

# Programmazione di Sistemi Embedded

## Jimmy Challenge Arduino Game

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

13 giugno 2016

## **Sommario**

In questa relazione si descriverà il gioco Jimmy Challenge.

Questo gioco è stato realizzato utilizzando molti componenti hardware/-software; sempre con l'obiettivo di sfruttare ed ottimizzare quando più possibile la board Arduino in un'ottica IoT.

**Capitolo 1 .**

# Capitolo 1

## Introduzione

**Jimmy Challenge**<sup>1</sup> è un gioco interattivo in cui un "ladro" tenta di *forzare* un lucchetto utilizzando un *grimaldello*.

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

Una volta che il settaggio 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.

## 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 due 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.

bbbbbb

#### 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.

### 2.2 Modeling

Il sistema si suddivide in più parti:

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

#### 2.2.1 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 distinguendoli in **sensori** ed **attuatori** 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.

### 2.2.2 Modellazione della parte logica - lato client

Per realizzare il comportamento dinamico del progettare, 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 migliore la comprensibilità del comportamento
- debugging semplificato

- supporto alla modificabilità, all'estensione ed al riuso del codice

Si è fatto uso della modellazione ad oggetti, modellando ogni task come una classe separata che estende da una classe base task. In ogni task viene fatto l'***inject*** del **comportamento** da avere ad ogni tick.

Il multi-tasking realizzato è cooperativo, tipo **round-robin** in quanto c'è parallelismo e le funzioni *tick()* sono chiamate sequenzialmente all'interno del loop dello Scheduler.

- 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 fare l'*inject* di codice da eseguire nei vari task.

Lo Scheduler utilizzato permette la cooperazione tra i task; per farlo tiene traccia della lista dei task che sono in esecuzione.

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 macchina 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 definita implicitamente dall'ordine di inserimento dei task nell'array.

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 tutti formattati in **JSON**.

### 2.2.3 Modellazione della parte fisica - lato server

TEST git

### 2.2.4 Modellazione della parte logica - lato server

## 2.3 Design

Il sistema che si suddivide in più aree:

1. I/O locale attraverso Arduino UNO
2. feedback remoto su browser
3. input per invio dei dati seriali al server attraverso async task Python
4. connessione USB da Arduino UNO verso il PC

5. connessione del PC ad un server
6. sito internet come GUI remota
7. gestione del server remoto

## 2.4 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.
- 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()**;

---

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

## Capitolo 3

# Giocare a Jimmy Challenge

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

L'obiettivo del giocatore è trovare e quindi scassinare il lucchetto nel minor tempo possibile. Questo gioco è giocabile **online** (uno contro uno) che **offline**.

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

Per trovare la posizione attuale del lucchetto 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 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 il lucchetto non è stato trovato.
- Quando **non** si è trovato il lucchetto:
  - 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 del lucchetto:
  - 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.

Dal punto di vista del giocatore il lucchetto è un'area nello spazio posta davanti al sensore (in linea orizzontale).

Per forzare il lucchetto è sufficiente utilizzare una mano posta davanti al sensore ad ultrasuoni per un tempo delimitato, 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 del lucchetto troppo presto, per esempio non si è ancora nello stato di scasso
- Se si rimane troppo tempo nella fase di scasso (il lucchetto è stato "rotto").

- Si può superare il livello:

- Se la mano resta fissa nella posizione in cui si trova il lucchetto, rispettando il tempo nello **stato di scasso** e poi la si agita sempre nell'area del lucchetto ("aprendolo").

### 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 il LED RGB.

- Ogni colore ha un significato:

- blu scuro: si sta scassinando il lucchetto
- verde: il lucchetto è scassinato.

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

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

## Capitolo 4

# Multi-Tasking

### 4.1 Task

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

- **Dal punto di vista del progettista:**
  - sono dei moduli software che impegnano il microprocessore per un certo periodo;
  - permettono di eseguire calcoli e/o azioni di 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 (4.1.1).

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 4.3 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 lo 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 ogni creazione, ogni task per essere eseguito deve essere aggiunto alla lista di esecuzione dello Scheduler; per farlo si usa `sched.addTask(<nome del task>)`.

### 4.1.1 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 definisce `[&]`, 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 viene specificato diversamente;
- Il body della funzione che si trova tra parentesi graffe rappresenta le azioni che devono essere eseguite.

### Vantaggi

- è possibile creare una classe "generica" che può avere più istante 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

Per utilizzare le lambda expression siamo stati costretti a rinunciare parzialmente all'*information hiding* della programmazione OO. Questo vincolo è legato al fatto che non è stato possibile passare oggetti come parametro di chiusura.

L'unico modo per passare gli oggetti e quindi i metodi all'interno di queste particolari funzioni è stato mettere tutto **public**.

#### 4.1.2 Generalizzazione del funzionamento di un task

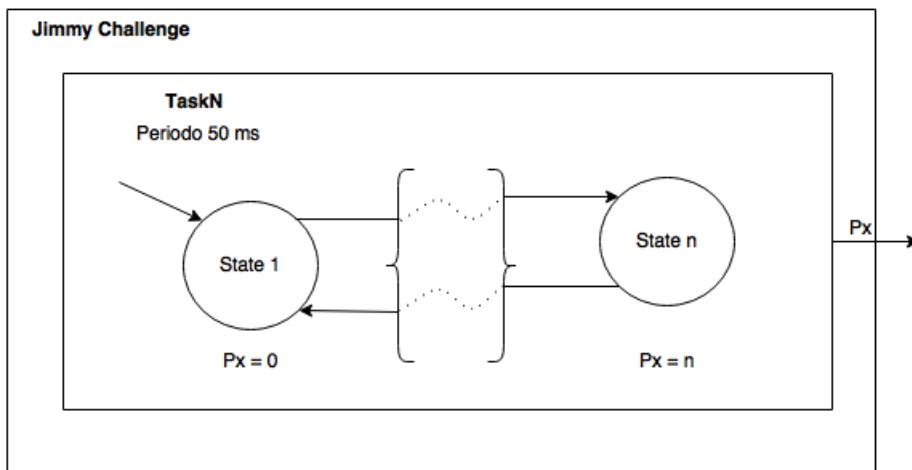


Figura 4.1: Generalizzazione dei task

### **Elenco dei Task sviluppati:**

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

#### **4.1.3 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 (4.3);
- in modo indiretto può variare l'evoluzione del sistema

Nella classe SonarTask viene definito il metodo `privare 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** (il lucchetto, cioè la distanza dal sensore)
- si riceve il **delta** (intervallo superiore/inferiore a partire dal lucchetto)
- 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 pContex->setDangerLevel(...) con un numero da 0 a 4, (che nel LedRgbTask indica il colore da visualizzare sul LED RGB).

```

void SonarTask::playLevel() {
    currentDistance = sonar->readDistance();
    pContext ->setCurrentDistance(currentDistance);
    int secretDistance = pContext ->getSecret();
    int currentLevel = pContext ->getLevel();
    uint8_t delta = pContext ->getDelta();
    int status = 0;
    int feedbackDistance = 0;

    feedbackDistance = abs(currentDistance - secretDistance
    );
    if (feedbackDistance == 0)
        feedbackDistance = 0;
    else if (currentDistance == 0)
        feedbackDistance = 100;

    if (currentDistance <= (secretDistance + delta)
    && currentDistance >= (secretDistance - delta)) {
        pContext ->setPadlockDetected(true);
        timeFound = (millis() / 1000) - timeOut; // 
        inizializzazione a zero
        switch (timeFound) {
            case 0:
                if (pContext ->isPadlockOpen()) {
                    msgService.sendMsg("Lucchetto
                    livello " + String(
                    currentLevel) + " APERTO",
                    F("all"));
                    pContext ->setNewLevel();
                    if (!pContext ->isGameOver()) {
                        status = 300 +
                        currentLevel + 1;
                        pContext ->
                        setPadlockOpen(
                        false);
                        pContext ->
                        setLockpicking(
                        false);
                    }
                } else
                    status = 101;
                break;
            case 1:
                status = 101;
                break;
            case 2:
                status = 102;
                break;
        }
    }
}

```

```

        case 3:
            status = 103;
            break;
        case 4:
            status = 104;
            pContext->setDangerLevel(0);
            pContext->setLockpicking(true);
            break;
        case 5:
            status = 104;
            pContext->setDangerLevel(0);
            break;
        case 6:
            status = 105;
            pContext->setDangerLevel(1);
            pContext->setPadlockOpen(true);
            break;
        case 7:
            status = 201;
            pContext->setDangerLevel(1);
            break;
        case 8:
            status = 202;
            pContext->setDangerLevel(2);
            break;
        case 9:
            status = 203;
            pContext->setDangerLevel(3);
            break;
        case 10:
            status = 204;
            pContext->setDangerLevel(4);
            pContext->setPadlockOpen(false);
            break;
        case 11:
            status = 205;
            pContext->setDangerLevel(4);
            break;
    }
} else {
    timeOut = millis()/1000;
    pContext->setPadlockDetected(false);
    pContext->setLockpicking(false);
}
msgService.sendInfo(currentDistance, status, currentLevel, F(
    "remote"));
}

```

Per la rilevazione della distanza della mano dal sensore è stata utilizzata la libreria **NewPing** 4.1.3.

## Libreria 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'esecuzione 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.

#### 4.1.4 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...

#### 4.1.5 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*).

#### 4.1.6 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.

#### 4.1.7 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.
- 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).

#### 4.1.8 LedRgbTask

### 4.2 Scheduler

### 4.3 Context

## Appendice A

# Elenco dei componenti utilizzati

### A.1 Componenti hardware

- Componenti lato client:
  - Arduino UNO
  - Breadboard
  - Cavi di collegamento
  - Resistori
  - Sensore di prossimità ad ultrasuoni HC-SR04
  - Buzzer
  - Potenziometro
  - Multiplexer CD4067B
  - Button
  - LED verde
  - LED RGB
  - LED rosso 6 pin
  - LED giallo 6 pin
- Componenti lato server:
  - Odroid C2
  - Monitor LCD

## A.2 Componenti software

Librerie Arduino:

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

IDE utilizzati:

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

Linguaggi di sviluppo:

- Wiring/C++, per lo sketch Arduino lato client
- Python
- JSON per la codifica della comunicazione USB e remota
- HTML ...

Altro:

- [Fritzing](#) per lo schema di collegamento

# Appendice B

## Schema di collegamento

### B.1 Sketch Arduino - lato client

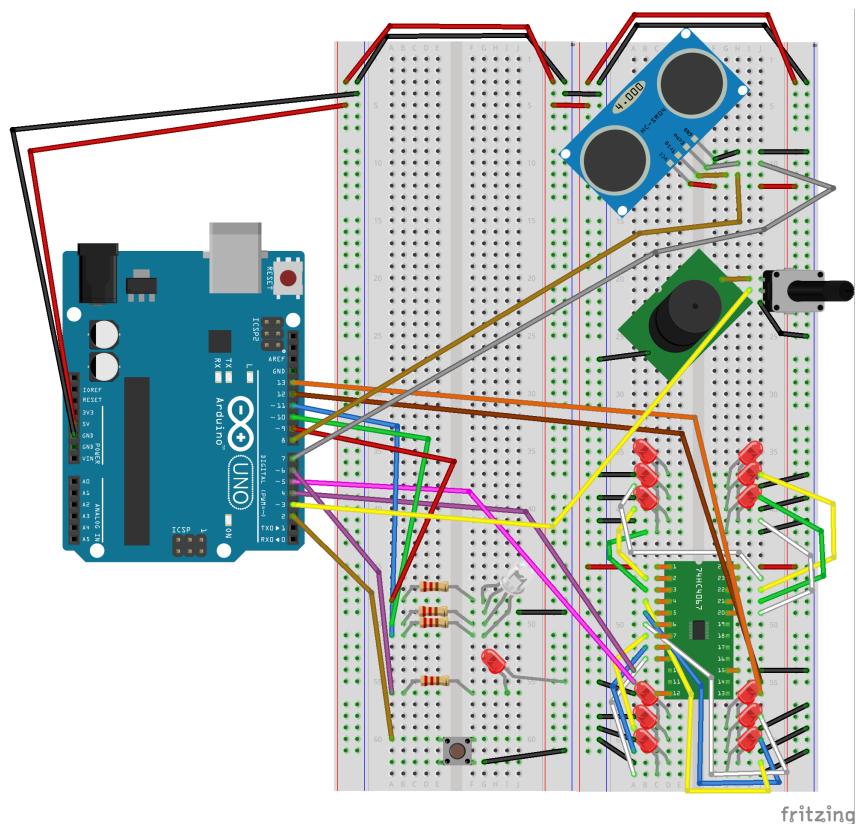


Figura B.1: Sketch Arduino - fritzing

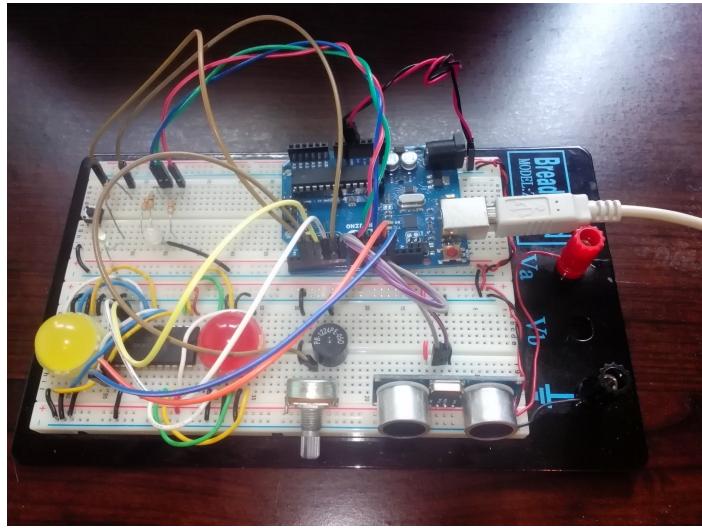


Figura B.2: Sketch Arduino - realizzazione reale

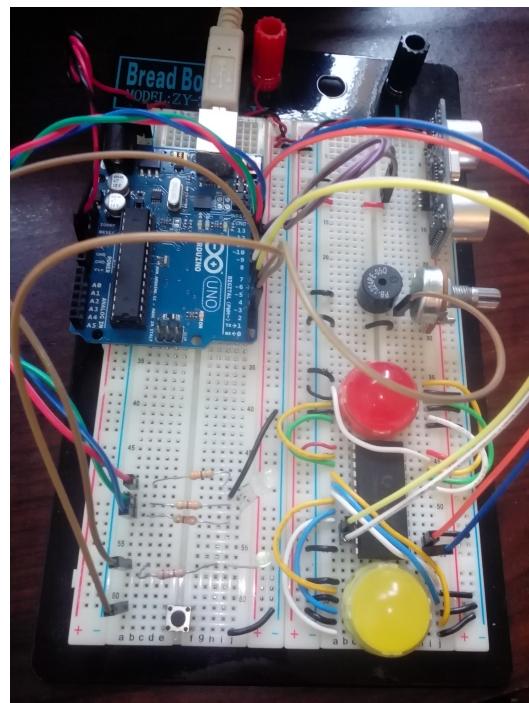


Figura B.3: Sketch Arduino - realizzazione reale

## B.2 Sketch Odroid - lato server

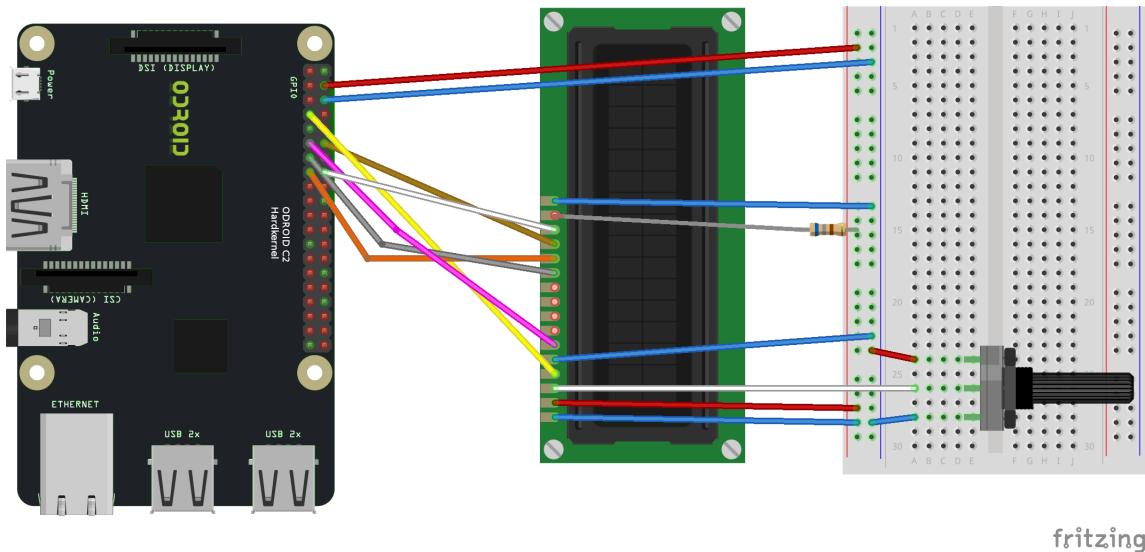


Figura B.4: Sketch Arduino - fritzing

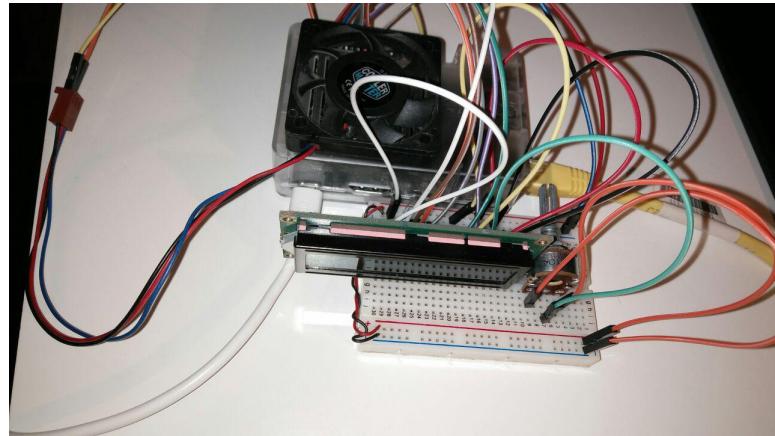


Figura B.5: Sketch Odroid - realizzazione reale

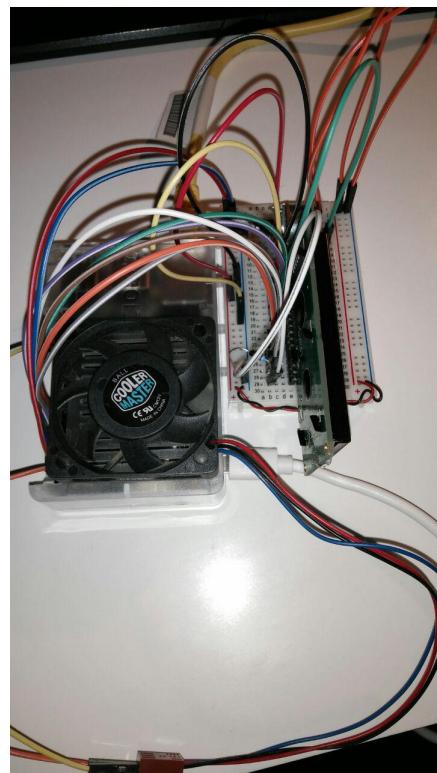


Figura B.6: Sketch Odroid - realizzazione reale