ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Sviluppo di un Framework per la Realizzazione di Micromondi Ciberfisici Distribuiti a supporto dell'Apprendimento del Pensiero Computazionale

Tesi di laurea in:
SISTEMI EMBEDDED E INTERNET-OF-THINGS

Relatore
Alessandro Ricci

Correlatore
Ylenia Battistini

Candidato
Mauro Pellonara

III Sessione di Laurea
Anno Accademico 2023-2024

Abstract

Questa tesi presenta lo sviluppo di un framework innovativo per la realizzazione di Micromondi Ciberfisici Distribuiti a supporto dell'apprendimento del pensiero computazionale, mirato a renderlo più efficace, pratico e interattivo. Più nello specifico, per rendere i Micromondi ciberfisici, si è scelto di integrarvi una rete di sistemi embedded.

I Micromondi sono ambienti virtuali dove gli studenti possono manipolare oggetti digitali e osservare immediatamente i risultati delle loro azioni. Questo approccio è basato sul costruzionismo, teoria dell'apprendimento sviluppata da Seymour Papert, che enfatizza l'importanza di imparare attraverso la costruzione attiva della conoscenza e la manipolazione di oggetti tangibili. È stato scelto Snap! come linguaggio di programmazione per i Micromondi per la sua natura visuale, che consente agli studenti di scrivere programmi tramite blocchi grafici, eliminando la necessità di memorizzare la sintassi di un linguaggio di programmazione tradizionale.

L'architettura del framework include un broker di messaggi che gestisce la comunicazione tra i Micromondi in esecuzione su Snap! e i sistemi embedded. Il framework consente agli studenti di controllare sensori e attuatori in tempo reale senza necessità di configurazione ed eliminando le complessità tradizionalmente associate alla programmazione di sistemi embedded.

La validazione del framework è stata effettuata tramite diversi casi di studio pratici, che ne hanno dimostrato l'efficacia e l'affidabilità. Molti di questi sono anche stati presentati alla Notte dei Ricercatori 2024, suscitando grande interesse e apprezzamento nel pubblico. I riscontri ottenuti indicano che l'integrazione dei sistemi embedded nei Micromondi su Snap! è coinvolgente e stimolante per gli utenti, con il potenziale di migliorare significativamente l'apprendimento del pensiero computazionale.

Indice

Abstract **iii**

1 Pensiero Computazionale e Micromondi **1**
1.1 Pensiero Computazionale 1
1.2 Il concetto di "Micromondo" 4
1.3 Micromondi e Sistemi Embedded 6

2 Un Framework per l'Integrazione della Piattaforma Snap! con una Rete di Sistemi Embedded **9**
2.1 Definizione e Descrizione 9
2.2 Requisiti funzionali 11
2.3 Requisiti non funzionali 11
2.4 Vincoli 12

3 Progettazione e Sviluppo del Framework **13**
3.1 Progettazione 13
3.1.1 Architettura 13
3.1.2 Design dettagliato 17
3.2 Implementazione 22
3.2.1 Broker di Messaggi 22
3.2.2 Integrazione con Snap! 26
3.2.3 Sistemi Embedded 32
3.3 Validazione 35
3.4 Utilizzo 42

4 Conclusione e Sviluppi Futuri **43**

Bibliografia **45**

Capitolo 1

Pensiero Computazionale e Micromondi

1.1 Pensiero Computazionale

Il pensiero computazionale consiste nella capacità di formulare problemi e soluzioni in modo tale che siano eseguibili da elaboratori e umani. Questo concetto è stato portato all'attenzione della comunità scientifica nel 2006 da un articolo di Jeannette Wing [30], in questo viene descritto il pensiero computazionale come una competenza fondamentale per tutti, non solo per i programmatori. Esercitare il pensiero computazionale comporta il compiere diversi processi mentali, tra cui:

- la decomposizione, ovvero suddividere problemi complessi in parti più piccole e facili da analizzare,
- l'astrazione, ovvero sintetizzare problemi in concetti fondamentali e isolarli dal resto, focalizzandosi solo sugli aspetti chiave,
- il riconoscimento di schemi, ovvero individuare similitudini e differenze tra problemi,
- la progettazione di algoritmi, ovvero ideare sequenze di istruzioni o azioni che risolvano un determinato problema,

- la generalizzazione, ovvero applicare la soluzione di un problema a un insieme più grande di problemi,

- l'individuazione e la risoluzione di errori.

Insieme, questi processi mentali aiutano a risolvere problemi complessi in modo sistematico e riproducibile, sono quindi generalmente utili per molte discipline [30].

Inoltre, attraverso l'esercizio del pensiero computazionale, è anche possibile sviluppare ed allenare capacità come il problem-solving, il pensiero logico e la creatività [30]. Queste capacità, insieme ai processi mentali prima elencati, sono essenziali per la vita quotidiana e le professioni tradizionali, quindi non solo per chi intraprende una carriera nell'informatica. Ad esempio, la capacità di decomporre un problema in parti più piccole è utile per un ingegnere del software ma anche per un capoufficio che deve assegnare i turni di lavoro ai propri dipendenti.

Il concetto di pensiero computazionale non proviene da Jeanette Wing, che ha solo contribuito a diffonderlo, ma esisteva già dal 1980 quando fu ideato da Seymour Papert [22], ricercatore nel campo dell'educazione e dell'informatica presso il MIT. Questo lo ha descritto per primo collocandolo nell'educazione, enfatizzando come questo approccio aiuti gli studenti a sviluppare competenze e capacità applicabili in molte discipline [22]. Infatti, nel mondo attuale, il pensiero computazionale trova largo campo nell'educazione, in particolare nelle discipline STEM (Scienza, Tecnologia, Ingegneria e Matematica) che stanno venendo sempre più valorizzate nel mondo del lavoro [5]. Queste discipline richiedono capacità esercitate durante l'applicazione del pensiero computazionale, come la solida comprensione delle metodologie di problem-solving, la padronanza del pensiero logico e le capacità di astrazione e decomposizione [5].

Nell'apprendimento del pensiero computazionale, generalmente si chiede agli studenti di ideare algoritmi e scrivere programmi per risolvere problemi o svolgere determinati compiti [22]. Per semplificare la scrittura dei programmi, Papert ha inventato LOGO, il primo linguaggio di programmazione a scopo didattico. Questo, essendo pensato per scopi educativi e orientato all'infanzia, è stato progettato per essere un linguaggio semplice e accessibile, permettendo agli studenti di esplorare concetti matematici e geometrici attraverso la programmazione.

Dal linguaggio LOGO è stato poi derivato un linguaggio di programmazione visuale chiamato Scratch, che utilizza blocchi grafici invece che istruzioni testuali [15]. Ogni blocco rappresenta un'istruzione o un comando e può essere combinato con altri blocchi in vari modi per creare programmi complessi. Ad esempio, un blocco chiamato "ripeti" verrebbe utilizzato per eseguire più volte una o più azioni, mentre un blocco chiamato "se" permetterebbe di eseguire azioni diverse in base a condizioni specifiche. Questo approccio visuale riduce la necessità di memorizzare la sintassi del linguaggio di programmazione, permettendo agli utenti di concentrarsi sulla logica del programma [29]. I blocchi rappresentano quindi concetti fondamentali della programmazione e del pensiero computazionale come cicli, condizioni e variabili. Questo approccio rende la programmazione accessibile a tutti e ne semplifica la comprensione, indipendentemente dall'età o dalle competenze tecniche degli utenti.

Per quanto Scratch sia completo per un uso semplice, non dispone di tutte le strutture necessarie per essere considerato un perfetto equivalente visuale dei linguaggi di programmazione moderni tipici delle Scienze Informatiche [24]. A risolvere questo ci ha pensato Snap!, una derivazione di Scratch sviluppata da UC Berkeley [18] che non richiede alcuna installazione poiché il suo ambiente di sviluppo, chiamato anch'esso Snap!, viene eseguito nel browser.

Una volta aperto, l'interfaccia presenta due barre laterali: quella a sinistra mostra i blocchi disponibili, quella a destra mostra un riquadro in cui viene presentato il risultato delle azioni eseguite dai blocchi. Per sviluppare un programma, è sufficiente trascinare i blocchi dalla barra laterale di sinistra verso l'area centrale. L'interfaccia si presenta così:

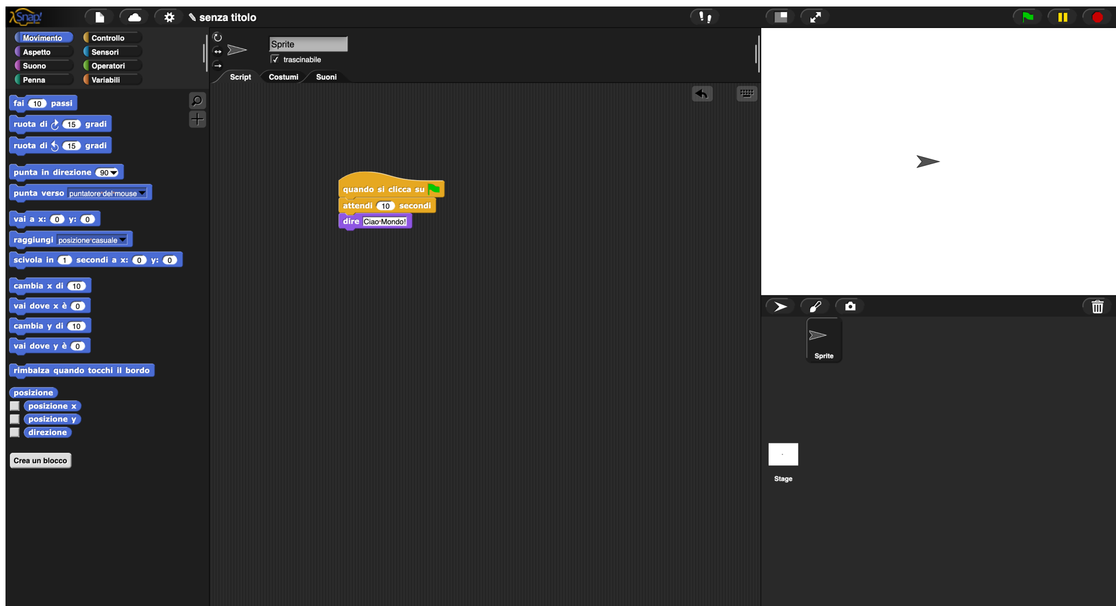

Figura 1.1: Schermata principale dell'ambiente di sviluppo di Snap!

Uno dei principali vantaggi di Snap! nell'insegnamento del pensiero computazionale è la sua interfaccia intuitiva, questa consente anche di osservare immediatamente il risultato delle azioni svolte dai blocchi. Ciò è cruciale per l'apprendimento, poiché permette agli studenti di dedicarsi subito alla ricerca e alla correzione degli errori. Facilitare l'approccio di *trial and error* è fondamentale per l'apprendimento degli studenti, poiché incoraggia l'esplorazione e la sperimentazione attraverso l'esperienza diretta [7].

Un ulteriore vantaggio di Snap! è la sua flessibilità e scalabilità. Può essere utilizzato per insegnare i concetti di base della programmazione ai principianti, ma è anche potente abbastanza da essere utilizzato per costruire progetti complessi. Questo lo rende uno strumento ideale per un'ampia varietà di contesti educativi, dalla scuola primaria fino all'università [18].

1.2 Il concetto di "Micromondo"

Nel contesto del pensiero computazionale e del linguaggio LOGO, Seymour Papert ha anche coniato il termine "Micromondo". Per spiegare in cosa consiste, è necessario introdurre un'altra creazione di Papert: la teoria del costruzionismo

nell'educazione [21]. Questa teoria si basa sull'idea che gli studenti apprendono meglio quando sono attivamente coinvolti nella costruzione di oggetti tangibili, permettendo loro di esplorare e sperimentare con idee in modo concreto. Non solo, questa si basa anche sulla teoria costruttivista che la conoscenza non è semplicemente trasmessa dai docenti agli studenti, ma è costruita attivamente da quest'ultimi attraverso esperienze e interazioni dirette con il mondo. Questo approccio promuove negli studenti:

- l'autonomia, in quanto il processo di costruzione di oggetti tangibili è guidato dagli studenti, che ne hanno la responsabilità, con il supporto dei docenti. Gli studenti sono quindi i protagonisti e i principali attori del proprio percorso di apprendimento,
- l'apprendimento attivo, in quanto gli studenti costruiscono conoscenza attraverso esperienze pratiche e concrete, permettendo una memorizzazione più duratura delle nozioni apprese,
- il pensiero critico, in quanto, durante la costruzione di oggetti tangibili, gli studenti sono portati a giudicare il funzionamento del prodotto creato da loro stessi,
- il problem-solving, in quanto gli studenti vengono posti davanti a problemi di cui devono trovare e costruire la soluzione,
- la creatività, sia nel trovare una soluzione al problema posto sia nel dirigere il processo di creazione di oggetti tangibili.

Insieme, queste competenze e atteggiamenti contribuiscono a rendere l'apprendimento più coinvolgente ed efficace [21].

Nel costruzionismo, l'uso di tecnologie educative come i computer e gli ambienti di programmazione visuale, come ad esempio Scratch e Snap!, è essenziale. Questo perché permettono agli studenti di creare e manipolare oggetti digitali, facilitando una comprensione più profonda delle nozioni da acquisire [21]. Proprio a questo scopo è stato creato il concetto di Micromondo.

Un Micromondo è essenzialmente un ambiente di apprendimento virtuale dove gli studenti possono manipolare oggetti digitali e osservare immediatamente i

risultati delle loro azioni. Questi ambienti sono progettati per essere accessibili e comprensibili a tutti, offrendo allo stesso tempo un alto potenziale creativo. Papert ha descritto questo concetto con l'espressione "Low floors, high ceilings" (pavimenti bassi, soffitti alti), sottolineando che debba essere facile iniziare a costruire Micromondi, ma è anche necessario offrire una vasta gamma di attività creative e complesse [22]. Questo approccio permette a ogni studente di esplorare e sviluppare le proprie idee, indipendentemente dal livello iniziale di competenza.

Mitchel Resnick, un altro importante ricercatore nel campo dell'educazione e dell'informatica presso il MIT, ha esteso il concetto di Papert introducendo l'espressione "Wide walls" (muri larghi). Resnick, noto per il suo lavoro sul progetto Scratch, ha enfatizzato l'importanza di creare ambienti di apprendimento che non solo abbiano "pavimenti bassi" e "soffitti alti", ma anche "muri larghi" per permettere agli studenti di esplorare nuovi concetti e idee [25]. Per raggiungere questo obiettivo, i Micromondi devono poter essere continuamente estendibili e ampliabili, un processo facilitato dai contributi e dalla condivisione di idee da parte degli studenti stessi.

Una piattaforma ideale per l'esecuzione di Micromondi è Snap!, che è derivato da Scratch, il progetto di Resnick, e ne condivide molti dei principi educativi. Gli utenti possono costruire Micromondi manipolando i blocchi di Snap! che sono progettati per essere utilizzati da tutti, indipendentemente dall'età o dal livello di competenza tecnica, riflettendo perfettamente il concetto di "Low floors" di Papert. Inoltre, grazie alla capacità di creare progetti complessi utilizzando sempre gli stessi semplici blocchi, Snap! riflette anche il concetto di "high ceilings". Infine, Snap! offre un forum con una comunità attiva giornalmente e numerosa [26]. Questo, combinato con la sua natura open-source, promuove pienamente il concetto di "Wide walls" di Resnick, consentendo un continuo scambio di idee e l'inclusione di nuove funzionalità e concetti nei Micromondi.

1.3 Micromondi e Sistemi Embedded

Per facilitare e rendere più efficace l'apprendimento del pensiero computazionale, questo può essere condotto all'interno di Micromondi, applicando così la teoria del costruzionismo. Tuttavia, l'ambiente offerto dai Micromondi è virtuale e non facil-

mente tangibile quanto la realtà fisica, il ché ne limita l'efficacia nell'applicazione della teoria del costruzionismo. Questa limitazione sarebbe superata se si potesse intersecare la realtà fisica con quella virtuale offerta dai Micromondi, creando quindi dei Micromondi Ciberfisici. In tal modo si mitigherebbe inoltre la natura astratta del pensiero computazionale rendendone l'apprendimento ancora più pratico e coinvolgente. A questo scopo possono essere impiegati i sistemi embedded, elaboratori compatti ed economici specializzati nello svolgere funzioni specifiche. Questi sistemi sono progettati per operare in tempo reale sull'ambiente circostante, compiendo azioni fisiche e monitorando fenomeni tramite sensori e attuatori. Quest'ultimi sono componenti cruciali dei sistemi embedded:

- I sensori raccolgono dati dall'ambiente esterno, come temperatura, movimento e pressione, e li convertono in segnali elettrici che possono essere elaborati dai sistemi embedded.
- Gli attuatori ricevono comandi dai sistemi embedded e compiono azioni fisiche, come muovere un motore, accendere una luce LED o emettere un suono.

L'integrazione di sensori e attuatori nei sistemi embedded permette loro di interagire attivamente con l'ambiente circostante, rendendoli fondamentali in molte applicazioni moderne [1].

Si potrebbero quindi creare Micromondi Ciberfisici che, attraverso i sistemi embedded, eseguano azioni fisiche tramite gli attuatori e monitorino fenomeni attraverso i sensori. Ad esempio, gli studenti potrebbero realizzare Micromondi in grado di controllare una matrice di LED, rilevare la presenza di persone in una stanza o condurre esperimenti scientifici. Questi esempi vanno oltre l'apprendimento del pensiero computazionale, estendendosi alle materie STEM. In questo modo, l'apprendimento non solo diventerebbe coinvolgente, ma anche profondamente interdisciplinare, contribuendo a fornire agli studenti una visione più ampia del mondo e della conoscenza, e permettendo loro di comprendere meglio le connessioni tra le diverse discipline.

Tuttavia, la programmazione di sistemi embedded può essere complessa e richiedere conoscenze tecniche avanzate, il ché potrebbe scoraggiare gli utenti meno esperti e compromettere il loro coinvolgimento [1]. Per sfruttare appieno il

potenziale dei sistemi embedded, è quindi fondamentale evitare le complessità tradizionalmente associate alla programmazione di questi sistemi, nascondendole agli utenti. Questo problema sarebbe risolto se fosse possibile pilotare i sistemi embedded utilizzando semplici blocchi di Snap! all'interno dei Micromondi, fornendo così un'interfaccia intuitiva che permetta agli studenti di concentrarsi sull'apprendimento.

Il progetto descritto in questa tesi mira quindi a integrare i sistemi embedded con i Micromondi in esecuzione su Snap!. Gli utenti dovranno poter pilotare i sistemi embedded senza necessità di configurarli e quindi concentrandosi esclusivamente sull'apprendimento del pensiero computazionale.

Capitolo 2

Un Framework per l'Integrazione della Piattaforma Snap! con una Rete di Sistemi Embedded

2.1 Definizione e Descrizione

Attualmente, esiste un'unica libreria di Snap! che permette di pilotare sistemi embedded, chiamata Snap4Arduino [2]. Tuttavia, questa libreria richiede che il programma scritto tramite i blocchi grafici venga compilato e caricato sui sistemi embedded a ogni modifica. Inoltre, non supporta pienamente l'uso di molteplici sistemi contemporaneamente, poiché consente di leggere l'output prodotto da un solo sistema alla volta, rendendo l'esperienza tediosa. Limitare l'uso a un unico sistema embedded non è un'opzione, poiché vincolerebbe eccessivamente la libertà e la creatività degli studenti.

Per superare questi ostacoli, si può applicare un approccio alternativo in cui agli utenti verrebbero forniti sistemi embedded con un programma pre-caricato capace di eseguire un insieme ben definito di azioni su attuatori e sensori. I Micromondi in esecuzione su Snap! si occuperebbero solo di comunicare quali azioni eseguire ai sistemi embedded e attendere eventuali risposte. Questo approccio richiede la definizione di un framework che consenta di inviare e ricevere messaggi tra i sistemi embedded e i Micromondi in esecuzione su Snap!.

Eliminando la necessità di caricare manualmente i programmi sui sistemi embedded, questi dispositivi non saranno più vincolati a cavi e a una locazione specifica. Se i sistemi embedded e i Micromondi con cui interagiscono fossero connessi alla stessa WLAN, sarebbe possibile spostare e riposizionare facilmente i sistemi embedded, purché rimangano all'interno di questa.

L'uso del framework non deve richiedere alcuna configurazione e deve essere intuitivo, rendendolo adatto anche a chi ha competenze tecniche limitate. Per garantire affidabilità, è anche necessario poter monitorare lo stato del framework, permettendo di controllare il suo andamento e facilitando il supporto remoto in caso di malfunzionamenti.

Considerando che l'ambito principale di utilizzo è educativo, l'impiego di una vasta gamma di sensori e attuatori può contribuire a stimolare la creatività e migliorare l'apprendimento del pensiero computazionale. I sensori e gli attuatori saranno collegati ai sistemi embedded che potranno essere molteplici. Inoltre, per non limitare sviluppi futuri, il framework deve permettere l'aggiunta di nuovi sensori e attuatori senza grosse complicazioni. Lo stesso vale anche per il supporto di diverse famiglie di sistemi embedded.

Si stabiliscono anche due casi di studio che insieme toccano tutti gli aspetti del progetto. Il primo prevede la creazione di un Micromondo che permetta di spegnere e accendere una matrice di LED con un colore casuale utilizzando un pulsante. La matrice di LED e il pulsante saranno connessi a un unico sistema embedded. Ci si auspica di poter realizzare questo caso di studio sviluppando un Micromondo semplice e intuitivo, privo di complessità o livelli di astrazione che potrebbero confondere gli utenti. Inoltre, è fondamentale che non vi siano ritardi nell'interazione, al fine di mantenere l'aspetto "magico" e coinvolgente dell'esperienza. Il funzionamento deve quindi essere semplice, rapido e riproducibile.

Il secondo caso di studio, che completa il primo, consiste nel mettere a disposizione cinque sistemi embedded, ognuno dotato di una matrice di LED, e permettere agli utenti di interagirci liberamente creando un loro Micromondo. Anche in questo scenario, il funzionamento deve essere privo di ritardi. Questo caso di studio mira a verificare la scalabilità del framework, dimostrando la sua capacità di gestire più sistemi embedded contemporaneamente. Inoltre, permette anche di valutare la sua versatilità, assicurando che gli utenti possano utilizzarlo liberamente per esprimere

la propria creatività e integrare i sistemi embedded nei loro Micromondi.

2.2 Requisiti funzionali

Si prevedono perciò una serie di requisiti funzionali fondamentali per garantire l'efficacia e l'utilità del framework:

- Disponibilità di blocchi Snap! semplici e intuitivi che permettano di interagire con sistemi embedded, facilitando notevolmente l'uso da parte di utenti con competenze tecniche limitate.
- Supporto per molteplici sistemi embedded, utilizzabili contemporaneamente da più Micromondi.
- Libertà di posizionamento dei sistemi embedded, per non vincolarli a dei cavi e quindi ad una specifica locazione.
- Assenza di configurazione per gli utenti, per garantire la fruibilità del framework a tutti.
- Compatibilità con un'ampia gamma di sensori e attuatori, per non limitare la creatività degli utenti, permettendo loro di sperimentare in più modi possibili.

2.3 Requisiti non funzionali

Per assicurare che il framework sia non solo efficace ma anche robusto, sono stati identificati i seguenti requisiti non funzionali:

- Interazioni reattive e veloci con i sistemi embedded, per garantire un'esperienza d'uso fluida e senza ritardi.
- Monitoraggio dello stato del framework, per agevolare il supporto remoto e quindi garantire maggiore affidabilità.

- Supporto per l'integrazione di nuovi sensori, attuatori e famiglie di sistemi embedded. Inoltre, il framework dovrebbe essere potenzialmente implementabile anche per piattaforme d'esecuzione di Micromondi diverse da Snap!.
- Usabile anche da persone senza competenze tecniche specifiche, per rendere il framework ideale per l'utilizzo nelle scuole e in contesti educativi.

2.4 Vincoli

Sono anche presenti alcuni vincoli che devono essere rispettati per garantire la fattibilità e l'adozione del framework:

- Utilizzo di sistemi embedded, sensori e attuatori economicamente accessibili, per non limitare una possibile diffusione su larga scala, specialmente nelle scuole.
- Compatibilità completa con il linguaggio Snap!, per garantire una base di utilizzo consolidata e diffusa in ambito educativo.

Capitolo 3

Progettazione e Sviluppo del Framework

3.1 Progettazione

3.1.1 Architettura

I sistemi embedded, per comunicare con i Micromondi con i quali interagiscono, devono essere connessi alla stessa WLAN di quest'ultimi. Inoltre, non potendo affidare la configurazione agli utenti, bisogna far si che i sistemi embedded si connettano automaticamente alla WLAN. Questa dovrà essere aperta da un Access Point che verrà fornito già configurato agli utenti insieme ai sistemi embedded.

L'utente dovrà anche poter distinguere un sistema embedded dagli altri per poterci interagire, è quindi necessario assegnare a ognuno un codice identificativo univoco all'interno del Micromondo in cui verrà utilizzato. In tal modo, sarà possibile fornire un Micromondo già configurato per i sistemi embedded disponibili all'utente.

Un possibile approccio per pilotare i sistemi embedded tramite WLAN consiste nell'invio da parte dei Micromondi di richieste HTTP contenenti un'azione da svolgere. Gli indirizzi IP dei sistemi embedded dovranno essere statici e determinabili a partire dai loro codici identificativi. Questo approccio comporterebbe che ogni sistema embedded debba eseguire un server Web localmente per ricevere le richieste HTTP.

Il principale problema è che un uso tipico dei sistemi embedded consiste nell'attendere il verificarsi di un evento (e.g. la pressione di un pulsante); in questo caso sarebbe necessario differire la risposta alla richiesta HTTP fino a quando non si verifichi l'evento (semmai avvenisse). Ciò non è una soluzione né affidabile né elegante, in quanto richiederebbe di impostare il *timeout* delle richieste HTTP a infinito e bloccherebbe il sistema embedded dallo svolgere altre azioni fino a quando non si verifichi l'evento desiderato [3] [13].

L'alternativa sarebbe inviare continuamente richieste HTTP (i.e. *polling*) ma questo, oltre a mettere sotto stress i sistemi embedded e la rete [19], non rileverebbe correttamente le azioni che avvengono tra una richiesta e l'altra.

Per risolvere questo problema, si può introdurre un broker di messaggi al quale si connetterebbero sia i Micromondi che i sistemi embedded. Il broker sarebbe perciò l'unico elaboratore a necessitare di un indirizzo IP statico e quello incaricato di gestire e scambiare messaggi. Il protocollo di riferimento per il brokeraggio di messaggi è MQTT, che offre dei canali chiamati *topic* ai quali i client si possono iscrivere per ricevere i messaggi pubblicati da altri client [16]. In questo caso, ogni sistema embedded si iscriverebbe a un topic di input per le richieste dai Micromondi e pubblicherebbe su un topic di output le risposte. I Micromondi, invece, pubblicherebbero richieste sui topic di input dei sistemi embedded e si iscriverebbero ai loro topic di output.

In questo modo si risolve il problema in quanto i sistemi embedded possono rispondere in modo asincrono alle richieste dei Micromondi pubblicando un messaggio sul proprio topic di output. Ulteriormente, dai codici identificativi dei sistemi embedded si potrà determinare il nome dei topic di input e output di quest'ultimi, non più il loro indirizzo IP che può anche essere dinamico.

La scelta di utilizzare MQTT sembra perfetta in quanto Snap! offre una libreria di blocchi progettata per rendere semplice e immediato il suo utilizzo [23]. Tuttavia, è importante notare che, se eseguito nel browser come nel caso di Snap!, MQTT utilizza implicitamente il protocollo WebSocket [20]. Pertanto, l'uso di MQTT rappresenta di fatto l'aggiunta di un inutile strato di *overhead*. Una soluzione più snella consiste nell'implementare la logica dei topic di MQTT utilizzando direttamente WebSocket e creare una libreria di blocchi su Snap! che permetta di utilizzare WebSocket in tal modo. Si evita così di sovrapporre protocolli e si

ottimizza la comunicazione in termini di velocità e risorse [20].

Con questo approccio, si ottiene che i sistemi embedded eseguirebbero un client WebSocket invece di un server Web. Inoltre, il server sul quale è in esecuzione il broker potrebbe anche occuparsi di aprire la WLAN e quindi rimuovere la necessità di un Access Point separato. Il broker, sul quale è quindi centralizzato il framework, puó anche occuparsi di offrire una panoramica generale di quest'ultimo per monitorarne facilmente lo stato. Questa potrebbe anche essere usata come strumento di diagnostica nel fornire supporto remoto agli utenti.

Infine, per semplicità d'uso e brevità, si è scelto di riferirsi ai sistemi embedded come "dispositivi". Non è necessario che l'utente finale conosca il nome tecnico degli strumenti utilizzati; è, invece, fondamentale evitare la possibile confusione causata dall'uso di termini sconosciuti.

L'architettura del framework si divide perciò in tre parti:

- il broker di messaggi,
- una o più istanze del Micromondo in esecuzione su Snap!,
- i sistemi embedded.

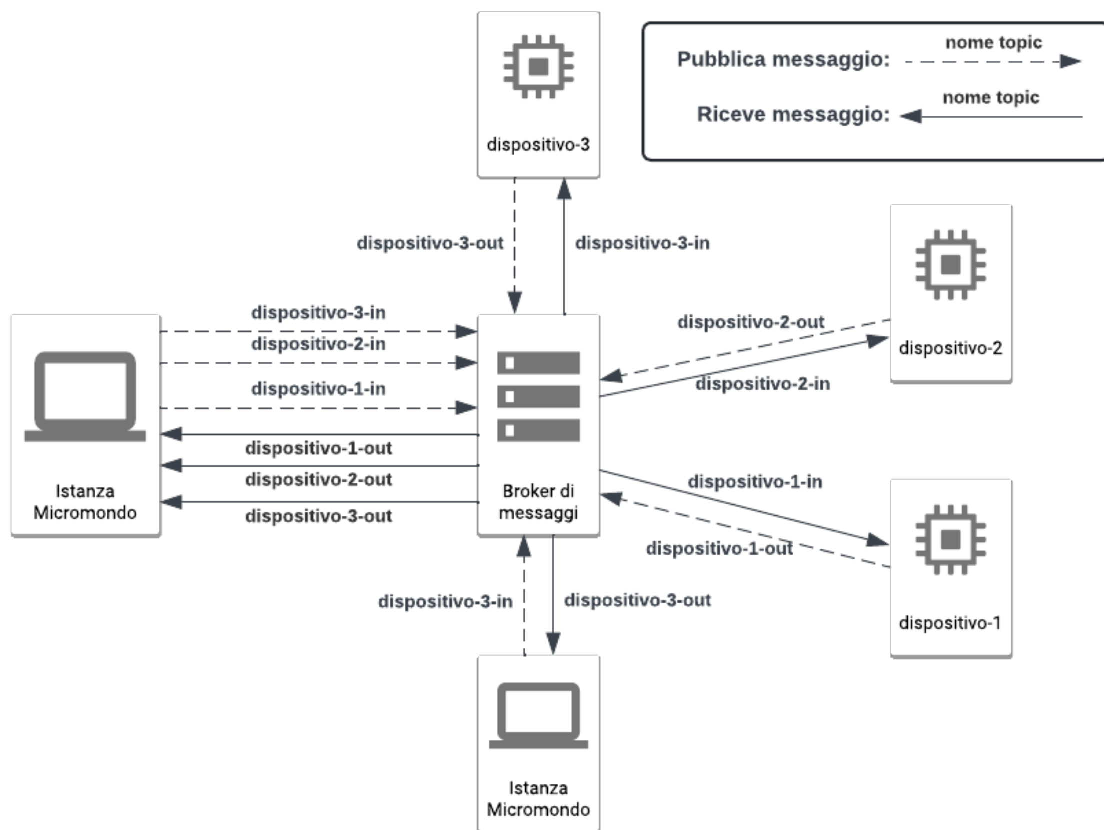

Figura 3.1: Architettura del framework

3.1.2 Design dettagliato

Broker di messaggi

Il broker di messaggi è in pratica un server WebSocket al quale si connettono tutti i sistemi embedded e i Micromondi. Siccome si vuole implementare la stessa logica di MQTT, il server WebSocket accetterà solo due tipi di messaggi: *publish* (pubblicazione) e *subscribe* (iscrizione). Nel caso riceva un messaggio di tipo *publish*, il broker lo inoltrerà a tutti i client iscritti al topic sul quale era stato pubblicato. Nel caso riceva un messaggio di tipo *subscribe*, il broker aggiungerà il client che ha inviato il messaggio alla lista di client iscritti al topic indicato.

Insieme ai messaggi di tipo *publish* e *subscribe* va incluso anche un *payload*, ovvero dei dati aggiuntivi che vengono trasmessi rispettivamente ai sistemi embedded e al broker di messaggi.

Il *payload* dei messaggi di tipo *subscribe* include:

- `clientName`: il nome utilizzato per identificare il client nel framework.

Il *payload* dei messaggi di tipo *publish* include:

- `target`: il nome del sensore o attuatore sul quale si vuole svolgere un'azione (e.g. la matrice di LED),
- `action`: il nome dell'azione da svolgere sul sensore o attuatore (e.g. accensione con un colore),
- `params`: dati necessari per svolgere l'azione (e.g. il colore con il quale accendere la matrice di LED).

Il broker leggerà tutti i messaggi nel framework e manterrà traccia di quali sistemi embedded e Micromondi sono connessi, il che permette, tramite un server Web, di offrire una pagina per visualizzare una panoramica generale del framework.

Sistemi embedded

I sistemi embedded ricevono messaggi tramite WebSocket, eseguono le azioni indicate e, se necessario, inviano indietro un messaggio con l'esito dell'azione. Ogni

sistema embedded ha un codice identificativo utilizzato per determinare i nomi dei topic di input e output. Il codice identificativo è costituito da un numero positivo preceduto da "dispositivo-". Il nome del topic di input è formato dal codice identificativo seguito da "-in", mentre il nome del topic di output è formato dal codice identificativo seguito da "-out". Ad esempio, il sistema embedded con codice "dispositivo-4" riceve messaggi dal topic "dispositivo-4-in" e pubblica messaggi sul topic "dispositivo-4-out".

Per non limitare la creatività degli utenti, si è scelto di permettere l'uso di diversi sensori e attuatori, tutti potenzialmente collegati a un unico sistema embedded. Questi sono:

- Pulsante generico: fornisce un modo semplice e diretto di interagire con i Micromondi.
- Sensore di distanza a ultrasuoni HC-SR04: misura la distanza in modo accurato, permettendo interazioni complesse con i Micromondi.
- Sensore di presenza a infrarossi FC-51: rileva la presenza di oggetti in pochi millisecondi.
- Servomotore SG90: permette di agire visibilmente sull'ambiente circostante.
- Matrice 8x8 WS2812 di LED 5050: permette di creare coinvolgenti giochi di luce.
- LED verde e rosso generici: permette di fornire un semplice riscontro positivo o negativo.

Per ogni sensore o attuatore si è cercato di scegliere il modello più comune ed economico disponibile sul mercato.

I sistemi embedded devono perciò potersi connettere alla WLAN e avere almeno 8 pin per interfacciarsi potenzialmente con tutti i sensori e attuatori allo stesso momento. Bisogna quindi scegliere dei microcontrollori consoni per i sistemi embedded.

La scelta è ricaduta su due dei microcontrollori più economici e comuni che soddisfino tutti i requisiti: l'ESP-8266 e l'ESP-32 [14]. Questi divergono leggermente

l'uno dall'altro [14] ma ciò non rappresenta un problema vista la versatilità del framework; infatti quest'ultimo richiede ai sistemi embedded, qualsiasi essi siano, solo la capacità di connettersi a un server WebSocket e interfacciarsi con i sensori e attuatori elencati sopra.

Micromondo

Un punto forte di Snap! è la possibilità di eseguire codice JavaScript all'interno dei blocchi accedendo a tutte le librerie offerte solitamente dai browser, tra cui quella per WebSocket [6] [28]. Inoltre, è possibile creare nuovi blocchi incorporandone altri, ciò permette di aumentare il livello di astrazione e l'intuitività d'uso.

L'approccio scelto è quello di sviluppare prima una libreria di blocchi per l'uso del protocollo WebSocket con JavaScript. Poi, sviluppare un'altra libreria di blocchi per l'uso di WebSocket in stile MQTT, incorporando i blocchi della prima libreria. Infine, un'ultima libreria di blocchi dedicata al pilotare i sistemi embedded, che incorpora all'interno i blocchi per l'uso di WebSocket in stile MQTT.

Queste librerie consentiranno a qualsiasi Micromondo su Snap!, indipendentemente dal suo scopo, di interagire con i sistemi embedded. Inoltre, ogni Micromondo potrà essere configurato per includere tutti o solo alcuni dei blocchi della libreria per pilotare i sistemi embedded, senza comprometterne il funzionamento. In questo modo, sarà possibile creare Micromondi specifici e mirati a determinati scopi, evitando confusione e dispersione per gli utenti che vi agiscono.

La libreria di blocchi per l'uso di WebSocket includerà tre blocchi che permetteranno di:

- connettersi a un server WebSocket,
- inviare un messaggio al server WebSocket,
- leggere i messaggi ricevuti dal server WebSocket.

La libreria di blocchi per l'uso di WebSocket in stile MQTT includerà quattro blocchi che permetteranno di:

- iscriversi a un topic,

- pubblicare un messaggio su un topic,
- attendere che arrivi un messaggio su un topic,
- leggere il prossimo messaggio su un topic.

La libreria di blocchi per pilotare i sistemi embedded includerà blocchi che permetteranno di:

- avviare il Micromondo, ovvero connettersi al server WebSocket,
- accendere la matrice di LED di un sistema embedded con un determinato colore,
- accendere con una sfumatura la matrice di LED di un sistema embedded con un determinato colore,
- spegnere la matrice di LED di un sistema embedded,
- attendere che venga premuto il pulsante di un sistema embedded,
- accendere o spegnere il LED rosso di un sistema embedded,
- accendere o spegnere il LED verde di un sistema embedded,
- ruotare il servomotore di un sistema embedded di un determinato angolo,
- misurare la distanza davanti a un sistema embedded,
- attendere che la distanza davanti a un sistema embedded rientri in un determinato intervallo,
- rilevare la presenza di un oggetto qualsiasi davanti a un sistema embedded,
- attendere che venga rilevata la presenza di un oggetto qualsiasi davanti a un sistema embedded.

Nel caso si desiderasse aggiungere un nuovo sensore o attuatore a quelli già supportati, sarà sufficiente assegnargli un nome (`target`) e definire per ciascuna azione che può compiere il nome (`action`) e i parametri accettati (`params`).

Successivamente, per ogni azione si dovrà aggiungere un blocco alla libreria per pilotare i sistemi embedded che invii un messaggio di tipo *publish* con i relativi `target`, `action` e `params` nel *payload*. Inoltre, andrà aggiornato il programma dei sistemi embedded per far si che implementi le nuove azioni e si interfacci con il nuovo sensore o attuatore.

Questo approccio prevede l'implementazione di due librerie universali per Snap!: una per l'uso generico di WebSocket e una per l'uso specifico di WebSocket in stile MQTT. Queste librerie saranno scritte in inglese per non limitarne una possibile diffusione a livello internazionale. Al contrario, la libreria di blocchi per pilotare i sistemi embedded sarà scritta in modo verboso e in italiano, poiché è necessario che sia intuitiva e facilmente comprensibile dagli utenti italiani alla quale è destinata.

Inoltre, è necessario sottolineare che i Micromondi possono essere eseguiti anche su altre piattaforme oltre a Snap!. L'unico requisito richiesto dal framework a queste piattaforme alternative è la capacità di connettersi a un server WebSocket e di inviare e ricevere messaggi. In realtà, questi requisiti possono essere soddisfatti da praticamente qualsiasi sistema che possa inviare messaggi tramite WebSocket e connettersi a una WLAN. Pertanto, il framework può essere utilizzato in qualsiasi contesto, non solo nei Micromondi, per pilotare molteplici sistemi embedded in modo intuitivo e senza configurazione.

3.2 Implementazione

3.2.1 Broker di Messaggi

Per facilitare l'uso del framework da parte degli utenti, si è scelto di configurare il server che esegue il broker di messaggi affinché apra anche la WLAN. In questo modo, all'avvio, il server aprirà la WLAN ed eseguirà contemporaneamente il broker e il server Web che offre la panoramica generale. Per quanto riguarda il server, è stato scelto uno degli elaboratori più economici, comuni e versatili per questo tipo di situazioni: il Raspberry Pi 4 [12].

Per rendere più semplice l'integrazione con la libreria di blocchi per l'uso di WebSocket scritta in JavaScript, è stato scelto lo stesso linguaggio di programmazione per il broker di messaggi. In particolare, l'ambiente di runtime JavaScript selezionato è *Node.js* per via delle molte librerie disponibili e della capacità di gestione asincrona degli eventi [27].

Inoltre, siccome sono due le parti del framework che eseguono codice in linguaggio JavaScript, è stato scelto JSON come formato per i messaggi inviati tramite WebSocket, in quanto si integra perfettamente con JavaScript ed è ampiamente riconosciuto e supportato [9].

Per implementare il server Web che offre la panoramica generale, è stato scelto il framework *Express* per la sua leggerezza [4]. È presente una singola pagina Web sviluppata utilizzando HTML, CSS e JavaScript. Nel caso siano presenti due sistemi embedded e un Micromondo, la pagina Web si presenta come segue:

Dashboard dei MicroMondi

Scarica resoconto

In questo momento è in esecuzione un solo MicroMondo.

Lista dei canali di informazione con i loro ascoltatori

- **dispositivo-2-in** (canale di input a dispositivo-2):
  - dispositivo-2 (192.168.1.33)
- **dispositivo-4-in** (canale di input a dispositivo-4):
  - dispositivo-4 (192.168.1.32)
- **dispositivo-4-out** (canale di output da dispositivo-4):
  - MicroMondo-v3ej9 (localhost)
- **dispositivo-2-out** (canale di output da dispositivo-2):
  - MicroMondo-v3ej9 (localhost)

Log del server

Orario di invio	Tipo di messaggio	Mittente	Canale di informazione	Contenuto
08/10/2024, 19:33:57	publish	localhost	dispositivo-2-in	{"action":"detectPresence","params":{},"target":"ir"}
08/10/2024, 19:33:57	subscribe	localhost	dispositivo-2-out	{"clientName":"MicroMondo-v3ej9"}
08/10/2024, 19:33:57	publish	localhost	dispositivo-4-in	{"action":"clear","params":{},"target":"ledMatrix"}
08/10/2024, 19:33:57	publish	localhost	dispositivo-4-in	{"action":"detectDistance","params":{"min":0,"max":50},"target":"sonar"}
08/10/2024, 19:33:57	subscribe	localhost	dispositivo-4-out	{"clientName":"MicroMondo-v3ej9"}
08/10/2024, 19:33:57	publish	localhost	dispositivo-2-in	{"action":"clear","params":{},"target":"ledMatrix"}
08/10/2024, 19:33:43	subscribe	localhost	dispositivo-4-in	{"clientName":"dispositivo-4"}
08/10/2024, 19:33:36	subscribe	localhost	dispositivo-2-in	{"clientName":"dispositivo-2"}

Figura 3.2: Pagina web che offre la panoramica generale del framework

Per il server WebSocket è stata scelta la libreria *ws* che consente di creare event listener per la gestione delle connessioni dei client e la ricezione dei loro messaggi.

Quando un client si connette al server WebSocket viene creato un event listener per la ricezione di messaggi da parte di quel client, all'interno di questo si controlla il tipo di messaggio ricevuto e si invoca la corrispondente funzione che lo gestisce.

```
server.on('connection', (client, req) => {
  client.on('message', (rawMessage) => {
      const message = JSON.parse(rawMessage);
      const { type, topic, payload } = message;
      let ipAddress = req.socket.remoteAddress.replace("::ffff:", "");
      if (ipAddress === "::1") ipAddress = "localhost";

      switch (type) {
          case 'subscribe':
              handleSubscribeMessage(topic, ipAddress, payload, client);
              break;
          case 'publish':
              handlePublishMessage(topic, ipAddress, payload);
              break;
          default:
              logMessage("error", "unknown message type", "");
              break;
      }
  });
});
```

La funzione che gestisce i messaggi di tipo *publish* inoltra il messaggio a tutti i client ancora connessi e iscritti al topic in cui è stato pubblicato.

```
const handlePublishMessage = (topic, ipAddress, payload) => {
  logMessage("publish", topic, ipAddress, payload);

  const message = JSON.stringify({
      topic,
      payload
  });

  // send message to all the clients subscribed to the topic
  topicsClients[topic]?.filter(client => client.readyState === WebSocket.OPEN).
      forEach(client => client.send(message));
}
```

La funzione che gestisce i messaggi di tipo *subscribe* crea il topic se non esiste e aggiunge il client che ha inviato il messaggio alla lista dei client iscritti. Inoltre, crea un event listener per la disconnessione del client che elimina il topic se non vi sono client iscritti.

```
const handleSubscribeMessage = (topic, ipAddress, payload, client) => {
  logMessage("subscribe", topic, ipAddress, payload);

  // if it's a new topic create a list to store the clients subscribed to it
  if (!topicsClients[topic]) {
      topicsClients[topic] = [];
  }

  // add the client to the list of clients subscribed to that topic
  if (!topicsClients[topic].includes(client)) {
      client.name = payload.clientName;
      client.ipAddress = ipAddress;
      topicsClients[topic].push(client);
  }

  // when the client closes the connection, remove it from the topic list
  client.on('close', () => {
      logMessage("unsubscribe", topic, ipAddress, payload);

      // checks that the topic still exists
      if (!topicsClients[topic]) return;

      topicsClients[topic] = topicsClients[topic].filter(c => c !== client);

      // delete the topic if there are no clients connected to it
      if (topicsClients[topic].length === 0) {
          delete topicsClients[topic];
      }
  });
}
```

Il codice per il broker di messaggi è visualizzabile all'indirizzo `https://github.com/Innova-Mente/distrib-mw/tree/pellonara`.

3.2.2 Integrazione con Snap!

La parte principale della libreria per l'uso di WebSocket su Snap! consiste nel codice JavaScript eseguito dal blocco per connettersi al server WebSocket. Oltre a ciò, questo tiene traccia anche dei messaggi ricevuti e per ognuno salva la data e l'ora di ricezione. Inoltre, genera un nome casuale per identificare il Micromondo.

```
function initializeWebSocket() {
  const client = new WebSocket("ws://" + serverAddress);
  const receivedMessages = [];
  const clientName = "Micromondo-" + (Math.random() + 1).toString(36).substring(7)
  let isConnectionOpen = false;

  client.onopen = (event) => {
      isConnectionOpen = true;
  };

  client.onmessage = (event) => {
      let data = JSON.parse(event.data);
      data["timestamp"] = new Date().toISOString();
      console.log("Received WebSocket message with data: ", data);
      receivedMessages.push(data);
  };

  client.onclose = (event) => {
      isConnectionOpen = false;
  };

  let sendMessage = (message) => {
      if (!isConnectionOpen) return;
      console.log("Sent WebSocket message with data: ", message);
      client.send(message);
  };

  return {
      client,
      clientName,
      receivedMessages,
      sendMessage,
      getIsConnectionOpen: () => isConnectionOpen
  };
};
// open connection if not there already or closed
if (!window.webSocket || !window.webSocket.getIsConnectionOpen()) {
  webSocket = initializeWebSocket();
  subscribedTopics = [];
}
```

Il blocco che stabilisce la connessione con il server WebSocket crea anche una variabile globale accessibile tramite JavaScript da qualsiasi blocco eseguito nello stesso Micromondo. Questa rappresenta il client WebSocket e consente di:

- accedere ai messaggi ricevuti finora,
- accedere al nome identificativo del Micromondo,
- inviare un messaggio,
- verificare se la connessione è ancora aperta.

La funzione di invio dei messaggi è stata incorporata in un blocco dedicato per maggiore chiarezza nella costruzione delle altre librerie, invece le altre funzioni sono accessibili solo tramite JavaScript.

La libreria per l'uso di WebSocket in stile MQTT include un blocco per pubblicare su un topic un messaggio che indichi di svolgere un'azione (`action`) su un sensore o attuatore (`target`) con dei certi parametri (`params`).

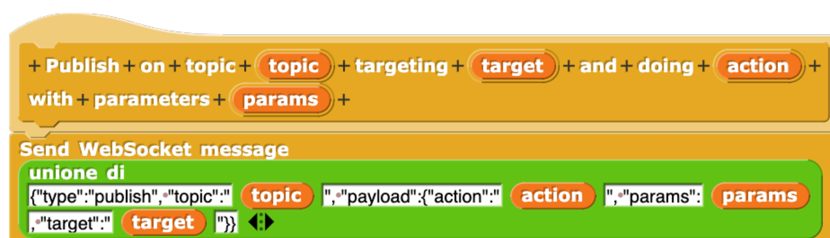

Figura 3.3: Blocco per la pubblicazione di un messaggio su un topic

Un esempio di utilizzo del blocco in Figura 3.3 è il blocco per accendere la matrice di LED di un sistema embedded con un determinato colore.

Figura 3.4: Blocco per l'accensione della matrice di LED di un sistema embedded

La libreria per l'uso di WebSocket in stile MQTT include anche un blocco per iscriversi ad un topic nel caso non si sia già iscritti. Per fare ciò si tiene aggiornata una lista di topic ai quali si è già iscritti.

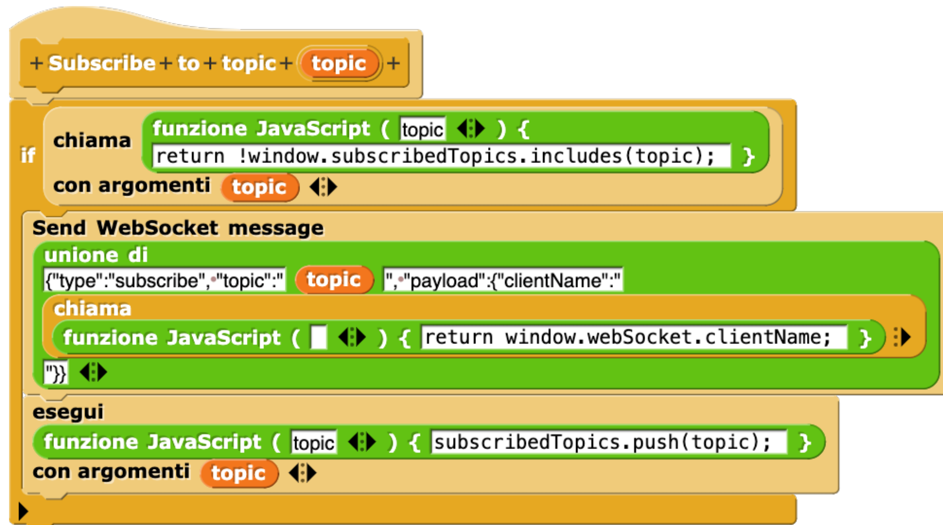

Figura 3.5: Blocco per l'iscrizione ad un topic

La libreria include anche un blocco per attendere il prossimo messaggio riguardante un'azione su un certo sensore o attuatore di un sistema embedded. Per fare ciò tiene conto di quando il blocco si è messo in attesa e controlla solo i messaggi ricevuti dopo. Proprio per questo motivo la libreria per l'uso di WebSocket tiene traccia della data e dell'ora di ricezione dei messaggi.

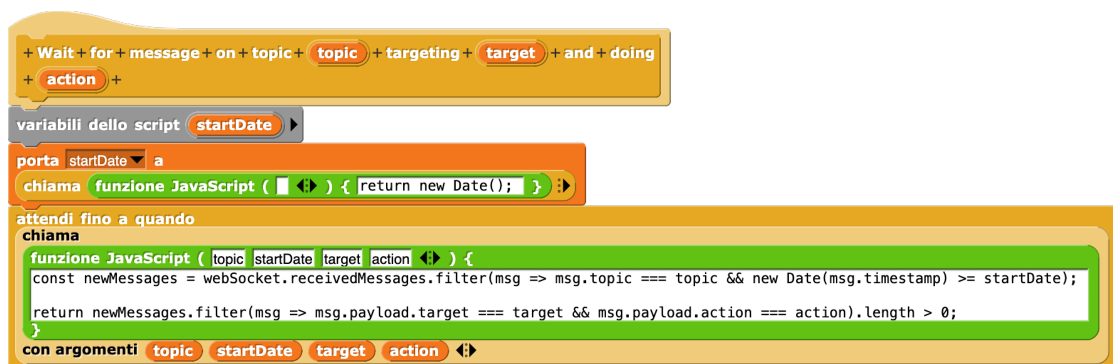

Figura 3.6: Blocco che attende il prossimo messaggio riguardante un'azione su un certo sensore o attuatore di un sistema embedded

La libreria include anche un blocco per leggere il prossimo messaggio riguardante un'azione su un certo sensore o attuatore di un sistema embedded. Il funzionamento è identico al blocco in Figura 3.6 con l'unica differenza che viene restituito il primo messaggio ricevuto e valido.

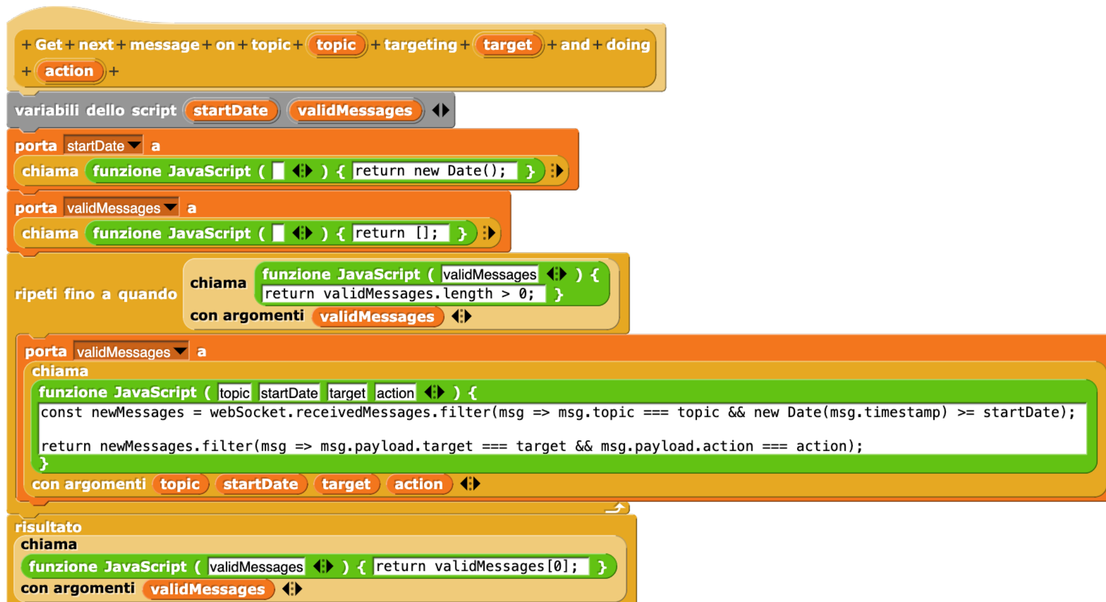

Figura 3.7: Blocco che legge il prossimo messaggio riguardante un'azione su un certo sensore o attuatore di un sistema embedded

Un esempio di utilizzo dei blocchi in Figura 3.3, Figura 3.5 e Figura 3.6 è il blocco per attendere che venga premuto il pulsante di un sistema embedded. Prima, si comunica al sistema embedded di monitorare la pressione del pulsante, poi si attende di ricevere un suo messaggio che conferma la pressione del pulsante.

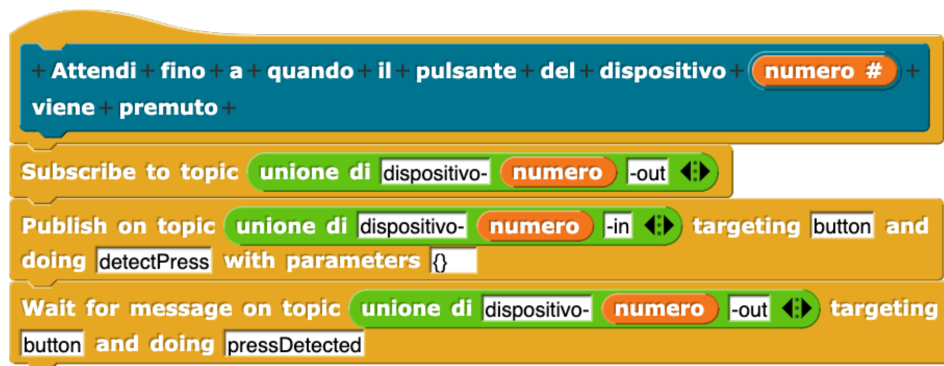

Figura 3.8: Blocco che attende che venga premuto il pulsante di un sistema embedded

Un esempio di utilizzo dei blocchi in Figura 3.3, Figura 3.5 e Figura 3.7 è il blocco per misurare la distanza davanti un sistema embedded. Il funzionamento è analogo a quello del blocco in Figura 3.8: prima, si comunica al sistema embedded di misurare la distanza davanti, poi si attende di ricevere un suo messaggio che contenga il valore misurato della distanza.

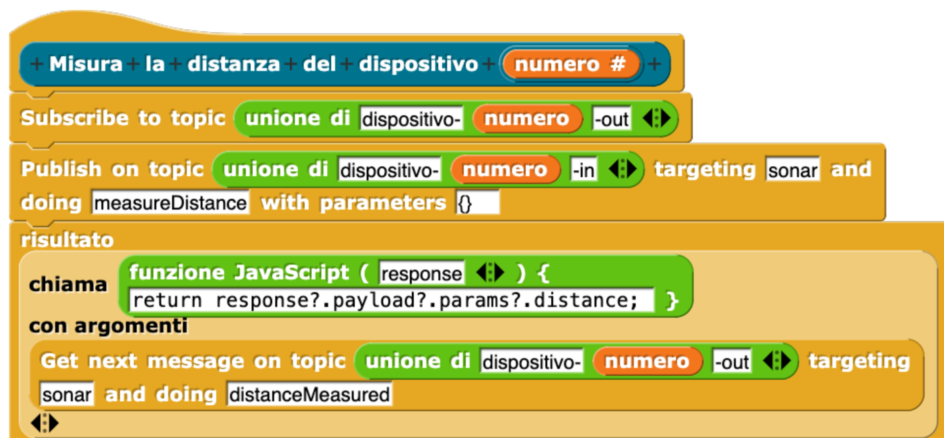

Figura 3.9: Blocco che misura la distanza davanti a un sistema embedded

Tutti i blocchi della libreria per pilotare i sistemi embedded hanno un colore ciano scuro per poter essere facilmente distinti dagli altri. Inoltre, sono scritti in modo dettagliato e verboso per renderne l'utilizzo il più intuitivo possibile. Si presentano come segue:

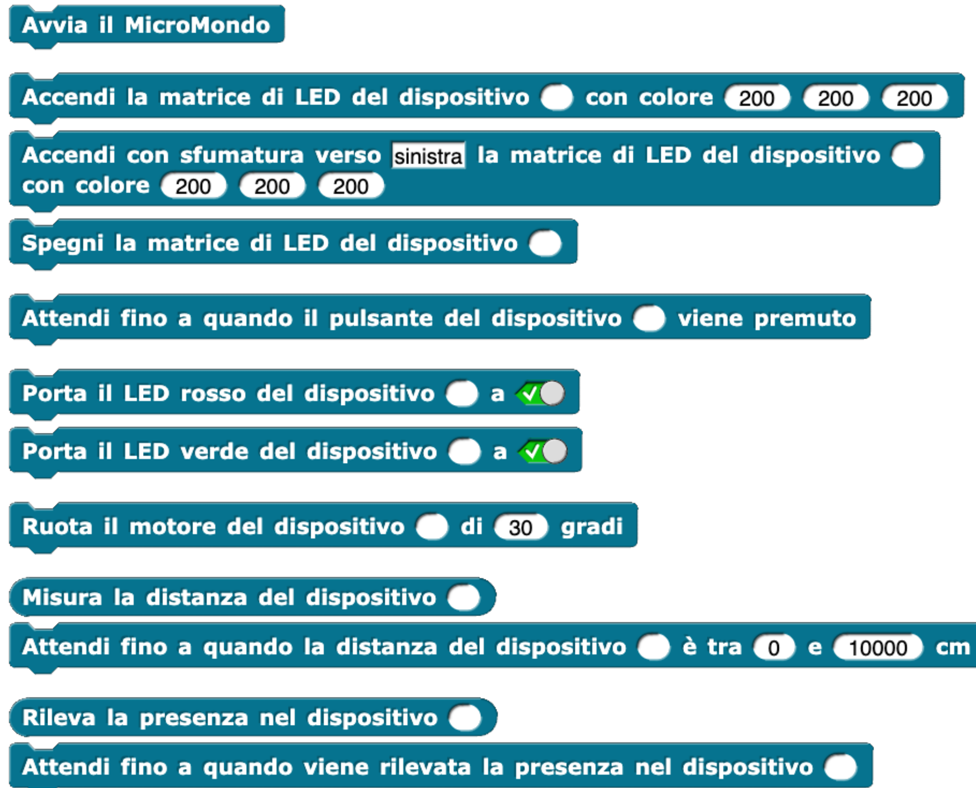

Figura 3.10: Libreria di blocchi per pilotare i sistemi embedded

È possibile scaricare un Micromondo completo di tutte le librerie di blocchi discusse all'indirizzo `https://github.com/Innova-Mente/micromondoteca/blob/pellonara/micromondo-completo.xml`.

3.2.3 Sistemi Embedded

Come linguaggio di programmazione è stato scelto C++ in quanto è lo standard per lo sviluppo di sistemi embedded vista la sua efficienza e ampia disponibilità di librerie [10].

La libreria WebSocket utilizzata consente di creare un listener per gestire gli eventi legati alle connessioni WebSocket. Non appena il sistema embedded si connette al broker che funge da server WebSocket, si iscrive al proprio topic di input per ricevere i messaggi provenienti dai Micromondi. Quando viene ricevuto un messaggio, questo viene convertito in formato JSON e una variabile di controllo globale viene aggiornata per segnalare l'arrivo di un nuovo messaggio da elaborare.

```
void webSocketEvent(WStype_t eventType, uint8_t *message, size_t messageLength)
{
  if (eventType == WStype_CONNECTED)
  {
    Serial.printf("Connected!\n");
    subscribeToTopic(webSocket, INPUT_TOPIC, DEVICE_NAME);
  }
  else if (eventType == WStype_TEXT)
  {
    deserializeJson(lastMessage, message);
    newMessageArrived = true;
    Serial.printf("Received message: \n%s\n", message);
  }
}
```

Quando possibile, sono stati utilizzati gli interrupt per rilevare dati dai sensori. In questo caso, solo il pulsante e il sensore di presenza potevano essere configurati con interrupt.

```
void IRAM_ATTR buttonInterrupt()
{
  pressDetected = true;
}
void IRAM_ATTR irInterrupt()
{
  presenceDetected = true;
}
```

Il superloop [17] si occupa di leggere l'eventuale nuovo messaggio ed eseguirne il parsing per capire quale azione è stata richiesta. Se l'azione richiesta è pressoché istantanea viene eseguita direttamente, altrimenti si imposta una variabile di controllo globale per segnalare di dover monitorare un determinato sensore. Di seguito è riportata una porzione della funzione che fa il parsing dei messaggi:

```
String target = lastMessage["payload"]["target"];
String action = lastMessage["payload"]["action"];

if (target == "ledMatrix")
{
  if (action == "clear")
  {
    ledMatrix->clear();
  }
  else if (action == "fillColor")
  {
    int r = lastMessage["payload"]["params"]["r"];
    int g = lastMessage["payload"]["params"]["g"];
    int b = lastMessage["payload"]["params"]["b"];

    ledMatrix->fillColor(RGB(r, g, b));
  }
}
else if (target == "button")
{
  if (action == "detectPress")
  {
    pinMode(BUTTON_PIN, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), buttonInterrupt, RISING);
  }
}
else if (target == "sonar")
{
  if (action == "detectDistance")
  {
    isSonarDetecting = true;
    minSonarDetectDistance = lastMessage["payload"]["params"]["min"];
    maxSonarDetectDistance = lastMessage["payload"]["params"]["max"];
  }
  else if (action == "measureDistance")
  {
    float distance = sonar->measure();
    publishMessage(webSocket, OUTPUT_TOPIC, "sonar", "distanceMeasured", String("{
        \"distance\": ") + distance + "}");
  }
}
```

Il superloop controlla quindi anche le variabili di controllo globali per determinare se deve monitorare dei sensori. Di seguito è riportata una porzione della funzione che si occupa di ciò e di compiere le corrispondenti azioni:

```
if (pressDetected)
{
  detachInterrupt(digitalPinToInterrupt(BUTTON_PIN));
  pressDetected = false;
  publishMessage(webSocket, OUTPUT_TOPIC, "button", "pressDetected", "{}");
}
if (isSonarDetecting)
{
  float distance = sonar->measure();
  if (distance >= minSonarDetectDistance && distance <= maxSonarDetectDistance)
  {
    isSonarDetecting = false;
    publishMessage(webSocket, OUTPUT_TOPIC, "sonar", "distanceDetected", "{}");
  }
}
```

L'architettura del codice dei sistemi embedded è quindi basata sugli eventi (*event-driven*). Il superloop non esegue nessuna azione a meno che non si verifichino eventi che modifichino delle variabili di controllo globali e costringano quindi il sistema embedded a compiere delle determinate azioni.

Per ogni sensore o attuatore, è stata sviluppata una libreria dedicata che astrae il concetto di pin, fornendo funzioni che riflettono le possibili azioni eseguibili da ciascun sensore e attuatore. Se ne guadagna che il codice è di più facile lettura e manutenibilità.

Infine, le uniche differenze tra il codice per il microcontrollore ESP-8266 e quello per l'ESP-32 riguardano i pin utilizzati per interfacciarsi con sensori e attuatori, la libreria utilizzata per controllare il servomotore e la libreria utilizzata per connettersi alla WLAN.

Il codice per il microcontrollore ESP-8266 è visualizzabile all'indirizzo `https://github.com/Innova-Mente/matrix-controller-embedded/tree/pellonara`, mentre quello per l'ESP-32 è visualizzabile all'indirizzo `https://github.com/Innova-Mente/matrix-controller-embedded/tree/pellonara-esp32`.

3.3 Validazione

Nel Capitolo 2, per validare il funzionamento del framework, sono stati concepiti due casi di studio che insieme toccano tutti gli aspetti del progetto. Il primo consiste nello spegnere e accendere una matrice di LED con un colore casuale utilizzando un pulsante. La configurazione hardware del sistema embedded necessario per la realizzazione di questo caso di studio è la seguente:

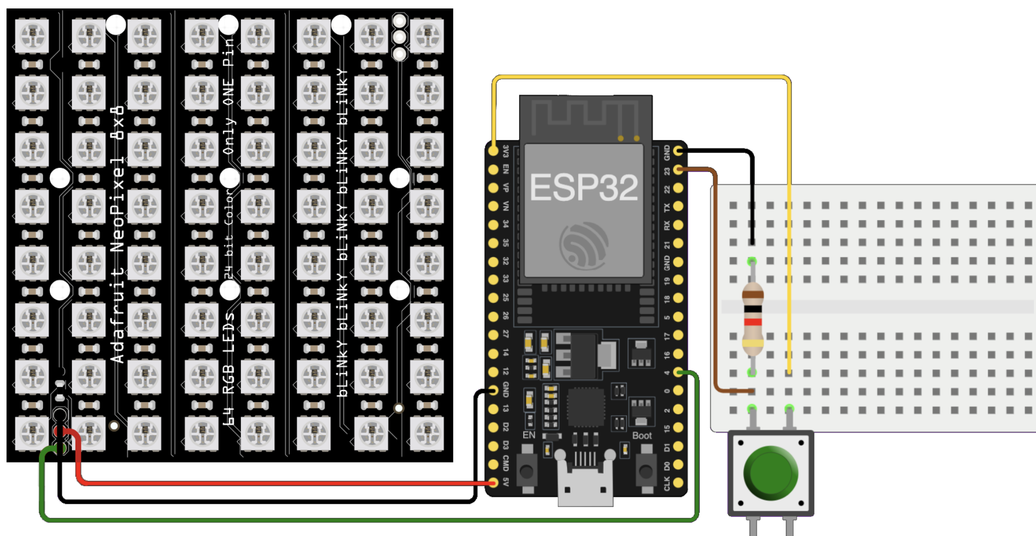

Figura 3.11: Circuito per il controllo di una matrice di LED tramite un pulsante

Il Micromondo progettato per questo caso di studio utilizza una sola variabile per mantenere lo stato della matrice di LED, risultando semplice e intuitivo, come auspicato. Il funzionamento è rapido, riproducibile e privo di complessità inutili. Il Micromondo si presenta come segue:

Figura 3.12: Micromondo per il controllo di una matrice di LED tramite un pulsante

Il framework è stato realizzato nel contesto del progetto Innova-Mente [11], coordinato dal Centro di Ricerche e studi per l'Informatica Applicata alla Didattica (CRIAD) [8] con sede a Cesena. Grazie a Innova-Mente e il CRIAD, è stato possibile presentare il framework e i suoi casi di studio più complessi alla Notte dei Ricercatori 2024 a Cesena. Durante l'evento, è stata allestita una vera e propria mostra sui Micromondi e la loro integrazione con i sistemi embedded resa possibile dal framework. Il pubblico, composto da famiglie e intere classi di bambini, ha potuto interagire con i diversi Micromondi, mostrando grande curiosità ed entusiasmo. In questo contesto, sono anche stati sviluppati e presentati molteplici casi di studio non previsti in fase di analisi e progettazione del framework, tutti eseguiti senza problemi, dimostrando la flessibilità e la versatilità di quest'ultimo.

Uno dei casi di studio presentati era in realtà il secondo caso di studio concepito nel Capitolo 2. Questo consisteva nel mettere a disposizione cinque sistemi embedded, ognuno dotato di una matrice di LED, e permettere agli utenti di interagirci a piacere, creando un loro Micromondo. L'obiettivo era consentire agli utenti di sperimentare liberamente e in modo coinvolgente con i blocchi messi a disposizione dai Micromondi integrati con i sistemi embedded. La configurazione hardware dei sistemi embedded necessari per la realizzazione di questo caso di studio è la seguente:

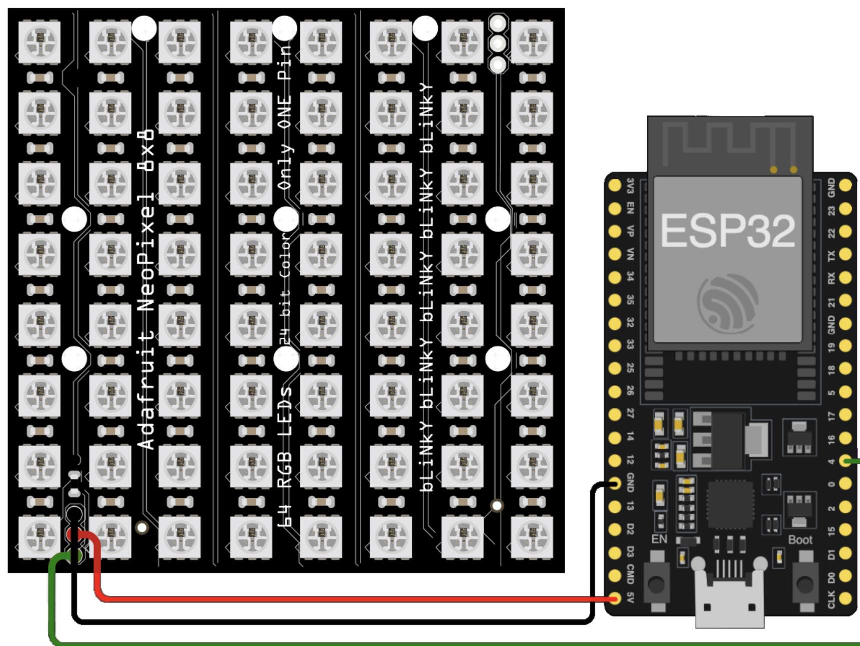

Figura 3.13: Circuito di un sistema embedded qualsiasi del secondo caso di studio concepito

Un esempio di Micromondo che dei ragazzi sono riusciti a creare consiste nel far lampeggiare una matrice di LED casuale per due secondi. In particolare, all'interno di un ciclo infinito, viene eseguito un ciclo *for* che spegne tutte le matrici di LED, poi attende 2 secondi, accende una matrice di LED a caso con colore RGB(200, 200, 200) e infine attende altri 2 secondi. Il Micromondo si presenta così:

Figura 3.14: Esempio di Micromondo creato tramite il secondo caso di studio concepito

Figura 3.15: Il secondo caso di studio come mostrato alla Notte dei Ricercatori

Questo dimostra e conferma il potenziale educativo del progetto, che ha permesso a ragazzi senza esperienze pregressa nella programmazione di apprendere rapidamente concetti come variabili e cicli *for*.

Un altro caso di studio presentato alla Notte dei Ricercatori è un gioco di reazione in cui, dopo un intervallo di tempo casuale, si accende con colore blu la matrice di LED di un sistema embedded principale. Non appena appare la luce blu, due sfidanti devono cercare di attivare per primi il sensore di presenza del proprio sistema embedded usando le mani. Il sistema embedded di ogni giocatore, oltre al sensore di presenza, dispone anche di una matrice di LED che si illumina di verde in caso di vittoria e di rosso in caso di sconfitta. La configurazione hardware del sistema embedded principale è la stessa di quella in Figura 3.13, quella dei sistemi embedded di ciascun giocatore è invece la seguente:

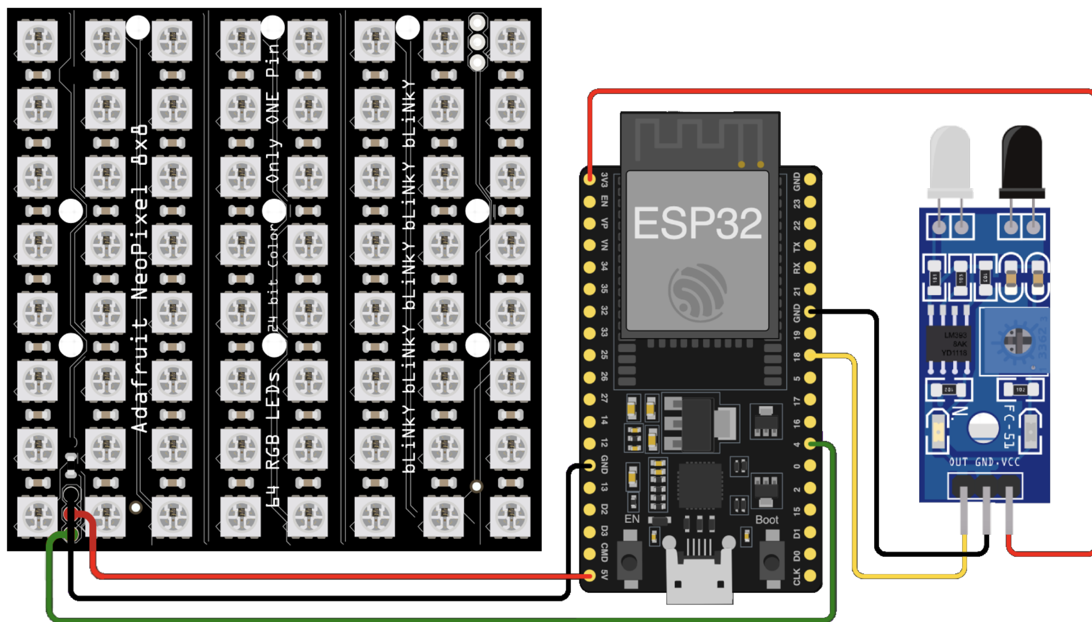

Figura 3.16: Circuito dei sistemi embedded di un giocatore del gioco di reazione

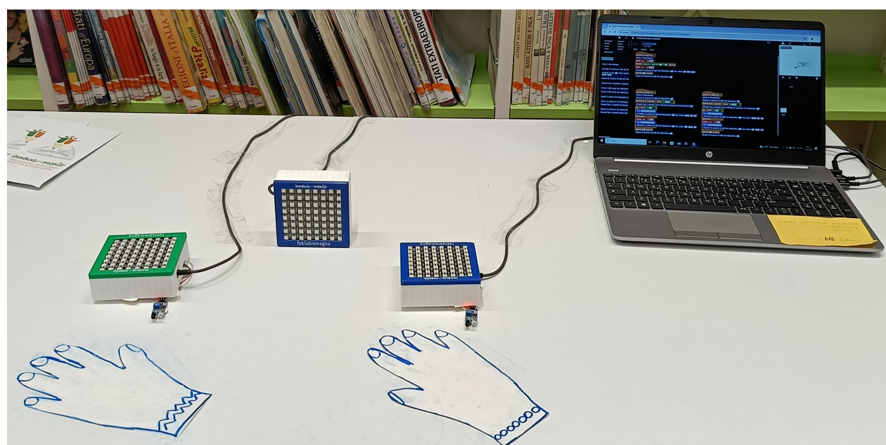

Figura 3.17: Il gioco di reazione come mostrato alla Notte dei Ricercatori

Il Micromondo è strutturato in tre flussi paralleli: il flusso principale, che avvia il gioco accendendo la luce blu, e i due flussi dedicati ai giocatori, che devono reagire il più velocemente possibile. Questi tre flussi condividono una variabile globale che tiene traccia dello stato della partita corrente.

Il flusso principale attende da 1 a 5 secondi e poi avvia la partita accendendo con colore blu la matrice di LED del sistema embedded principale:

Figura 3.18: Flusso principale del Micromondo per il gioco di reazione

Il flusso di ciascun giocatore attende l'inizio della partita e, una volta avviata, verifica se il sensore di presenza viene attivato. Quando il sensore viene attivato, controlla se la partita è già terminata. In caso non lo sia, il giocatore ha reagito per primo alla luce blu e viene eletto vincitore. In caso contrario, il giocatore ha reagito per secondo e quindi perde. Il flusso su Snap! si presenta come segue:

Figura 3.19: Flusso di ciascun giocatore del Micromondo per il gioco di reazione

Questo dimostra che è possibile creare Micromondi che interagiscono in modo articolato con i sistemi embedded senza compromettere l'intuitività e la semplicità d'uso. Inoltre, in questo caso è stata verificata la velocità e l'attendibilità del framework, che doveva reagire in modo rapido e preciso agli input degli utenti.

Ogni caso di studio presentato è stato prima realizzato concretamente, poi ne è stato verificato il funzionamento e, in tutti i casi tranne il primo, è stato anche mostrato al pubblico della Notte dei Ricercatori, dove erano presenti 15 sistemi embedded attivi simultaneamente. Questi sistemi embedded, distribuiti tra 5 Micromondi in esecuzione su 5 elaboratori diversi, erano tutti collegati alla stessa WLAN e utilizzavano lo stesso broker di messaggi. Anche durante situazioni di intenso traffico di messaggi, non sono stati riscontrati malfunzionamenti, dimostrando la robustezza e l'affidabilità del framework stesso. Inoltre, attraverso la panoramica generale offerta dal broker, era possibile monitorare in tempo reale lo stato del framework, e quindi verificare quali sistemi embedded erano connessi e quali messaggi stavano inviando, garantendo così controllo completo.

3.4 Utilizzo

Il framework è stato quindi utilizzato in tutto il suo potenziale durante la Notte dei Ricercatori 2024 a Cesena. Per l'occasione, sono stati progettati degli involucri per contenere i sistemi embedded e nasconderli al pubblico, lasciando visibile solo la matrice di LED collegata ed eventuali cavi necessari per connettere sensori e attuatori. Se aperto, l'involucro appariva così:

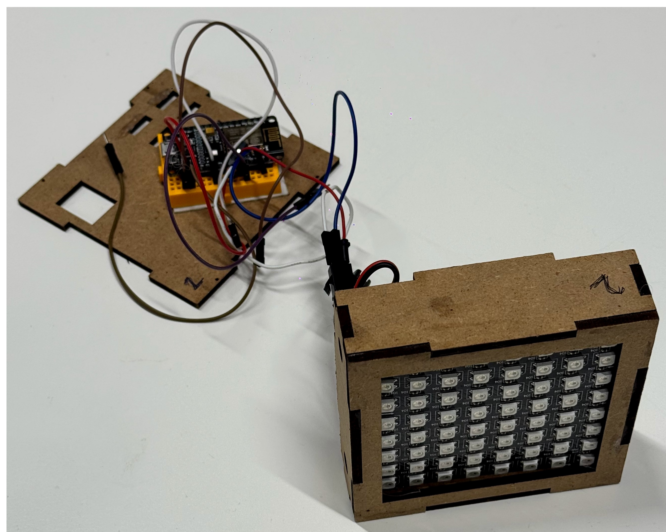

Figura 3.20: Involucro aperto con un sistema embedded e una matrice di LED

Inoltre, per semplificare e velocizzare il processo di configurazione del framework, sono state scritte delle guide per ogni sua parte.

La guida per la configurazione del broker di messaggi è disponibile all'indirizzo: `https://github.com/Innova-Mente/distrib-mw/blob/pellonara/README.md`. La guida per la configurazione e il caricamento del programma apposito sui sistemi embedded ESP-32 è disponibile all'indirizzo: `https://github.com/Innova-Mente/matrix-controller-embedded/blob/pellonara-esp32/README.md`. La guida per la configurazione e il caricamento del programma apposito sui sistemi embedded ESP-8266 è disponibile all'indirizzo: `https://github.com/Innova-Mente/matrix-controller-embedded/blob/pellonara/README.md`. La guida per la configurazione e l'esecuzione dei Micromondi su Snap! è disponibile all'indirizzo: `https://github.com/Innova-Mente/micromondoteca/blob/pellonara/README.md`.

Capitolo 4

Conclusione e Sviluppi Futuri

Conclusioni

Il framework sviluppato ha raggiunto l'obiettivo principale di integrare i sistemi embedded nell'apprendimento del pensiero computazionale, permettendone il controllo in maniera semplice ed efficace tramite blocchi grafici di Snap!. Questi blocchi possono essere inclusi nei Micromondi, fondendo il mondo virtuale con quello reale.

Il progetto ha presentato numerose sfide tecniche, richiedendo competenze in programmazione di sistemi embedded, ingegneria del software, programmazione web e networking. La progettazione dell'architettura del framework è stata una delle sfide principali. È stata concepita tenendo in considerazione la versatilità e l'estendibilità futura, agevolando future aggiunte di nuovi sensori, attuatori, famiglie di sistemi embedded e piattaforme per l'esecuzione di Micromondi.

Un'altra sfida significativa è stata garantire la reattività del framework. Per evitare ritardi, il programma scritto per i sistemi embedded implementa una serie di *best practices* tipiche di quest'ultimi come *interrupt* e l'architettura *event-driven*. La stessa attenzione è stata riservata al broker di messaggi, che durante situazioni di carico elevato, come la Notte dei Ricercatori 2024, ha gestito decine di messaggi al secondo senza ritardi o malfunzionamenti.

La validazione del framework attraverso casi di studio pratici e la Notte dei Ricercatori ha dimostrato la sua efficacia e affidabilità, generando anche un di-

screto entusiasmo tra il pubblico, composto principalmente da giovani studenti. Inoltre, tramite il secondo caso di studio concepito nel Capitolo 2 è stato anche confermato il potenziale educativo del progetto, evidenziando come l'integrazione di Micromondi con sistemi embedded possa rendere l'apprendimento del pensiero computazionale più efficace, stimolante e coinvolgente.

Infine, il framework realizzato può essere anche utilizzato fuori dal contesto dei Micromondi in quanto richiede solo che gli ambienti in cui viene integrato possano connettersi a una WLAN e inviare messaggi tramite WebSocket. Il framework, nato per l'apprendimento, può quindi essere applicato in qualsiasi contesto, grazie alla sua versatilità e flessibilità.

Sviluppi Futuri

Nonostante i risultati ottenuti siano ampiamente soddisfacenti, vi sono diverse aree in cui il framework può essere ulteriormente ampliato e migliorato:

- Espansione della libreria Snap! per pilotare i sistemi embedded: includere ulteriori sensori e attuatori per dare ancora più spazio alla componente fantasiosa e creativa degli utenti.

- Supporto per più famiglie di sistemi embedded: estendere la compatibilità oltre l'ESP-32 e l'ESP-8266 per aiutare la diffusione del framework.

- Supporto multilingua: tradurre in più lingue la libreria Snap! per pilotare i sistemi embedded per permettere una diffusione internazionale del framework, ampliando la comunità di utenti e contributori.

- Collaborazioni con scuole e università: stabilire collaborazioni per validare e migliorare il framework in ambienti educativi reali raccogliendo riscontri preziosi da docenti e studenti.

L'espansione della libreria Snap! e la collaborazione con istituzioni educative saranno fondamentali per continuare a migliorare e adattare il framework alle esigenze di docenti e studenti. Questi sviluppi futuri contribuiranno a rendere il framework ancora più versatile, accessibile e diffuso, promuovendo l'apprendimento del pensiero computazionale e aiutando a formare gli studenti del futuro.

Bibliografia

[1] José David Alvarado Moreno and Luis Garcia. Embedded systems for internet of things (iot) applications: A review study. In *2018 Congreso Internacional de Innovación y Tendencias en Ingeniería (CONIITI)*, pages 1–6, 2018. doi: 10.1109/CONIITI.2018.8587092. URL http://dx.doi.org/10.1109/CONIITI.2018.8587092.

[2] Snap 4 Arduino. Snap 4 arduino: Arduino integration for snap!, 2024. URL https://snap4arduino.rocks/.

[3] Martin F Arlitt and Carey L Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, 1997. doi: 10.1109/90.649565. URL https://doi.org/10.1109/90.649565.

[4] Ethan Brown. *Web Development with Node and Express: Leveraging the JavaScript Stack*. O'Reilly Media, 2019. ISBN 9781492053514. URL https://www.oreilly.com/library/view/web-development-with/9781492053507/.

[5] Li Cheng, Xiaoman Wang, and Albert D. Ritzhaupt. The effects of computational thinking integration in stem on students' learning performance in k-12 education: A meta-analysis. *Journal of Educational Computing Research*, 61(2):416–443, October 2022. ISSN 1541-4140. doi: 10.1177/07356331221114183. URL http://dx.doi.org/10.1177/07356331221114183.

[6] Mozilla Contributors. Websocket. Mozilla Developer Network, 2024. URL https://developer.mozilla.org/en-US/docs/Web/API/WebSocket.

[7] Andrée-Ann Cyr and Nicole D. Anderson. Learning from your mistakes: does it matter if you're out in left foot, i mean field? *Memory*, 26(9):1281–1290, April 2018. ISSN 1464-0686. doi: 10.1080/09658211.2018.1464189. URL http://dx.doi.org/10.1080/09658211.2018.1464189.

[8] Centro di Ricerche e studi per l'Informatica Applicata alla Didattica (CRIAD). Centro di ricerche e studi per l'informatica applicata alla didattica (criad), 2024. URL https://www.criad.org/.

[9] R. Almeida et al. Towards flexible retrieval, integration and analysis of json data sets through fuzzy sets: A case study. *Information*, 12(7):258, 2021. doi: 10.3390/info12070258. URL https://doi.org/10.3390/info12070258.

[10] Dominic Herity. C++ in embedded systems: Myth and reality. *Embedded Systems Programming*, 11(2):48–71, 1998. URL https://forums.parallax.com/discussion/download/86053&d=1318932104.

[11] Innova-Mente. Innova-mente, 2024. URL https://www.innova-mente.org/.

[12] K. Mohaideen Abdul Kadhar and G. Anand. Introduction to the raspberry pi. In *Data Science with Raspberry Pi*, pages 49–78. Apress, 2021. doi: 10.1007/978-1-4842-6825-4_3. URL https://doi.org/10.1007/978-1-4842-6825-4_3.

[13] Edward A Lee. What's ahead for embedded software? *Computer*, 33(9):18–26, 2000. doi: 10.1109/2.868693. URL https://doi.org/10.1109/2.868693.

[14] A Maier, A Sharp, and Y Vagapov. Comparative analysis and practical implementation of the esp32 microcontroller module for the internet of things. In *2017 Internet Technologies and Applications (ITA)*, pages 143–148. IEEE, 2017. doi: 10.1109/ITECHA.2017.8101926. URL https://doi.org/10.1109/ITECHA.2017.8101926.

[15] Profesor N., José Luis-Masabanda, Elizabeth Morales-Urrutia, and José Goitia. *Scratch as a Tool to Promote Computational Thinking in Technological Education*, pages 525–533. 10 2023. ISBN 978-981-99-5413-1. doi:

10.1007/978-981-99-5414-8_48. URL http://dx.doi.org/10.1007/978-981-99-5414-8_48.

[16] Nitin Naik. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7, 2017. doi: 10.1109/SysEng.2017.8088251. URL https://doi.org/10.1109/SysEng.2017.8088251.

[17] Krzysztof Niedźwiedź. Embedded systems programming: A foreground-background ("superloop") architecture, 2023. URL https://intechhouse.com/blog/embedded-systems-programming-a-foreground-background-superloop-architecture/.

[18] Mark Noone and Aidan Mooney. First programming language: Visual or textual?, 2017. URL https://arxiv.org/abs/1710.11557.

[19] K.E. Ogundeyi and C Yinka-Banjo. Websocket in real time application. *Nigerian Journal of Technology*, 38(4):1010, December 2019. ISSN 0331-8443. doi: 10.4314/njt.v38i4.26. URL http://dx.doi.org/10.4314/njt.v38i4.26.

[20] Guilherme M. B. Oliveira, Danielly C. M. Costa, Ricardo J. B. V. M. Cavalcanti, Josiel P. P. Oliveira, Diego R. C. Silva, Marcelo B. Nogueira, and Marconi C. Rodrigues. Comparison between mqtt and websocket protocols for iot applications using esp8266. In *2018 Workshop on Metrology for Industry 4.0 and IoT*. IEEE, April 2018. doi: 10.1109/metroi4.2018.8428348. URL http://dx.doi.org/10.1109/METROI4.2018.8428348.

[21] S. Papert and Massachusetts Institute of Technology. Epistemology & Learning Research Group. *Constructionism: A New Opportunity for Elementary Science Education*. Massachusetts Institute of Technology, Media Laboratory, Epistemology and Learning Group, 1986. URL https://books.google.it/books?id=0N8-HAAACAAJ.

[22] Seymour Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, January 1981. ISBN 0465046274. URL http://www.amazon.fr/exec/obidos/ASIN/0465046274/citeulike04-21.

[23] Pixavier. Mqtt4snap!, a snap! library for using mqtt, 2024. URL https://github.com/pixavier/mqtt4snap.

[24] ProgKids. Snap! vs scratch - differences and advantages, 2024. URL https://www.progkids.com/en/blog/snap-vs-scratch.

[25] Mitchel Resnick. Designing for wide walls, 2017. URL https://mres.medium.com/designing-for-wide-walls-323bdb4e7277.

[26] Snap! Snap! forum, 2024. URL https://forum.snap.berkeley.edu/.

[27] Stefan Tilkov and Steve Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010. doi: 10.1109/MIC.2010.145. URL https://doi.org/10.1109/MIC.2010.145.

[28] WHATWG. Websocket api. WHATWG Living Standard, 2024. URL https://websockets.spec.whatwg.org/#the-websocket-interface.

[29] K. N. Whitley and Alan F. Blackwell. Visual programming: the outlook from academia and industry. In *Papers presented at the seventh workshop on Empirical studies of programmers - ESP '97*, ESP '97, page 180–208. ACM Press, 1997. doi: 10.1145/266399.266415. URL http://dx.doi.org/10.1145/266399.266415.

[30] Jeannette M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, March 2006. ISSN 1557-7317. doi: 10.1145/1118178.1118215. URL http://dx.doi.org/10.1145/1118178.1118215.

Riconoscimenti

Desidero ringraziare il relatore e la correlatrice di questa tesi, Alessandro Ricci e Ylenia Battistini, per l'aiuto e la disponibilità offerti in tutti i passi del percorso di scrittura della tesi. Sono particolarmente grato per aver avuto la possibilità di prendere parte a questo progetto, che ritengo possa avere e avrà un impatto considerevole nella formazione degli studenti del futuro.

Ringrazio inoltre la mia famiglia che mi ha sempre supportato durante questi tre bei anni. Ringrazio quindi mia madre Anna Maria, mio padre Paolo, mia sorella Alessia, la mia fidanzata Annabel e i miei nonni Graziella, Pietro, Adriana e Romolo. Un abbraccio a nonna Adriana, con noi sempre sarai.