ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

STUDIO E REALIZZAZIONE DI UN LABORATORIO REMOTO PER L'IoT

Relatore:
Chiar.mo Prof.
Marco Di Felice

Correlatori:
Dott. Leonardo Montecchiari
Dott. Luca Sciullo

Presentata da:
Nicholas Carlotti

Sessione III
2019/2020

Abstract

Il mondo dell'Internet of Things ha subìto, negli anni, un forte sviluppo. Grazie alla pervasività delle tecnologie coinvolte, sono nate innumerevoli applicazioni sia in campo domestico che a livello industriale. L'alto interesse per la materia ha portato diverse istituzioni educative nel mondo ad introdurre l'insegnamento delle tecnologie IoT ai propri studenti, affiancandole ad essenziali attività pratiche laboratoriali. L'acesso ai laboratori è di vitale importanza per l'insegnamento di questi corsi, è quindi di particolare interesse far si che questi possano essere sfruttati anche in circostanze che ne limitano l'utilizzo, come ad esempio le attuali misure di didattica a distanza adottate in risposta allo scoppio della pandemia di COVID-19. In questa tesi verrà discussa la realizzazione di un sistema che permetta lo svolgimento di attività di laboratorio, in ambito IoT, da remoto. Il sistema consentirà agli studenti di prenotare l'utilizzo di un microcontrollore, situato all'interno del laboratorio del proprio istituto, e di accedervi remotamente. Durante queste sessioni lo studente potrà interagire col dispositivo caricandovi i programmi da lui realizzati ed osservandone il comportamento tramite un flusso video in tempo reale.

Introduzione

IoT (**I**nternet **o**f **T**hings) è un termine che indica la rete di oggetti fisici che sono connessi alla rete internet e scambiano dati tramite essa in maniera costante. Alcuni esempi sono i dispositivi per la domotica, videocamere di sorveglianza o sistemi di monitoraggio remoti. In genere, uno sviluppatore IoT, si occupa di mettere in collegamento il mondo fisico con quello di Internet. Per farlo utilizza dispositivi embedded capaci di interagire col mondo reale, tramite sensori ed attuatori, e di connettersi in rete.

Nell'ultimo decennio vi è stato uno slancio nel settore hobbistico e professionale con la diffuzione di diverse famiglie di questi dispositivi, le quali offrono un insieme eterogeneo di campi d'applicazione. Alcuni esempi sono: *Arduino UNO*[1], un microcontrollore riprogrammabile che permette di realizzare circuiti prototipali; *RaspberryPi*[2], un computer ARM di piccole dimensioni; ed *Intel Edison*[3], una scheda pensata per la realizzazione di dispositivi indossabili.

Un altro settore che ha potuto sfruttare ampiamente queste innovazioni è quello didattico. Le dimensioni contenute ed i costi accessibili hanno permesso a diverse scuole ed università di introdurre corsi di programmazione IoT nei loro piani didattici, compresa l'Università di Bologna.

Da tempo era sorta la necessità, per i docenti del corso di Internet of Things della Laurea Magistrale in Informatica dell'Università di Bologna, di permettere ai propri studenti di condurre attività laboratoriali da remoto, fornendo loro una maggiore flessibilità nel condurre esperimenti.

L'importanza di un sistema simile è stata accentuata dall'avvento della

pandemia di *SARS-CoV-2*: Per contenere i danni dell'emergenza sanitaria, settori come lavori d'ufficio e didattica hanno dovuto riformulare il loro svolgimento in modo da poter essere condotti a distanza, in alcuni casi con impatti piuttosto peggiorativi. Un esempio sono le attività laboratoriali didattiche che, nella maggioranza dei casi, non possono essere svolte in altro modo se non in loco.

Questi avvenimenti hanno reso essenziale l'esistenza di un sistema che permetta agli studenti di interagire remotamente con i dispositivi presenti nel laboratorio del corso di IoT.

In questa tesi verrà discussa la realizzazione di INTRA (**IN**ternet of **T**hings **R**emote l**A**boratory), un sistema che permette a studenti e docenti di organizzare attività laboratoriali in ambito Internet of Things. In particolare INTRA permette di partecipare a sessioni di programmazione di dispositivi embedded remoti, installati in un laboratorio fisico dell'istituto, senza dovervi interagire fisicamente.

Allo stesso tempo il sistema risulta molto utile nel caso in cui un istituto disponga di hardware costoso, difficile da reperire o semplicemente di cui gli studenti non sono dotati. In questo caso INTRA può essere usato per rendere il dispositivo utilizzabile, previa prenotazione, da chiunque ne dovesse avere bisogno.

Dopo una discussione sullo stato dell'arte nel campo dell'Internet of Things, nel capitolo 2 verranno identificati i requisiti del sistema, i casi d'uso e le relative soluzioni architetturali identificate. Successivamente, nel capitolo 3, verranno analizzati i dettagli implementativi più delicati dei nodi principali presenti nel sistema realizzato. Infine, nel capitolo 4, verranno mostrati alcuni risultati derivati dai primi test di collaudo effettuati sul sistema finale. Nel capitolo 5 saranno identificate le possibili evoluzioni del sistema nel capitolo.

Indice

Introduzione **iii**

Descrizione . . . iii

Motivazione . . . iii

1 Stato dell'arte **1**

1.1 Campi di applicazione . . . 4

1.2 IoT nel mondo educational . . . 7

2 Progettazione **9**

2.1 Obiettivi . . . 9

2.1.1 Studenti . . . 9

2.1.2 Professori . . . 10

2.2 Analisi . . . 11

2.2.1 Ruolo server centrale . . . 11

2.2.2 Compilazione sketch . . . 13

2.2.3 Il client . . . 15

2.3 Architettura . . . 15

2.3.1 Overseer . . . 16

2.3.2 Microcontrollore . . . 17

2.3.3 Client . . . 17

2.3.4 Struttura sessione . . . 18

2.3.5 Streaming video . . . 19

3 Implementazione **21**
3.1 Client . . . 21
3.1.1 Funzionalità native . . . 22
3.1.2 Utilizzo Arduino . . . 24
3.1.3 Invio codice eseguibile . . . 25
3.2 Server . . . 26
3.2.1 REST API . . . 26
3.2.2 Svolgimento sessioni . . . 28
3.3 Microcontrollore . . . 33
3.3.1 Dati seriali . . . 33
3.3.2 Upload sketch . . . 34
3.3.3 Streaming video . . . 35

4 Validazione **39**
4.1 Delay sessione . . . 39
4.2 Casi d'uso . . . 41

5 Conclusioni **45**

Conclusioni **45**
5.0.1 Possibili sviluppi . . . 46

Elenco delle figure

1.1 Stima dispositivi IoT connessi negli anni [8]. 2
1.2 Modello di funzionamento MQTT [9] 3
1.3 Xiaomi Mi Smart Band 4, Google Home Mini, Ring Doorbell . 5
1.4 Scheda Arduino e UDOO Neo 6

2.1 Diagramma dei principali componenti del sistema e delle loro interazioni 16

3.1 Architettura applicazione Electron.
Fonte: Kristian Poslek `https://medium.com/developers-writing/building-a-desktop-application-with-electron-204203eeb658` 23
3.2 Handshake WebRTC [47] 36

4.1 Delay video misurato tramite connessione di rete mobile . . . 40
4.2 Creazione di sessioni multiple 42
4.3 Schermata di accesso alle sessioni 43
4.4 Utilizzo della console seriale durante una sessione 43

Capitolo 1

Stato dell'arte

Il termine IoT fu coniato nel 1999 da Dave Ashton [4] durante il suo incarico alla Procter&Gamble. Secondo Ashton, la rete Internet sarebbe dovuta essere sfruttata in modo diverso, facendo in modo che non fossero più esclusivamente gli esseri umani a produrre e consumare i dati, ma che fossero soprattutto le macchine a creare e scambiare dati tra di esse. Ashton propose anche di fare in modo che fossero le macchine stesse a raccogliere i dati dal mondo reale, soprattutto sfruttando la tecnologia RFID. Infatti gli utenti umani hanno conoscenza, capacità e tempo limitati, la capacità delle maccchine di essere in possesso di dati relativi al mondo fisico è limitata da essi.

Sebbene il termine sia stato coniato nel 1999, il primo esempio documentato di un oggetto del mondo reale ("cosa") connesso ad Internet (allora ARPANET) fu il distributore di bibite gassate remoto creato alla Carnegie Mellon University nel 1982 da David Nichols [5] [6]. Il dispositivo consisteva in un normale distributore di bibite al quale erano stati connessi dei sensori, a loro volta collegati al mainframe del dipartimento di informatica. Così facendo era possibile, per gli studenti, cosultare lo stato delle bibite e la loro disponibilità da remoto, attraverso i loro terminali.

Da allora il mondo dell'IoT ha subìto uno sviluppo esponenziale. Come traspare dai dati, al giorno d'oggi vi sono più macchine connesse ad Internet

che persone sulla terra, le stime convergono sulla cifra di circa 33 miliardi [7] di dispositivi connessi alla rete. Più della metà di questi dispositivi sono IoT (Figura 1.1). Infatti lo sviluppo di standard di comunicazione, miniaturizzazione di dispositivi e abbassamento dei costi hanno permesso di rendere i dispositivi IoT sempre più pervasivi ed accessibili al pubblico.

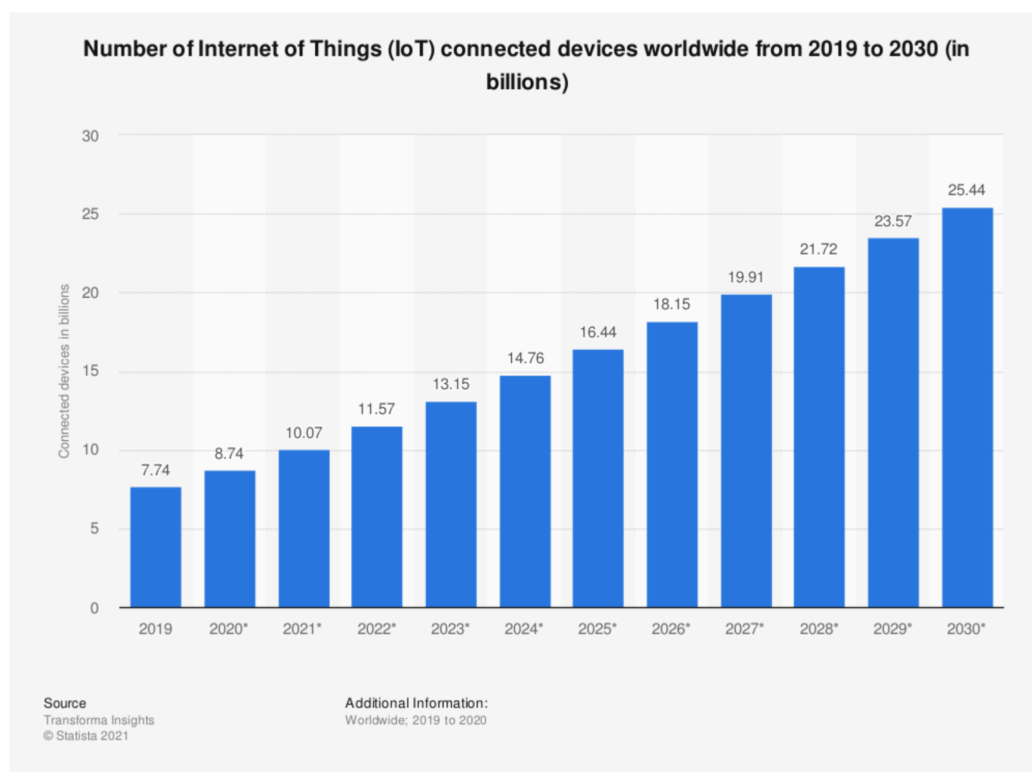

Figura 1.1: Stima dispositivi IoT connessi negli anni [8].

Tecnologie di rete come BLE, ZigBee e RFID hanno permesso di applicare l'IoT a contesti diversificati e nuovi, in alcuni casi portando alla creazione di Personal Area Network (PAN) composte da numerosi dispositivi che comunicano tra loro e direttamente col cloud.

Inoltre vi è stato un notevole progresso nel campo software grazie alla realizzazione di nuovi protocolli di comunicazione, pensati per la diffusione di dati da parte di dispositivi embedded con capacità ed autonomia limita-

te: un esempio di questi protocolli è Message Queuing Telemetry Transport (MQTT) [9]. MQTT è un protoocllo basato sul paradigma publish/subscribe, ovvero un sistema in cui vi sono uno o più creatori di dati (*publishers*) che generano messaggi dotati di una determinata etichetta. Questi messaggi vengono inviati ad un nodo speciale detto *broker*. I nodi interessati ai messaggi, i *subscribers*, dialogano direttamente col broker al quale specificano le categorie di dati che desiderano ottenere. Il ruolo del broker è quindi quello di instradare i messaggi generati dai publisher ai subscriber che ne hanno espresso interesse.

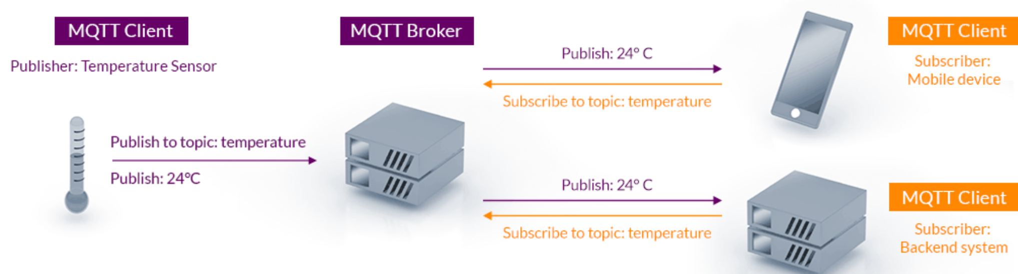

Figura 1.2: Modello di funzionamento MQTT [9]

.

La proliferazione di sistemi IoT adottati dal pubblico ha introdotto la necessità, da parte delle aziende produttrici, di monitorare, aggiornare e rendere sicuri i dispositivi da loro creati. Per soddisfare queste esigenze sono stati realizzati pacchetti software integrati (Azure IoT Hub, Cloud IoT Core e AWS IoT Device Management) che consentono agli sviluppatori di sistemi IoT di comunicare in modo sicuro, gestire i dati generati e distribuire aggiornamenti ai propri dispositivi.

Come conseguenza di questi progetti vi è stato un aumento del numero di aziende e progetti dedicati al mondo IoT. Un problema identificato recentemente riguarda l'interoperabilità dei sistemi IoT prodotti dalle diverse aziende [10]. Infatti, vista la mancanza di convenzioni unificate per la consu-

mazione dei dati e dei servizi offerti dai dispositivi prodotti, si sono originati ostacoli sia per gli utenti che per gli sviluppatori. La mancanza di un protocollo comune impedisce a produttori terzi di creare soluzioni che sfruttino o si integrino con prodotti di altre aziende e questo porta a diversi svantaggi che si ripercuotono sugli utenti finali:

- L'utilizzo di protocolli proprietari da parte delle aziende impedisce l'acquisto di prodotti di terze parti per la sostituzione in caso di guasto.
- La chiusura dei protocolli danneggia l'innovazione e limita la varietà di prodotti disponibili sul mercato.
- L'assenza di interoperabilità non consente agli utenti di utilizzare prodotti diversificati, impedendo di realizzare sistemi composti da dispositivi che non appartengano alla stessa casa produttrice.

Diversi tentativi per correggere il problema sono stati proposti negli anni, il più celebre è la Web of Things (WoT) [11] proposta dal W3C. Questa soluzione aspira ad astrarre le diversità di protocolli e rappresentazione dei dati tramite l'utilizzo di un vocabolario comune ed un formato standard per le informazioni.

1.1 Campi di applicazione

L'evoluzione delle tecnologie sopra citate ha permesso l'utilizzo sempre più pervasivo di sistemi IoT, con un particolare successo per i prodotti ideati per i piccoli consumatori.

A livello del consumatore, negli ultimi anni vi è stato un forte sviluppo dell'ecosistema di prodotti disponibili. Da semplici accessori personali, come smartwatch, fitness band e bilance pesapersona a sistemi come home assistant, termostati e monitor per la qualità dell'aria.

Figura 1.3: Xiaomi Mi Smart Band 4, Google Home Mini, Ring Doorbell

Grazie al loro basso costo ed alla possibilità di controllo e gestione tramite smartphone, oggetto ormai posseduto dalla quasi totalità della popolazione [12], questi dispositivi sono stati via via adottati da sempre più persone e famiglie.

Un altro fattore che ha permesso una larga adozione di questi sistemi è la possibilità, offerta da alcune aziende, di controllarli tramite i più diffusi assistenti personali, come Amazon Echo Dot e Google Home.

Un altro ambito di grande successo per l'IoT è il campo industriale. Nella produzione industriale vi è la necessità di controllare gli sprechi, ottimizzare i processi ed applicare una manutenzione attiva. Grazie alle dimensioni contenute ed alla varietà di tecnologie le possibilità sono numerose.

Negli anni sono state proposte soluzioni differenti, come sistemi IoT per il controllo della filiera di produzione di olio d'oliva [13] e sistemi per la manutenzione predittiva [14].

Oltre alle proposte vi sono numerosi esempi di sistemi IoT realmente usati, alcuni di questi sono il sistema cloud di monitoring impiegato da Volkswagen [15] o dispositivi per la sorveglianza dello stato di salute di coltivazioni [16].

Anche in campo medico è stato possibile trovare applicazioni per dispositivi IoT. Infatti, utilizzando dispositivi già commercialmente disponibili, o

realizzandone esemplari ad-hoc, sono stati proposti sistemi di sorveglianza dello stato di salute dei pazienti [17] [18] [19]. Questi sistemi permettono di registrare con precisione i parametri vitali degli utenti e di trasmetterli in tempo reale ai relativi medici. Il vantaggio consiste in un monitoring più attento ed allo snellimento delle procedure mediche, consentendo ai pazienti di contattare i medici per ottenere un loro parere senza dovervisi recare fisicamente.

La conseguenza è una maggiore tutela della salute della popolazione ed uno screening più accurato, che viene potenziato dalla maggiore disponibilità di dati e dall'utilizzo di algoritmi predittivi per lo stato di salute.

Un fattore rilevante per lo studio di nuove soluzioni IoT è stata l'introduzione di dispositivi e schede prototipali a basso costo. Infatti sia il settore hobbistico che quello professionale hanno giovato dalla diffusione di microcontrollori come Arduino UNO [1], Teensy 4.0 [20] e AdaFruit Flora [21] e dai diversi Single Board Computer (SBC) presenti sul mercato, ad esempio RaspberryPi [2] e UDOO Neo [22]. Questi prodotti permettono a programmatori ed ingegneri di interfacciarsi con diversi dispositivi elettronici ad un costo contenuto. Le conseguenze più notevoli sono osservabili in campo hobbistico grazie al grande numero di progetti DIY consultabili online [23] [24].

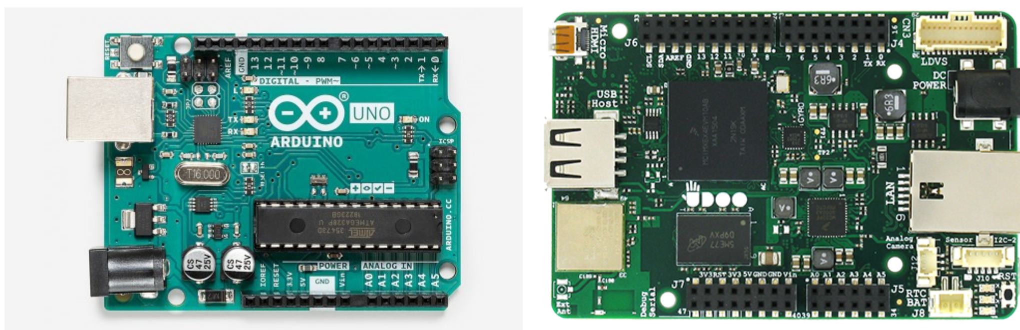

Figura 1.4: Scheda Arduino e UDOO Neo

1.2 IoT nel mondo educational

Quando si parla di IoT nel mondo dell'educazione vanno distinti due utilizzi: il primo è l'uso di dispositivi IoT per permettere o complementare attività didattiche, il secondo riguarda l'effettivo insegnamento della materia IoT.

Del primo caso, in letteratura, vi sono diversi esempi, il carattere generale di questo tipo di impiego è però incentrato su quello laboratoriale. Sono comuni realizzazioni di esperimenti laboratoriali che puntano a migliorare l'apprendimento di un determinato argomento, fornendo un mezzo interattivo e controllato per la sperimentazione dei princìpi da apprendere. Alcuni esempi sono il controllo di un motore elettrico, al quale sono collegati diversi sensori per la misurazione di alcuni parametri, tramite una dashboard interattiva [25]; la regolazione ed il monitoraggio di un sistema di ventilazione filtrante [26]; e sistemi di controllo per esperimenti di diversa natura all'interno di un laboratorio [27].

Ma l'IoT nel mondo educational gioca anche un ruolo da protagonista. Infatti la proliferazione di sistemi IoT in diversi ambiti della vita quotidiana e lavorativa ha reso la materia dell'Internet of Things necessaria nel curricolum degli studenti di informatica. Essendo un tema fortemente incentrato sulle tecnologie, l'aspetto sperimentale e laboratoriale è essenziale allo studio della materia e, generalmente, nei corsi universitari sono previste attività pratiche in laboratorio. L'utilizzo di laboratori fisici comporta alcuni limiti:

- La finestra temporale nella quale è possibile usufruire dei dispositivi è limitata dagli orari di apertura del laboratorio.

- Manipolando e riconfigurando direttamente i dispositivi utilizzati negli esperimenti è possibile danneggiarli

- Vi è la necessità di accedere fisicamente ai laboratori in modo da utilizzare i microcontrollori

- Relativamente al punto precedente, nel caso di progetti di gruppo è necessario che gli studenti si organizzino in modo che ogni componente sia presente durante la sessione di laboratorio.

Per aggirare questi limiti è necessario far si che i microcontrollori siano programmabili senza necessariamente avervi accesso fisicamente, a tal fine esistono due approcci: simulare un microcontrollore e i dispositivi ad esso connessi (laboratorio virutale) o rendere accessibile l'hardware presente in laboratorio da remoto (laboratorio remoto).

Per quanto riguarda la simulazione, la soluzione più valida per l'insegnamento dell'IoT sembra essere IoTIFY Virtual Lab [28], un sistema che permette la programmazione di un RaspberryPi emulato tramite codice Python e la realizzazione di semplici circuiti elettronici ad esso collegati. Un sistema virtuale presenta diversi vantaggi come costo, flessibilità e la disponibilità dei dispositivi. Il limite di questi sistemi è rappresentato dalla fedeltà della simulazione.

Un sistema remoto invece è un buon compromesso tra costi, accessibilità e fedeltà. In letteratura non vi sono molti esempi di questi sitemi, i più lampanti sono i laboratori VREL [29], sviluppati dal consorzio iot-open [30], e il sistema proposto da Fernández-Pacheco et al. [31] Nell'articolo gli autori propongono un sistema di programmazione remota di schede Arduino [32], utilizzando un RaspberryPi[2] come mezzo per mettere in comunicazione la scheda con la rete internet. Non vengnono però forniti dettagli su come realizzare la condivisione delle risorse tra gli utenti del sistema.

Capitolo 2

Progettazione

L'obiettivo di questa tesi è descrivere e realizzare un sistema per lo svolgimento di esperienze laboratoriali (**sessioni**) in remoto. Le sessioni prevedono la programmazione di un dispositivo IoT in tempo reale, con la possibilità di visualizzarne lo stato fisico e consultarne eventuali messaggi testuali. In questo capitolo verranno discussi i requisiti del sistema da realizzare, le scelte architetturali prese e i motivi per cui sono state fatte. I requisiti derivano dalle necessità didattiche espresse dai docenti del corso Internet of Things dell'Università di Bologna.

2.1 Obiettivi

Nel sistema analizzato sono presenti due utenti principali: studenti e professori. Per meglio capire gli obiettivi del sistema da realizzare, verranno elencati i bisogni e i casi d'uso, dal punto di vista di entrambi gli attori principali, che il software finale soddisferà.

2.1.1 Studenti

La prima interazione che uno studente ha col sistema è la (RF1) consultazione di una lista di sessioni disponibili e prenotabili. (RF2) Per ogni sessione è indicato il tipo di microcontrollore, l'inizio e la fine della sessione.

(RF3) Lo studente può eventualmente prenotare una di queste sessioni ed (RF4) annullare la prenotazione in caso non potesse più parteciparvi.

(RF5) Lo studente deve potersi connettere ad una sessione e di conseguenza accedere ad un microcontrollore remotamente da un computer personale. (RF6) All'inizio di una sessione, lo studente deve trovare il microcontrollore in uno stato pulito, ovvero senza che vi sia alcun programma in esecuzione. (RF7) Durante una sessione lo studente scrive sketch di codice destinati ad essere eseguiti sul microcontrollore assegnato. (RF8) Inoltre deve essere possibile ispezionare lo stato fisico del dispositivo tramite un feed video e (RF9) consultarne l'output seriale. Lo studente deve anche (RF10) poter consultare il pinout dello specifico microcontrollore con cui sta interagendo. Deve essere possibile (RF11) includere librerie aggiuntive negli sketch creati.

Una sessione può terminare (RF12) spontaneamente allo scadere del tempo o può (RF13) essere conclusa manualmente dallo studente stesso. (RF14) Dopo il termine di una sessione lo studente deve poter consulare il codice scritto durante il suo svolgimento, e (RF15) deve poter accedere agli sketch creati in sessioni precedenti.

Infine, (RF16) uno studente deve poter segnalare guasti ed (RF17) essere avvisato nel caso in cui esistano segnalazioni di guasti relativi al microcontrollore al quale sta accedendo.

2.1.2 Professori

In questa sezione verranno descritti i requisiti dal punto di vista dei professori, tutor o degil assistenti al corso. Per semplicità verrà usato il termine "admin" per identificare questa categoria di utente.

(RF18) Un admin può creare delle sessioni ed assegnarle a determinati microcontrollori, successivamente può (RF19) eliminarle (RF20) o cancellarne la prenotazione, per compiere queste azioni (RF21) un amministratore deve poter ispezionare lo stato di prenotazione degli slot da lui creati. Inoltre (RF22) un amministratore può assegnare uno slot ad uno dei suoi studenti.

(RF23) Un admin può ispezionare lo stato del pinout di un microcontrollore durante una sessione passata e (RF24) consultare la lista di segnalazioni di guasti relativi ai microcontrollori da lui gestiti.

(RF25) Un amministratore può inserire un nuovo microcontrollore nel sistema specificandone (RF26) il tipo ed (RF27) il pinout. Queste informazioni possono essere (RF28) modificate in un secondo momento ed (RF29) il microcontrollore eliminato. Chiaramente queste modifiche non possono essere consentite (o non devono avere alcun impatto) se effettuate durante una sessione in cui uno studente sta interagendo col microcontrollore.

Al fine di identificare possibili anomalie, un amministratore può (RF30) sospendere tutte le prenotazioni relative ad uno specifico microcontrollore, (RF31) visualizzarne il feed video (RF32) e quello seriale, (RF33) esegurvi sketch ed infine (RF34) permettere nuovamente di effettuare prenotazioni e/o sessioni.

2.2 Analisi

Procediamo ora con l'analisi dei requisiti espressi nella sezione precedente. Il sistema sarà composto da tre nodi: un client utilizzato dagli utenti, un server con funzione di repository centrale per i dati inerenti a microcontrollori ed utenti, e infine i microcontrollori stessi. Sono da definire le dinamiche di interazione tra questi nodi e l'architettura generale da realizzare. Per comprendere le ragioni che hanno portato all'architettura finale verranno analizzati alcuni aspetti cruciali del sistema.

2.2.1 Ruolo server centrale

In prima analisi si può osservare che il server avrà il compito di memorizzare la lista di microcontrollori disponibili, le sessioni, i dati sui guasti e gli account degli utenti.

Non è chiaro invece come il server ed i microcontorllori vadano messi in comunicazione. Durante il processo di analisi sono state individuate due pos-

sibili soluzioni: collegare i microcontrollori fisicamente al server o utilizzare una connessione di rete.

Un'architettura centralizzata, ovvero una dove tutti i microcontrollori del sistema sono collegati fisicamente (ad esempio tramite un cavo USB) al server, potrebbe rivelarsi particolarmente difficile da gestire: Si consideri il fatto che per ogni microcontrollore è necessaria una webcam che lo riprenda in modo da generare il feed video, una breadboard e i dispositivi elettronici (led, servo-motori, sensori) collegati ad essa. Questo implicherebbe il collegare diversi dispositivi al server creando ovvi problemi di gestione dello spazio.

La soluzione migliore sembra quindi essere quella distribuita. Il microcontrollore è un dispositivo molto semplice e con una memoria flash limitata, si rivelerebbe poco pratico utilizzarlo come dispositivo standalone che riceva del codice tramite una connessione di rete e lo esegua; questo richiederebbe la scrittura di un firmware sofisticato con l'effetto di limitare le capacità hardware messe a disposizione degli studenti. Un'idea promettente è quella di collegare il microcontrollore ad un secondo dispositivo embedded, preferibilmente più avanzato, che abbia il compito di renderlo accessibile da remoto. Anticipando una scelta implementativa sarà giustificata in seguito, d'ora in avanti questo dispositivo verrà chiamato **RaspberryPi** (alternativamente **Raspberry** o semplicemente **pi**).

Dopo aver definito il grado di disaccoppiamento fisico tra server e microcontrollore è necessario analizzare quello logico. In particolare, si possono fare diverse considerazioni sul ruolo che gioca il server nella comunicazione studente-microcontrollore. Ci sono due potenziali approcci:

- Rendere il server un'entità che permette di fare discovery di microcontrollori e lasciare che gli utenti vi si connettano autonomamente. Tramite un'API esposta dal server, i Raspberry sarebbero in grado di eseguire gli opportuni controlli (autenticazione, validazione della sessione) e permettere agli studenti di programmare le schede.

- Il server fa da tramite tra utenti e microcontrollori in qualunque fase delle loro interazioni.

I vantaggi del primo approccio consistono solamente nella scalabilità del sistema: delegando il compito di gestire una sessione ai singoli microcontrollori è possibile alleviare il carico sul server centrale e permettere lo svolgimento di un maggior numero di sessioni in contemporanea. Il vantaggio però viene a discapito di una minore complessità di implementazione, inoltre questo vantaggio non è particolarmente rilevante per l'ambito d'uso di questo progetto: difficilmente ci saranno così tanti microcontrollori da permettere un numero elevato di sessioni contemporanee.

Il secondo approccio invece è il più adatto al caso d'uso in questione ed è il più semplice da realizzare. Il server farà da intermediario tra microcontrollore e utente e ne medierà lo scambio di ogni messaggio. Il vantaggio consiste in una maggiore facilità nell'attuare controlli e validazione sulle interazioni utente-dispositivo.

2.2.2 Compilazione sketch

Il microcontrollore deve ricevere un programma compilato da poter eseguire. La compilazione è un aspetto delicato del sistema e va considerato attentamente a quale nodo affidarne il compito. Una prima soluzione potrebbe essere quella di assegnare il compito al RaspberryPi, d'altronde è il nodo direttamente connesso al microcontrollore e il primo a conoscerne le specifiche (la più importante di queste è l'architettura della scheda). Facendo così nascerebbero però alcuni problemi:

- Ritardi nel ciclo code-compile-run svolto durante una sessione.
- Difficoltà nell'inclusione di librerie esterne.

Il problema del ritardo consiste nel fatto che la compilazione avverrebbe su una macchina remota e con potenza di calcolo decisamente inferiore rispetto a quella di sviluppo, di conseguenza i risultati della compilazione

verrebbero presentati allo studente con un certo ritardo. Considerando che, durante la fase di sviluppo di un programma accade spesso di procedere alla compilazione, l'introduzione di un ritardo ad ogni compilazione porterebbe ad un accumulo consistente di tempo in cui lo studente rimane in attesa. In più vi è da considerare un'aspetto più pratico: si pensi al caso in cui il programma scritto contenga un errore banale (ad esempio un punto e virgola ";" mancante), dover attendere diversi secondi per la notifica da parte del compilatore può risultare particolarmente frustrante.

Per quanto riguarda le librerie può succedere che uno studente, per sua necessità, decida di utilizzare una libreria esterna. Se si optasse per la compilazione a bordo del RaspberryPi sarebbe necessario fare in modo che questo ottenga le librerie utilizzate dal programma e, trattandosi di librerie arbitrarie, l'unica opzione possibile è quella di trasmetterle direttamente ad esso. Questo si rivelerebbe poco pratico nel caso di librerie di dimensioni non banali.

Come secondo tentativo si potrebbe considerare l'overseer come il responsabile della compilazione. In questo caso però si porrebbero non solo gli stessi problemi della compilazione su RaspberryPi, ma anche un possibile problema di performance derivato dal fatto che un singolo nodo, ovvero il server, si occuperebbe della compilazione dei programmi di tutte le sessioni attive in un determinato momento.

La scelta che è stata presa è dunque quella di compilare sulla macchina dello studente. In questo modo si hanno diversi vantaggi:

- Compilazione distribuita sui singoli nodi, non creando un unico punto di sovraccarico
- Possibilità di usare qualunque libreria, il microcontrollore ed il RaspberryPi non si interessano del processo di compilazione
- Feedback immediato nel caso di possibili errori di compilazione.

2.2.3 Il client

L'obiettivo di INTRA è quello di proporre un ambiente user friendly e quanto più astratto possibile per lo sviluppo di applicazioni IoT da remoto. Per fare ciò va posta particolare attenzione al client e a come l'utente finale interagirà col sistema. Considerando anche le sezioni precedenti, i compiti del client possono essere riassunti nei seguenti punti:

- Acquisire il codice scritto dall'utente
- Gestire la compilazione del programma ed inviare un eseguibile al microcontrollore
- Ricevere e visualizzare un feedback video e seriale in tempo reale dal dispositivo.

Da queste considerazioni emerge che il client sarà necessariamente un'applicazione nativa eseguita sulla macchina dell'utente che dovrà, in qualche modo, consentire all'utente di scrivere codice, raccoglierlo, convertirlo in un programma eseguibile dal microcontrollore e mostrare allo studente video e messaggi di testo in tempo reale.

Inoltre servirà un modo per interagire con la parte più gestionale del sistema (prenotazioni, guasti, interfaccia amministratori). Includere queste funzionalità nel client permetterebbe di ottenere una soluzione più integrata.

2.3 Architettura

In questa sezione verrà descritta la struttura generale del sistema finale. A seguito dell'analisi condotta precedentemente, è possibile individuare 3 elementi architetturali fondamentali del sistema:

- Client
- Overseer
- Microcontrollore

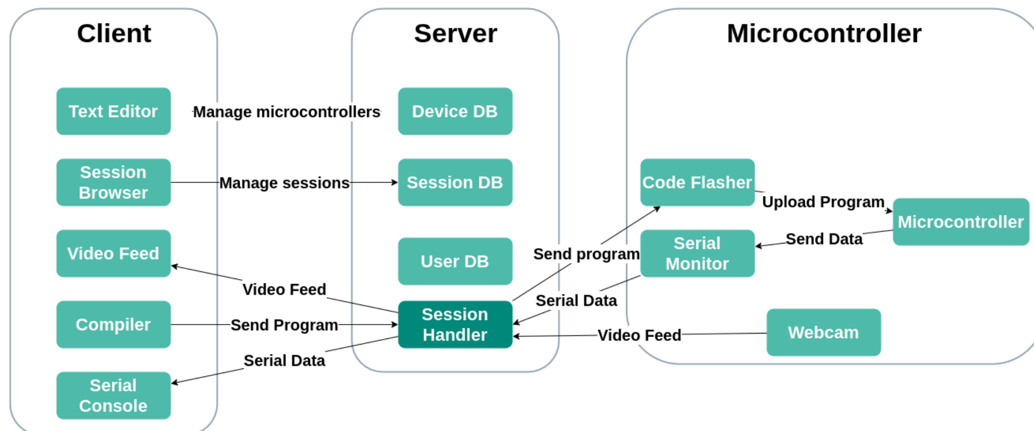

Figura 2.1: Diagramma dei principali componenti del sistema e delle loro interazioni

2.3.1 Overseer

Il server avrà il compito di gestire dati relativi a client, microcontrollori, sessioni e segnalazioni, questo consisterà nel permettere agli utenti, professori e studenti, di eseguire operazioni CRUD su questi dati. È stato quindi deciso di realizzare il server attraverso un servizio web.

Come accennato in fase di analisi, il server avrà il compito di fare da mediatore tra studenti e microcontrollori durante lo svolgimento di una sessione. Qualora una delle due parti volesse inviare un messaggio all'altra, questa invierà il messaggio al server centrale, il quale si occuperà di indirizzarlo al destinatario. Oltre a permettere la comunicazione tra questi due attori, il server dovrà mediare il ciclo di vita della sessione, eseguendo tutte le operazioni necessarie al momento della sua terminazione. Il server dovrà essere dotato di un componente che si occuperà di mettere in atto la terminazione di una sessione nel momento in cui giunga la sua fine (o qualora lo studente decidesse di terminarla manualmente).

2.3.2 Microcontrollore

Come anticipato nella sezione 2.2.1, il microcontrollore sarà collegato ad un SBC (single board computer) che gli permetta di comunicare col mondo esterno. Un SBC si è rivelata un'ottima soluzione: un sistema embedded capace di dialogare col microcontrollore e di comunicare attraverso la rete internet permette di rendere la scheda operabile remotamente a costi contenuti.

Durante lo svolgimento di una sessione l'SBC deve ricevere comandi in tempo reale dal client (esecuzione di un programma, richiesta del flusso video), ed anche trasmettere messaggi, come ad esempio l'output seriale del microcontrollore. Per permettere la creazione di un flusso video, l'SBC dovrà essere dotato di un dispositivo di acquisizione.

Infine l'SBC caricherà i programmi inviati dallo studente sulla scheda, per farlo sfrutterà l'interfaccia seriale di cui sarà fornito (caratteristica molto diffusa su questo tipo di dispositivo). Inoltre si occuperà di gestire la fase di reset del microcontrollore, compilando e caricandovi un programma vuoto.

Per quanto riguarda la parte prettamente elettronica del microcontrollore, ovvero l'insieme di sensori ed attuatori a questo azionati, i collegamenti saranno decisi dal professore al momento della creazione del laboratorio (o della sua riconfigurazione). Lo studente potrà solo consultare la lista di porte e dispositivi ad esse collegati documentata dal professore, senza poter compiere modifiche.

2.3.3 Client

Il client deve permettere lo svolgimento di operazioni gestionali (creazione e prenotazione di sessioni, gestione microcontrollori) e delle sessioni di programmazione. Dovrà inoltre consentire la visualizzazione del flusso video trasmesso dall'SBC e trattare il codice scritto dallo studente, crearne un eseguibile ed inviarlo al microcontrollore. La soluzione più integrata consisterebbe nel creare un software che combini la parte gestionale del sistema con

un IDE, fornito di editor di testo, una console per visualizzare l'output seriale del dispositivo ed un compilatore. Queste funzionalità vengono già fornite da un software per la programmazione di microcontrollori, l'IDE Arduino [32].

ArduinoIDE permette la creazione di sketch di codice, l'inclusione di librerie esterne, la compilazione ed il caricamento dei programmi sul microcontrollore. È inoltre possibile utilizzarlo in modalità portable, consentendo di sfruttarne gli strumenti anche in un software esterno che desideri incorporarne le funzionalità.

La soluzione finale individuata per il client è dunque un software che esponga un'interfaccia per la parte gestionale del sistema e che, durante lo svolgimento di una sessione, espone un'interfaccia grafica che consente di interagire col microcontrollore e utilizzare una versione autocontenuta dell'IDE Arduino per la scrittura degli sketch di codice.

Utilizzando l'IDE Arduino come mezzo per la scrittura e la compilazione di sketch, si trae il vantaggio di un vasto ecosistema di librerie e schede supportate ufficialmente dall'IDE o dalla community. Questo permetterà ai professori di costruire un laboratorio con un discreto grado di flessibilità per quanto riguarda la scelta dell'hardware da impiegare.

2.3.4 Struttura sessione

La sessione prevede una lunga serie di interazioni tra studenti e microcontrollori, per ognuna delle azioni permesse vanno identificati i dati e i passi da intraprendere a seguito della loro ricezione. Alcuni esempi sono la richiesta, da parte dello studente, di eseguire un programma sul microcontrollore e l'invio di dei dati inviati tramite la porta seriale di questo. Il protocollo usato durante una sessione è riassunto nella tabella 2.1. La lista non è esaustiva in quanto, per alcune necessità implementative, è stato inevitabile espandere il protocollo, senza però modificare o rimuovere ciò che è qui elencato.

Azione	**Mittente**	**Messaggio**
Richiesta flusso video	Studente	`start_video_data`
Invio programma	Studente	`run_code`
Invio dati seriali	Microcontrollore	`send_serial_data`
Stop flusso video	Studente	`stop_video_data`
Termina sessione	Studente	`stop_session`
Segnala stato scheda	Microcontrollore	`device_status`

Tabella 2.1: Protocollo utilizzato durante le sessioni

2.3.5 Streaming video

Una scelta che ha determinato la natura di altri elementi del sistema, in particolare client e server, è lo streaming video. Il client infatti deve ricevere un flusso video in tempo reale dello stato del microcontrollore durante lo svolgimento di una sessione. Le caratteristiche di questo flusso sono bassa latenza ed una qualità sufficiente a distinguere i particolari del circuito elettronico visualizzato. La scelta è ricaduta su una tecnologia peer to peer chiamata WebRTC [33]. Questa tecnologia permette di stabilire una connessione diretta, quando possibile, tra due web browser in modo da scambiare video, audio o messaggi arbitrari a bassa latenza. Si tratta quindi di una soluzione pensata, ma non limitata, al mondo del web.

Capitolo 3

Implementazione

Prima di parlare dei dettagli implementativi di ogni macro-componente del sistema, verrà data una panoramica della soluzione finale e delle tecnologie coinvolte:

L'overseer è un'applicazione web che espone una REST API per la parte gestionale del sistema (creazione sessioni, microcontrollori), ed un server websocket [34] per permettere la comunicazione in tempo reale tra studente e microcontrollore durante lo svolgimento delle sessioni.

Il client è implementato da un'applicazione Electron che racchiude una versione **portable** dell'IDE Arduino e che comunica col server attraverso la REST API ed il protocollo websocket.

Il microcontrollore invece è formato da una scheda (generalmente una scheda Arduino-compatibile) collegata ad un RaspberryPi tramite cavo USB. Il raspberry mantiene una connessione costante col server ed attende istruzioni da quest'ultimo. Il dispositivo di acquisizione video scelto è il Raspberry camera module[35].

3.1 Client

Come già accennato, il client consiste in un'applicazione Electron, la cui interfaccia è stata realizzata con *VueJS*[36]. La scelta di Electron è stata

guidata dai seguenti motivi:

- L'ambiente web-like di Electron permette l'utilizzo di implementazioni native di protocolli come WebRTC
- La possibilità di realizzare applicazioni desktop multipiattaforma astraendone i particolari.
- Possibilità di raggiungere rapidamente una versione prototipale

Il client permette allo studente di svolgere tutte le attività previste dal sistema. Insieme ad esso viene distribuita anche una versione autocontenuta dell'IDE Arduino, che verrà utilizzata sia in modo esplicito dall'utente che implicitamente dal client qualora a questo venisse richiesto di compilare un programma.

3.1.1 Funzionalità native

La struttura dell'applicazione è influenzata dall'utilizzo di Electron e, in maggior parte, da VueJs. Un'applicazione Electron segue uno schema generale che è illustrato nella figura.

L'aspetto fondamentale è la divisione del progetto in due parti: il main process ed i renderer processes. Il primo è la parte dell'applicazione che è autorizzata ad interagire col sistema operativo, potendo compiere operazioni come la lettura di file o l'esecuzione di processi esterni. I rendered processes invece si occupano di implementare le funzionalità dell'applicazione, agli occhi dello sviluppatore questa parte corrisponde a tutti gli effetti ad un web browser.

Per permettere ai processi main e render di comunicare si sfrutta la interprocess comunication (*IPC*), meccanismo che costituisce il cuore di un'applicazione Electron. Al fine di facilitare l'accesso alle funzionalità offerte dal main process, è stato creato un modulo che espone un'API semplificata per eseguire operazioni come la lettura di file o esplorare il contenuto di

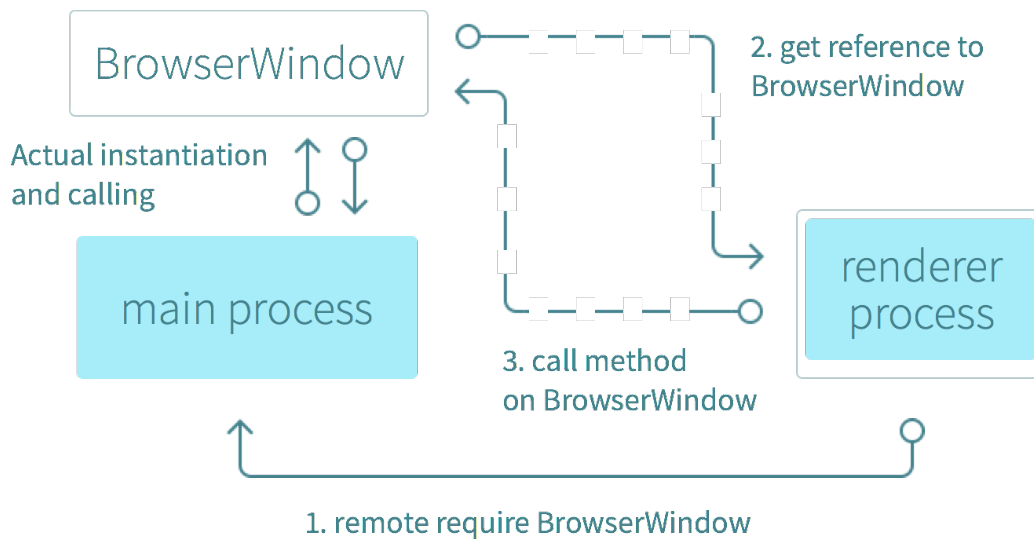

Figura 3.1: Architettura applicazione Electron.
Fonte: Kristian Poslek https://medium.com/developers-writing/building-a-desktop-application-with-electron-204203eeb658

una cartella, di seguito viene riportato un esempio di funzione contenuto nel modulo:

```
1  /**
2   * Read the files contained in the directory given as
      parameter and returns
3   * them as an array of AppFile
4   *
5   * @param {String} directory The path for the directory to be
       read.
6   * @returns {AppFile[]} An array representing all of the
      folder's files.
7   */
8  async function sendReadDirMessage(directory) {
9    let fileNames = await ipcRenderer.invoke(requestChannels.
      readDir, directory);
10   let parsedFiles = fileNames.map(name => {
11     return new AppFile(name, directory);
```

```
12   })
13   return parsedFiles;
14 }
```

Listing 3.1: Esempio di funzione del modulo per gestire l'IPC

3.1.2 Utilizzo Arduino

Come già menzionato alcune volte, l'applicazione funziona unitamente all'IDE Arduino. Agli occhi dell'utente questa collaborazione constiste semplicemente nel fornire un editor per la creazione di sketch di codice. In realtà il motivo è più complesso: Integrato all'IDE è presente un compilatore pensato per la traduzione di sketch Arduino in programmi per microcontrollori.

Questo strumento, chiamato *arduino-builder*, semplifica alcuni aspetti della compilazione come l'inclusione di librerie esterne e la corretta creazione di codice eseguibile per l'architettura del microcontrollore scelto. Dal momento che questo tool viene inserito nell'installazione classica dell'IDE, è naturale che sia pensato per essere utilizzato in tale contesto. Vi è infatti un alto grado di accoppiamento tra IDE e builder, tale che questi due attingono agli stessi file di configurazione e alle stesse risorse (liberie, schede installate).

Per questo motivo fornire agli studenti l'IDE Arduino consiste in un doppio vantaggio: Non solo potranno usufruire di un editor dedicato allo sviluppo IoT, ma il processo di compilazione viene ampiamente semplificato dal fatto che librerie esterne e board aggiuntive verranno installate direttamente dallo studente, tramite l'IDE, e saranno pronte per essere utilizzate dal builder, il quale verrà invocato direttamente dal client.

Il processo di compilazione è ampiamente astratto dall'arduino-builder, occorre infatti invocarne l'eseguibile al quale vengono passati diversi parametri, il più degno di nota tra questi è il *fully qualified board name* (FQBN), una stringa univoca che identifica l'architettura per la quale andrà compilato il programma fornito in input. Questa stringa viene fornita al client dal server al momento della connessione ad una sessione.

Il risultato del processo di compilazione è un insieme di file eseguibili caricabili sul microcontrollore programmato durante la sessione.

```
$: arduino-builder -fqbn arduino:avr:uno
  -tools hardware/tools \
  -tools tools-builder \
  -libraries libraries \
  -hardware hardware \
  -build-path /tmp/intra_build/ \
   portable/sketchbook/sessione_01
Sketch uses 924 bytes (2%) of program storage space. Maximum
   is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving
   2039 bytes for local variables. Maximum is 2048 bytes.
$:
```

Listing 3.2: Esempio di compilazione di uno sketch con arduino-builder

3.1.3 Invio codice eseguibile

Dal compilatore vengono generate diverse versioni del programma fornito come input, alcune costituiscono il semplice programma in formato binario, altre contengono anche il bootloader per il microcontrollore al quale sono destinati questi file. Per rimanere il più generali possibile si è scelto di inviare tutti l'intero output del compilatore al RaspberryPi il quale, seguendo i propri parametri di configurazione, sceglierà i file adatti ad essere caricati sul microcontrollore. Da ciò segue che il payload del messaggio inviato sarà un archivio compresso di questi file.

La dimensione dell'archivio è variabile e dipende dal tipo di architettura programmata. Durante la fase di testing è stato rilevato che la dimensione del payload, oltre un certo limite, causa la perdita dei dati inviati. La natura di questa perdita non è ancora chiara, la teoria più valida è che possa dipendere da una delle due implementazioni di Socket.IO usate. La soluzione è stata quella di suddividere il messaggio in frammenti da 5KB l'uno e

di accompagnare ogni frammento con un indice che ne fornisca la posizione all'interno del messaggio completo.

3.2 Server

Procediamo ora ad illustrare i dettagli implementativi del server. Il server è una web application scritta in Python con l'ausilio del framework Django [37] e, in particolar modo vista la natura di servizio REST, con la libreria Django Rest Framework [38].

A fianco del servizio web viene eseguita un'applicazione Socket.IO [39]. Si tratta di un framework che permette al server web e ad un client di comunicare bidirezionalmente in tempo reale utilizzando opportunamente un polling HTTP o il protocollo websocket.

Di seguito verranno descritti in dettaglio gli aspetti salienti dell'implementazione del server.

3.2.1 REST API

La REST API esposta dal server è il mezzo tramite il quale il mondo esterno può interagire col lato gestionale dell'applicazione. Vengono esposti 2 namespace:

- `/microcontrollers/` Che espone gli endpoint per la gestione dei microcontroller
- `/sessions/` Contiene tutti i punti di accesso per la creazione, prenotazione e consultazione delle sessioni.

Le operazioni definite sulle risorse del sistema sono piuttosto basilari, come ad esempio la creazione di un microcontrollore.

```
1 "request": {
2     "method": "POST",
3     "url": "http://domain.example/microcontrollers/",
4     "httpVersion": "HTTP/1.1",
```

```
5       "headers": [
6           {
7               "name": "Authorization",
8               "value": "Bearer {api_token}",
9           }
10      ],
11      "postData" : {
12          "mimeType": "application/json",
13          "text" : "{name: '{Microcontroller name}',
    description: '{Description and pinout}'}",
14      },
15  }
```

Listing 3.3: Formato richiesta per la creazione di un Microcontrollore in HAR

L'unica eccezione è la creazione di sessioni multiple, ovvero un numero definito di sessioni al giorno che si ripetono per un numero di giorni definito dal professore.

```
1   "request": {
2       "method": "POST",
3       "url": "http://domain.example/sessions/",
4       "httpVersion": "HTTP/1.1",
5       "queryString": [
6           {
7               "name": "bulk",
8               "value": "1",
9               "comment": "Enable bulk creation"
10          },
11      ],
12      "headers": [
13          {
14              "name": "Authorization",
15              "value": "Bearer {api_token}",
16          }
17      ],
18      "postData" : {
19          "mimeType": "application/json",
```

```
20            "text" : "{first_session: '2021-01-01T09:00:00.000Z',
        sessions_duration: '900',  per_day_count: '5', days_count
        : '5', microcontrollers: [1,2]}",
21        },
22 }
```

Listing 3.4: Formato richiesta per la creazione di sessioni multiple in HAR

Per ogni endpoint è richiesta l'autenticazione, fornibile attraverso JWT [40], ad eccezione del punto di login.

3.2.2 Svolgimento sessioni

La parte più delicata della logica del server è la gestione delle sessioni. Una sessione è un evento di durata prolungata, durante il quale si avvicendano interazioni tra studenti e microcontrollori. Per fare chiarezza sul lato implementativo del loro svolgimento è necessario descrivere il loro ciclo di vita.

Ciclo di vita

La sessione ha inizio quando uno studente presenta la richiesta al server di connettersi al microcontrollore e questo ne verifica l'identità confermando la sua richiesta. Una volta iniziata la sessione, il client potrà richiedere l'inizio del flusso video del microcontrollore al RaspberryPi.

A questo punto lo studente potrà procedere a programmare il dispositivo. Durante questa fase vengono in genere scambiati due tipi di messaggio: la richiesta, da parte del client, di caricare ed eseguire un programma sul microcontrollore e i messaggi inviati dal dispositivo tramite la sua interfaccia seriale.

Se il Raspberry dovvesse improvvisamente disconnettersi dalla sessione (per una perdita di connessione o un errore dell'applicazione), verrà inviato automaticamente dal server, al client, il messaggio `peer_disconnected`. In caso di riconnessione il messaggio inviato sarà `peer_reconnected`. Viceversa,

se il client dovesse disconnettersi il server comanderà al Raspberry di chiudere il flusso video, ma non di resettare la scheda. Conseguentemente alla riconnessione di uno dei due partecipanti verranno scambiati nuovamente i messaggi necessari a inizializzare la trasmissione video da Raspberry a client.

La fine di una sessione può avvenire in due modi:

- Naturalmente al suo orario di fine
- Manualmente, su richiesta dello studente.

In entrambi i casi il processo di terminazione è il medesimo, l'unica differenza è che nel secondo caso il processo è originato dall'invio di un messaggio da parte del client. Giunta la fine di una sessione, il server compie le opportune operazioni di pulizia e sucessivamente invia il messaggio di terminazione a client e Raspberry. A questo punto il microcontrollore verrà resettato caricandovi un programma vuoto.

Namespace Socket.IO

Spieghiamo adesso come è stato implementato quanto descritto nella sezione 3.2.2. La comunicazione in tempo reale tra client e raspberry avviene tramite un canale websocket con l'ausilio di Socket.IO. Sebbene ai due partecipanti alla sessione possa sembrare di avere un contatto diretto l'uno con l'altro, ogni messaggio scambiato tra essi viene inviato al server che, dopo aver eseguito alcuni controlli, lo invia al destinatario corretto. Questo dettaglio deriva dal meccanismo di base del protocollo websocket e non da una scelta implementativa, anche se risulta particolarmente utile al nostro caso d'uso.

Socket.IO è, in origine, un protocollo la cui implementazione ufficiale esiste solo per il linguaggio Javascript. Per renderne possibile l'utilizzo sul server Python viene sfruttata la libreria python-socketio [41]. Questa libreria permette di sviluppare applicazioni client e server che utilizzino il protocollo Socket.IO. La libreria fornisce sia API standard che un'implementazione asincrona basata sul framework asyncio [42] di Python.

Lo svolgimento di una sessione avviene attraverso il namespace socket.io `/microcontrollers/`. Questo namespace è implementato da una singola classe all'interno della quale vengono definiti gli event listener per i possibili messaggi inviati. Un listener non è altro che un metodo la cui firma segue una determinata convenzione e che accetta fino a due parametri: Il primo è il *sid*, ovvero un codice univoco che identifica il singolo client che ha inviato il messaggio; il secondo è opzionale ed è il payload del messaggio. Al momento della loro connessione sia ai client che ai raspberry viene automaticamente assegnato un sid.

```
@authenticated
async def on_running_code(self, sid):
    session = await self.__get_iot_session(sid)
    student_sid = self.get_peer_sid(session, sid)
    await self.emit("running_code", room=student_sid)
```

Listing 3.5: Gestione messaggio di esecuzione di un programma

Unitamente a Socket.IO viene utilizzato anche un server Redis [43] per la memorizzazione di informazioni riguardanti le sessioni. Nello store Redis viene infatti salvata una struttura dati, associata all'id della sessione in corso, contenente le informazioni necessarie al suo svolgimento.

```
{
    microcontroller: "{Raspberry's sid}",
    student: "{Student's sid}",
    session: "{Session's primary key}",
    end_date: "{Nominal session's end}"
}
```

Listing 3.6: Formato della struttura salvata nello store Redis per ogni sessione

Queste informazioni vengono consultate ogni volta che un messaggio viene ricevuto dal server al fine di verificarlo, autenticarlo ed indirizzarlo al destinatario appropriato.

Per risalire alla sessione corrispondente ad un sid, ovvero la sessione a cui il peer con un determinato sid sta partecipando, è necessario accedere all'identificativo della sessione stessa. Per farlo viene sfruttato un oggetto

di sessione (che per non essere confuso con il concetto di sessione fin qui menzionato verrà chiamato “sessione socketio”) che viene associato ad ogni sid dal framework Socket.IO. In questo oggetto vengono salvati due dati: l’id della sessione partecipata e il ruolo di quel determinato client Socket.IO. Utilizzando l’id della sessione è quindi possibile risalire alla più completa struttura contenuta nello store Redis.

Ogni evento ricevuto dal server viene gestito allo stesso modo, salvo che per alcune eccezioni:

1. Autenticazione del peer verificando inoltre che la sessione alla quale sta partecipando sia ancora in corso.
2. Estrazione del sid del co-partecipante alla sessione del mittente
3. Inoltro del messaggio al destinatario

Le variazioni di questo flusso consistono nella presenza di operazioni aggiuntive per messaggi speciali, come quello di terminazione della sessione o di disconnessione.

Autenticazione Verrà adesso illustrato come avviene l’autenticazione di studenti e microcontrollori una volta che questi si connettono al canale websocket. Una connessione websocket viene creata in modo analogo ad una connessione HTTP, con la differenza che non è possibile allegarvi header. L’unica parte in cui possono essere inclusi dati arbitrari è la lista di parametri query dell’URL.

Al momento della connessione, il client dello studente allega all’interno dei query parameter della richiesta: il token di autenticazione (precisamente un token JWT), il ruolo con il quale si presenta (per lo studente questo valore è **student**) e l’id della sessione alla quale ci si desidera connettere. Sebbene i parametri di query non siano un mezzo sicuro per la trasmissione di dati sensibili, finchè la connessione avviene attraverso SSL nessun dato sensibile sarà esposto. Una volta che il server verifica l’identità dello studente ed i suoi permessi a connettersi alla sessione specificata, esso invierà conferma al

client e la sessione potrà avere inizio, altrimenti la connessione verrà rifiutata ed un messaggio di errore sarà recapitato al client.

Per quanto riguarda il raspberry la procedura è analoga, le differenze sono due: una riguarda il metodo con cui avviene l'autenticazione, la seconda è l'omissione del parametro indicante la sessione alla quale ci si sta connettendo in quanto il Raspberry rimane sempre connesso al server anche quando non è impegnato in una sessione. Il raspberry si autentica inviando una pre-shared key che consiste in un codice UUID.

Terminare una sessione

Il sopraggiungere della fine naturale di una sessione è un evento asincrono rispetto alla logica di esecuzione del server. Per questo motivo è necessario affidare il compito di terminare una sessione ad un task esterno, che venga azionato al momento opportuno. Data la semplicità del contesto d'uso del sistema è stata scelta una soluzione contenuta: viene utilizzato un task asyncio che rimane dormiente fino a pochi momenti prima della fine nominale di una sessione, per poi risvegliarsi e compiere tutte le operazioni necessarie alla terminazione. Questa soluzione presenta alcuni limiti, il più rilevante è l'impossibilità di eseguire più processi server in contemporanea dal momento che i task sono legati al processo server che li ha creati, di conseguenza task non potrebbero essere manipolati da processi che non ne sono i creatori (impedendo operazioni come la cancellazione di uno di questi task nel caso in cui lo studente abbia deciso di terminare la sessione manualmente).

Per superare questo limite occorrerebbe inserire i task di terminazione all'interno di una job queue esterna ed utilizzare Redis per la coordinazione tra task e processi server.

```
1 async def session_stop_task(self, duration: timedelta,
    session_id):
2     logging.debug("Scheduling session stop")
3     end_time = int(time() + duration.seconds)
4     await sleep (duration)
5     session_data = self.redis_session_get(session_id)
```

```
6        student_sid = session_data['student']
7        microcontroller_sid = session_data['microcontroller']
8        await self.emit('stop_session', room=student_sid)
9        await self.emit('stop_session', room=microcontroller_sid)
10       await self.terminate_session(session_id, mode=
        SESSION_TERMINATION_CAUSE_CHOICES[1][0])
11       self.termination_tasks.pop(session_id)
```

Listing 3.7: Task di terminazione di una sessione

3.3 Microcontrollore

Concludiamo il capitolo sull'implementazione del sistema parlando del microcontrollore. Con questo termine si intende la combinazione tra RaspberryPi e una qualunque scheda per la quale sia possibile creare programmi sfruttando l'ecosistema Arduino. Questi due dispositivi sono connessi tra loro tramite un cavo USB il quale permette di instaurare un collegamento seriale tra essi.

Il software in esecuzione sul RaspberryPi è composto da diversi componenti: uno per la gestione dei messaggi in arrivo o uscita tramite protocollo websocket, un altro ha il compito di monitorare possibili messaggi inviati dal microcontrollore tramite interfaccia seriale (**monitor seriale**) ed infine un componente che gestisce il caricamento di programmi sul dispositivo. In generale l'applicazione è basata sul framework asyncio e questo aspetto è rispecchiato nei singoli componenti che sono stati scritti seguendo questo paradigma. Analogamente al server, la libreria utilizzata per sfruttare la tecnologia webscoket è python-socketio nella sua versione asincrona.

3.3.1 Dati seriali

Durante una sessione è possibile che uno studente carichi un programma che faccia uso dell'interfaccia seriale per inviare messaggi testuali dal microcontrollore a se stesso. Per catturare questi messaggi viene fatto uso della

libreria aioserial [44], un wrapper asincrono a pyserial [45]. Attualmente l'interfaccia seriale è pensata per essere usata in sola lettura da parte dello studente, esso infatti non può inviare dati dal client al microcontrollore, per questo il monitor seriale è eseguito in modalità sola lettura.

Quando un messaggio viene ricevuto dal monitor, questo viene formattato in modo da includerne il timestamp di ricezione e viene poi fornito ad una funzione di callback che si occuperà di inoltrarlo allo studente.

Il timestamp di un messaggio coincide col momento in cui il primo (o i primi) caratteri di una nuova riga di output viene letta dal monitor. Con riga si intende una qualunque stringa terminata dalla sequenza di caratteri "`\r\n`".

3.3.2 Upload sketch

La funzionalità chiave del Raspberry è il caricamento di programmi sulla scheda ad esso collegata, operazione che avviene a seguito della completa ricezione di un messaggio `run_code` da parte dello studente. Il processo di caricare un programma sul microcontrollore viene affidato ad un tool da riga di comando chiamato Arduino CLI [46]. Si tratta di un tool che permette di gestire librerie, schede, compilare sketch ed eseguirne l'upload su di un dispositivo compatibile collegato. Lo scopo per il quale viene utilizzato dall'applicazione è l'upload di programmi.

Trattandosi di un'operazione bloccante, di discreta durata e della è necessario conoscere l'esito appena questo è noto, Arduino CLI viene invocato tramite la versione asincrona della primitiva popen, `create_subprocess_shell`, fornita da asyncio.

```
1 $: arduino-cli upload --fqbn arduino:avr:uno -p /dev/ttyUSB0
    --input-dir /path/to/binary_dir/
```

Listing 3.8: Invocazione ArduinoCLI per caricamento di un programma

3.3.3 Streaming video

Durante lo svolgimento di una sessione, il Raspberry ha il compito di inviare un flusso video dello stato fisico del microcontrollore allo studente. Questa trasmissione avviene tramite WebRTC. Affinchè lo stream possa essere inizializzato, il protocollo prevede una fase di handshake durante la quale vengono concordati i parametri dei flussi multimediali scambiati. L'handshake, detto *Signaling*, consiste nello scambi di due messaggi chiamati *SDP offer* e *SDP answer*. Lo standard non specifica in che modo questo debba avvenire questo scambio, comunemente viene sfruttato il protocollo websocket ed è esattamente quello che avviene all'inizio di una sessione: Appena allo studente viene confermato l'inizio della sessione, il suo client genera un'offerta SDP e la invia tramite il messaggio `start_video_stream`. Una volta ricevuto il messaggio il Raspberry ne utilizzerà il contenuto per generare la sua risposta ed inviarla all'interno del messaggio `start_video_stream_answer`.

Come il nome suggerisce, WebRTC è stato pensato per lo streaming multimediale P2P in ambiente web, per sfruttarlo su Python viene usata una libreria che ne fornisce un'implementazione: aiortc [48]. Questa libreria mira ad esporre un'API simile a quella per il linguaggio Javascript, con il supporto per la cattura di flussi multimediali da webcam e microfoni.

Implementando questo componente del microcontrollore è venuta alla luce una limitazione, relativamente al nostro caso d'uso, della libreria: WebRTC consente di trasmettere video nei formati H264 o VP8/9, il flusso video ottenuto dalla webcam deve essere quindi codificato in uno di questi due formati. I moduli di aiortc che si occupano di questa operazione sono stati pensati per eseguire una codifica software dei video, con la conseguenza che una macchina dalle risorse limitate come il Raspberry soffra un grave degrado della performance.

Per risolvere il problema sono state provate diverse strade, quella più promettente è stata quella di sfruttare il modulo webcam del Raspberry insieme alle librerie create per l'acquisizione video da tale dispositivo. Il Raspberry infatti è dotato di un video encoder hardware, accedervi però è un compi-

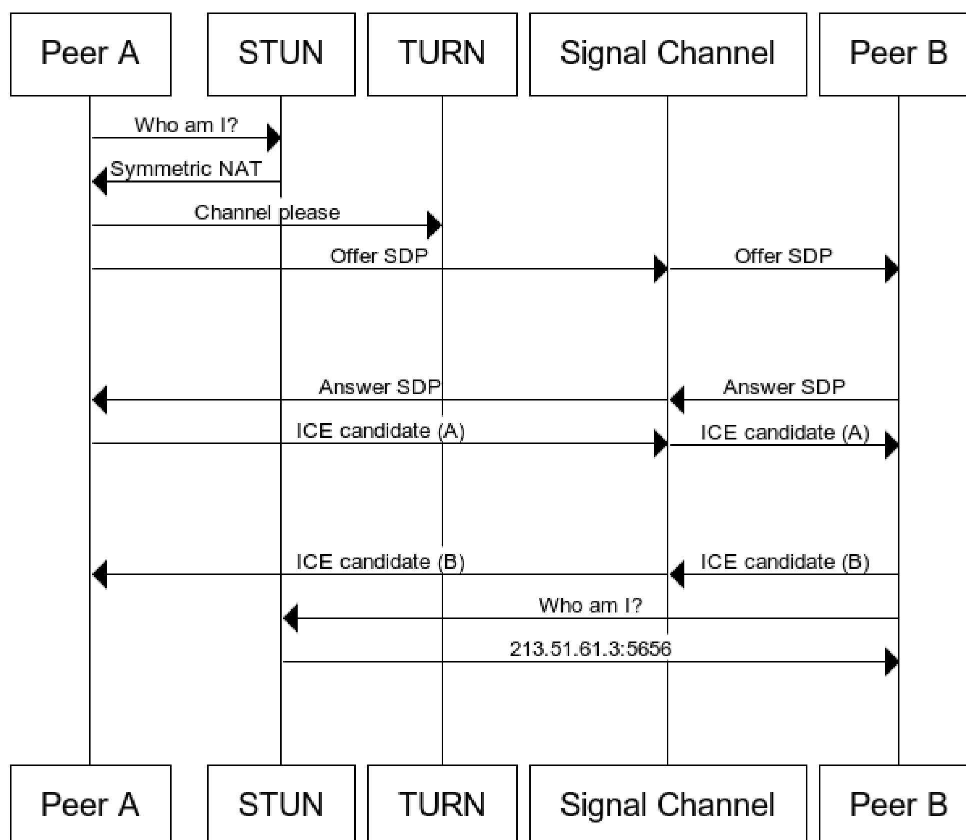

Figura 3.2: Handshake WebRTC [47]

to delicato, che richiede l'utilizzo di codice specializzato e di basso livello. Una libreria che astrae da queste complicazioni è picamera [49], pensata per interagire, tramite applicazioni Python, con il modulo webcam. Tra le sue numerose funzionalità, questa libreria permette l'acquisizione di flussi video codificati in H264 tramite l'acceleratore hardware presente sul SBC.

A questo punto è stato necessario modificare la pipeline multimediale di aiortc in modo da inserire il video generato tramite picamera, per fare ciò è stata scritta una versione personalizzata della classe usata da aiortc per l'acquisizione video. Piuttosto che accedere alla webcam, la classe

`RaspberryCamTrack` accede al flusso video generato da picamera e lo inserisce nella pipeline di aiortc. Inoltre è stato necessario eseguire il monkey patching della classe che, all'interno di aiortc, si occupa di eseguire la codifica in H264 in modo da bypassare completamente questa procedura in quanto non più necessaria.

```
1  class RaspberryCamTrack(VideoStreamTrack):
2      """
3      Custom :class:aiortc.mediastreams.VideoStreamTrack
   implementation
4      made for getting the h264 encoded video frames from the
   raspberry
5      camera without any codec overhead put onto the CPU
6      """
7
8      def __init__(self, camera_resolution=(640, 320),
   camera_framerate=20):
9          super().__init__()
10         self.log = logging.getLogger(type(self).__name__)
11         self.camera = get_picamera()
12         self.camera.resolution = camera_resolution
13         self.camera.framerate = camera_framerate
14         self.camera.vflip = True
15         self.frames = CustomBuffer(self.camera)
16         self.log.info("Starting camera")
17         self.camera.start_recording(self.frames.buffer,
   format='h264',
18                                     profile='baseline',
19                                     intra_period=
   camera_framerate * 2)
20
21     async def recv(self):
22         frame = await self.frames.get_frame()
23         return (frame[0], int(frame[1] / (10 ** 2)))
24
25     def stop(self):
26         super().stop()
27         self.frames.close()
```

```
28          self.camera.stop_recording()
29          self.log.info("Custom video track closed")
```

Listing 3.9: Implementazione classe customizzata per la generazione di un flusso video adatto ad aiortc

Capitolo 4

Validazione

Concludiamo l'esposizione del sistema con il capitolo della validazione. In questo capitolo saranno illustrati elementi sia di carattere quantitativo che qualitativo al fine di illustrare l'efficacia del sistema realizzato. Trattandosi di un software sperimentale non sono presenti dati sul reale utilizzo da parte degli utenti finali, ciononostante è possibile fare alcune considerazioni sulle verifiche condotte durante la fase di testing.

4.1 Delay sessione

Un fattore importante per la valutazione della qualità del sistema è il ritardo che viene inserito nel normale flusso di programmazione di un microcontrollore. Le tempistiche da analizzare sono due: La latenza del feed video ed il tempo impiegato per caricare il programma scritto dallo studente sul microcontrollore. Per compiere queste misurazioni sono stati condotti alcuni test con la seguente configurazione:

- Il server viene eseguito su una macchina connessa alla rete locale tramite un collegamento WiFi
- Il Raspberry è connesso alla rete locale tramite WiFi
- Il client si connette al server tramite una connessione dati 4G

Ritardo feed video La misurazione della latenza video è avvenuta tramite l'uso di due orologi sincronizzati, uno inquadrato dal Raspberry e l'altro posto a fianco della macchina dello studente come riferimento. Successivamente lo studente si è conensso ad una sessione creata per programmare il microcontrollore di tale Raspberry, una volta inizializzato il flusso video si è potuto comparare la differenza tra l'orario mostrato dall'orologio inquadrato e quello di riferimento. La differenza misurata è di circa 1 secondo.

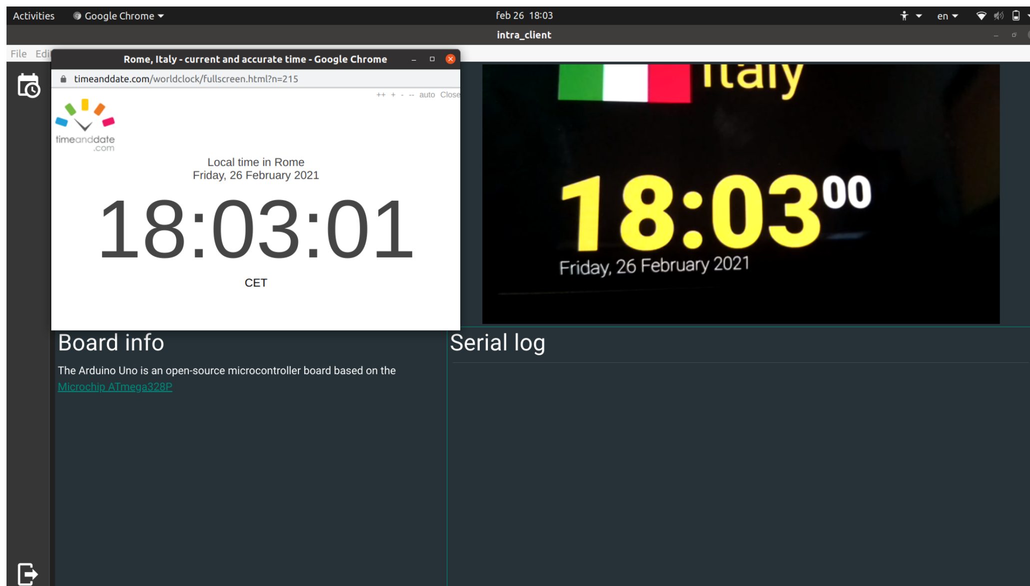

Figura 4.1: Delay video misurato tramite connessione di rete mobile

Ritardo di esecuzione Con ritardo di esecuzione si intende il tempo che impiega lo sketch scritto dallo studente ad essere caricato sul microcontrollore una volta che ne viene richiesta l'esecuzione. La misura di questo tempo è dato dalla somma di diversi fattori: Il tempo di compilazione, di creazione dell'archivio compresso, di trasmissione, decompressione e, finalmente, del caricamento sul microcontrollore.

Un fattore che gode di una certa variabilità è il tempo di compilazione. Durante la fase di sviluppo sono state usate diverse architetture di microcon-

trollori, alcune delle quali richiedevano una compilazione di 10 secondi. Per questo test è stato utilizzato un Arduino UNO[1], per il quale il processo di compilazione non supera il secondo.

Il tempo rilevato in questo test è stato di circa 9 secondi che, sommati al secondo del ritardo video, si traduce in un tempo di 10 secondi dal momento in cui lo studente richiede l'esecuzione del programma a quando ne possa osservare gli effetti.

Ripetendo il test in rete locale, ovvero utilizzando una macchina collegata alla stessa rete del server e del microcontrollore, il tempo si riduce a 7 secondi (quindi 8 prima che lo studente possa osservare risultati). Da questo si può dedurre che il processo di compressione e decompressione inserisce un cospicuo ritardo al processo. Uno possibile sviluppo futuro comporterebbe quindi la rimozione della necessità di inviare multipli file al Raspberry.

4.2 Casi d'uso

Verranno ora illustrati i 2 casi d'uso più frequenti per il sistema per illustrarne il suo funzionamento.

Creazione esercitazione É possibile che un professore, terminato un modulo del proprio corso, ritenga sia necessario permettere agli studenti di eseguire un'esercitazione pratica. In questo caso potrebbe decidere di rendere disponibili alcuni microcontrollori ai propri studenti per lo svolgimento di queste attività. Lo scopo del professore è quindi quello di creare un numero di sessioni tale da permettere agli studenti di accedere ai microcontrollori. Il professore può quindi creare delle sessioni multiple usando la funzione "Bulk creation" del client.

Il professore può specificare quali microcontrollori impegnare, l'orario di inizio delle sessioni, la loro durata e per quanti giorni ripeterle.

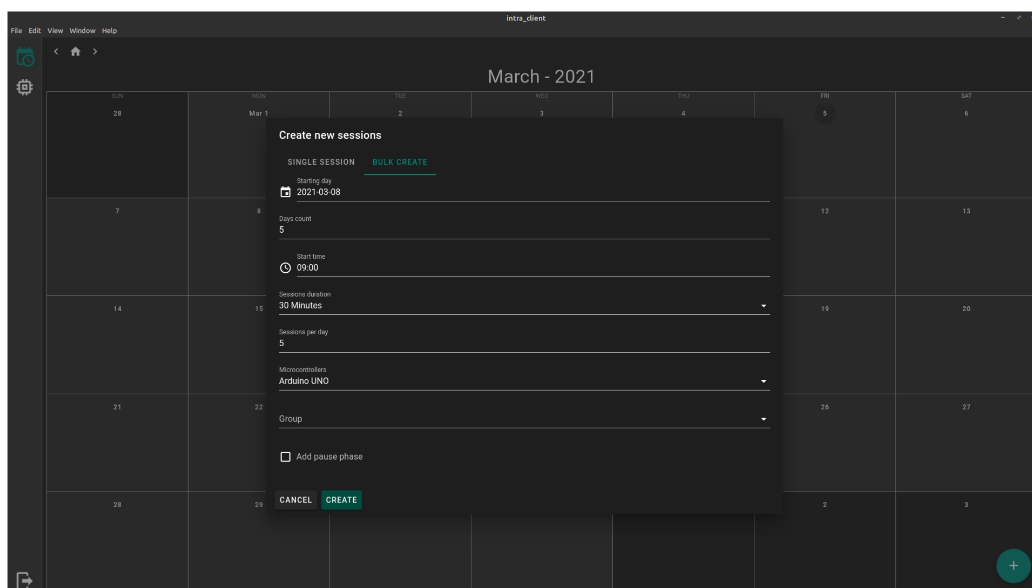

Figura 4.2: Creazione di sessioni multiple

Scrittura programma seriale Al fine di illustrare le funzionalità del client durante una sessione, viene presentato un esempio di scrittura di programma che faccia uso dell'interfaccia seriale del microcontrollore.

Inizialmente lo studente effetua il login ed accede alla sezione delle sessioni prenotate, qui potrà decidere di collegarsi ad una sessione in corso a lui assegnata.

Una volta entrato nella sessione lo studente si ritroverà davanti alla schermata di sessione, composta dal file picker, il feed video, una sezione informativa e la console seriale. A questo punto lo studente apre l'IDE Arduino tramite il pulsante "Open Arduino" e creerà un nuovo sketch che comanderà la scheda di inviare alcuni messaggi attraverso la propria interfaccia seriale.

Una volta scritto il programma tramite l'IDE, lo studente potrà ritornare alla finestra del client, selezionare lo sketch appena creato e farlo eseguire al microcontrollore cliccando sul pulsante "Compile and run". Dopo una breve attesa il microcontrollore inizierà ad inviare messaggi seriali al client, come illustrato nella Figura 4.4

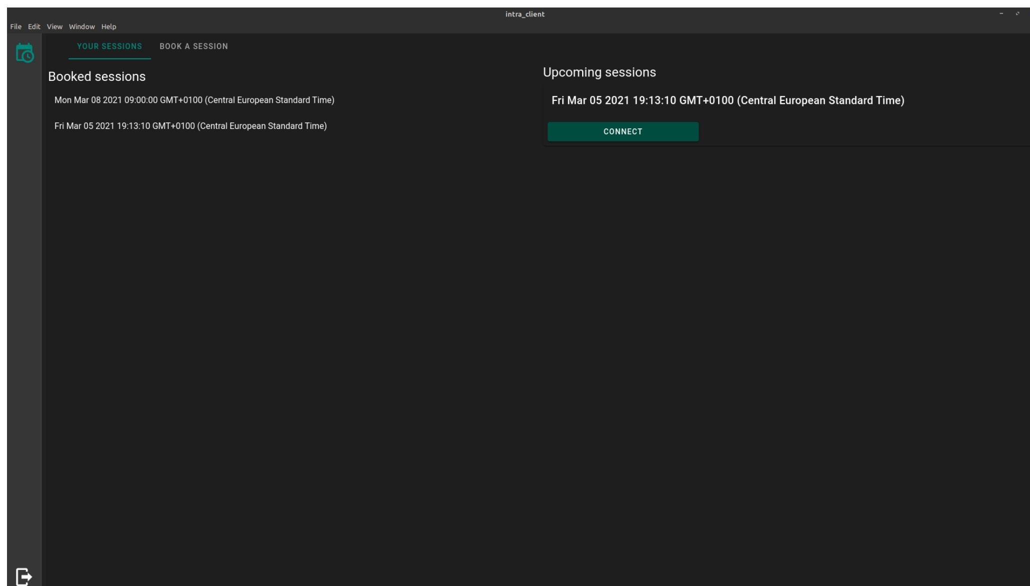

Figura 4.3: Schermata di accesso alle sessioni

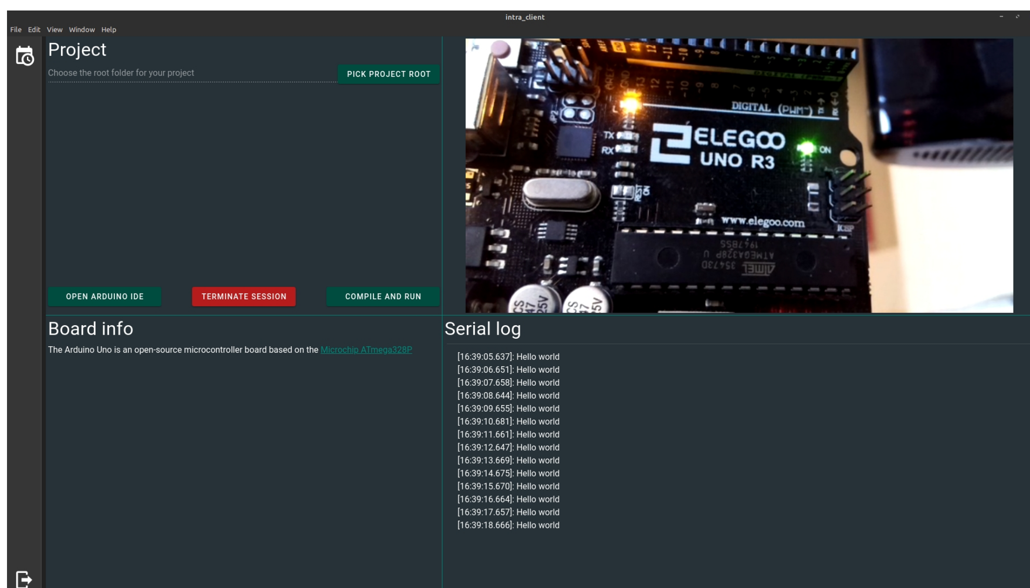

Figura 4.4: Utilizzo della console seriale durante una sessione

Capitolo 5

Conclusioni

In questa tesi è stato illustrato un sistema di programmazione remota di microcontrollori mirato al mondo dell'educazione in tema Internet of Things. In particolare è stato realizzato un software che permette ai professori di creare laboratori remoti con i microcontrollori disponibili nei laboratori fisici del proprio istituto. La necessità di un tale sistema era sorto da diverso tempo, la sua realizzazione è stata però accelerata in risposta alle misure di didattica a distanza, adottate in seguito allo scatenarsi della pandemia di COVID-19. Sono stati analizzati requisiti, componenti ed implementazione del sistema con l'obiettivo di rendere i dispositivi, presenti nei laboratori, in grado di essere programmati, a turno ed in remoto, dagli studenti che ne prenotino l'utilizzo.

Sebbene questo sia l'uso previsto del sistema è anche possibile collegare un qualsiasi microcontrollore al server centrale e renderlo programmabile, ad esempio, un professore potrebbe disporre di alcuni microcontrollori nella propria abitazione che vorrebbe fornire remotamente ai propri studenti, possibilità vantaggiosa se si considerano la necessità di condurre lezioni in didattica a distanza durante (come accade durante l'attuale epidemia). Allo stato attuale di implementazione il sistema è consono alla gestione di alcune sessioni parallele senza che vi siano grossi colli di bottiglia, i passi da compiere per risolvere i problemi di scalabilità sono però principalmente tecnologici

con risvolti limitati sull'architettura del sistema.

Attualmente vi è l'intenzione di impiegare il sistema per permettere lo svolgimento del corso di Internet of Things della laurea magistrale in Informatica all'Università di Bologna.

5.0.1 Possibili sviluppi

Parliamo ora degli aspetti dell'applicazione di cui si intuiscono possibili espansioni. Vi sono diverse idee che vanno dalle semplici migliorie al sistema alla creazione di una rete distribuita di microcontrollori utilizzabile da una comunità di studenti.

Monitor di fine sessione

Come accennato nella sezione 3.2.2, il meccanismo utilizzato per terminare le sessioni è un task creato tramite il framework asyncio [42] che rimane dormiente fino alla fine nominale della sessione per poi svolgere le operazioni opportune.

Come già specificato si tratta di una soluzione non scalabile e poco elegante. Una soluzione migliore riguarda l'utilizzo di un task scheduler esterno, come Django Background Tasks [50] o Celery [51]. I task assegnati a questo scheduler verrebbero azionati alla terminazione di una sessione e potrebbero svolgere gli stessi compiti svolti dai task asyncio attuali, in particolare:

- Rimuovere l'oggetto di sessione dallo store Redis
- Inviare il messaggio di fine sessione al client
- Inviare il segnale di reset al Raspberry

Questi ultimi due compiti implicano l'utilizzo del canale websocket, per inviare messaggi su questo tramite un processo esterno al server (com'è il caso per i task esterni) si può sfruttare Redis come message queue: python-socketio permette l'utilizzo di Redis come coda di invio per i propri messaggi,

questo significa che qualunque processo che abbia acesso al medesimo store Redis di un'applicazione Socket.IO può inviare messaggi tramite essa [52].

Riduzione overhead esecuzione

Una importante limitazione è il tempo impiegato dalla compilazione all'esecuzione dello sketch durante una sessione. Questo ritardo è largamente influenzato dal tempo di compressione e, più importante, dal tempo di decompressione da parte del Raspberry. Questo processo può essere bypassato se, per ogni microcontrollore, si specificasse quale tipo di file binario occorre per procedere col caricamento del programma. Così facendo, il client dello studente potrebbe scegliere il singolo file da inviare e non doversi occupare della sua compressione. Analogamente il Raspberry potrebbe caricare direttamente il file ricevuto senza dover prima decomprimere un archivio.

Ambiente totalmente integrato

La soluzione attualmente implementata prevede l'utilizzo di due applicazioni desktop durante una sessione: il client e l'IDE Arduino. Una soluzione più auspicabile consiste nell'utilizzare un unico programma sia per la scrittura di codice che per l'interazione col microcontrollore. Questo è possibile grazie ad una moderata ristrutturazione del client: Si può realizzare un singolo IDE che permetta di scrivere sketch di codice e che utilizzi ArduinoCLI come backend per compilazione e gestione di librerie.

Questa strada prevede lo studio dell'interfaccia gPRC offerta dal tool di riga di comando, trattandosi di un'interfaccia sperimentale è soggetta a cambiamenti.

Rete distribuita

Ricollegandosi al discorso fatto durante l'introduzione di questo capitolo, una funzionalità interessante sarebbe quella di permettere anche ai singoli studenti di contribuire al sistema offrendo i propri microcontrollori. Con

poche accortezze si potrebbe far cadere il requisito tecnico di utilizzare un RaspberryPi per abilitare le schede ad essere programmate da remoto e permettere di utilizzare un qualunque computer (durante le fasi iniziali dello sviluppo era questo il caso). Trattandosi di dispositivi che si trovano fuori dal controllo dei professori, questi sarebbero usati in maniera più informale, senza la possibilità di sessioni, ma in modo diretto tra gli studenti di un corso. Questa espansione tornerebbe molto utile a studenti che si ritrovano a lavorare in gruppo dove uno o più componenti non dispone di un microcontrollore, permettendo così ad un singolo componente di "prestare" il proprio dispositivo ad un compagno.

Bibliografia

[1] Arduino. Arduino UNO. https://store.arduino.cc/arduino-uno-rev3.

[2] Raspberry Pi Foundation. Raspberrypi website. https://www.raspberrypi.org/.

[3] Arduino. Intel Edison. https://www.arduino.cc/en/ArduinoCertified/IntelEdison.

[4] Kevin Asthon. That 'internet of things' thing.

[5] Jordan Teicher. The little-known story of the first iot device.

[6] The Carnegie Mellon University Computer Science Department Coke Machine. The "only" coke machine on the internet. https://www.cs.cmu.edu/~coke/history_long.txt.

[7] 33 billion internet devices by 2020. Technical report.

[8] statista.com. Number of internet of things (iot) connected devices worldwide from 2019 to 2030. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

[9] OASIS. Mqtt: The standard for iot messaging. https://mqtt.org/.

[10] Mohammed Atiquzzaman Mahda Noura and Martin Gaedke. Interoperability in internet of things: Taxonomies and open challenges.

[11] W3C WoT Working Group. Web of things (wot) architecture. Technical report.

[12] Pew Research Center. Mobile fact sheet. `https://www.pewresearch.org/internet/fact-sheet/mobile/`.

[13] A. Arena, A. Bianchini, P. Perazzo, C. Vallati, and G. Dini. Bruschetta: An iot blockchain-based framework for certifying extra virgin olive oil supply chain. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 173–179, 2019.

[14] Shamayleh A., Awad M., and Farhat J. Iot based predictive maintenance management of medical equipment. 2020.

[15] Volkswagen. Industry 4.0: We make it happen! `https://www.volkswagen-newsroom.com/en/stories/industry-40-we-make-it-happen-4779`.

[16] Arable. Arable - Decision Agricolture. `https://www.arable.com/`.

[17] Hoa Hong Nguyen, Farhaan Mirza, M Asif Naeem, and Minh Nguyen. A review on iot healthcare monitoring applications and a vision for transforming sensor data into real-time clinical feedback. In *2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 257–262. IEEE, 2017.

[18] F. Wu, T. Wu, and M. R. Yuce. Design and implementation of a wearable sensor network system for iot-connected safety and health applications. In *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pages 87–90, 2019.

[19] P. Pace, G. Aloi, G. Caliciuri, R. Gravina, C. Savaglio, G. Fortino, G. Ibanez-Sanchez, A. Fides-Valero, J. Bayo-Monton, M. Uberti, M. Corona, L. Bernini, M. Gulino, A. Costa, I. De Luca, and M. Mortara.

Inter-health: An interoperable iot solution for active and assisted living healthcare services. In *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pages 81–86, 2019.

[20] Usb development board. `https://www.pjrc.com/teensy/`.

[21] Adafruit. Flora is adafruit's fully-featured wearable electronics platform. `https://learn.adafruit.com/category/flora`.

[22] SECO S.P.A. The iot maker board. `https://www.udoo.org/udoo-neo/`.

[23] Joop Brokking. The arduino auto-level quadcopter. `http://www.brokking.net/ymfc-al_main.html`.

[24] Arduino. Arduino project hub. `https://create.arduino.cc/projecthub`.

[25] T. S. El-Hasan. Internet of thing (iot) based remote labs in engineering. In *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 976–982, 2019.

[26] T. V. Tran, H. Takahashi, T. Narabayashi, and H. Kikura. An application of iot for conduct of laboratory experiment from home. In *2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*, pages 1–4, 2020.

[27] M. Poongothai, P. M. Subramanian, and A. Rajeswari. Design and implementation of iot based smart laboratory. In *2018 5th International Conference on Industrial Engineering and Applications (ICIEA)*, pages 169–173, 2018.

[28] Iotify. IoTIFY - cloud based IoT simulator and IoT testing platform. `https://iotify.io/iot-virtual-lab/`.

[29] www.roboticlab.eu. Welcome to DistanceLab e-environment. `http://distance.roboticlab.eu/`.

[30] IOT-OPEN.EU consortium. VREL - remote access distant labs. http://iot-open.eu/io3/.

[31] A. FernÃ¡ndez-Pacheco, S. Martin, and M. Castro. Implementation of an arduino remote laboratory with raspberry pi. In *2019 IEEE Global Engineering Education Conference (EDUCON)*, pages 1415–1418, 2019.

[32] Arduino. Arduino. http://arduino.cc.

[33] Google. Real-time communication for the web. https://webrtc.org/.

[34] IETF. *The Websocket protocol.* https://tools.ietf.org/html/rfc6455.

[35] Raspberry Pi foundation. Camera module v2. https://www.raspberrypi.org/products/camera-module-v2/.

[36] Evan You. The Progressive JavaScript Framework. https://vuejs.org/.

[37] Django Software Foundation. The web framework for perfectionists with deadlines. https://www.djangoproject.com/.

[38] Encode OSS Ltd. Django rest framework. https://www.django-rest-framework.org/.

[39] Socket.IO. Socket.IO. https://socket.io.

[40] Auth0©Inc. Json web tokens. https://jwt.io/.

[41] Miguel Grinberg. Python implementation of the socket.io realtime client and server. https://github.com/miguelgrinberg/python-socketio.

[42] Python Software Foundation. *Asynchronous I/O.* https://docs.python.org/3/library/asyncio.html.

[43] Redis Labs. Redis. https://redis.io/.

[44] Henry Chang. pyserial-asyncio for humans. https://github.com/changyuheng/aioserial.

[45] Chris Liechti. Python serial port extension. https://pypi.org/project/pyserial/.

[46] Arduino.cc. Arduino CLI (Command Line Interface) Application. https://www.arduino.cc/pro/cli.

[47] Altanai. *WebRTC Integrator's Guide*. Preso dall'anteprima web: https://subscription.packtpub.com/book/web_development/9781783981267/1/ch01lvl1sec09/running-webrtc-without-sip.

[48] Miguel Grinberg. Web Real-Time Communication (WebRTC) and Object Real-Time Communication (ORTC) in Python. https://github.com/aiortc/aiortc.

[49] Dave Jones. A pure python interface for the raspberry pi camera module. https://picamera.readthedocs.io/.

[50] John Montgomery arteria GmbH. Database backed asynchronous task queue. https://pypi.org/project/django-background-tasks/.

[51] Celery. Distributed task queue. https://github.com/celery/celery.

[52] Miguel Grinberg. Emitting from external processes. https://python-socketio.readthedocs.io/en/latest/server.html#emitting-from-external-processes.