**Programmazione per l'IoT - Laurea Magistrale in Informatica Applicata - Università degli Studi di Urbino Carlo Bo**
**Data di pubblicazione 15/03/2022 - DOI: 10.5281/zenodo.6359424**

# *ParkMe!*: una soluzione IoT low cost di *smart parking*

Pio Mario Jonathan Maiori[1]*, Emanuele Lattanzi[2]

**Sommario**

Nella realtà esistono già *smart city* con sistemi di parcheggio che offrono servizi di *smart parking*, ma molti di questi si basano su tecnologie costose o che richiedono una manutenzione speciale del manto stradale, con costi proibitivi per piccoli enti o per i privati.

Il progetto *ParkMe!*, realizzato come prova finale per l'esame di *Programmazione per l'Internet of Things* presso l'*Università degli Studi di Urbino Carlo Bo*, analizza le dinamiche presenti in un qualsiasi parcheggio e vuole proporsi come soluzione alternativa low cost di *smart parking*, utilizzando componenti hardware tipiche dell'IoT, economiche ma efficienti, e configurazioni modulari per ridurre la complessità della manutenzione.

Dopo un'attenta analisi sullo stato attuale dell'arte e sui requisiti di sistema riguardanti le dinamiche gestionali tipiche di un parcheggio, è stato realizzato un modellino in scala 1:45 con tutte le componenti necessarie alla simulazione. A tal proposito, sono stati sviluppati anche i software per l'edge e per il cloud computing e implementate le interfacce di comunicazione tra le varie parti.

Lo studio condotto ha così portato alla realizzazione di una piattaforma di *smart parking* dinamica, modulare e a costi relativamente bassi, capace di offrire funzionalità di base come la ricerca e la prenotazione di un posto libero all'interno di un parcheggio.

Tale progetto può essere facilmente replicato nella realtà purché lo si applichi all'interno di parcheggi al coperto. Da qui potranno muoversi ulteriori studi e analisi sulle dinamiche presenti all'aperto, sulle criticità riscontrabili e sulle possibili soluzioni per adattare questo progetto a qualsiasi parcheggio.

**Keywords**

IoT — Smart parking — Smart city — Edge computing — Cloud computing — Raspberry Pi — HC-SR04

[1] *Laurea Magistrale in Informatica Applicata, Università degli Studi di Urbino Carlo Bo, Urbino, Italia*
[2] *Docente di Programmazione per l'Internet of Things, Università degli Studi di Urbino Carlo Bo, Urbino, Italia*
***Corresponding author**: p.maiori@campus.uniurb.it

## Introduzione

Il progetto *ParkMe!* è stato realizzato con l'obiettivo di mettere in pratica tutte le nozioni apprese durante il corso di *Programmazione per l'Internet of Things*, tenuto dal docente *Emanuele Lattanzi* nell'A.A. 2021 / 2022 del corso di *Laurea Magistrale in Informatica Applicata* dell'*Università degli Studi di Urbino Carlo Bo*.

Il sistema progettato è in grado di mettere in comunicazione sensori, attuatori, single-board computer, server e client in un unico ambiente complesso, completo ed eterogeneo.

L'idea alla base intende analizzare le dinamiche che sussistono, realmente o potenzialmente, all'interno di un parcheggio intelligente: dare la possibilità ai guidatori di veicoli di ricercare un posto auto libero in uno dei parcheggi attivi e affiliati al sistema e di prenotarlo, tramite una piattaforma digitale, con pochi semplici click.

Nella realtà esistono già esempi di *smart city* con parcheggi intelligenti in grado di offrire servizi analoghi di *smart parking* [1], ma molti di essi si basano su tecnologie costose o che richiedono una manutenzione speciale del manto stradale, con costi fattibili per la gestione nelle grandi città ma proibitivi per i comuni più piccoli o per i privati.

*ParkMe!* vuole proporsi come soluzione low cost di *smart parking*, adatto a chiunque voglia rendere intelligente un qualsiasi parcheggio pubblico o privato e offrire un servizio di pubblica utilità ai guidatori.

La presente relazione analizza le fasi di realizzazione del progetto, le componenti in gioco, le interfacce implementate e i protocolli di comunicazione impiegati. Illustra i risultati ottenuti nella fase di sperimentazione ed elenca alcuni spunti e riflessioni sull'utilità, sull'innovazione e sulla replicabilità di tale progetto nel mondo reale.

## 1. Panoramica sul sistema

Il sistema di parcheggio intelligente è stato progettato realizzando un modellino in scala 1:45 di un ipotetico parcheggio di 10 posti auto collocati su un unico livello, tenendo conto

di tutte le proporzioni e le dimensioni reali dei posti, delle pertinenze e dei veicoli coinvolti. [2]

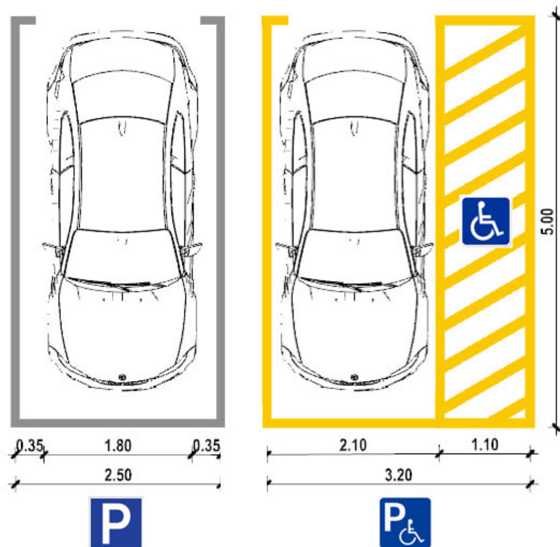


**Figura 1.** Dimensioni reali di posti auto standard e riservati ai portatori di handicap.

I posti auto contemplati nel progetto appartengono a una delle seguenti tipologie:

- **Standard:** posti standard, accessibili da chiunque in possesso di un veicolo idoneo, senza alcuna limitazione;
- **Rosa:** posti riservati ai veicoli di donne incinte e di genitori con almeno un bambino di età inferiore ai 2 anni [3];
- **Handicap:** posti riservati ai veicoli di portatori di handicap.

Ai fini del progetto non sono state considerate altre tipologie di posti (ad es.: carico / scarico merci, bus, taxi) e altre categorie di veicoli (ad es.: trasporto urbano, furgoni, forze dell'ordine), ma data la modularità delle componenti impiegate (sia hardware che software) è naturalmente possibile riadattare il progetto per una loro inclusione.

Ciascun posto è stato mappato all'interno del parcheggio e catalogato nel sistema con tutti i dati relativi alle dimensioni e al posizionamento, per poter essere successivamente renderizzato dal client tramite il viewer dinamico sviluppato, grazie al quale l'utente finale può visualizzarne lo stato di disponibilità e, se in possesso dell'opportuno permesso, prenotarlo.

Gli utenti registrati nel sistema possono essere in possesso di uno o più permessi, ad esempio pass di accesso con data di inizio e di fine validità, i quali concedono la possibilità di prenotare il posto della tipologia associata.

Lo stato sulla disponibilità dei posti può assumere uno dei seguenti valori:

- **Libero:** posto disponibile, pronto a essere occupato fisicamente da un veicolo idoneo o a essere prenotato da un utente con permesso valido;
- **Occupato:** posto già occupato fisicamente da un veicolo e non prenotabile;
- **Prenotato:** posto non ancora occupato fisicamente da un veicolo ma prenotato a distanza da un utente con permesso valido;
- **Fuori servizio:** posto il cui stato di disponibilità è ignoto a causa di disservizi tecnici e, pertanto, non prenotabile.

Con queste premesse iniziali, la topologia del parcheggio progettato assume i seguenti connotati:

- I posti auto 1, 2, 3, 6, 7 e 8 sono di tipo standard;
- I posti 4, 5 e 10 sono di tipo handicap;
- Il posto 9 è di tipo rosa.

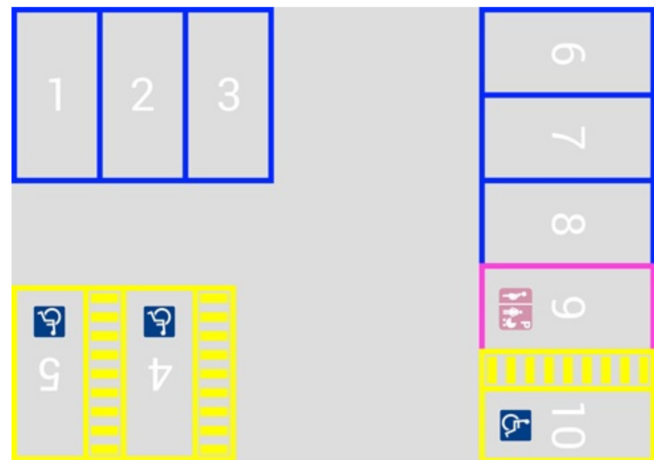


**Figura 2.** Topologia del parcheggio progettato.

## 2. Componenti

Di seguito sono elencate tutte le componenti coinvolte all'interno del modellino, complete di specifiche hardware e funzionali.

**Materiale da bricolage** Per la realizzazione della struttura fisica del modellino è stato impiegato del materiale tipico da bricolage: fogli di cartoncino e di carta, elementi in legno e colla.

**Breadboard a 830 punti [4]** Alla base di qualsiasi collegamento fisico tra le varie componenti, le breadboard a 830 punti permettono di collegare le varie componenti hardware senza il bisogno di ricorrere alla saldatura. Sono state utilizzate 3 breadboard per collegare tutte le componenti ai single-board computer.

**Cavetti Jumper [5]** Colorati e con pin di tipologia omologa (maschio / maschio, femmina / femmina) o eterologa (maschio / femmina), i cavetti jumper collegano le componenti hardware alle breadboard e, tramite queste, ai single-board computer.

**Resistori [6]** Necessari per abbassare la tensione elettrica all'interno dei circuiti, i resistori consentono alle componenti elettroniche di ricevere la corrente con il giusto voltaggio. Sono state utilizzate resistenze a 1 kΩ, 470 Ω, 220 Ω, 100 Ω, 10 Ω e, laddove necessario, combinate insieme in serie per ottenere il livello di tensione desiderato.

**LED RGB [7]** Impiegati come attuatori, i LED permettono di visualizzare lo stato dei posti mediante un colore differente[1]. Sono stati utilizzati 10 LED RGB di tipo catodo, uno per ogni posto.

**Sensori di prossimità a ultrasuoni HC-SR04 [8]** Impiegati per la fase di sensing, i sensori di prossimità a ultrasuoni sono in grado di misurare la distanza con l'ostacolo più vicino emettendo un segnale a ultrasuoni e segnalando il ritorno dell'eco, in un intervallo spaziale tra i 2 e i 500 cm e con una risoluzione di 0.3 cm. Sono stati utilizzati 10 sensori, uno per ogni posto.

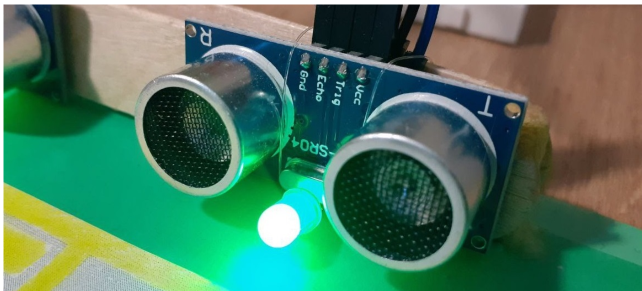


**Figura 3.** Sensore di prossimità e LED installati per un posto.

**Raspberry Pi 3A+ [9]** Alla base dell'edge computing trova posto il Raspberry Pi 3A+: dotato di un processore quadcore 64bit a 1.4 GHz, RAM da 512 MB, dual WiFi a 2.4 e a 5 GHz ma porta Ethernet assente, è un computer single-board in grado di raccogliere ed elaborare i dati in loco e di comunicare al server i risultati dell'elaborazione soltanto quando necessario. Sono stati utilizzati 2 Raspberry, ognuno in grado di gestire al massimo 5 posti auto.

**USB Dongle WiFi TP-Link TL-WN823N [10]** Per la realizzazione della topologia di rete lineare multi-hop, i Raspberry sono stati configurati come wireless access point grazie al dongle WiFi TL-WN823N della TP-Link. I driver sono stati scaricati e compilati appositamente per il processore ARM del Raspberry.

**Server [11]** Il server utilizzato per il cloud computing, offerto da IONOS, è un hosting virtuale condiviso senza permessi di root. Accessibile tramite un URL dedicato, la piattaforma può contare su 2 GB di RAM, su memoria SSD e su un processore virtuale scalabile. NGINX è il web server abilitato in combinazione con Apache e con OPcache attiva. Supporta PHP 7.4 e MySQL 5.7 con motore MariaDB.

**Client** La fase finale di consultazione e di fruizione dei dati elaborati avviene tramite un'interfaccia web semplice e responsiva, adatta sia alla visualizzazione tramite desktop che da mobile. A corredo del front-end, è stata sviluppata anche una semplice web app per Android, compilata con target SDK 31.

## 3. Assemblaggio

Di seguito sono elencati i passaggi salienti del processo di assemblaggio del modellino.

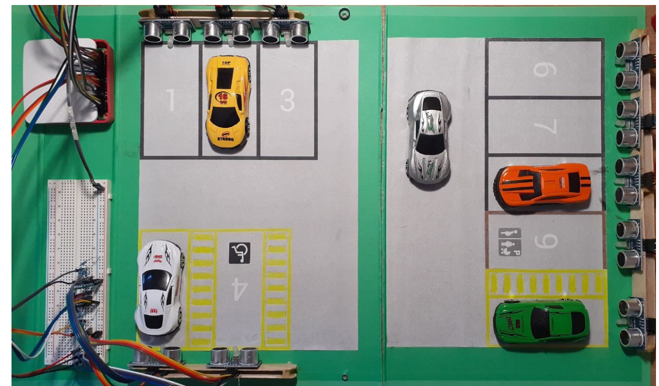


**Figura 4.** Modellino completo di componenti e di elementi di test.

1. **Progettazione struttura:** la piantina del parcheggio è stata realizzata tenendo conto sia di tutte le proporzioni in scala della topologia del parcheggio progettato che delle dimensioni delle componenti elettroniche da reperire. Successivamente la piantina è stata stampata e incollata su supporti in cartoncino;

2. **Reperimento componenti:** sulla base dei posti auto così stabiliti sono state acquistate tutte le componenti elettroniche richieste e tutto il materiale da bricolage necessario alla realizzazione del modellino;

3. **Realizzazione modellino:** il modellino è stato realizzato considerando la topologia della piantina e il posizionamento delle componenti elettroniche, predisponendo i punti di installazione delle stesse. Per una maggior praticità e per garantire una certa modularità, il modellino è stato suddiviso in 2 settori da 5 posti ciascuno;

[1] Rosso - occupato, verde - libero, blu - prenotato e spento - fuori servizio.

4. **Installazione componenti:** tutte le componenti elettroniche sono state installate, fissate nel modellino, collegate alle breadboard e, queste ultime, ai Raspberry;

5. **Test preliminare di collaudo:** i Raspberry sono stati avviati: alcuni snippet di codice sono stati scritti per testare la piena funzionalità di tutte le componenti elettroniche e porre rimedio a collegamenti errati o difettosi.

## 4. Sviluppo dei programmi

Al termine del processo di assemblaggio del modellino, è seguita la fase di progettazione e implementazione delle soluzioni software adottate per il funzionamento di tutte le componenti del progetto.

### 4.1 Raspberry Pi

La programmazione dei due Raspberry Pi ha avuto come obiettivo l'installazione e lo sviluppo, per ciascuno di essi, del software strettamente necessario a renderli completamente indipendenti e autonomi, richiedendo a regime un apporto pari a zero di manutenzione da parte di utenti esterni. Grazie alla straordinaria economicità del modello 3A+ (27 € circa) [12], quest'ultimo consente di rendere intelligente qualsiasi aspetto della vita quotidiana a un costo davvero irrisorio.

Si è deciso di installare il sistema operativo Raspberry Pi OS Lite (versione 5.10) [13] che include il kernel di base Linux con poche altre dipendenze e librerie accessorie, con accesso tramite CLI. In tal modo, rinunciando a tutte le librerie grafiche per la renderizzazione del desktop e di tutti i tool di gestione degli strumenti tramite GUI, il sistema operativo è leggero, veloce e occupa davvero poco spazio sulla scheda microSD da 8 GB installata e, in esecuzione, sui 512 MB di RAM.

A questo, è stato aggiunto e installato InfluxDB (versione 1.8.10) [14] – compatibile con Raspberry Pi 3 – per il salvataggio della sequenza temporale dei dati rilevati tramite i sensori e per la fase successiva di interrogazione degli stessi per l'invio dei risultati al server remoto. Si sarebbe preferito installare la versione più recente di InfluxDB – ovvero la versione 2.1 – ma come riportato nella documentazione ufficiale [15] non è compatibile con il sistema considerato. Ad ogni modo, la versione 1.8 installata lavora egregiamente per le finalità del progetto e senza troppi problemi: l'esecuzione e l'utilizzo diretto sul Raspberry fa sì che i dati puntuali siano salvati e letti a tempo zero, evitando qualsiasi latenza di rete che si sarebbe presentata invece durante il salvataggio in remoto su un server esterno di InfluxDB. Con tale configurazione, si è pertanto deciso di gestire personalmente la lettura e la scrittura dei dati e di non installare il plugin Telegraf [16] in quanto avrebbe inutilmente appesantito il sistema.

Successivamente, è stato installato Python (versione 3.9.2) e tutte le librerie e le dipendenze funzionali per l'utilizzo dei sensori, dei LED e di InfluxDB all'interno degli script applicativi.

Per il Raspberry configurato come wireless access point, si è provveduto a compilare e a installare i driver Linux per processore ARM per il dongle WiFi della TP-Link [17]. In seguito, sono state configurate le interfacce di rete `wlan0` e `wlan1` e le relative sottoreti tramite `networkd` per il collegamento del Raspberry a Internet e per il routing dei pacchetti tra il dongle WiFi e Internet. Ecco in sintesi la configurazione delle interfacce:

- `wlan0` è stata configurata come interfaccia principale per essere associata al modulo WiFi on-board della Raspberry Pi;
- `wlan1`, invece, è stata configurata come interfaccia accessoria per essere associata al dongle WiFi come wireless access point.

Con tale configurazione, in caso di malfunzionamento della chiavetta il Raspberry è comunque in grado di accedere a Internet e comunicare eventualmente il malfunzionamento al server remoto (funzionalità non implementata in questo progetto).

Sono state attivate le regole di firewall tramite `iptables` per la gestione dei flussi di INPUT, OUTPUT e FORWARD dei pacchetti e sono state abilitate le porte e i specifici protocolli di rete impiegati per la comunicazione. La policy impostata di default per i flussi è DROP.

Il programma sviluppato è stato progettato secondo il paradigma OOP (Orientato a oggetti) e ciò ha consentito di sviluppare classi modulari per rappresentare e gestire tutte le componenti in gioco. Inoltre, nessun parametro di configurazione è stato inserito nel codice: tutte le proprietà dei sensori e i parametri di connessione a InfluxDB e al server remoto sono stati collocati in un file di configurazione `config.ini` che viene aperto e letto dal programma nella fase di inizializzazione. Sempre in questa fase, vengono istanziati gli oggetti per la gestione di InfluxDB, della connessione HTTPS con il server remoto e dei posti auto (con i sensori e i LED).

A questo punto, avendo tutte le componenti istanziate, verificate e pronte all'uso, il programma entra nella fase di esecuzione con modalità **while-loop**:

1. Per ogni posto viene effettuato il sensing sulla distanza rispetto all'ostacolo più vicino (un'auto), viene calcolato lo stato di disponibilità mediante un valore soglia prestabilito e viene comunicato tramite il colore del LED a seconda se il posto sia libero oppure occupato. Tra le sequenziali operazioni di sensing è stata impostata una fase di `sleep` di durata minima (0.1 secondi) per ridurre al minimo eventuali interferenze dovute dal ritardo dell'eco prodotto dai sensori precedenti;

2. Il dato calcolato relativo alla distanza viene momentaneamente salvato come dato puntuale all'interno di una sequenza temporale nell'oggetto manager di InfluxDB e, pertanto, non ancora scritto sul database;
3. Quando tutti i posti hanno terminato la fase di sensing, viene invocata la scrittura della sequenza temporale dei dati su InfluxDB (una sola scrittura per tutti i posti coinvolti) e viene svuotata la sequenza temporale nel manager;
4. A timeout scaduto di sincronizzazione con il server, il manager di connessione HTTPS avvia la funzione che interroga InfluxDB per ottenere i dati elaborati sullo stato dei posti auto, crea il payload JSON per l'invio dei dati al server e attende, in modalità sincrona, la risposta, salvando i dati ricevuti e aggiornando lo stato dei posti auto se prenotati. Inoltre, salva l'esito di sincronizzazione su InfluxDB per un impiego successivo di analisi statistiche e ripristina il timeout di sync periodico.

Di fondamentale attenzione è stata la gestione della CPU e il consumo energetico che ne consegue.

Utilizzando la funzione `time.sleep()` di Python, il programma sospende temporaneamente l'esecuzione senza occupare la CPU inutilmente [18], risparmiando computazione e, quindi, energia. Inoltre, se il programma viene mandato in esecuzione con il comando `nice` e con parametro di nice positivo, il sistema schedula il programma per essere ancor meno CPU intensivo e per mantenere quanto più libera la CPU per altri programmi e servizi.

Il programma viene eseguito con il seguente comando:

```
nice -n 19 bash /home/pi/project/execute.sh
```

Tale comando è stato inserito nel file `/etc/rc.local` di Linux per essere eseguito a ogni avvio del sistema.

```
top - 23:49:31 up 10:01,  2 users,  load average: 0.00, 0.12, 0.11
Tasks: 120 total,   1 running, 119 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.2 us,  0.4 sy,  0.4 ni, 97.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    427.6 total,     56.3 free,    137.9 used,    233.5 buff/cache
MiB Swap:    100.0 total,     98.7 free,      1.2 used.    302.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  750 root      39  19   28760  20988   8948 S   3.3   4.8  24:08.21 python
 2483 pi        20   0   11112   2924   2520 R   3.3   0.7   0:00.73 top
    1 root      20   0   33704   8512   6744 S   0.0   1.9   0:06.68 systemd
```

**Figura 5.** Carico medio della CPU mostrato tramite comando `top`.

## 4.2 Server

Tutti i dati raccolti dai Raspberry vengono elaborati e inviati al server remoto che effettua, a sua volta, una rielaborazione minima, aggiorna i dati nel database relazionale e risponde ai singoli Raspberry con l'esito di comunicazione e con i dati di risposta sugli stati rielaborati dei posti auto.

La rielaborazione dei dati interessa gli stati di disponibilità di ciascun posto aderendo a questa logica:

- In assenza di prenotazione viene confermato lo stato di disponibilità (libero o occupato) del posto;
- In presenza di prenotazione, solo se lo stato è libero viene contrassegnato come prenotato, altrimenti (essendo occupato fisicamente da un altro veicolo) la prenotazione viene cancellata e confermato lo stato occupato.

Il database MySQL è strutturato per contenere tutti i dati applicativi senza ridondanze applicando la normalizzazione dei dati e rispetta i seguenti requisiti:

- Ogni parcheggio, collocato in una certa località, ha un nome ed è costituito da uno o più piani (in assenza di piani, si considerano tutti i posti su un unico livello);
- Ogni piano, avente un nome specifico (ad es.: ”Piano Terra”, ”Livello C”) è costituito da uno o più posti;
- Ogni posto, avente un nome specifico (può essere un numero, una lettera ecc. . . ) e dotato di un identificativo univoco, ha determinate dimensioni (lunghezza, larghezza) ed è collocato all'interno del piano ad una certa posizione (rotazione, offset). Per semplicità, un posto viene considerato il pivot dell'intero piano e tutti i posti vengono collocati in relazione al posto pivot più vicino[2]. Ciascun posto è di una determinata tipologia (come visto prima: standard, handicap, rosa ecc. . . ) e ha uno specifico stato di disponibilità (libero, occupato, prenotato), e può essere fuori servizio (se il sensore o il Raspberry non funzionano o se ci sono problemi di connessione);
- Ogni utente, avente uno specifico nome, può accedere alla piattaforma mediante email e password ed è in grado di visualizzare i posti all'interno di un parcheggio e di prenotare i posti liberi solo se in possesso del relativo permesso;
- Ogni permesso associa un utente alla relativa tipologia di posto, e ha un intervallo di validità temporale;
- Ogni prenotazione, con una validità di 15 minuti, associa un utente a un posto libero il cui stato passa su prenotato se e solo se nel frattempo il posto non è stato già occupato da un altro autoveicolo (per l'eventuale latenza tra la ricezione della prenotazione e l'applicazione della stessa al posto selezionato) oppure se nel frattempo un'altra prenotazione concorrente ha già riservato il posto.

[2] Ogni posto viene collocato con un certo offset orizzontale e verticale dal posto pivot associato.

L'applicazione server è stata programmata in PHP e progettata secondo il paradigma OOP e il modello di architettura software MVC (Model View Controller). Data la limitata configurabilità del server (nessun permesso di root né accesso al terminale), non è stato installato alcun framework e, pertanto, le classi utilizzate per il tipo di architettura sono state scritte ex novo e minimali.

Il PDO (PHP Data Object) è stato impiegato per l'accesso al database in localhost, per la gestione della connessione, per la preparazione e l'esecuzione delle query.

L'accesso ai servizi da parte dei Raspberry e dai client avviene mediante API di tipo HTTPS/REST. Si rimanda alla sezione *Protocolli di comunicazione* per una visione più dettagliata di tali dinamiche.

### 4.3 Client

La visualizzazione finale e la fruizione dei dati avviene tramite un'interfaccia web responsive intuitiva, raggiungibile con un qualsiasi browser web o programma in grado di visualizzare una webview.

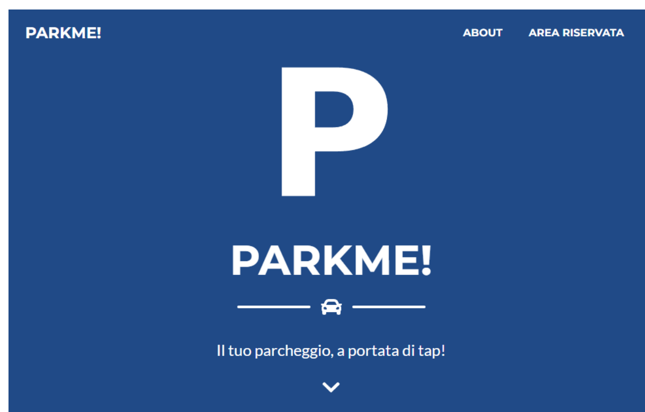


**Figura 6.** Home del portale web.

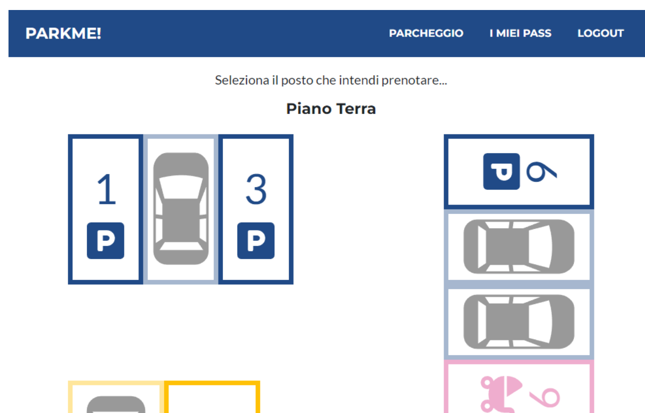


**Figura 7.** Visualizzazione del parcheggio e selezione del posto da prenotare.

Utilizzando un template gratuito [19] di Bootstrap, l'interfaccia è stata programmata utilizzando HTML5, CSS3, JavaScript e jQuery. Le dinamiche di accesso ai dati real-time sono state implementate mediante tecnologia AJAX. Tutte le chiamate alle API di tipo HTTPS/REST sono garantite ai soli utenti loggati, tramite sessione attiva, e rese sicure grazie al token CSRF (Cross-Site Request Forgery), per prevenire attacchi di tipo Cross-Site.

L'applicazione sviluppata per Android (con target SDK 31) incorpora la webview per un accesso da mobile immediato e dedicato. Non sono state previste funzionalità avanzate tramite app.

## 5. Protocolli di comunicazione

L'intero flusso di comunicazione tra i sensori, gli attuatori e gli utenti fruitori della piattaforma avviene mediante un sistema di connessioni tra i Raspberry e il server e tra quest'ultimo e i client. Ogni comunicazione avviene mediante il protocollo di comunicazione sicuro HTTPS, con interfaccia API del cloud server di tipo REST.

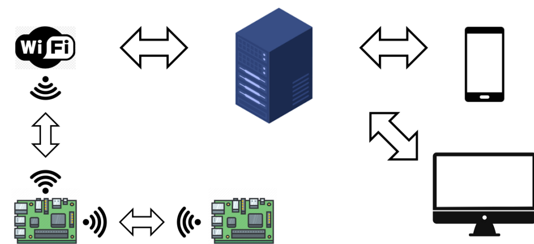


**Figura 8.** Flusso di comunicazione del sistema.

### 5.1 Raspberry Pi $\leftrightarrow$ Server

Ogni Raspberry Pi è collegato a Internet tramite connessione WiFi (protocollo standard 802.11 e varianti). Premesso che l'impiego di un sistema del genere nel mondo reale può avvenire utilizzando un hotspot WiFi mobile dotato di modulo SIM per l'accesso ai servizi di telefonia mobile, è difatti impossibile contare su un indirizzo IP statico. Installare un router con indirizzo IP statico e cablare un intero parcheggio con cavi Ethernet è ovviamente possibile, ma è una soluzione che richiede l'attivazione di una linea telefonica per il parcheggio, la gestione di una manutenzione periodica sul cablaggio di rete e, ovviamente, Raspberry dotati di porta Ethernet.

In questo scenario, l'unico a possedere un indirizzo IP statico e un dominio stabile è il server remoto, raggiungibile da chiunque conosca l'URL di accesso alla piattaforma. Si è reso quindi necessario implementare un sistema di **poll periodico** a opera dei Raspberry (con timeout di 5 secondi) i quali:

1. Raccolti ed elaborati i dati, invocano l'endpoint delle API REST del server inviando un token di autenticazione nell'header della richiesta e un payload JSON nel body contenente gli aggiornamenti sugli stati di disponibilità dei posti;
2. In risposta alla richiesta da parte dei Raspberry, il server verifica il token di autenticazione e, se valido, aggiorna lo stato di disponibilità dei posti. Approfittando della connessione attiva e sfruttando la tecnica del *piggybacking*, comunica ai Raspberry gli aggiornamenti sugli stati rielaborati tenendo conto anche delle prenotazioni in carico.

In tal modo, con una sola comunicazione stabilita, entrambe le parti coinvolte comunicano e ricevono aggiornamenti dalla rispettiva controparte.

Nei payload, gli identificativi univoci dei posti si basano su stringhe di hash generate in fase di configurazione dei Raspberry e associate sulla piattaforma in cloud con le entità nel database relazionale. Questa scelta consente ai posti di possedere un identificativo univoco immutabile nel tempo e permette una gestione dal cloud di eventuali riconfigurazioni (operazioni di aggiunta, modifica o eliminazione di record nel database potrebbero portare a un aggiornamento della chiave primaria, facendo saltare l'associazione con i posti fisici).

Ecco un esempio di payload di risposta dal server durante la chiamata API `setParkUpdate`:


```
{ "status":"ok", "piggyback": {
"L":[ "VG1wRmQwMXFiRGhOVVQwOWZEQT18Mg==",
"VG1wRmQwMXFiRGhOVVQwOWZEQT18NA==",
"VG1wRmQwMXFiRGhOVVQwOWZEQT18NQ==" ],
"O":[ "VG1wRmQwMXFiRGhOVVQwOWZEQT18MQ==" ],
"P":[ "VG1wRmQwMXFiRGhOVVQwOWZEQT18Mw==" ]
} }
```


Con i vincoli di ottimizzazione imposti dal mondo dell'IoT (payload di dati più piccoli comportano minor traffico dati e, quindi, minor consumo di energia), il payload è stato impostato elencando gli ID dei posti per ogni stato[3]. L'altra possibilità era quella di segnalare lo stato di disponibilità per ogni posto, ma avrebbe fatto crescere inutilmente la dimensione del payload.

### 5.2 Raspberry Pi ↔ Raspberry Pi

Nell'ottica di installare un solo hotspot WiFi per l'accesso a Internet all'interno di un vasto parcheggio, soltanto i Raspberry che rientrano nel raggio di azione sono in grado di connettersi, mentre per quelli più lontani si presenta il problema della scarsità o della totale assenza di segnale. Per questo motivo, tramite la configurazione di wireless access point per i Raspberry dotati del dongle WiFi è stato possibile realizzare una rete lineare multi-hop in cui ciascun Raspberry si connette al successivo tramite WiFi, rendendo la riconfigurazione della rete molto semplice. In caso di guasti o di malfunzionamenti su un nodo, con una tale soluzione modulare è facile riconfigurare la rete e sostituire il Raspberry difettoso con uno funzionante.

[3] L - libero, O - occupato, P - prenotato.

### 5.3 Client ↔ Server

Gli utenti finali possono accedere alla piattaforma, come già ribadito, grazie alla webview e, pertanto, tramite un qualsiasi browser web moderno e con protocollo HTTPS. Data la recente versione (Javascript ES6) dei linguaggi utilizzati per la programmazione del client, è sconsigliato l'utilizzo di browser web obsoleti (Internet Explorer o, in generale, browser non aggiornati). Il reperimento iniziale delle informazioni sui parcheggi e il continuo sync con il server avviene mediante API REST con tecnologia AJAX e **poll periodico**. Grazie all'ottimizzazione del payload JSON impiegato nel sync, il consumo di dati per ciascuna richiesta è davvero esiguo.

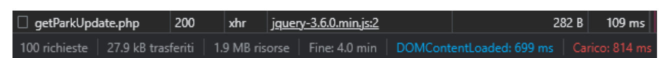


**Figura 9.** Analisi sulla dimensione dei dati trasferiti (27.9 kB in 100 richieste totali).

Ecco un esempio di payload di risposta dal server durante la chiamata API `getParkUpdate`:


```
{ "status": "ok", "data": [
{ "id": "1", "stato": "O" },
{ "id": "2", "stato": "L" },
...
{ "id": "9", "stato": "O" },
{ "id": "10", "stato": "L" } ] }
```


## 6. Risultati

Dopo aver messo in piedi il sistema è seguita la fase sperimentale in cui il sistema è stato sottoposto a test ripetuti. I feedback raccolti sono serviti successivamente per apportare ulteriori migliorie sulla gestione delle componenti, dei dati e dei flussi di comunicazione tra le varie parti, arrivando ai risultati qui di seguito presentati.

### 6.1 Analisi dei dati

Tutti i dati raccolti dai Raspberry sono stati salvati localmente su InfluxDB all'interno del database `parking_garage` mediante utenza con opportuni privilegi di lettura e scrittura. Il linguaggio utilizzato per l'inserimento e la selezione dei dati è InfluxQL [20].

Sono state utilizzate due `measurement`, denominate rispettivamente `parking_lot` e `logs`: la prima adibita alla raccolta di tutte le misure sulle distanze da parte dei sensori a ultrasuoni per ogni singolo posto, mentre la seconda a scopo di debug per verificare lo stato di connessione alle API del server remoto.

**Query su `parking_lot`** Nella fase di sync con il server, la query impostata calcola la media delle rilevazioni degli ultimi 5 secondi per ogni posto. Ottenuto il `resultset`, il programma decreta lo stato di disponibilità (libero, occupato) dei posti mediante i parametri di soglia sulla distanza e imposta il payload JSON.

```
> SELECT ROUND(MEAN("distance") * 100) / 100 AS "distance" FROM "parking_lot" WHERE "time" > now() - 5s AND
 "distance" < 500 GROUP BY "pid";
name: parking_lot                              name: parking_lot
tags: pid=VG1wRmQwMXFiRGhOVVQwOWZEQT18MTA=     tags: pid=VG1wRmQwMXFiRGhOVVQwOWZEQT18Nw==
time                distance                   time                distance
----                --------                   ----                --------
1642510522759139448 47.67                      1642510522759139448 4.47
```

**Figura 10.** Query che calcola la media delle distanze per ogni posto.

In tal modo, il server remoto non necessita dei valori raw sulla distanza, anche perché ogni posto possiede una configurazione distinta e, pertanto, un differente valore soglia. Lato applicativo, ciò che interessa è semplicemente lo stato di disponibilità di ciascun posto. È, quindi, garantita la separazione dei contesti tra edge e cloud computing.

**Query su `logs`** Le query sui log, eseguite soltanto da terminale e a scopo di debug, sono state utili per rilevare le percentuali di connessioni riuscite e fallite con le API server nell'ultima ora e nell'ultimo giorno.

```
> SELECT COUNT("payload") AS "total" FROM "logs" WHERE "time" > now() - 1h GROUP BY "context", "category";
name: logs                              name: logs
tags: category=error, context=http      tags: category=success, context=http
time                total               time                total
----                -----               ----                -----
1642506813806568551 15                  1642506813806568551 494
```

**Figura 11.** Query che calcola il numero di connessioni riuscite e fallite in un'ora.

```
> SELECT COUNT("payload") AS "total" FROM "logs" WHERE "time" > now() - 1d GROUP BY "context", "category";
name: logs                                 name: logs
tags: category=error, context=http         tags: category=success, context=http
time                total                  time                total
----                -----                  ----                -----
1642423898671429508 177                    1642423898671429508 7627
```

**Figura 12.** Query che calcola il numero di connessioni riuscite e fallite in un giorno.

Durante la fase di sperimentazione, la percentuale media giornaliera di tutte le connessioni riuscite è stata pari al 97,73% e, conseguentemente, tutte quelle fallite al 2,27%. La percentuale media oraria ha seguito lo stesso trend (97,05% riuscite e 2,95% fallite), facendo intuire che la reperibilità della connessione e/o del server è di circa il 97% durante tutto il giorno, senza aggravi in specifici momenti della giornata.

### 6.2 Gestione degli errori

Considerata l'incidenza non irrisoria di circa il 3% di connessioni fallite sul totale, è stato deciso di applicare un timeout di connessione HTTPS relativamente basso (5 secondi), onde evitare di far attendere troppo tempo il programma per la ricezione dei dati (e, in alcuni casi, la non ricezione). L'intero snippet di codice adibito a tale funzionalità è stato ovviamente incapsulato in un blocco `try - catch` per gestire eventuali eccezioni sollevate dal manager HTTPS causate da errori di connessione.

Una logica simile ha interessato anche la gestione del manager di InfluxDB, il quale solleva eccezioni qualora non riesca a connettersi al database (soprattutto nella fase iniziale di startup in cui InfluxDB richiede del tempo prima di essere operativo) o non riesca a effettuare query.

### 6.3 Considerazioni sullo sviluppo

Nello sviluppo di software basato sulla tecnica di esecuzione while-loop, è di fondamentale importanza garantire la gestione di qualsiasi errore ed evitare potenziali situazioni di stallo, mantenendo il programma sempre in esecuzione. Nel mondo dell'IoT questa condizione è accentuata dal fatto che i single-board computer utilizzati e collocati nei posti più disparati (ad es.: sotto terra) devono essere completamente autonomi e devono richiedere una manutenzione minima (se non addirittura nulla). Non tutti, inoltre, possiedono un indirizzo IP statico o di una connessione a Internet e, pertanto, non possono essere raggiunti tramite terminale SSH e riconfigurati con pochi comandi dal computer dell'ufficio tecnico.

Ecco qui elencate alcune considerazioni emerse durante la realizzazione del progetto.

**While-loop semplici** Realizzare while-loop semplici contribuisce a rendere il codice mantenibile e a evitare errori di esecuzione che possono bloccare il programma. Da evitare la concorrenzialità di più while-loop sullo stesso single-board computer, soprattutto se questi gestiscono risorse in comune o se il cambiamento di stato di uno o più parametri influenza l'esecuzione di altri loop.

**Paradigma OOP** Sebbene sia sempre possibile programmare l'IoT in maniera procedurale (anzi, in molti ambiti è l'unico paradigma a disposizione), è preferibile impostare lo sviluppo mediante il paradigma di programmazione orientato a oggetti per incapsulare le soluzioni sviluppate in opportune classi adibite a uno specifico aspetto della soluzione finale. Così facendo, anche la gestione di errori e di eccezioni diventa più semplice e intuitiva e consente una maggiore mantenibilità del codice.

**Check componenti** I sensori e gli attuatori sono soggetti a usura e a deterioramento come qualsiasi componente elettronico, soprattutto se collocati all'esterno. Per un sistema reattivo ai guasti, è opportuno che i single-board computer effettuino dei check periodici sui sensori e sugli attuatori (se questi presentano interfacce tali da avviare routine automatiche) o verifichino la validità e la fattibilità dei dati raccolti dagli stessi (ad es.: controllo su valori fuori intervallo). Tali informazioni sullo stato di salute delle componenti (inclusi i single-board computer) dovranno essere comunicate in cloud per gestire i guasti da parte di operatori esterni e calendarizzare le manutenzioni per riparare o sostituire le componenti danneggiate.

**Ottimizzazione payload** Il mondo dell'IoT presenta un contesto particolare per la programmazione, vincolando gli sviluppatori a scrivere programmi efficienti da eseguire su poca

RAM e con poca energia. Un passo fondamentale nello sviluppo di software di questa natura riguarda l'ottimizzazione dei payload di richiesta e di risposta di una qualsiasi connessione in ingresso e in uscita. Dovendo minimizzare la quantità di dati da scambiare, è opportuno effettuare a monte una scelta oculata sulla natura dei dati da scambiare e sulla frequenza di invio periodico degli stessi. Sebbene la minimizzazione risultante potrebbe apparire criptica o poco intuitiva, a parità di informazioni scambiate è sempre bene utilizzare payload di dimensione più piccola: meno byte da scambiare significa meno energia da spendere nella comunicazione e meno tempo di attesa per inviare una richiesta e per ricevere una risposta.

**Protocolli di comunicazione** Anche la scelta dei protocolli di comunicazione deve tenere conto di tutte le limitazioni imposte dall'IoT, anche in relazione alle capacità messe a disposizione dai single-board computer impiegati e ai requisiti di sistema richiesti. Sulla base della natura e della direzionalità della comunicazione (unidirezionale o bidirezionale), a parità di funzionalità conviene scegliere il protocollo di comunicazione con minor sovraccarico computazionale e con una qualità del servizio tale da garantirne sempre la connettività.

**Sync o async?** La gestione di task concorrenti, invocati all'interno del flusso di esecuzione del programma, può avvenire in modalità sincrona o asincrona a seconda se sia necessario attendere il risultato della loro esecuzione o se questi possono agire indipendentemente e terminare senza vincoli. Se si decide di impostare una gestione sincrona di queste attività (ad es.: invocazione delle API del server remoto), bisogna considerare di impostare un timeout onde evitare di condurre l'esecuzione del programma in stallo. In questi casi va gestita anche un'azione di fallback non banale, capace di prendere provvedimenti affinché l'eventuale problema riscontrato possa essere risolto o, quantomeno, notificato.

In linea di massima, una gestione ottimizzata e controllata del software e dell'hardware e lo sviluppo di soluzioni semplici e modulari aiutano ad abbattere le risorse richieste, sia in termini di capitale umano che economico, e a rendere fattibili e mantenibili progetti che, altrimenti, rimarrebbero solo su carta.

## 7. Conclusioni

Il progetto *ParkMe!* così impostato può essere facilmente replicato nella realtà purché l'installazione avvenga in parcheggi coperti (al chiuso o sotterranei) e con accesso a Internet mediante un singolo hotspot WiFi. Purtroppo non sono stati effettuati dei test all'aperto in cui condizioni meteo avverse quali pioggia, nebbia o neve potrebbero incidere significativamente sull'attività di sensing e far registrare valori anomali sulla distanza calcolata dai sensori.

Una possibile soluzione può essere sviluppata applicando analisi statistiche più dettagliate sulle misurazioni effettuate da un singolo sensore, calcolandone la variazione temporale e scartando valori fuori scala od oscillazioni troppo repentine non riconducibili all'ingresso e all'uscita di un veicolo. Tali operazioni possono essere applicate anche sui dati raccolti da posti con 2 sensori installati, aggregandoli nella maniera più opportuna per costruire nuovi coefficienti di misurazione.

Un altro problema presente nella realtà riguarda la modalità di segnalazione della prenotazione del posto ai guidatori che sono presenti in loco. Nel progetto sono stati utilizzati i LED come attuatori che, con una luce blu, segnalano la prenotazione presa in carico dal sistema per un determinato posto, ma manca ovviamente un sistema di protezione di tale posto prenotato, in grado cioè di proibire l'accesso ad altri utenti. L'installazione di dissuasori all'ingresso di ogni posto e la configurazione di questi con i Raspberry può essere una soluzione a questo problema, ma cosa succede se un posto auto viene occupato da un motociclo malgrado il dissuasore attivo? Oppure, come deve essere gestito il passaggio di una persona o di un animale davanti al sensore di un posto prenotato?

Ulteriori migliorie possono essere apportate anche sulla modalità di sblocco del posto all'arrivo dell'utente autorizzato (ovvero, con la prenotazione a suo carico), al momento manuale e tramite un apposito pulsante nella webview.

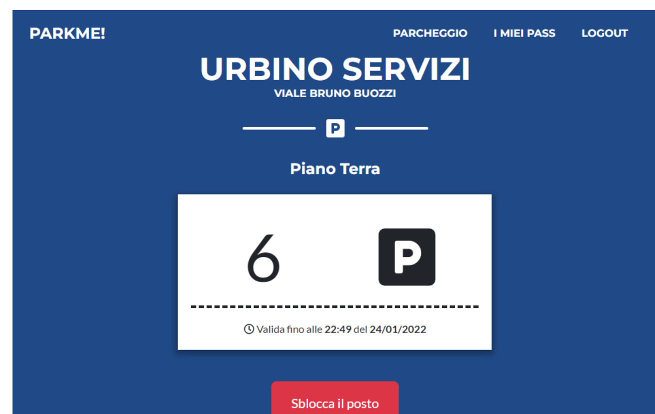


**Figura 13.** Ticket virtuale di prenotazione effettuata e tasto di sblocco del posto.

L'installazione di una telecamera all'ingresso di ogni posto e lo sviluppo di software in grado di riconoscere autonomamente la targa del veicolo può automatizzare tale processo, ma richiede ulteriori componenti hardware, un carico maggiore sui Raspberry e una gestione dei veicoli associati per ogni utente, rendendo più costosa la configurazione di tale sistema per ciascun posto e più complicata la fase di registrazione dell'utente.

Per quanto riguarda il client, un ulteriore sviluppo dell'applicazione potrebbe essere dedicato all'implementazione di notifiche push per comunicazioni dirette e in tempo reale da parte del sistema, in merito ad avvisi sulla scadenza della prenotazione o sulla disponibilità di posti liberi nei propri parcheggi preferiti e in una certa fascia oraria di interesse. Inoltre, funzionalità avanzate come la prenotazione automatica di un

posto in giorni e in fasce orarie prestabilite potrebbero risultare molto utili agli utenti lavoratori sempre alla ricerca di un posto libero nei pressi del proprio luogo di lavoro.

Tanti altri aspetti di questo progetto possono essere migliorati, sviluppati e adattati a seconda delle necessità riscontrate nella realtà. Ora, spetta al lettore decidere su quale ambito continuare la ricerca.

## Riferimenti bibliografici

[1] Economyup. *Che cos'è lo smart parking: le soluzioni e le tecnologie per il parcheggio intelligente*. https://www.economyup.it/mobilita/che-cose-lo-smart-parking-le-soluzioni-e-le-tecnologie-per-il-parcheggio-intelligente/.

[2] BibLus-BIM. *Progetto parcheggi: normativa, dwg e pdf relazioni tecniche*. https://bim.acca.it/progetto-parcheggi-dwg/.

[3] Studio Cataldi. *Parcheggio rosa: cos'è e come funziona*. https://www.studiocataldi.it/articoli/42797-parcheggio-rosa-cos-e-e-come-funziona.asp.

[4] ELEGOO. *ELEGOO Solderless Breadboard Kit*. https://www.elegoo.com/products/elegoo-solderless-breadboard-kit.

[5] ELEGOO. *ELEGOO Multicolored Dupont Wire Kit*. https://www.elegoo.com/products/elegoo-multicolored-dupont-wire-kit.

[6] ELEGOO. *ELEGOO 17 Values 1% Resistor Kit Assortment*. https://www.elegoo.com/products/elegoo-resistor-kit.

[7] ELEGOO. *ELEGOO LED Assortment Kit*. https://www.elegoo.com/products/elegoo-led-assortment-kit.

[8] ELEGOO. *ELEGOO Ultrasonic Distance Sensor Module Kit*. https://www.elegoo.com/products/elegoo-ultrasonic-sensor-kit.

[9] Raspberry Pi. *Buy a Raspberry Pi 3 Model A+*. https://www.raspberrypi.com/products/raspberry-pi-3-model-a-plus/.

[10] TP-Link Italia. *TL-WN823N — Mini Scheda Wireless N300 USB*. https://www.tp-link.com/it/home-networking/adapter/tl-wn823n/.

[11] IONOS. *WordPress Hosting - Un anno a 1 €/mese*. https://www.ionos.it/hosting/hosting-wordpress.

[12] Melopero Electronics. *Raspberry Pi 3 Model A+*. https://www.melopero.com/shop/raspberry-pi/boards/single-boards/raspberrypi3modela/?src=raspberrypi.

[13] Raspberry Pi. *Operating system images*. https://www.raspberrypi.com/software/operating-systems/.

[14] InfluxData. *Downloads*. https://portal.influxdata.com/downloads/.

[15] InfluxData. *InfluxDB OSS 2.1 Documentation*. https://portal.influxdata.com/downloads/.

[16] InfluxDB. *Telegraf Open Source Server Agent*. https://www.influxdata.com/time-series-platform/telegraf/.

[17] TP-Link Italia. *Scarica per TL-WN823N*. https://www.tp-link.com/it/support/download/tl-wn823n/.

[18] Python 3.10.2 documentation. *Time access and conversions*. https://docs.python.org/3/library/time.html#time.sleep.

[19] Start Bootstrap. *Freelancer - One Page Theme*. https://startbootstrap.com/theme/freelancer.

[20] Influx Query Language (InfluxQL). *InfluxDB OSS 1.8 Documentation*. https://docs.influxdata.com/influxdb/v1.8/query_language/.