

# Programmazione di Sistemi Embedded

## Jimmy Challenge Arduino Game

Edoardo Rosa - Matr. 707922  
Federico Torsello - Matr. 702619

15 giugno 2016

## Sommario

In questa relazione si descriverà il progetto del gioco interattivo **Jimmy Challenge**, descrivendo le fasi implementative e le scelte di progettuali.

Jimmy Challenge è stato realizzato utilizzando diversi componenti hardware/software, con l'obiettivo di sfruttare ed ottimizzare quando più possibile la board Arduino in un'ottica IoT.

**Capitolo 1** In questo capitolo si introduce il progetto e l'interazione giocatore-sistema.

**Capitolo 2**

# Capitolo 1

## Introduzione

**Jimmy Challenge**<sup>1</sup> è un gioco interattivo che si ispira all'attività di *lock-picking*: aprire un lucchetto o una serratura usando ad esempio un **grimaldello** per manipolare i pistoncini interni per simulare la presenza della chiave originale.

Per poter giocare è necessario alimentare l'**Arduino UNO**<sup>2</sup> e attendere qualche secondo di setup.

Una volta che il setup è completo, l'utente può interagire con Arduino utilizzando diversi componenti hardware che mutano il loro comportamento in base al contesto.

---

<sup>1</sup> "jimmy" in inglese vuol dire grimaldello, da questo il nome del gioco.

<sup>2</sup> Arduino UNO - è una scheda elettronica di piccole dimensioni con un microcontrollore ATmega, utile per creare rapidamente prototipi e per scopi hobbyistici, didattici e professionali. ([https://it.wikipedia.org/wiki/Arduino\\_%28hardware%29](https://it.wikipedia.org/wiki/Arduino_%28hardware%29))

## Capitolo 2

# Progettazione di un sistema Embedded

### 2.1 Analisi dei requisiti

#### 2.1.1 Requisiti funzionali

Il sistema realizzato deve permettere ad uno o più giocatori di competere per sbloccare dei lucchetti. Ad ogni lucchetto sbloccato si supera il livello fino a quando i livelli non terminano e il gioco si dice finito. Durante il gioco sono presenti delle penalità e dei bonus.

#### 2.1.2 Requisiti non funzionali

Le performance utili per la buona riuscita del progetto a cui non si può rinunciare riguardano la rilevazione della distanza e l'invio dei feedback locali/remoti. Il tempo gioca un ruolo importante in questo progetto, quindi si devono evitare inutili ritardi nella rilevazione ed invio delle informazioni.

Tabella 2.1: Caso d'uso 1

ID	UC1
<b>ATTORI</b>	Giocatore
<b>PRECONDIZIONI</b>	Il giocatore vuole iniziare una partita single player
<b>SEQUENZA DEGLI EVENTI</b>	<ol style="list-style-type: none"> <li>1. Il caso d'uso inizia quando un giocatore vuole giocare</li> <li>2. Il giocatore deve alimentare l'Arduino per poter giocare</li> <li>3. Il programma inizia ad eseguire.</li> </ol>
<b>POSTCONDIZIONI</b>	1. Il giocatore gioca.

## 2.2 Casi d'uso

### 2.2.1 Caso d'uso 1

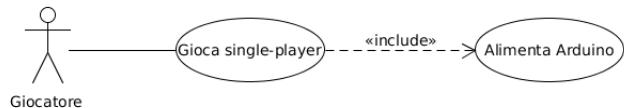


Figura 2.1: Caso d'uso 1

Tabella 2.2: Caso d'uso 2

ID	UC2
<b>ATTORI</b>	Giocatore 1, Giocatore 2
<b>PRECONDIZIONI</b>	I due attori vogliono giocare una partita a Jimmy Challenge multipiattaforma.
<b>SEQUENZA DEGLI EVENTI</b>	<ol style="list-style-type: none"> <li>1. Il caso d'uso inizia quando i due giocatori vogliono giocare online.</li> <li>2. I giocatori devono collegare al PC l'Arduino.</li> <li>3. I giocatori devono far avviare lo script in Python per inviare i dati.</li> <li>4. I giocatori devono effettuare l'accesso al sito.</li> <li>5. I giocatori possono chattare.</li> <li>6. I giocatori devono invitarsi ad iniziare una nuova partita.</li> <li>7. I giocatori iniziano a giocare.</li> <li>8. Ogni giocatore può controllare lo stato della partita dell'altro giocatore.</li> </ol>
<b>POSTCONDIZIONI</b>	1. I due giocatori giocano.

### 2.2.2 Caso d'uso 2

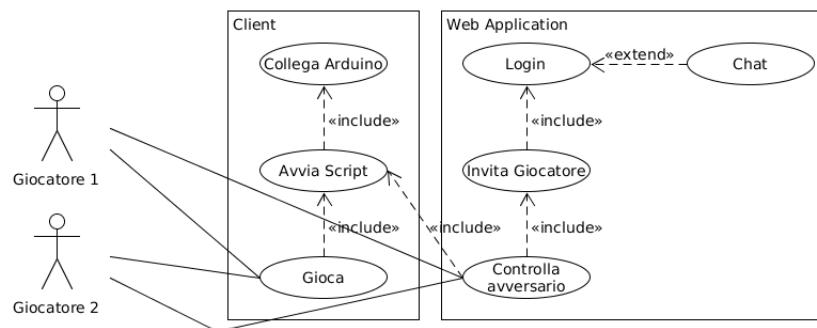


Figura 2.2: Caso d'uso 2

## 2.3 Modeling

Il sistema si suddivide in più parti:

- parte fisica lato client/server;
- parte software lato client/server;
- networking lato client/server.

### 2.3.1 Modellazione della parte fisica - lato client

Dal punto di vista fisico, si ha un Arduino UNO:

- connesso ad una breadboard e a dei componenti hardware attraverso i sui pin digitali
- connesso ad un PC mediante la porta USB (da cui riceve l'alimentazione a 5v).

Ogni componente della breadboard ha un proprio impiego e può essere distinto in **sensore** ed **attuatore** secondo la visione di *sistema reattivo*.

**Reactive system:** è l'ambiente esterno che determina gli eventi che condizionano l'esecuzione del sistema

La connessione seriale via USB serve per inviare i dati campionati dall'Arduino verso il PC. A loro volta questi dati possono servire per creare una interfaccia utente locale (per esempio su terminale) o remota, su browser.

### 2.3.2 Modellazione della parte logica - lato client

Per realizzare il comportamento dinamico del progetto, nella parte software si è replicato il funzionamento di una macchina a stati finiti (*FSM*) che esegue dei task.

**Macchine a stati finiti** (o *automi a stati finiti*): sono il modello nel discreto più utilizzato per modellare sistemi embedded.

Ogni FSM opera in una sequenza di passi (discreti) e la sua dinamica è caratterizzata da sequenze di eventi (discreti).

Un evento discreto avviene ad un determinato istante e non ha durata.

**Decomposizione in task:** principio di progettazione importante, rende modulare il sistema in quanto:

- ogni modulo è rappresentato da un task (compito da eseguire)

- un task può essere decomposto in sotto-task in modo ricorsivo o un task complesso può essere definito come composizione di sotto-task più semplici

Vantaggi:

- separation of concerns
- l'utilizzo di singoli moduli per una migliore comprensibilità del comportamento
- debugging semplificato
- supporto alla modificabilità, all'estensione ed al riuso del codice

Si è fatto uso della programmazione ad oggetti, modellando ogni task (es. *LedTask*, *BuzzerTask*, etc.) come una classe separata che estende la classe base *Task*. In ogni *task* viene fatto l'***inject*** del comportamento da avere. La simulazione della sensazione di lavorare su un sistema multi-tasking e cooperativo è realizzata tramite una schedulazione **round-robin** dei vari task. È stato necessario quindi introdurre uno *Scheduler* per richiamare ciclicamente le funzioni *tick()* dei vari task.

- si è fatto uso della programmazione Object-Oriented per rendere la struttura del codice ingegneristica, lineare e scalabile;
- si sono utilizzate le lambda expression per definire il comportamento di ogni task.

La comunicazione tra i task avviene attraverso variabili condivise. Queste variabili appartengono alla classe Context.

Per quanto riguarda la connessione USB, per scelta progettuale, tutti i dati inviati dall'Arduino al PC sono formattati in **JSON**.

### 2.3.3 Modellazione della parte fisica - lato server

A lato server si ha un Odroid C2 (prodotto da [Hardkernel](#)) con sistema operativo Arch Linux ARM:

- CPU: ARM Cortex-A53 Quad Core 2GHz;
- RAM: 2GB DDR3;
- 40pin GPIOs + 7pin I2S;
- Gigabit Ethernet;
- 4 x USB 2.0.

Il dispositivo è collegato in rete e funge da server per abilitare la modalità multiplayer di Jimmy Challenge dalla parte client tramite lo script python in esecuzione sul dispositivo collegato all'Arduino.

In un'ottica più orientata all'hardware, al server è collegato un LCD 16x2 che mostra alcune statistiche sullo stato attuale dei servizi e delle risorse.

Lo schema B.4 presenta il monitor LCD collegato ai pin GPIO del Odroid con un potenziometro per l'aggiustamento del contrasto.

Al fine di collegare in rete il server si è aggiunto un modulo WiFi (TP-Link TL-WN722N) tramite una porta USB.

#### 2.3.4 Modellazione della parte logica - lato server

Per permettere agli utenti di giocare tra di loro online, il server mette a disposizione:

- una interfaccia per visualizzare gli utenti loggati;
- una chat globale in cui poter invitare gli utenti a giocare;
- una view con lo stato del gioco.

La piattaforma riceve messaggi JSON dallo script lato client tramite richieste POST ad un'interfaccia PHP RESTful. Una volta che le informazioni sono inviate al server, sono elaborate e pubblicate ai vari subscriber tramite Server-Sent Events (sempre utilizzando JSON come modello).

La modalità multiplayer di Jimmy Challenge in sostanza consiste in una applicazione web, quasi real-time, in cui i giocatori possono vedere, tramite l'apposita view, lo stato e l'avanzamento dell'avversario nel gioco.

Per garantire massime performance e sicurezza sono state adottate queste tecnologie:

- **NGINX** come web server;
- **PHP 7** come backend e motore delle parti dinamiche dell'applicazione e che implementa l'interfaccia RESTfull;
- **Javascript** per elaborazione lato client;
- **MySQL** per lo store degli account degli utenti;
- **Redis** come staging area per i dati temporanei da condividere con **tutti** i subscriber;
- **JSON** come modello per i messaggi scambiati;
- **HTTPS over TLS**
- **JSON Web Signature** per i token JWS

Il monitor LCD visualizza alcune statistiche sulle risorse usate dal sistema:

- RAM usata;
- Spazio disco usato della partizione di root ('/');
- Temperatura e percentuale di carico della CPU;
- Indirizzo IP interno;
- Traffico in upload e in download.

I dati stampati sullo schermo LCD sono collezionati e formattati tramite uno script python che utilizza un wrapper del porting di [WiringPi](#) per i dispositivi Odroid.

## 2.4 Design

Il sistema si suddivide in più aree:

1. connessione USB da Arduino UNO verso il PC;
2. I/O locale attraverso Arduino UNO;
3. input per invio dei dati seriali al server attraverso Python;
4. connessione del PC ad un server;
5. feedback remoto su browser;
6. sito internet come GUI remota;
7. gestione del server remoto.

## 2.5 Analysis

Potenzialità software di Arduino UNO sfruttate:

- programmazione Object-Oriented con [Wiring](#)<sup>1</sup>.
  - modularizzazione in classi;
  - utilizzo di oggetti e metodi;
  - information hiding;
  - ereditarietà;
  - polimorfismo.

---

<sup>1</sup>Wiring - linguaggio di programmazione semplice e intuitivo derivato dal C e dal C++

- utilizzo di *lambda expressions*

Potenzialità hardware di Arduino UNO sfruttate:

- sono stati utilizzati tutti i 12 pin di I/O digitale;
- si è cercato di limitare l'utilizzo di **delay** per mantenere le prestazioni ottimali;
- in alcuni casi al posto dei *delay* si è fatto ricorso a dei **custom timer** impiegando il metodo **millis()**;

## Capitolo 3

# Giocare a Jimmy Challenge

### 3.1 Giocare con la mano e con i sensi

L’obiettivo del giocatore è trovare la giusta posizione dei pistoncini e quindi scassinare il lucchetto nel minor tempo possibile. Questo gioco è sia un multiplayer **online** che un single game **offline**.

La posizione dei pistoncini viene assegnata in modo random ad ogni nuovo livello e rimane fissa fino al suo superamento.

Per trovare la posizione attuale dei pistoncini al giocatore basta muovere la mano orizzontalmente in direzione del sensore ad ultrasuoni. (La rilevazione del lucchetto è spiegata più avanti).

Durante le varie fasi di gioco l’utente ha la possibilità di rendersi conto dell’evoluzione del gioco ascoltando i suoni emessi dal buzzer e/o guardando i colori dei LED.

#### 3.1.1 Significato dei suoni e dei colori

- All’avvio del gioco:
  - i LED a 12 pin giallo e rosso fanno un carosello;
  - il LED verde e il buzzer si comportano come quando la corretta posizione del pistoncino non è stata trovata.
- Quando **non** si è trovato il pistoncino:
  - il LED verde emette una luce pulsante;
  - il LED RGB emette una luce continua di color blu chiaro;
  - il buzzer suona due note in modo frenetico.
- Quando si è nell’area della posizione corretta del pistoncino:
  - il LED verde emette una luce fissa;

- il LED RGB continuerà ad emettere una luce continua blu chiaro, ma solo fino a quando non entrerà nello stato di scasso;
- il buzzer suona due note meno freneticamente.

### 3.1.2 Superare un livello

Per superare il livello il ladro deve forzare il lucchetto trovando la giusta posizione dei pistoncini.

Dal punto di vista del giocatore ogni livello rappresenta un lucchetto da aprire e la posizione del pistoncino che compone ogni lucchetto è rappresentata dal range di spazio posta davanti al sensore (in linea orizzontale). Una volta trovata la corretta posizione inizierà il processo di forzatura e di scasso del lucchetto e non si deve muovere per non rischiare di perdere la posizione del pistoncino.

Per forzare il lucchetto è sufficiente tenere una mano davanti al sensore ad ultrasuoni per un tempo limitato, avviando lo stato di scasso. Se il tempo di scasso non viene rispettato o la mano viene rimossa troppo presto, il livello riparte senza salvare i progressi.

- **Non** si supera il livello:

- Se la mano viene spostata dall'area corretta del pistoncino troppo presto, per esempio non si è ancora nello stato di scasso
- Se si rimane troppo tempo nella fase di scasso (il grimaldello è stato "rotto").

- Si può superare il livello:

- Se la mano resta fissa nella posizione in cui si trova il pistoncino, rispettando il tempo dello **stato di scasso** e poi la si agita sempre nell'area del lucchetto per simulare il processo di apertura.

### 3.1.3 Stato di scasso

Se si rispetta il tempo nello stato di scasso e quindi si apre il lucchetto, si supera il livello.

Per indicare lo stato di scasso si è utilizzato un LED RGB.

- Ogni colore ha un significato:

- blu scuro: si sta iniziando a scassinare il lucchetto
- verde: il lucchetto è scassinato.

NB: per passare al livello successivo si deve togliere la mano e riposizionarla nell'area del pistoncino [come se si girasse la "*chiave*"].

- arancio: attenzione, se non si toglie la mano ora si rischia di rompere il grimaldello
- giallo: pericolo di rottura ancora più elevato
- rosso: il grimaldello è stato rotto, quindi il livello deve essere ricominciato.

# Capitolo 4

## Classi astratte

### 4.1 Interfacce sviluppate

1. Audio.h;
2. Input.h;
3. Light.h;
4. LightPwm.h.

#### 4.1.1 Audio.h

Questa classe è l'astrazione della gestione di eventi sonori, in cui ogni suono da emettere dovrebbe avere un numero univoco assegnato. All'interno del sistema viene implementata dalle classi Buzzer.

#### Metodi

- public virtual void playSound(int) = 0

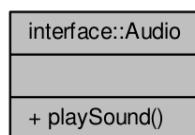


Figura 4.1: Collaboration Diagram - interface::Audio

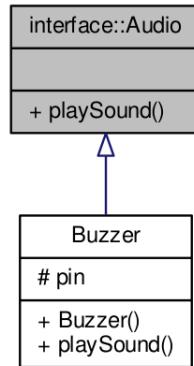


Figura 4.2: Inheritance Diagram - interface::Audio

#### 4.1.2 Input.h

Questa classe è l'interfaccia da implementare per gestire gli eventi di input da parte dei vari device, che nel progetto sono il sonar e il button.

##### Metodi

- `virtual bool readBool() {  
 return false;  
};`
- `virtual int readDistance() {  
 return 0;  
};`

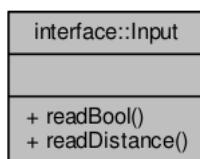


Figura 4.3: Collaboration Diagram - interface::Input

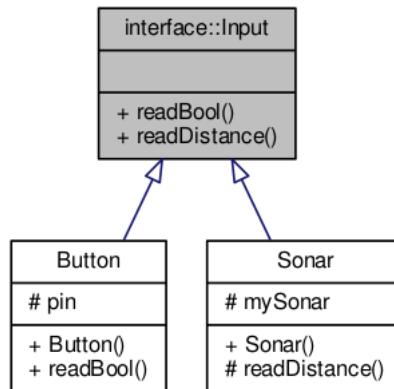


Figura 4.4: Inheritance Diagram - `interface::Input`

#### 4.1.3 Light.h

Questa interfaccia rappresenta la gestione dei LED attraverso la loro accensione/spegnimento al fine di dare un feedback visivo al giocatore.

##### Metodi

- `public virtual void switchOn() = 0;`
- `public virtual void switchOff() = 0;`

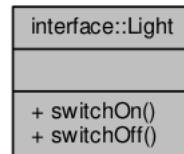


Figura 4.5: Collaboration Diagram - `interface::Light`

#### 4.1.4 LightPwm.h

Interfaccia che estende l'interfaccia Light, permettendo di gestire il feedback luminoso dei LED come la variazione dell'intensità luminosa, agendo sui pin PWM.

##### Metodi

- `public virtual void setIntensity(uint8_t) = 0;`

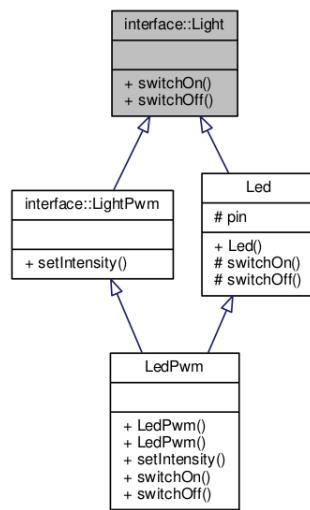


Figura 4.6: Inheritance Diagram - `interface::Light`

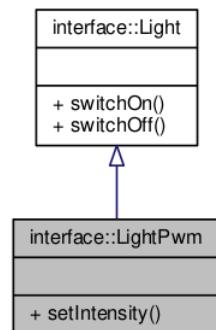


Figura 4.7: Collaboration Diagram - `interface::LightPwm`

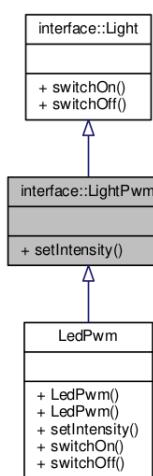


Figura 4.8: Inheritance Diagram - `interface::LightPwm`

# Capitolo 5

## Classi concrete

### 5.1 Classi concrete sviluppate

- Classi utilizzate per l'**input**:
  1. Sonar.cpp, Sonar.h;
  2. Button.cpp, Button.h.
- Classi utilizzate per l'**output**:
  1. Buzzer.cpp, Buzzer.h;
  2. Led.cpp, Led.h;
  3. LedPwm.cpp, LedPwm.h;
  4. LedRgb.cpp, LedRgb.h;
  5. MessageService.cpp, MessageService.h;
  6. Multiplexer.cpp, Multiplexer.h.
- Classi utilizzate per coordinare il **multi-tasking**:
  1. Context.h;
  2. Scheduler.cpp, Scheduler.h;
  3. Timer.h, Timer.cpp;
  4. Task.h.
- Classi dei **task**:
  1. ButtonTask.cpp, ButtonTask.h;
  2. BuzzerTask.cpp, BuzzerTask.h;
  3. LedPwmTask.cpp, LedPwmTask.h;
  4. LedRgbTask.cpp, LedRgbTask.h;
  5. LedTask.cpp, LedTask.h;
  6. SonarTask.cpp, SonarTask.h.

## 5.2 Gestione dell'input

### 5.2.1 Sonar.cpp, Sonar.h

Questa classe permette di leggere la distanza tra la mano del giocatore ed il sensore ad ultrasuoni.

Per ottenere un input molto performante dal punto di vista del tempo e dell'accuratezza, è stata sfruttata la libreria [NewPing](#).

In particolare nel costruttore di Sonar viene istanziato un oggetto NewPing a cui sono passati tre parametri:

- `trigPin` = pin settato come output a cui è fisicamente collegato il trigger del sonar;
- `echoPin` = pin settato come output a cui è fisicamente collegato l'echo del sonar;
- `maxDistance` = limita massimo di distanza gestito oltre il quale la mano non viene rilevata e non si possono creare lucchetti.

#### Metodi

- `int readDistance()`: lettura istantanea della distanza mano-sensore espressa in cm.

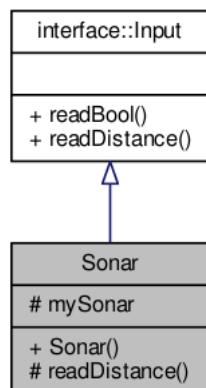


Figura 5.1: Collaboration Diagram - Sonar

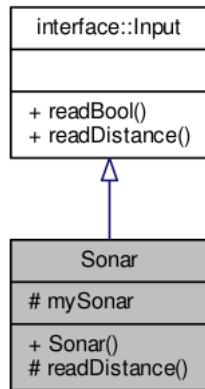


Figura 5.2: Inheritance Diagram - Sonar

### 5.2.2 Button.cpp, Button.h

Anche questa classe implementano l’interfaccia Input. Il loro scopo è riconoscere la pressione del button, evitando gli effetti del bouncing.

Il pin è stato configurato come *INPUT\_PULLUP* in modo da sfruttare la resistenza interna dell’Arduino, ottenendo:

- dei campionamenti con un rumore ridotto;
- floating del button limitato.

Questa scelta di progetto è confermata dal tutorial [Arduino Internal Pull-Up Resistor](#).

#### Metodi

- **bool readBool()** : prende come input il valore del pin disposto in configurazione INPUT\_PULLUP e lo nega. Se tale valore è diverso da true, alla variabile `lastDebounceTime` viene passato il valore `millis()` in modo da resettare il *debouncing*. Se `millis() - lastDebounceTime` è maggiore uguale al `debounceDelay` passato dal file .ino, si ritorna `true`, altrimenti `false`.

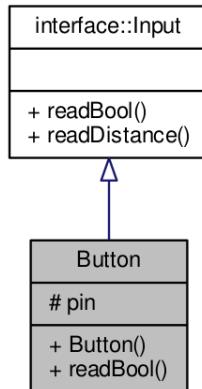


Figura 5.3: Collaboration Diagram - Button

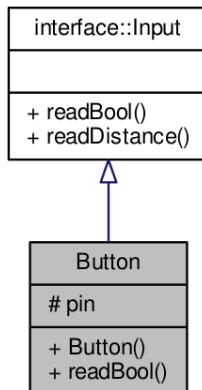


Figura 5.4: Inheritance Diagram - Button

## 5.3 Gestione dell'output

### 5.3.1 Buzzer.cpp, Buzzer.h

Questa classe gestisce il buzzer, permettendo di modulare i suoi da lui emessi. Nel progetto è stato utile per far comprendere al giocatore se il lucchetto è in stato di scasso.

#### Metodi

- `public void playSound(const int sound)` : in base al numero intero passato a questo metodo, viene calcolato il suono che il buzzer deve emettere;
- `private void playMarioTheme()` : una divertente easter egg;

- **private void buzz(int, int)**: attraverso dei `delayMicroseconds(...)` contenuti in un ciclo for, regola il tempo in cui il buzzer si trova nello stato HIGH o LOW;

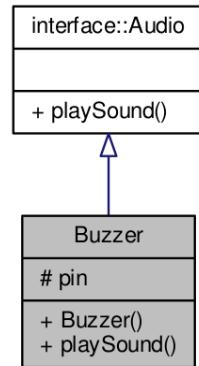


Figura 5.5: Collaboration Diagram - Buzzer

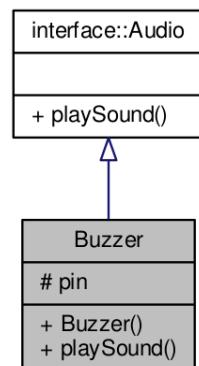


Figura 5.6: Inheritance Diagram - Buzzer

### 5.3.2 Led.cpp, Led.h

Classi per gestire l'accensione/spegnimento dei LED a 2 pin (controllo e massa) connessi all'Arduino.

#### Metodi

- `protected void switchOn()` : metodo che regola l'accensione istantanea del LED;
- `protected void switchOff()` : metodo che regola lo spegnimento istantaneo del LED;

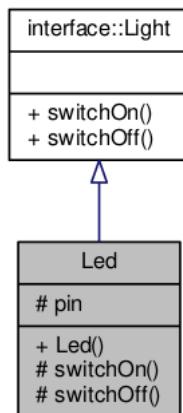


Figura 5.7: Collaboration Diagram - Led

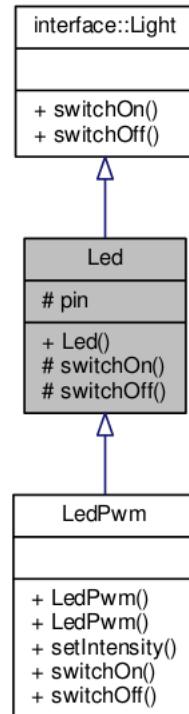


Figura 5.8: Inheritance Diagram - Led

### 5.3.3 LedPwm.cpp, LedPwm.h

Queste classi permettono il controllo l'accensione/spegnimento e dimmeraggio dei LED connessi ai pin PWM di Arduino

#### Metodi

- `public void setIntensity(uint8_t)` : metodo che regola l'intensità luminosa emessa dal LED (compresa tra 0 e 255);
- `public void switchOn()` : metodo che regola l'accensione istantanea del LED;
- `public void switchOff()` : metodo che regola lo spegnimento istantaneo del LED;

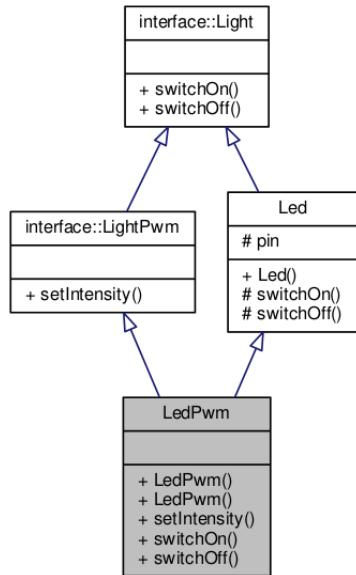


Figura 5.9: Collaboration Diagram - LedPwm

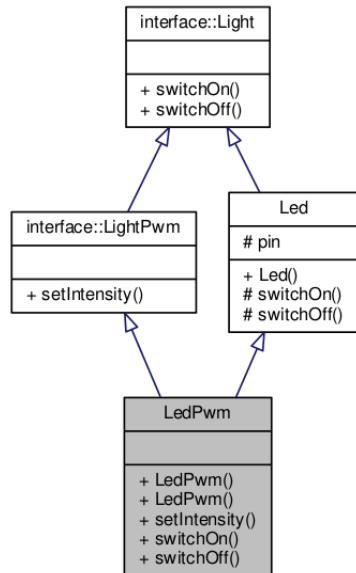


Figura 5.10: Inheritance Diagram - LedPwm

### 5.3.4 LedRgb.cpp, LedRgb.h

Queste classi permettono il controllo dei LED RGB, restituendo un **colore** in base all'intensità luminosa assegnata ad ogni singolo pin.

Nel dettaglio, il controllo viene eseguito istanziando tre oggetti `LedPwm` legati ad ogni colore/pin.

## Metodi

- `public void setColor(int, int, int)` : setta il colore da far vedere al giocatore;
- `protected void switchOn()` : accensione istantanea del LED RGB;
- `protected void switchOff()` : spegnimento istantaneo del LED RGB.

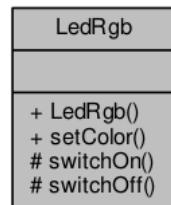


Figura 5.11: Collaboration Diagram - LedRgb

### 5.3.5 Multiplexer.cpp, Multiplexer.h

Classe che permette di gestire il multiplexer e quindi le sue uscite digitali. In questo progetto il multiplexer è strettamente legato ai LED a 12 pin regolando l'accensione/spegnimento di ogni LED interno.

#### Metodi

- `public void switchOn(int)` : serve a riportare un numero da intero a decimale sulle uscite del multiplexer. Nel progetto è stato utile per visualizzare il livello a cui si sta giocando o il carosello sui LED a 12 pin;
- `public void carouselYellow(int)` : carosello del LED a 12 pin giallo, cioè si illumina uno alla volta ogni LED interno per il periodo passato come intero;
- `public void carouselRed(int)` : carosello del LED a 12 pin rosso, con funzionamento identico al `carouselYellow(int)`.

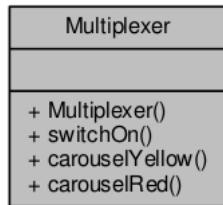


Figura 5.12: Collaboration Diagram - Multiplexer

### 5.3.6 MessageService.cpp, MessageService.h

Questa classe permette di gestire l'invio e la ricezione di messaggi formattati in JSON attraverso la seriale. Per codificare/decodificare i messaggi JSON è stata utilizzata la libreria [ArduinoJsonA.2.1](#), facilitando le operazioni di I/O su seriale.

#### Metodi

- `public void init(const int, const String &)` : inizializza la seriale e stampa il messaggio di welcome;
- `public void setMessage(String)` : quando un messaggio viene letto, è parsato se valido viene inviato un *ACK*;
- `public void errorMsg()` : invia un messaggio di errore utilizzando JSON;
- `public void ackMsg(const String)` : invio di un *ACK* in JSON;
- `public void sendMsg(const String, const String)` : crea ed invia un messaggio JSON al destinatario.
- `public void sendInfo(const int, const int, const uint8_t, const String)` : invia un *service message* in JSON.

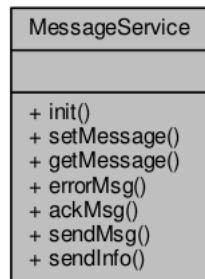


Figura 5.13: Collaboration Diagram - MessageService

## 5.4 Coordinamento dei task

## 5.5 Scheduler

Lo Scheduler coordina e fa cooperare i task del sistema.

Al suo interno è presente il vettore `taskList[...]` in cui sono aggiunti i task da eseguire grazie al metodo `bool addTask(Task* task)`.

Nel caso in cui nello Scheduler non ci fossero task o fossero stati tutti bloccati, il sistema rimarrebbe apparentemente fermo (anche se in realtà il microcontrollore continuerebbe ad eseguire il suo `void loop()`).

Lo Scheduler è **cooperativo** in quanto una volta selezionato un task, questo viene eseguito fino al suo completamento (*run-to-completion*). Ad ogni tick della FSM vengono eseguite atomicamente le azioni associate al nuovo stato.

Lo scheduling è a **priorità statica**, in quanto ad ogni task viene assegnata una priorità che non cambia durante l'esecuzione. La si può considerare come definita implicitamente dall'ordine di inserimento dei task nella *task list*.

## 5.6 Context.h

La classe Context contiene tutti le variabili di stato del programma. I metodi al suo interno sono accessibili da tutti i task in modo da far evolvere il sistema; di seguito la lista di tutti i metodi e del loro significato.

### Metodi

- `bool isPadlockOpen()` : serve a sapere se il giocatore ha aperto il lucchetto;
- `void setPadlockOpen(bool padlockOpen)` : setting dello stato del lucchetto aperto/chiuso;
- `bool isPadlockDetected()` : serve a sapere se il giocatore ha trovato il lucchetto, cioè la distanza giusta dal sensore;
- `void setPadlockDetected(bool padlockDetected)` : set dello stato del padlock trovato o no;
- `void setCurrentDistance(int currentDistance)` : set della distanza del padlock da aprire;
- `int getCurrentDistance()` : restituisce la distanza a cui si trova la mano;

- `void setButtonPressed(bool buttonPressed)` : set dello stato del bottone, se premuto o no;
- `bool isButtonPressed()` : restituisce lo stato del bottone;
- `void setNewLevel()` : crea il nuovo livello da giocare;
- `uint8_t getDelta()` : margine di errore come scarto;
- `uint8_t getLevel()` : restituisce il livello a cui si sta giocando;
- `int getSecret()` : restituisce la distanza a cui si trova il lucchetto;
- `void newRandomNumber()` : genera un nuovo numero random;
- `void setGameOver(bool gameOver)` : setta lo stato del gioco, cioè se è finito o no;
- `bool isGameOver()` : restituisce lo stato del gioco, finito o no;
- `void setDangerLevel (uint8_t dangerLevel)` : set del livello di pericolo nello stato di scasso;
- `uint8_t getDangerLevel()` : restituisce il livello di pericolo nello stato di scasso;
- `void setLockpicking (bool state)` : set dello stato di scasso;
- `bool isLockpicking()` : indica se ci si trova nello stato di scasso;
- `void carousel(uint8_t delay1, uint8_t delay2)` : esegue un carosello temporizzato dei due LED a 12 pin.

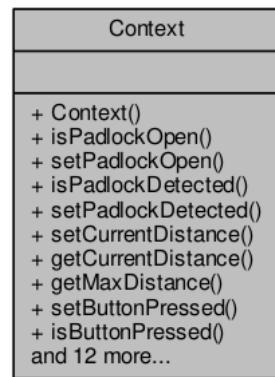


Figura 5.14: Collaboration Diagram - Context

### 5.6.1 Scheduler.cpp, Scheduler.h

#### Metodi

- `public void init(int)` : inizializza il base period dello Scheduler;
- `public virtual bool addTask(Task *)` : aggiunge alla lista dei task ogni singolo task da eseguire;
- `public virtual void schedule()` : pianifica l'esecuzione dei vari task, eseguendoli uno alla volta. Nel caso in cui un task non sia abilitato, viene semplicemente ignorato (non eseguendo il suo tick) e quindi si passa immediatamente al successivo della lista.

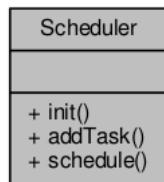


Figura 5.15: Collaboration Diagram - Scheduler

### 5.6.2 Timer.h, Timer.cpp

#### Metodi

- `public void setupPeriod(int)` : disabilita gli *interrupt*, setta i registri del Timer1 (configurando il prescale) e riabilita gli interrupt;
- `public void waitForNextTick()` : permette di eseguire il task.

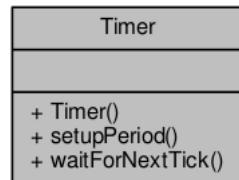


Figura 5.16: Collaboration Diagram - Timer

### 5.6.3 Task.h

#### Metodi

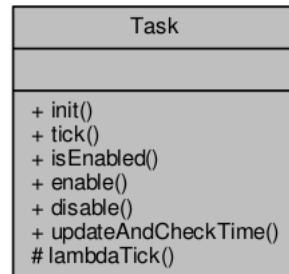


Figura 5.17: Collaboration Diagram - Task

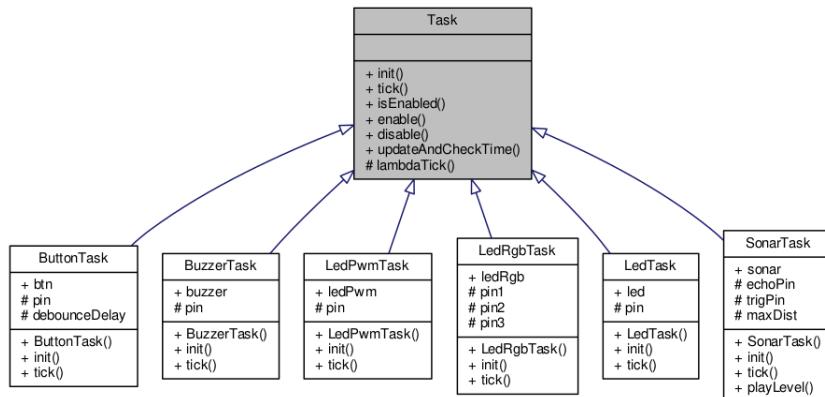


Figura 5.18: Inheritance Diagram - Task

# Capitolo 6

## Multi-Tasking

### 6.1 Task

I task rappresentano le azioni da eseguire concorrentemente nel sistema, pianificate dallo Scheduler.

- **Dal punto di vista del progettista:**
  - sono delle entità attive dotate di un flusso di controllo autonomo che svolgono un determinato compito e che impegnano il microprocessore per un certo periodo;
  - permettono di elaborare dati provenienti dai sensori ed agire sugli attuatori e/o gestire eventi I/O;
  - possono essere sostituiti, alterati, inibiti o modificati in base alle necessità.
- **Dal punto di vista dell'utente** invece:
  - creano l'illusione di avere un unico sistema monolitico;
  - la semplice esistenza e quindi la loro gestione è trasparente all'utente.

Per la descrizione del comportamento di ogni task, si è deciso di utilizzare le lambda expression (6.1.1).

In sintesi i vari task sono istanziati definendo all'interno del *body* del metodo `init(...)` ogni singolo *behaviour*. Per farlo sono utilizzati:

- i metodi propri del task;
- i metodi del Context `??` come mezzo di comunicazione tra i diversi task.

Questo ci ha permesso di avere più gradi di libertà nella definizione del comportamento dei task e quindi del codice da eseguire. In quest'ottica non vi è necessità di creare una classe per ogni singolo comportamento, ma basta istanziare due o più volte la classe e definire diversi body.

### Esempio

Si vogliono controllare due LED utilizzando il task LedTask; in particolare:

- se viene trovato il lucchetto, il led1 viene acceso e il led2 viene spento,
- altrimenti, se il lucchetto non viene trovato, il led1 passa a spento e il led2 diventa acceso.

```
ledT0 = new LedTask(led1, pContext);
ledT0->init(50, [] {
    if (pContext->isPadlockDetected()) {
        ledT0->led->switchOn();
    } else {
        ledT0->led->switchOff();
    }
});
sched.addTask(ledT0);

ledT1 = new LedTask(led2, pContext);
ledT1->init(50, [] {
    if (pContext->isPadlockDetected()) {
        ledT1->led->switchOn();
    } else {
        ledT1->led->switchOff();
    }
});
sched.addTask(ledT1);
```

Dopo la creazione, il task per eseguire deve essere aggiunto alla lista di esecuzione dello Scheduler; per farlo si usa `sched.addTask(<nome del task>)`.

#### 6.1.1 Le espressioni Lambda in Wiring/C++11

Una **Lambda expression** (*lambda closure*) è una funzione anonima definita al momento della chiamata.

Sintassi accettate dal compilatore di Arduino:

```
[ capture-list ] ( params ) { body }
[ capture-list ] { body }
```

### Caratteristiche

- La *capture list* è la lista di variabili che è possibile utilizzare oltre agli argomenti della funzione.

- Se si passa `[&]`, tutte le variabili locali saranno passate per riferimento.
- Se non viene specificato niente, la *lambda function* non ha variabili come argomenti e viene indicata con `[]`;
- Il tipo di ritorno è `void`, a meno che non venga specificato diversamente;
- Il body della funzione che si trova tra parentesi graffe.

## Vantaggi

- è possibile creare una classe "generica" che può avere più istanze a cui assegnare più *behaviour*;
- si evita di dover creare per ogni comportamento una classe specifica;
- codice modulare, (diverse istanze dello stesso oggetto possono avere diversi body);
- definizione del comportamento direttamente dal file `*.ino`.

## Contro

Utilizzare le lambda expression ci ha portati ad un *trade-off*, cioè siamo stati costretti a rinunciare parzialmente all'*information hiding* della programmazione OO al fine di ottenere i vantaggi sopra citati. Questa rinuncia è legata al fatto che non è stato possibile passare oggetti come chiusura a causa di alcuni vincoli del linguaggio usato da Arduino.

L'unico modo per utilizzare i metodi richiamati all'interno di queste particolari funzioni è stato dichiarare tali metodi come `public`.

### 6.1.2 Generalizzazione del funzionamento di un task

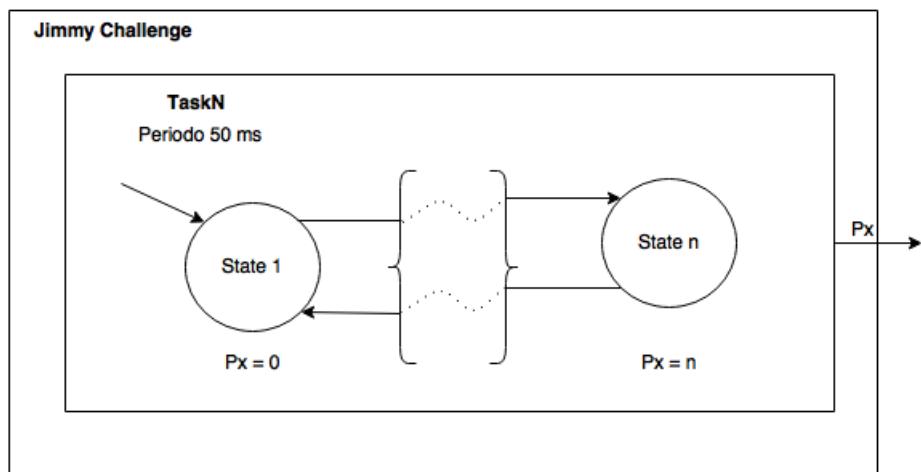


Figura 6.1: Generalizzazione dei task

## 6.2 Scheduler

Lo Scheduler coordina e fa cooperare i task del sistema.

Al suo interno è presente il vettore `taskList[...]` in cui sono aggiunti i task da eseguire grazie al metodo `bool addTask(Task* task)`.

Nel caso in cui nello Scheduler non ci fossero task o fossero stati tutti bloccati, il sistema rimarrebbe apparentemente fermo (anche se in realtà il microcontrollore continuerebbe ad eseguire il suo `void loop()`).

Lo Scheduler è **cooperativo** in quanto una volta selezionato un task, questo viene eseguito fino al suo completamento (*run-to-completion*). Ad ogni tick della FSM vengono eseguite atomicamente le azioni associate al nuovo stato.

Lo scheduling è a **priorità statica**, in quanto ad ogni task viene assegnata una priorità che non cambia durante l'esecuzione. La si può considerare come definita implicitamente dall'ordine di inserimento dei task nella *task list*.

## 6.3 Context

La classe Context contiene tutti le variabili di stato del programma. I metodi al suo interno sono accessibili da tutti i task in modo da far evolvere il sistema; di seguito la lista di tutti i metodi e del loro significato.

### 6.4 Elenco dei Task sviluppati:

1. SonarTask
2. ButtonTask
3. BuzzerTask
4. LedTask
5. LedPwmTask
6. LedRgbTask

#### 6.4.1 SonarTask

È il task più importante del progetto in quanto:

- permette l'interazione utente-sistema sfruttando il sensore ad ultrasuoni;
- aggiorna la `currentDistance` del Context (??);
- in modo indiretto può variare l'evoluzione del sistema

Nella classe SonarTask viene definito il metodo `private playLevel()` in cui:

- si legge la `currentDistance` dal sonar;
- si setta tale `currentDistance` nel Context;
- si gestiscono gli `status` da inviare alla seriale

Dal Context:

- si riceve il **numero segreto**: la distanza del pistoncino a partire dal punto 0, ovvero il sonar);
- si riceve il **delta**: intervallo di errore in cui la posizione del pistoncino è considerata corretta;
- si riceve il **livello attuale** a cui si sta giocando;
- si controlla se il lucchetto è aperto con `pContext->isPadLockOpen()`

Sul Context:

- se il lucchetto è stato aperto si crea un nuovo livello con `pContext->setNewLevel()`;
- se il padLock non è stato completamente sbloccato viene settato come chiuso con `pContext->setPadlockOpen(false)`;
- se il padLock non è stato completamente sbloccato viene resettato lo stato di scasso con `pContext->setLockpicking(false)`;
- in base al tempo passato nello scassinare il lucchetto (e quindi allo `status` in cui ci si trova), viene settato `pContext->setDangerLevel(...)` con un numero da 0 a 4, (che nel LedRgbTask indica il colore da visualizzare sul LED RGB).

Per la rilevazione della distanza della mano dal sensore è stata utilizzata la libreria **NewPing A.2.1**.

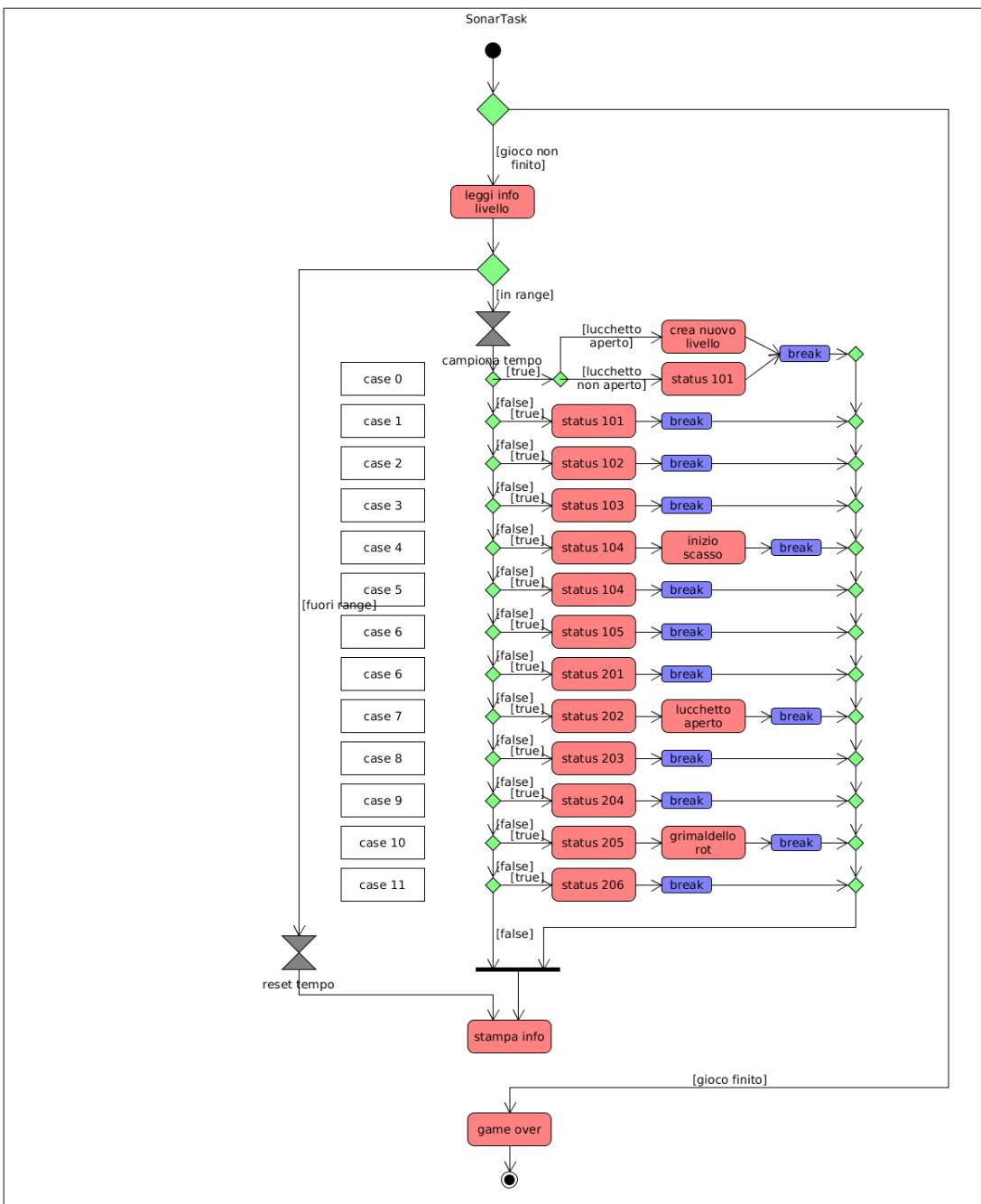


Figura 6.2: Diagramma di attività - `playLevel()`

#### 6.4.2 ButtonTask

Questo task controlla se è avvenuta la pressione del *button*. Finché il gioco non è finito è possibile stampare su terminale un messaggio che indica la posizione del lucchetto.

Se il gioco è finito, ma il bottone viene comunque premuto, si avvia una divertente *easter egg* legata al BuzzerTask e ad un famoso gioco degli anni '80...

#### 6.4.3 BuzzerTask

Questo task controlla i suoni che il *buzzer* deve emettere attraverso il metodo `buzzerT0->buzzer->playSound(<num>)`

- Se il gioco non è finito e il lucchetto non è ancora stato trovato, viene emesso il suono 0;
- Se il gioco non è finito e il lucchetto è stato trovato, viene emesso il suono 1;
- Se il gioco è finito e il pulsante premuto, viene emesso il suono 2 (*easter egg*).

#### 6.4.4 LedTask

Questo task controlla l'accensione e lo spegnimento del LED verde.

- Se il gioco non è finito e il lucchetto è stato trovato, il LED viene acceso;
- Se il gioco non è finito e il lucchetto non è ancora stato trovato, il LED viene spento.

#### 6.4.5 LedPwmTask

Questo task gestisce il LED verde utilizzando il pin PWM grazie ai seguenti metodi:

- `ledPwmT0->ledPwm->setIntensity(<num>)` per gestire l'intensità luminosa (anche temporizzata) del LED;
- `ledPwmT0->ledPwm->switchOff()` per spegnere immediatamente il LED.

Analizzando questo task nel dettaglio:

- Se il lucchetto non è stato aperto e non è stato trovato:

- si aumenta l'intensità del LED usando un ciclo **for** inizializzato a 64 (in modo da non partire dal LED completamente spento) per arrivare a 255 (valore massimo consentito).
  - al termine del ciclo il LED viene spento

L'obiettivo di questo frammento di codice è far capire al giocatore che il sistema è in funzione facendo illuminare e subito spegnere il LED, come se questo emettesse tanti flash luminosi.

- Se il gioco è finito:
  - si aumenta l'intensità del LED usando un ciclo **for** inizializzato a 10 (riducendo il tempo necessario per arrivare al massimo).
  - all'interno di questo ciclo è stato inserito un **delay** di 3 ms per rendere più visibile il dimmeraggio
  - al termine del ciclo for, il LED viene spento.

#### 6.4.6 LedRgbTask

Questo task gestisce il funzionamento del LED RGB.

In fase di creazione prende in input i pin PWM dell'Arduino a cui è collegato fisicamente il LED RGB (costanti **LED\_RGB\_R**, **LED\_RGB\_G**, **LED\_RGB\_B**).

Funzionamento:

- Finché il gioco non è finito:
  - Se si è in fase di scasso, si imposta un colore nel costrutto **switch-case** in base al numero intero di **pContext->getDangerLevel()**
    - Nel dettaglio:
      - \* 0 = **blu scuro** (inizio stato di scasso);
      - \* 1 = **verde** (lucchetto sbloccato);
      - \* 2 = **giallo** (attenzione, rischio di rompere il lucchetto);
      - \* 3 = **rosa** (attenzione, elevato pericolo di rottura);
      - \* 4 = **rosso** (rottura del lucchetto).
    - Altrimenti si imposta un colore ***light blue*** costante.

## 6.5 Networking e Sicurezza

Il mondo embedded e IoT sta avendo un forte sviluppo in questi anni grazie alla disponibilità di dispositivi dal rapporto prestazioni/prezzo vantaggioso ha creato un grande bacino di utenza di esperti o semplici neofiti interessati al mondo dell'elettronica/informatica dedicata. Un effetto positivo di questa evoluzione è lo sviluppo di librerie open-source e di sistemi software orientati all'IoT.

Questo trend ha creato un mercato sempre più vasto di produttori di dispositivi più performanti, economici e dalle dimensioni ridotte (perfetti per realizzazioni embedded). A causa di questo sviluppo, per certi versi incontrollato, "tutto" è virtualmente collegato/collegabile *online* e sembra essere passata in secondo piano la progettazione di hardware e software che tiene conto della **sicurezza dei sistemi**. I primi attacchi mirati a queste tecnologie hanno avuto facilità di esecuzione e una rapidissima diffusione<sup>1</sup>.

In questo scenario, dalla parte dei produttori hardware si registra una mancanza o addirittura una resistenza per quanto riguarda la correzione di falle o il rilascio di aggiornamenti firmware. Questo potrebbe essere dovuto all'utilizzo di codice proprietario o la totale mancanza di supporto per il dispositivo che si utilizza. In alcuni casi si potrebbe parlare di **obsolescenza programmata**.

Anche da parte degli utilizzatori finali dei sistemi non si evince una particolare attenzione ai problemi relativi alla sicurezza; probabilmente a causa delle scarse conoscenze del sistema in uso (perché complesso o non studiato) e delle tecnologie usate. È ormai tristemente noto che lo sviluppo di questi **dispositivi perennemente connessi**, non prende quasi mai in considerazione le problematiche relative alla sicurezza derivata dalla connessione a sistemi più *fragili* o legate all'interazione con altri device (perdita di privacy, prestazioni o utilizzo improprio di risorse di rete).

Jimmy Challenge è stato sviluppato garantendo la sicurezza del sistema e dei giocatori.

In questo capitolo verranno presentate alcune delle soluzioni adottate e delle tecnologie utilizzate per l'implementazione dell'applicativo lato server e dei collegamenti di reti. Per scelta progettuale ed etica, ove possibile, si è preferito utilizzare esclusivamente software *open-source*.

### 6.5.1 Sistema Operativo

Le mansioni di server sono fisicamente compiute da un **Odroid C2**) il quale utilizza come OS una versione ARM a 64-bit della distribuzione Linux **Arch Linux ARM**.

---

<sup>1</sup>Breve lista di attacchi a sistemi IoT nel 2015 <http://tinyurl.com/honlko2>.

## Arch Linux ARM

- è *bleeding edge* per quanto riguarda l’upstream degli aggiornamenti (sempre aggiornata all’ultima versione dei software disponibile);
- generalmente è considerata sicura;
- fornisce una libertà maggiore per la configurazione del sistema;
- una forte e grande community.

## Software installato orientato alla sicurezza

- **SSH** con autenticazione chiave pubblica-privata RSA;
- firewall **UFW**, per rendere disponibili all’esterno soltanto alcuni servizi;
  - 80: Web Server;
  - 22: SSH;
  - 443: HTTPS.
- **Cerbot** per i certificati SSL/TLS;
- **Netdata** per una visualizzazione delle risorse del sistema da remoto.

Oltre all’installazione e alla configurazione dei software sono stati configurati diversi parametri per l’*hardening* del kernel tramite *sysctl* ed un sistema di *logging* per la registrazione degli accessi.

### 6.5.2 NGINX e HTTPS

**NGINX** è un web server orientato alle performance, al ridotto consumo di risorse e alla facilità di configurazione. In un sistema come l’Odroid con potenza computazionale ridotta (rispetto ai classici server) l’utilizzo di applicativi con ridotto consumo di risorse garantisce una maggiore reattività del sistema e un migliore risultato agli occhi dell’utente.

La configurazione di NGINX prende in considerazione tutti i maggiori problemi legati all’esposizione di un sito e di un web server in rete: buffer overflow, DDoS, bruteforcing, sniffing, etc. e cerca di mitigare o prevenire ogni tipo di attacco senza generare intralcio o rallentamenti all’utilizzo standard del web server.

Grazie a *Cerbot* è possibile ottenere certificati SSL/TLS gratuitamente e garantire un canale di comunicazione protetto tra client e server.

La configurazione di NGINX è visibile [qui](#).

L’intera configurazione di NGINX cerca di essere compatibile con gli ultimi protocolli e limitare quelli obsoleti e poco sicuri, rischiando però di ridurre

la compatibilità con i browser minori. Per garantire ulteriori performance NGINX è abilitato ad instaurare connessioni tramite il recente protocollo HTTP/2 con i client compatibili.

### 6.5.3 MySQL, Redis

MySQL e Redis:

- non sono esposti verso l'esterno;
- sono protetti da password;
- sono configurati per rispettare il POLP (*principle of least privilege*).

#### MySQL

Con MySQL si gestisce il database per lo storage delle credenziali degli utenti e la lista degli utenti connessi online.

```
CREATE TABLE IF NOT EXISTS users (
    id int(3) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    created DATETIME DEFAULT CURRENT_TIMESTAMP,
    username varchar(15) COLLATE utf8_general_ci NOT NULL,
    password varchar(255) COLLATE utf8_general_ci NOT NULL,
    email varchar(30) COLLATE utf8_general_ci NOT NULL,
    keepalive DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
        CURRENT_TIMESTAMP,
    logged boolean NOT NULL DEFAULT 0,
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_general_ci;

ALTER TABLE users ADD INDEX logged_index(logged);

SET GLOBAL event_scheduler = ON;

DELIMITER $$
CREATE EVENT IF NOT EXISTS `keepalive_logged`
ON SCHEDULE EVERY 1 MINUTE STARTS '2016-06-01 00:00:00'
ON COMPLETION PRESERVE
DO BEGIN
    UPDATE users SET logged = 0
    WHERE TIMESTAMPDIFF(SECOND, keepalive, now()) >= 20;
END;$$
DELIMITER ;
```

Un event MySQL controlla periodicamente se un utente è ancora collegato o meno.

#### Redis

**Redis** è un applicativo per lo store di strutture dati chiave-valore in memoria. Il principale vantaggio di Redis è la sua velocità in lettura e scrittura e la facilità di utilizzo.

#### 6.5.4 Sito Web, back-end e API RESTful

Il sito web (sia front-end che back-end) è stato sviluppato tenendo a mente le linee guida del W3C e del OWASP.

Al fine di rendere l'applicativo il più possibile portatile, sono state implementate delle API RESTful in PHP7; sia l'applicativo Python lato client, sia il sito web (tramite AJAX) utilizzano queste API.

Una volta che un utente si è loggato, gli viene assegnato un *token* JWS (JSON Web Signature) che deve essere validato ad ogni richiesta che farà da quel momento in poi. Questo token permette di verificare l'identità dell'utente ad ogni richiesta ed evitare usi improrii della API da utenti malintenzionati. Le API possono essere invocate solo tramite richieste POST e comunicano tramite messaggi JSON appositamente formattati.

In seguito al login effettuato, il sito web si presenta all'utente come un'unica pagina in cui è possibile inviare messaggi a tutti gli altri utenti collegati tramite una chat globale e selezionare un giocatore da sfidare.

Una volta iniziata una partita, nella view principale della pagina appariranno i dati relativi allo stato della partita dell'avversario. Ogni aggiornamento dello stato verso i client è effettuato tramite Server-Sent Events (SSE).

Grazie a SSE, per mezzo del pattern **publish-subscribe**:

- si riducono i consumi di risorse computazionali
- si limita l'utilizzo della banda
- si eliminano i problemi di gestione di altre tecnologie come WebSocket o WebRTC.

Per farlo si instaura un canale mono direzionale verso il client in cui è possibile istruire il browser per controllare periodicamente l'aggiornamento dello stato dei dati o forzare l'invio di dati aggiornanti da parte del server.

Per il back-end sono state utilizzate diverse librerie per fornire alcune funzionalità come JWS e SSE scelte appositamente per la loro compatibilità con gli ultimi standard e versioni di PHP al fine di garantire maggiori performance e sicurezza:

- [jose](#): per i token JWS;
- [libSSE-php](#): per SSE;
- [http-foundation](#) e [predis](#): come dipendenze di "libSSE-php".

Le interazioni tramite il database MySQL avvengono per mezzo di una connessione locale, utilizzando le varie tecniche di mitigazione per SQLi di primo e secondo livello fornite dal PHP.

Per proteggere la privacy degli utenti, le password sono salvate sul database come hash (usando bcrypt).

### 6.5.5 Python

Come anticipato nella relazione, per poter inviare al server i dati letti dalla seriale di Arduino è stato sviluppato uno script Python, in particolare **Python 3.5**.

```
└ ./.jimmy.py
Ctrl-C to close the program...
Username: fede
Password:
```

Figura 6.3: Login da terminale - lato client

Tenendo a mente la visione dei sistemi embedded, dalle capacità computazionali ridotte rispetto ad un laptop o ad un PC desktop, si è deciso di utilizzare:

- **pyserial** : per la lettura dei dati dalla seriale;
- **ujson**: per i messaggi JSON;
- **asyncio** : per eseguire le routine in maniera asincrona;
- **aiohttp**: come HTTP client per asyncio.

Per poter utilizzare lo script è necessario autenticarsi e ricevere il token JWS al fine di effettuare le richieste HTTP.

#### Pyserial

La libreria **pyserial** consente la connessione all'Arduino tramite porta seriale e la lettura dei messaggi in maniera continua grazie all'utilizzo di un thread che agisce da produttore di messaggi per l'intero script.

L'elaborazione di questi messaggi viene effettuata da un routine che dopo aver *parsato* il messaggio lo instrada al corretto destinatario. Se si deve inviare un messaggio al server, viene composto il messaggio in JSON e viene inviato tramite una richiesta HTTP POST al server.

#### Asyncio

A parte il thread creato per la lettura dei messaggi da Arduino, le altre *routine*, o meglio, **coroutine** sono eseguite all'interno dell'event loop di asyncio.

Asyncio fornisce una infrastruttura single-threaded per la scrittura di codice concorrente utilizzando coroutine, in particolare è consigliato per **programmi concorrenti IO-bound**.

## Aiohttp

Aiohttp fornisce un supporto per richieste HTTP asincrone tramite l'infrastruttura messa a disposizione da asyncio.

Questo tipo di approccio:

- permette di ridurre i consumi di risorse;
- aumentare le performance dell'applicazione.

Questo risultato si ottiene perché la maggior parte dell'esecuzione dello script è basata sull'invio di dati al server.

L'obiettivo di rendere completamente asincrono il codice ed eliminare il thread della libreria pyserial teoricamente sarebbe possibile utilizzando delle API per asyncio, ma al momento dello sviluppo del progetto queste soluzioni sono incomplete e non affidabili, quindi non sono state scartate.

# Capitolo 7

## Testing

### 7.1 Web Application

#### 7.1.1 Fase di gioco

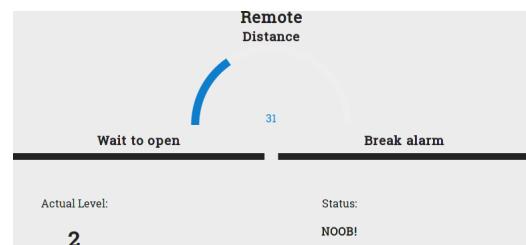


Figura 7.1: Pistoncino non trovato

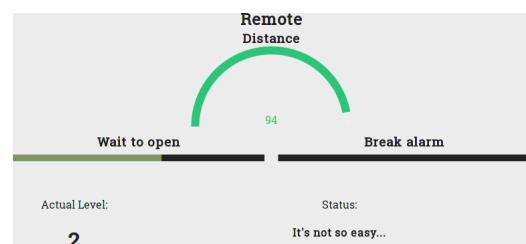


Figura 7.2: Pistoncino trovato, inizio stato di scasso

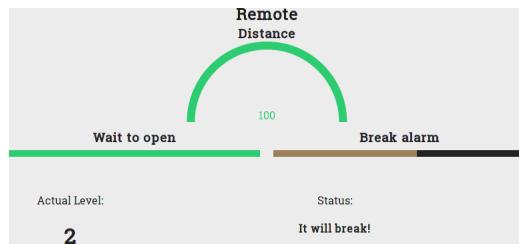


Figura 7.3: Stato di scasso, rischio rottura

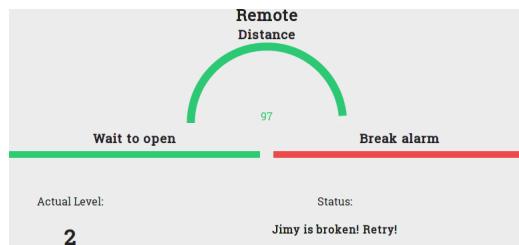


Figura 7.4: Grimaldello rotto

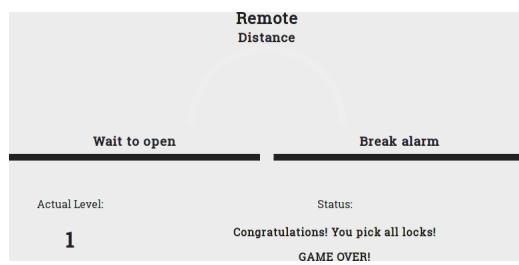


Figura 7.5: Livello superato, lucchetto aperto

### 7.1.2 Fase di login



Figura 7.6: Login al proprio account

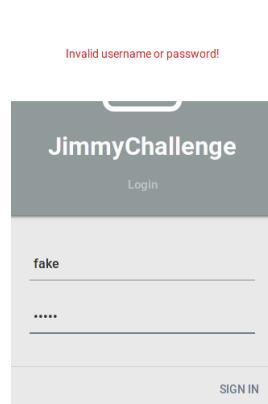


Figura 7.7: Errore di login

## 7.2 Partita single player

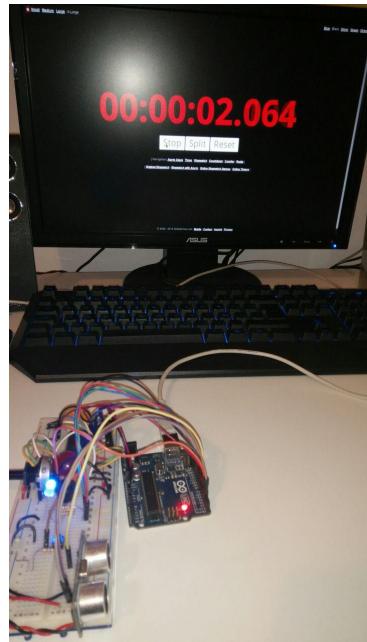


Figura 7.8: Testing cronometrato

## Appendice A

# Elenco dei componenti utilizzati

### A.1 Componenti hardware

- Componenti lato client:
  - Arduino UNO;
  - resistori;
  - sensore di prossimità ad ultrasuoni HC-SR04;
  - buzzer;
  - potenziometro;
  - multiplexer CD4067B;
  - button;
  - LED verde;
  - LED RGB;
  - LED rosso a 12 pin;
  - LED giallo a 12 pin;
  - breadboard;
  - cavi di collegamento;
- Componenti lato server:
  - Odroid C2;
  - monitor LCD;
  - potenziometro;
  - breadboard;
  - cavi di collegamento.

## A.2 Componenti software

### A.2.1 Librerie Arduino

- [NewPing](#);
- [ArduinoJson](#).

#### NewPing

##### Caratteristiche

- Funziona con diversi modelli di sensori ad ultrasuoni: SR04, SRF05, SRF06, DYP-ME007 e Parallax Ping<sup>TM</sup>;
- Non ha un **lag** di un secondo se non si riceve un ping di eco
- Ping coerente e affidabile fino a 30 volte al secondo.
- Timer interrupt method per sketch event-driven
- Metodo di filtro digitale Built-in `ping_median()` per facilitare la correzione degli errori.
- Utilizzo dei registri delle porte durante l'accesso ai pin per avere un'escuzione più veloce e dimensioni del codice ridotte.
- Consente l'impostazione di una massima distanza di lettura del ping "in chiaro".
- Facilita l'utilizzo di più sensori.
- Calcolo distanza preciso, in centimetri, pollici e uS.
- Non fa uso di `pulseIn`, in quanto lento e con alcuni modelli di sensore a ultrasuoni restituisce risultati errati.
- Attualmente in sviluppo, con caratteristiche che vengono aggiunte e bug/issues affrontati.

#### ArduinoJson

##### Caratteristiche

- codifica/decodifica JSON;
- API di facile utilizzo;
- allocazione di memoria fissa (senza malloc);
- nessuna *data duplication* (evitando la copia);

- portatile (scritto in C++98);
- non ha dipendenze esterne;
- libreria leggera;
- MIT License.

#### A.2.2 IDE utilizzati

- [Atom](#) con [PlatformIO](#);
- [Arduino IDE](#) (per alcuni test sulla comunicazione seriale).

#### A.2.3 Linguaggi di sviluppo

- Wiring/C++;
- Python;
- JSON;
- HTML;
- PHP;
- Javascript/AJAX.

#### A.2.4 Altre

- [Fritzing](#) per riportare gli schemi di collegamento.

# Appendice B

## Schema di collegamento

### B.1 Sketch Arduino - lato client

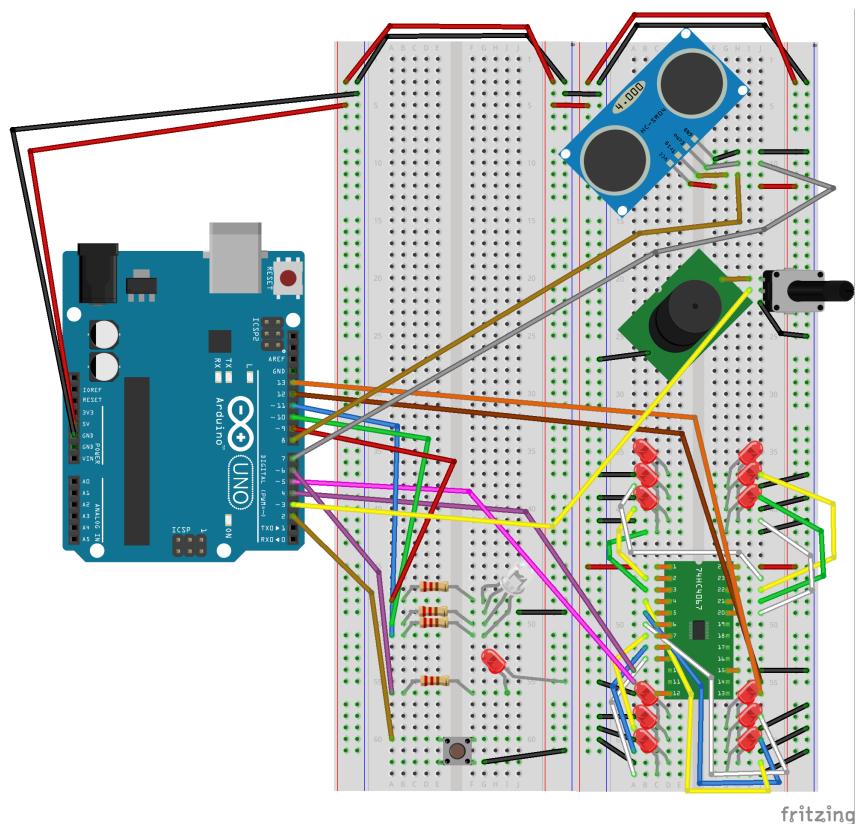


Figura B.1: Sketch Arduino - fritzing

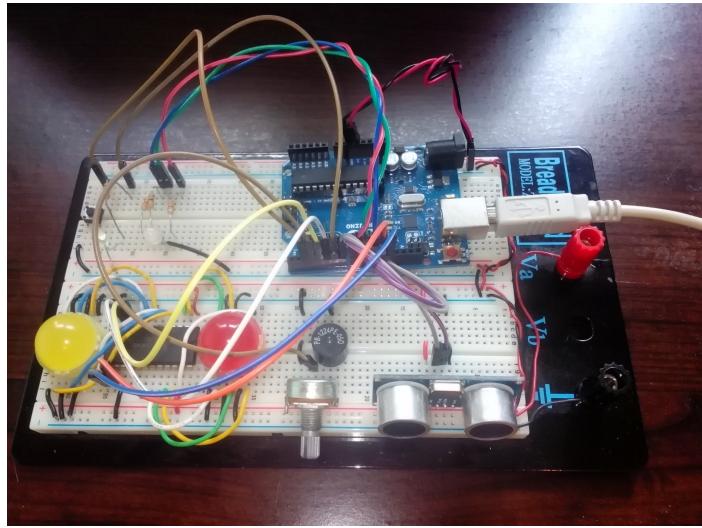


Figura B.2: Sketch Arduino - realizzazione reale

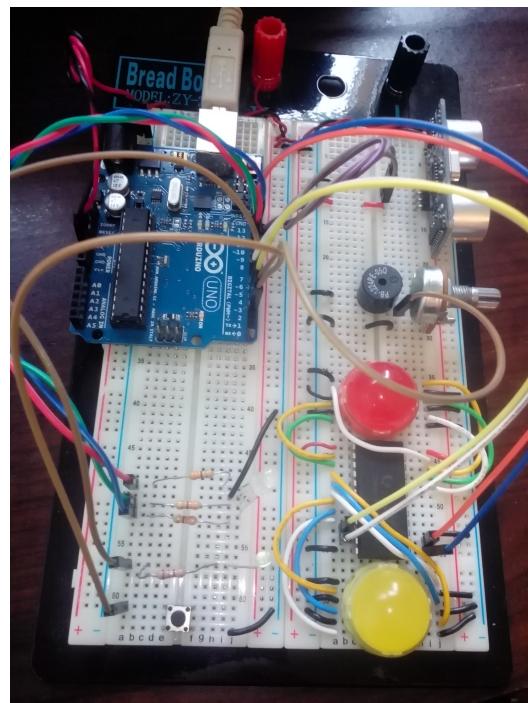


Figura B.3: Sketch Arduino - realizzazione reale

## B.2 Sketch Odroid - lato server

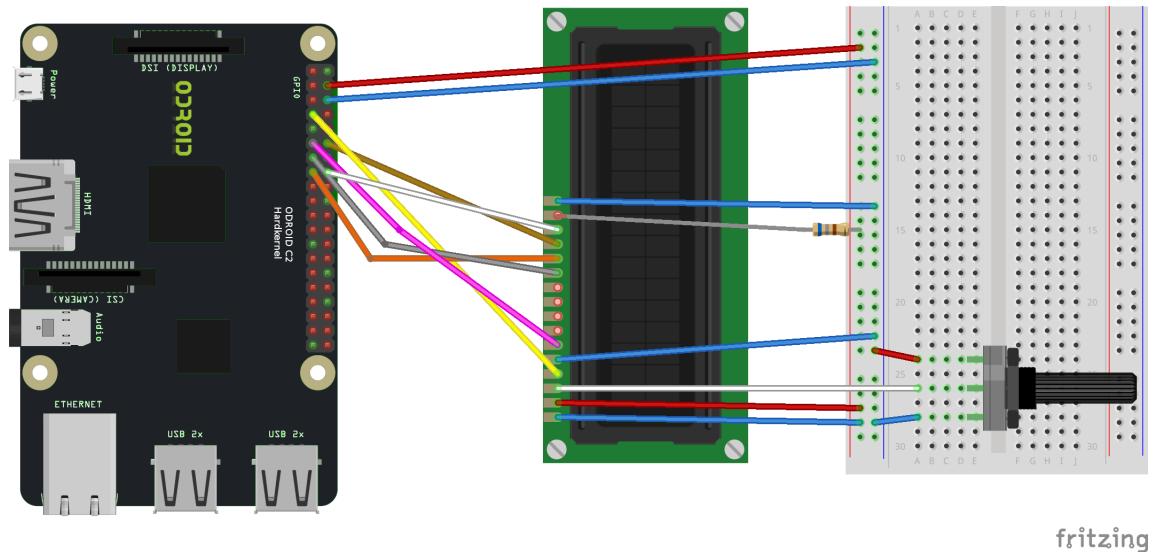


Figura B.4: Sketch Arduino - fritzing

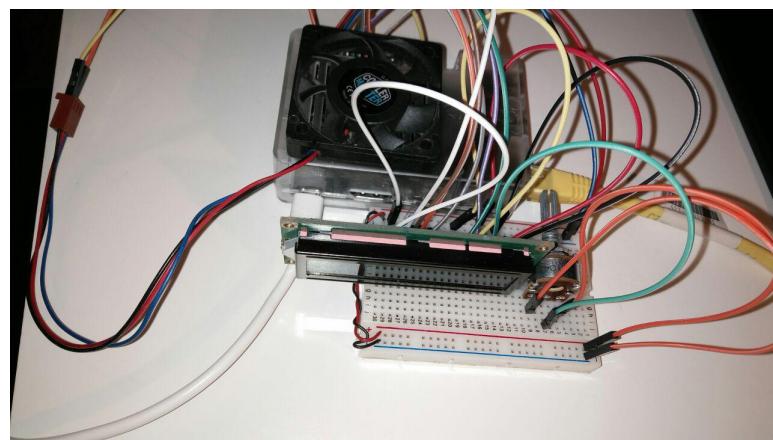


Figura B.5: Sketch Odroid - realizzazione reale

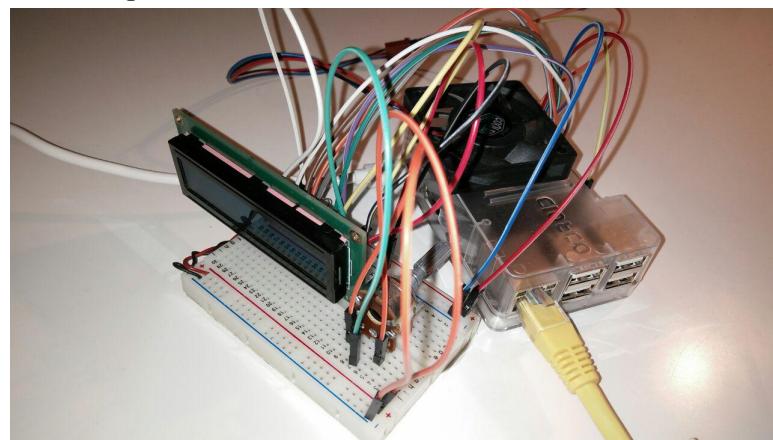


Figura B.6: Sketch Odroid - realizzazione reale