione adottata

## Diagramma di allocazione

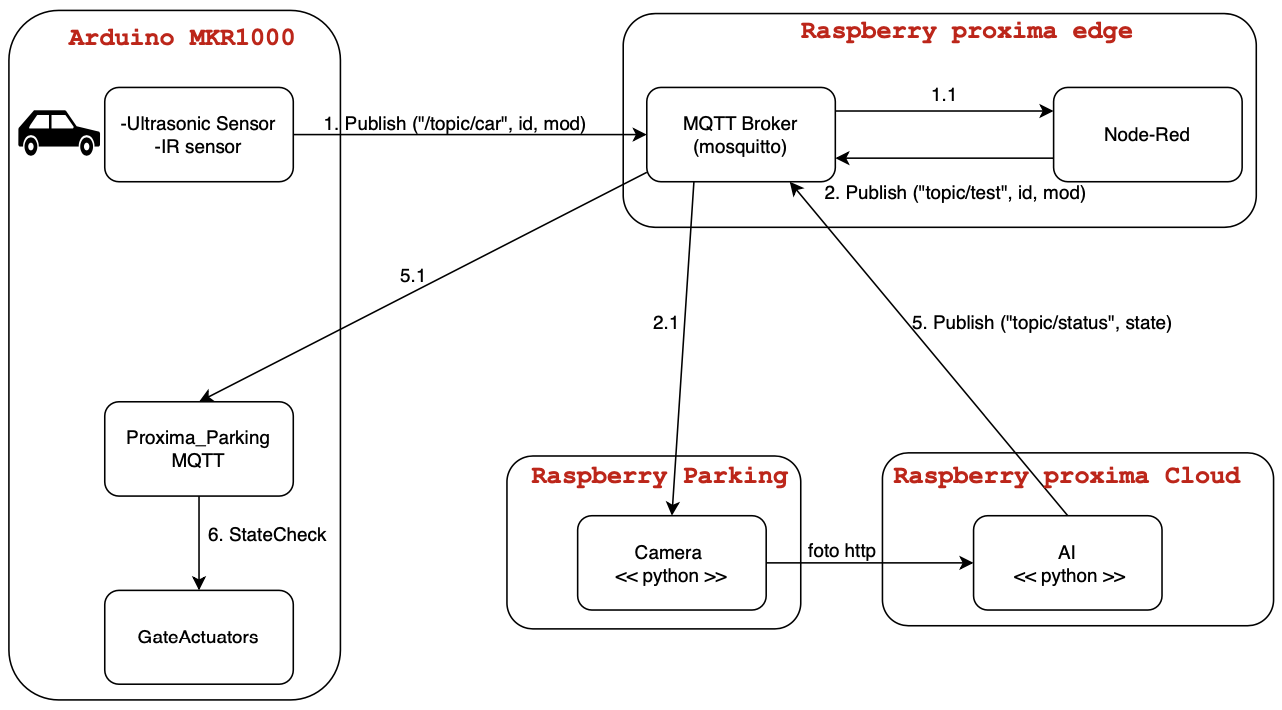


Figura 21 - Diagramma di allocazione della soluzione finale

Nel seguente diagramma è mostrata la dislocazione fisica di tutte le componenti hardware e di tutte le componenti software della soluzione adottata.

1. ISTALLAZIONE SU PROXIMA CITY

Per avviare il simulatore è necessario eseguire il software a partire dalla classe Home. Tale classe è una interfaccia che permette di monitorare un veicolo e richiama una seconda interfaccia per inserire le informazioni del veicolo. In particolare sono richiesti:

1. Sitografia

1. Mosquitto, [https://mosquitto.org](https://mosquitto.org/)  
2. Node-RED, [https://nodered.org](https://nodered.org/)

3. JMeter, [https://jmeter.apache.org](https://jmeter.apache.org/)  
4. Integrazione di un Gate, <https://github.com/angelopetraccaro/ProgettoASSD>