Sistemi Embedded e Internet-Of-Things

Smart GreenHouse A.A. 2018-2019

Nicholas Ciarafoni
Giovanni Perini

Si vuole realizzare un sistema embedded integrato che rappresenti una versione semplificata di una serra smart.
Il compito della serra smart è l'irrigazione automatizzata (di un certo terreno o pianta) implementando una strategia che tenga conto dell'umidità percepita, con la possibilità di controllare e intervenire manualmente mediante mobile app.
Il sistema è costituito da 5 parti (sotto-sistemi):

- ***GreenHouse Server (PC)***
  - *Contiene la logica che definisce e attua la strategia di irrigazione*
- ***GreenHouse Controller (Arduino)***
  - *Permette di controllare l'apertura e chiusura degli irrigatori (pompe acqua), quindi della quantità di acqua erogata al minuto*
- ***GreenHouse Edge (ESP)***
  - *Permette di percepire l'umidità del terreno*
- ***GreenHouse Mobile App (Android)***
  - *Permette di controllo manuale della serra*
- ***GreenHouse Front End (PC)***
  - *Front end per visualizzazione/osservazione/analisi dati*

GreenHouse Server

Il server scritto in linguaggio Java, consente di pilotare la nostra serra smart tramite un'architettura basata ad eventi, dove è implementata una macchina a stati finiti asincrona.
Per la realizzazione del Server è stato utilizzato IntelliJ IDEA, come IDE di supporto, in quanto ci garantiva un'approccio più fluido e coerente a quello già utilizzato in Android Studio per la realizzazione dell'applicativo richiesto in questo progetto, **SDK** Amazon **Corretto 21.0.3**.
La tecnologia che sta alla base di tutto il progetto lato server, si basa su di un toolkit per lo sviluppo di applicazioni reattive su JVM, chiamato **Eclipse Vert.x**.
Vert.x gestisce eventi tramite un event loop, ottimizzando la scalabilità e la reattività delle applicazioni, e permette la creazione di microservizi, comunicazioni via HTTP, webSocket e messaging con EventBus.
Come caratteristiche principali abbiamo un modello di concorrenza non bloccante, alta disponibilità e tolleranza ai guasti che favoriscono le applicazioni moderne ad alte prestazioni e bassa latenza.

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

Sistemi Embedded e Internet-Of-Things

Smart GreenHouse A.A. 2018-2019

Nicholas Ciarafoni
Giovanni Perini

Netty o Vert.x
differenze e motivazione di scelta

Netty e **Vert.x** sono due framework popolari per la costruzione di applicazioni di rete ad alte prestazioni e scalabili. Sebbene abbiano obiettivi simili, presentano diverse differenze chiave.
Netty gestisce la concorrenza in modo diverso rispetto a Vert.x utilizzando un modello thread-per-channel, dove ogni connessione è associata a un thread dedicato. Grande controllo, ma meno efficiente all'aumentare delle connessioni.
Vert.x invece usa un modello di event loop con I/O non bloccante, impiegando pochi thread per gestire molte connessioni e migliorando quindi l'uso di risorse all'aumentare del carico di richieste.
Sia Netty che Vert.x supportano gli stessi protocolli come HTTP, TCP, UDP, WebSocket e SSL, solo che Vertx offre un livello di astrazione superiore e più indirizzata all'utente, include server e client HTTP e un supporto per WebSocket a messaggistica asincrona.
In Vert.x abbiamo inoltre un EventBus distribuito, che permette la comunicazione asincrona tra le diverse parti dell'applicazione, attraverso un pattern di pubblicazione-sottoscrizione, dove se un produttore vuole pubblicare un messaggio verso un determinato indirizzo, tutti i consumatori ad esso sottoscritti, riceveranno il messaggio in modo asincrono.
La scelta quindi per esigenze, verte sull'utilizzo di Vert.x, dove il lato di gestione asincrona funge da perno per tutto l'applicativo.

Per il lato server sono state utilizzate alcune delle librerie fornite a lezione e in aggiunta integrate librerie esterne.
Per l'inclusione di queste librerie abbiamo fatto uso di **Gradle** versione **8.7**, uno strumento di automazione della build, basato su linguaggio di scripting Groovy o Kotlin, che ci assicura un modo rapido per configurare il progetto.
Gradle, è pensato per progetti multi-modulo di grandi dimensioni, supporta inoltre build incrementali, e riconosce le parti aggiornate del build saltando i processi non necessari, riducendo quindi il tempo di costruzione. Solo le attività con modifiche rispetto all'ultima build vengono eseguite nuovamente.

Le dipendenze aggiunte nel lato server sono:

- **io.vertx:vertx-web:4.5.7**
    - Aggiunge Vert.x Web, una libreria per costruire applicazioni web.
      Questa libreria offre strumenti per costruire applicazioni web con Vert.x.
      Include supporto per routing, gestione delle richieste HTTP, template engines, autenticazione, e gestione delle sessioni.
- **io.vertx:vertx-core:4.0.0.CR2**
    - Aggiunge il core di Vert.x in una versione release candidate.
      È il cuore del framework Vert.x e contiene le funzionalità di base per la

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

gestione degli eventi, il modello di concorrenza non bloccante, e la gestione delle reti.

- **io.vertx:vertx-web-client:4.0.0.CR2**
  - Aggiunge Vert.x Web Client, una libreria per fare richieste HTTP client-side. Permette di fare chiamate HTTP/HTTPS asincrone, supportando operazioni come GET, POST, PUT, DELETE e altre. È utile per consumare API RESTful e comunicare con altri servizi web
- **io.github.java-native:jssc:2.9.2**
  - La JSSC è una libreria per la comunicazione seriale, che permette all'interno del nostro progetto di interfacciarsi con il microcontrollore, ed effettuare lo scambio di messaggi.

Interfacce e implementazioni:

- **CommChannel**
  - Interfaccia di base per la gestione di un canale di comunicazione seriale, all'interno i suoi metodi comprendono: **sendMsg, receiveMsg,isMsgavaiable**.
- **Event**
  - Interfaccia evento personalizzata, dove l'implementazione viene gestita all'interno delle classi che la richiamano, abbiamo quindi 3 tipologie di eventi: **MsgEventFromWifi, MsgEventFromSerial,Tick.**
- **Observer**
  - Interfaccia del pattern Observer per la notifica di un evento, nel loop centrale di controllo dell'applicativo, rispettivamente la classe BasicEventLoopController implementiamo Observer per offrire alla coda di eventi l'evento notificato per essere successivamente processato.

Classi:

- ★ **DataPoint**
  - Crea un oggetto di tipo DataPoint formato da un valore, un campo di tipo temporale, e un place per individuare da quale parte dell'applicativo un evento messaggio proviene.
- ★ **MsgService**
  - Gestore dei messaggi, tenta di collegarsi ad un canale seriale di comunicazione avendone porta e baud rate, si mette in ascolto, notificando i messaggi in ricezione, attraverso un Observable.
    In invio, inoltra i messaggi attraverso la stessa porta seriale.
- ★ **SerialCommChannel**
  - Creazione concreta del canale di comunicazione seriale, dove implementiamo una coda a blocchi, su cui inseriamo e togliamo eventi in testa, successivamente la classe apre e configura la porta seriale.Tutti gli

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

eventi che arrivano alla porta, vengono controllati se sono di tipo RXCHAR, andando a scartare eventuali interferenze, togliendo successivamente l'interruttore di linea e prendendo l'effettivo payload del messaggio, usando come byte di riconoscimento per il termine del contenuto il carattere di nuova linea '\n'.
    - Nell'invio, aggiunge il carattere '\n' al messaggio da inviare, che verrà utilizzato come terminatore in ricezione dagli altri dispositivi.
- ★ **ObservableTimer**
    - In questa classe generiamo un evento tick ad intervalli specifici, utilizzando un scheduler, nel metodo start avviamo la generazione degli eventi e tramite il metodo scheduleTick, pianifichiamo la generazione di ogni evento tick, dopo un certo periodo di tempo deltaT di 5 secondi come da richiesta di progetto. Estendiamo Observable per notificare l'avvenuto evento tick durante l'esecuzione tramite il ScheduleExecutorService dove gestisce un pool di thread che vengono utilizzati per eseguire le attività di generazione di eventi tick.
- ★ **Tick**
    - Evento di notifica per la chiusura dell'erogazione della nostra pompa d'acqua, dopo che il nostro scheduler determina il passaggio dei **deltaT** millisecondi.
- ★ **GreenHouseAgent**
    - Questo agente viene utilizzato come controllore per la serra automatizzata, usando Vertx per la gestione degli eventi asincroni e supportando la comunicazione, sia con arduino che con un server HTTP, per il monitoraggio e il controllo in tempo reale dei dati elaborati e sullo stato della serra.
    - MsgService gestisce la comunicazione seriale con Arduino
    - EventBus gestisce la comunicazione asincrona tra le componenti
    - HttpClient invia dati ad un server HTTP per la visualizzazione in tempo reale
    - State currentState gestisce lo stato della serra che può essere Manuale o Automatico in base alle specifiche del progetto
    - ObservableTimer gestisce un timer osservabile che notifica un evento Tick per comunicare lo spegnimento dell'erogazione, qui la richiesta **POST** HTTP che viene invocata, viene utilizzata per visualizzare sulla pagina web associata al nostro applicativo attraverso l'indirizzo "**/api/data**" la notifica di arresto dell'erogazione, usando anche la variabile timerIsAlive per effettuare il controllo sullo stato del timer.
    - **msgFromWifiEventBridge** contiene il valore del messaggio ricevuto da WiFi, che vogliamo inoltrare tramite **EventBus** alla classe principale del server dove può elaborare bene i dati e visionarli all'interno della pagina web.
    - Impostiamo lo stato iniziale su automatico, e consumiamo i nuovi dati ricevuti tramite un EventBus all'indirizzo "**data**.**new**".
      Tramite il metodo principale processEvent andiamo a processare gli eventi in coda, in base allo stato corrente, manuale o automatico:

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

Sistemi Embedded e Internet-Of-Things

Smart GreenHouse A.A. 2018-2019

Nicholas Ciarafoni
Giovanni Perini

- Manuale

    - **MsgEventFromSerial**: se il messaggio ha l'header 'A', cambia lo stato in automatico. Se il messaggio è un errore e lo vediamo tramite una intestazione "[**ERRORE**]", spegne la pompa, altrimenti invia i dati ad Arduino e al server HTTP, che li visionerà sulla pagina Web.
    - **MsgEventFromWifi**: invia il messaggio ad android "bypassando" Arduino.Il messaggio viene inviato ad Arduino, ma avendo un intestazione unica, viene gestito come messaggio da inoltrare tramite bluetooth ai dispositivi connessi, quindi di fatto bypassa arduino per giungere all'applicativo su Android.

- Automatico
    - Se riceve un evento Tick, invia la notifica al server HTTP e chiude la pompa.
    - Se un evento di tipo Msg Event From Serial, contiene un'intestazione "**B**", cambia lo stato in manuale.
    - MsgEventFromWifi, sono i dati ricevuti continuamente dal modulo **ESP** sullo stato dell'irrigazione, simulando con un potenziometro l'umidità di una pianta o una serra completa.
      I dati dell'umidità non vengono sovrascritti dai valori impostati da mobile, in questa simulazione poiché se avessimo voluto che: impostando un valore da dispositivo mobile e comunicato tramite bluetooth, la nostra umidità cambiasse immediatamente al valore ricevuto, non avrebbe rispettato il caso reale del giusto innalzamento d'umidità di una pianta o serra che sia.
      Sarebbe stato sbagliato e inverosimile, quindi nelle fasi di debug, lato manuale, vedremo un valore che rispecchia l'umidità effettiva data dal valore del potenziometro, e non da un sensore d'umidità reale in quanto sarebbe stato troppo complesso da gestire e creare.
      L'altro valore che viene gestito in questo caso è un valore impostato da mobile che cambia la quantità di acqua erogabile in percentuale (0 - 100%) della pompa.

★ **ServerService**

- In questa classe è racchiuso il cuore centrale del server Vert.x, che gestisce dati sensoriali tramite API RESTful.
  Come attributi principali abbiamo una porta in cui il server è in ascolto, la "**8080**", che gli passiamo nella classe **RunService**, il nostro main.
  E una LinkedList di 10 oggetti DataPoint, usata per memorizzare i 10 rilevamenti dell'umidità.

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

Sistemi Embedded e Internet-Of-Things

Smart GreenHouse A.A. 2018-2019

Nicholas Ciarafoni
Giovanni Perini

- Start()
  - Metodo principale invocato all'avvio del verticle, configura il router per gestire le richieste HTTP, implementato i **CORS,**
    (*Tecnica utilizzata dai browser web per consentire o limitare le richieste effettuate da una fonte diversa rispetto a quella impostata dal server in cui siamo accedendo, una misura di sicurezza che previene le richieste non autorizzate da altri domini.*
    *Se il server riceve delle richieste da un origine diversa, risponde con intestazioni HTTP specifiche, indicando se la richiesta è permessa o negata,permettendo la condivisione delle risorse, o bloccando l'accesso.***)** e permettendo le richieste dall'indirizzo "http://192.168.1.12:8080", dove tramite un BodyHandler gestiamo il corpo delle richieste, altri handlers utilizzati sono:
    - **POST**
      - Invia i dati ricevuti all'indirizzo "/api/data" a **handlerAddNewData**.
        Prende il payload, verifica che non sia nullo o vuoto, estrae quindi i valori interni alle chiavi "value", "place" e aggiunge il timestamp corrente. Con questi 3 valori, gli stessi di un oggetto DataPoint li inserisce dentro un oggetto JSON.
        Dato che abbiamo tre tipologie di posti da cui arrivano i dati, a seconda se riceviamo un evento dal Bluetooth, dal dispositivo o dal Server gestiamo le casistiche.
        - **home** l'oggetto JSON creato viene pubblicato successivamente sull'EventBus per renderlo disponibile ai consumer.
        - **Bluetooth** stampa a video del valore ricevuto.
        - **Server** Stampa dell'erogazione interrotta, senza valori da visualizzare a video.
      - I dati estratti dal payload vengono inseriti all'interno di una LinkedList di DataPoint, e una volta raggiunto il massimo(10 oggetti DataPoint) della lista, rimuoviamo l'elemento meno recente, così da non perdere traccia dei rilevamenti più recenti.
    - **GET**
      - Richiede i dati e li inoltra a **handlerGetData.**
        Che crea un Array JSON con i dati memorizzati e lo invia come risposta.
      - imposta index.html come file di route "/"
    - **handlerFailure**
      - Gestisce gli errori inviando una risposta JSON con un messaggio d'errore.

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

- Log()
  - Funzione di stampa dei messaggi con un formato specifico. Abbiamo pensato di formattare ogni messaggio visualizzabile a video con una intestazione a seconda di quale componente ha richiamato la stampa.

★ **BasicEventLoopController**

- Architettura che sta alla base dell'applicativo, gestisce gli eventi in un event-loop continuo e non bloccante, implementa una coda per memorizzare e processare gli eventi in modo sicuro e sincronizzato tra Thread. Utilizziamo una coda bloccante di capacità 50, per dare spazio a più eventi ricevuti nello stesso istante da più fonti di essere accolti e non perderli, la cosa viene utilizzata per la gestione degli eventi in modo thread-safe, garantendo che l'accesso alla stessa avvenga in modo sincronizzato. Durante il metodo di run() estraiamo un evento dalla coda ad eventi e lo processiamo tramite processEvent che gestisce la logica di elaborazione degli eventi ricevuti.

GreenHouse Edge - ESP32

Per simulare la rilevazione dell'umidità del terreno, abbiamo collegato un potenziometro analogico al nostro modulo ESP in modalità station, la quale consente al modulo di connettersi come un dispositivo client in collegamento ad un AccessPoint WiFi, dove si collega trasmettendo i rilevamenti al nostro PC all'indirizzo **192.168.1.12:8080**.
Il sorgente fa uso principalmente di due librerie:

- **HTTPClient.h**
  - Libreria per le richieste HTTP indirizzate verso un server web.
    - Tramite **http.begin**(address + "/api/data") inizializziamo la connessione all'url formato d indirizzo (http://192.168.1.12:8080 a cui aggiungiamo un path /api/data)
    - **http.POST**(msg) viene utilizzato per inviare i messaggi in formato JSON attraverso il collegamento WiFi creato.In questo caso vengono inviati i valori ricevuti dal potenziometro, e aggiungiamo un posto "home", come secondo campo dell'oggetto JSON, per individuare da quale parte del codice il nostro evento viene inviato al server.
    - **http.end()** è invocato per liberare le risorse dopo averle utilizzate.
- **WiFi.h**
  - Libreria che fornisce le funzioni necessarie a gestire la connessione WiFi, configurando le impostazioni e monitorando lo stato della connessione.
    - Tramite **wifi.begin**(ssid,password) avviamo il processo di connessione
    - Utilizzando **wifi.status**, verifichiamo se l'ESP è connesso ad una rete.

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

Come campi fondamentali abbiamo:

- **ssidName**
  - Contiene il SSID del nostro modem a cui il modulo ESP32 si dovrà collegare.
- **pwd**
  - Contiene la password del modem.
- **address**
  - Per avere il giusto collegamento è necessario scoprire l'indirizzo specifico della macchina su su cui il server sarà eseguito, nel codice abbiamo impostato una stampa con l'indirizzo IP specifico della macchina, che dovrà essere inserito all'interno di questo campo per il giusto funzionamento dell'applicativo.

GreenHouse Front End

Per consentire all'utente di analizzare e tenere traccia dei rilevamenti effettuati dalla nostra applicazione si è creata una pagina web che riceve dei dati tramite delle funzioni AJAX.
La pagina web si presenta in due parti principali rispettivamente:

- Per i rilevamenti automatici con sfondo light-blue sulla sinistra,
- I comandi manuali con sfondo light-green sulla destra
- Nel lato sinistro in alto, si mantiene traccia dello stato della pompa, sia in automatico che in manuale.

- **Rilevamenti modulo ESP**
  - I rilevamenti effettuati dal sensore di umidità simulato con un potenziometro, e poi inoltrati tramite Wi-Fi questi rilevamenti direttamente all'indirizzo "**http://192.168.1.12:8080/api/data**".
    Dove tramite la GET di cui sopra, vengono ricevuti e visualizzati concretamente all'interno della view gestita dalla pagina web.
  - Le fetch Data eseguite all'interno dello script, riescono ad ottenere tramite delle richieste GET, indirizzate all'indirizzo **/api/data** del server, i rilevamenti.
  - Le fetch Data hanno un periodo di 1000ms dove allo scadere, vengono richiamate, una volta ricevuti i dati in formato JSON, li passano ad una funzione che li elabora, estrapolando e facendo visualizzare nel formato corretto i dati, nel placeholder giusto.
  - A seconda del valore alla chiave "place" dell'oggetto JSON, inseriamo delle classi per aggiungere degli effetti stilistici, come per esempio, un dato che proviene da "server" ha come sfondo light-red per notificare l'interruzione d'erogazione della pompa dell'acqua.
- **Stato pompa**
  - In questo caso una fetch Supply State tramite una richiesta GET all'indirizzo: "**http://192.168.1.12:8080/api/supplystate**" riceve i dati sullo stato della pompa, e sulla durata della sua ultima erogazione, questi dati vengono

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

visualizzati in modo dinamico in base allo stato della pompa, nel caso la pompa sia chiusa, viene visualizzato il valore di pompa chiusa e la durata di questo stato, invece nel caso sia aperta la pompa non visualizza nessun dato sull'erogazione.

- **Notifiche bluetooth**
  - Quando il nostro sistema è in modalità manuale, e connesso ad un dispositivo mobile tramite bluetooth, gestiamo una fetch Data Bluetooth che all'indirizzo "**http://192.168.1.12:8080/api/bluetooth**" riceve i dati in formato JSON e li estrapola, assegnandogli alle strutture dati che li visualizzano nel giusto ordine, come scelta implementativa abbiamo deciso di inserire i dati come una coda inserendo in fondo, tenendo così uno storico dal primo valore inviato al quinto che è in erogazione, abbiamo scelto di tenere traccia di solo 5 elementi, poiché stimiamo che un sistema automatico, in quanto tale, debba ricevere meno interventi esterni possibili, anche per una politica indirizzata all'ottimizzazione della parte automatica, garantendo sicurezza al cliente o azienda che ha commissionato questo sistema d'irrigazione, in eventuali ulteriori e future nuove implementazioni questi valori qui storicizzati e memorizzati possono essere inseriti in un grafico Real Time che favorisce la consultazione e l'analisi del sistema da remoto.
  - I dati bluetooth mantengono al loro interno tutte le informazioni salienti, per gestire correttamente la durata dell'erogazione per ogni dato inviato.

GreenHouse Mobile App - Android

L'applicativo mobile, sfrutta le librerie utilizzate a lezione precisamente quelle dell a.a. 2022. Avendo un dispositivo Lenovo Tablet con un sistema operativo Android 11, abbiamo optato per utilizzare delle versioni di gradle:

- Android Gradle Plugin **8.4.1**
- Java **8**
- Gradle **8.6**
- SDK **API 30 per Android 11**

Senza l'uso di queste specifiche, ottenevamo degli errori di configurazione che non ci permettevano l'utilizzo di determinate funzioni compatibili con un sistema operativo Android 11.
La mobile app si presenta con uno stile molto semplice e intuitivo, consente di collegarsi tramite un pulsante al dispositivo bluetooth associato in precedenza; il dispositivo della applicazione si chiama "SmartGreenHouse" e funge da server Bluetooth,a cui le richieste di connessione vengono indirizzate, una volta ottenuta la connessione il sistema parte aggiornando la UI view con gli elementi che formano il pannello di controllo remoto della serra smart.
Si può quindi visualizzare lo stato della connessione, il dispositivo a cui è associata, corredata da un valore centrale relativo all'umidità rilevata da un sensore simulato da un potenziometro.

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

Sistemi Embedded e Internet-Of-Things

Smart GreenHouse A.A. 2018-2019

Nicholas Ciarafoni
Giovanni Perini

- Main Activity
    - Contiene tutte le variabili utilizzate per ancorare gli elementi disposti nella view, tramite i container layout appositi, queste variabili vengono poi riorganizzate tramite delle proprietà specifiche, a seconda se si è connessi, se è scollegato il dispositivo o se si vuole passare da una modalità automatica a quella manuale.
    - Ogni parte della mobile app, si attiva e si disattiva in base allo stato in cui l'app verte in quell'istante, precisamente se si trova in modalità automatica, visualizza sempre un valore centrale d'umidità, e dei pulsanti che consentono di passare alla modalità manuale, o di disconnettersi dall'applicazione, o di scollegare il bluetooth.
      Se si trova in uno stato manuale consente all'utente di gestire una SeekBar, per impostare un valore che corrisponde il litraggio al minuti di una serra, essendo che la portata massima di un rubinetto collegato ad un acquedotto di una abitazione varia da 5 ai 15 litri al minuto, considerando una serra di svariate piante, abbiamo optato di mantenere i valori **da 0 a 100** come litri al minuto, nel caso l'utente decidesse di impostare un valore pari a 0, la pompa si chiude, per ogni altro valore che viene impostato, il dato verrà visualizzato sia a console lato server, che tramite la pagina web collegata.
    - L'utente è completamente libero di mettere quanti più dati vuole, l'applicativo riesce a gestirli in modo lineare, senza creare rallentamenti o cogestione di eventi, ogni valore viene disposto in maniera continuativa al precedente, senza perdite di eventi.

La view, potrà essere soggetta ad ulteriori aggiunte, come per il lato front end, su cui non ci siamo voluti soffermare più di tanto, in quanto il nostro obiettivo era capire più la dinamica che sta alla base del sistema in sé, piuttosto che aggiungere features non necessarie in termini di consegna, anche se il sistema è progettato per rimanere saldo a nuove aggiunte future.

- Utils.c
    - Contiene dei file di configurazione interni per impostare il nostro server name, e UUID.
- bt-lib
    - Unica directory dei sorgenti utilizzata per gestire un canale bluetooth reale, sono state applicate alcune modifiche sullo stato della porta, e aggiunti dei log di controllo e gestione delle eccezioni.
    - Bluetooth Channel
        - public boolean **isClosed()**
            - Usata per verificare se il worker è stato cancellato o nullo
    - RealBluetoothChannel
        - **isClosed()**

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

        - Modifica un Flag **running**, usato dal isClosed() della classe precedente per assicurarci che la socket sia chiusa.
    - **closeSocket()**
        - Se abbiamo una socket la chiude, e stampa a video un LOG in caso di successo, se si verificano delle eccezioni, come la socket è già chiusa o comunque ha un valore impostato a null, stampa a video l'errore.
    - **run()**
        - Modificato il run con il controllo sulla socket, finchè abbiamo una socket attiva, con running, esegue la lettura, poi estrapola i dati, invece se la socket è chiusa, non legge alcun valore dal canale bluetooth.
- Manifest
    - Sono stati inseriti i permessi generici per l'utilizzo del bluetooth,consentono di non dover gestire tutte le politiche relative ai permessi, checkPermission() delle API 33.
    - Gestiamo l'activity .MainActivity
- values
    - Contiene la gran parte dei placeholder delle stringhe, che sono state utilizzate per la view, modificabili e traducibili in più lingue, adatte per un sistema volto ad un pubblico di varie nazionalità su scala globale.
- Colors & Styles
    - Configurano lo stile e i colori della applicazione.

GreenHouse Controller - Arduino

Utilizzando il microcontrollore Arduino abbiamo realizzato una macchina a stati finiti asincrona che gestisce la simulazione di irrigazione di una serra in modalità automatica, offrendo la possibilità di intervenire manualmente sul sistema, utilizzando un dispositivo mobile; l'utente con il proprio dispositivo, una volta trovatosi ad una determinata distanza, comunica dati al modulo HC-06 tramite bluetooth, in questo modo Arduino si occupa anche di fare da ponte per la trasmissione di messaggi tra la ***Mobile App*** e la parte ***Server*** del sistema.

Per svolgere tale compito abbiamo implementato due task:

- **AutomaticState**(*ledAuto*, *ledPump*, *Sonar proxy*, *pState*, *pump*):
  Questo task descrive il comportamento iniziale del sistema che parte dallo stato automatico. Al costruttore vengono passati 5 parametri i quali rappresentano oggetti che verranno poi utilizzati nel task:

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

- ***ledAuto*** : oggetto di tipo Led che può essere solo spento o acceso, e rappresenta lo stato del sistema in automatico.
- ***ledPump*** : oggetto di tipo Led il quale viene utilizzato per simulare la portata (litri al minuto) che fuoriesce dalla pompa, pertanto il led può assumere 3 livelli di intensità in questo task (*Pmin*, *Pmid*, *Pmax*).
- ***proxy*** : oggetto di tipo Sonar il quale viene utilizzato per controllare se il dispositivo bluetooth, si trova ad una distanza adeguata dal sistema.
- ***pState*** : oggetto di tipo ShareState, si occupa di segnalare il passaggio da uno stato ad un altro, utilizzando i propri metodi *setAutomatic*() e *setManual*().
- ***pump***: oggetto di tipo ServoPump, il servo viene utilizzato per simulare l'apertura della pompa per l'irrigazione, può assumere 4 posizioni (**closePump()**, ***CAPACITY_MIN***, ***CAPACITY_MED***, ***CAPACITY_MAX***).

Il task effettua l'accensione del ledAuto, successivamente svolge un controllo sul canale di comunicazione bluetooth, ove vi sia un messaggio in arrivo (e non vuoto), controllandone il contenuto (che deve essere uguale a "B") e la distanza percepita dal sonar(che dovrà essere minore di DIST), se il controllo dovesse andare a buon fine, verrà inviata al Server java la richiesta di passare in modalità manuale, e la segnalazione alla **Mobile App** che la richiesta ha avuto esito positivo.
Se è in arrivo un messaggio (non vuoto) sul canale seriale, svolgiamo una manipolazione della stringa con il metodo "*substring()",* ricavandone due variabili distinte, la prima rappresenta l'intestazione del messaggio, che tramite la funzione *switch(**) imposterà l'intensità del ledPump e l'apertura della pompa, oppure cambierà stato del sistema, mentre l'altra variabile contiene i dati dell'umidità inviati da ESP (tramite il server java) che verranno instradati verso la **Mobile App**.

- **ManualState** (*ledManual*, *ledPump*, *proxy*, *pState*, *pump*):

  Questo task descrive il comportamento dello stato manuale del sistema. Anche a questo costruttore vengono passati 5 parametri:

  - ***ledManual*** : oggetto di tipo Led che può essere solo spento o acceso, e sta a segnalare che lo stato del sistema è manuale.
  - ***ledPump*** : oggetto di tipo Led, viene utilizzato per simulare la portata (litri al minuto) che fuoriesce dalla pompa, pertanto il led può assumere tutto lo spettro dell'intensità (da 0 a 255) in questo task.
  - ***proxy*** : oggetto di tipo Sonar, viene utilizzato per controllare se il dispositivo si trova ad una distanza adeguata dal sistema.
  - ***pState*** : oggetto di tipo ShareState, si occupa di segnalare il passaggio da uno stato ad un altro utilizzando i propri metodi *setAutomatic*() e *setManual*().
  - ***pump*****:** oggetto di tipo ServoPump, il servo viene usato per simulare l'apertura della pompa per l'irrigazione, può assumere tutte le

Nicholas Ciarafoni
Giovanni Perini

posizioni possibili del servo, che in questo caso sono da **750**(sta per 0) a **2250**(180) poiché usiamo la libreria **ServoTimer2** per evitare conflitti di timer, all'interno delle librerie **Timer.h** e **Servo.h.**

Anche in questo task come nel precedente, il sistema effettua l'accensione del ledManual per segnalare lo stato del sistema e controlla il canale di comunicazione bluetooth, in caso sia presente un messaggio (non vuoto) esso verrà subito redirezionato al server java tramite il canale di comunicazione seriale.
Verifica se il dispositivo bluetooth è ancora all'interno della distanza accettabile dal sistema (*DIST*), se non lo è, viene inviata una segnalazione al server per cambiare stato.
Controlla il canale seriale se è in arrivo un messaggio (non vuoto), mediante una manipolazione del payload come nel task precedente per separare l'intestazione dal messaggio, solo che in questo task è presente un controllo maggiore sulla stringa per rilevare la presenza di una diversa intestazione, così da non incorrere nella perdita di nessun evento.
Le possibilità all'interno della funzione switch sono 3:

- Se l'intestazione è **'p'** significa che i dati che lo seguono si riferiscono alla portata con cui va aperta la pompa.
- Se l'intestazione è **'r'** significa che i dati vanno inviati sul canale bluetooth.
- Se l'intestazione è **'a'** significa che lo stato deve passare ad automatico.

★ **Canali di comunicazione**:
I due canali di comunicazione sopra citati, sono di due metodi di trasmissione uno seriale ed uno bluetooth, vengono gestiti rispettivamente dalle classi **MsgServiceBT** e **MsgService**.
Entrambe le classi possiedono gli stessi metodi fondamentali:

- ***sendMsg****(Msg msg)*: invia il messaggio sul canale bluetooth o seriale.
- ***receiveMsg****()*: riceve il messaggio in byte, fino all'arrivo del carattere di terminazione della stringa.
- ***isMsgAvailable****()*: controlla se c'è un messaggio in arrivo sul canale.

★ La classe **SharedState** viene utilizzata per tenere traccia dello stato attuale del sistema, tramite due variabili booleane che non saranno mai vere contemporaneamente e vengono impostate tramite due metodi setAutomatic() (setta lo stato del sistema ad automatico) e setManual() (setta lo stato del sistema a manuale).
★ La classe **Led** definisce il comportamento degli oggetti del suddetto tipo, utilizzando i metodi *switchOn*() e *switchOff*() rispettivamente per accendere e spegnere i led, mentre *setIntensity*(int *) viene usato per impostare la luminosità ed è utilizzato solo da ledPump.
★ La classe **Sonar** possiede un solo metodo, *getDistance*() il quale ritorna la distanza di un oggetto dal sonar, esso funziona inviando un impulso e calcolando il tempo

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

che l'impulso impiega a ritornare dopo aver "rimbalzato" sulla superficie dell'oggetto.

- ★ La classe **ServoPump** possiede solo due metodi: uno per chiudere la pompa closePump() (settare l'angolo del servo a 0) e uno per settare una portata generica setAngle().
- ★ Le classi **Task**, **Scheduler** e **Timer** si occupano di definire i task, il tempo che intervalla tra una chiamata alla funzione tick() e la successiva, e di organizzarne le chiamate in una lista contenente le task ed il periodo trascorso dalla sua ultima esecuzione che verrà aggiornata continuamente.

Conclusioni

Sebbene abbiamo impiegato più tempo del necessario per portare a termine questo assignment, siamo riusciti a comprendere gran parte delle componenti interne ad un sistema IoT.
L'utilizzo di nuove tecnologie come *Vert.x*, ci hanno fornito delle domande e altrettante risposte sul funzionamento di un server Java, essendo per noi il primo interfacciamento ad un sistema complesso come questo, è stata una bella sfida che ci ha accompagnato per diverso tempo.
Lato *Arduino* è stato interessante comprendere l'uso dello scheduler, i diversi timer, e la gestione delle stringhe nei canali trasmissivi.
Il progetto ci ha aperto gli occhi su quanto lavoro e quanta ricerca occorre effettuare per giungere alla soluzione di una problematica, niente meno che ad una successione di problematiche di diversa complessità.
Lato *Android*, abbiamo avuto qualche intoppo, soprattutto per capire, come l'applicazione doveva gestire i permessi per il bluetooth, e grandi problemi iniziali con l'uso di *Gradle*, e delle sue versioni, in quanto in base alla versione del gradle e delle relative *API* scelte, dovevamo inserire all'aumentare dell'*API*, permessi sempre più specifici, quando nelle versioni precedenti (le stesse utilizzate da noi), bastavano permessi generici che ci hanno consentito di utilizzare il file 2022 delle esercitazioni di laboratorio.
Sono state utilizzate slide e lezioni registrate per avere il giusto bagaglio teorico dell'anno corrente per l'elaborato, e alcune degli anni successivi.

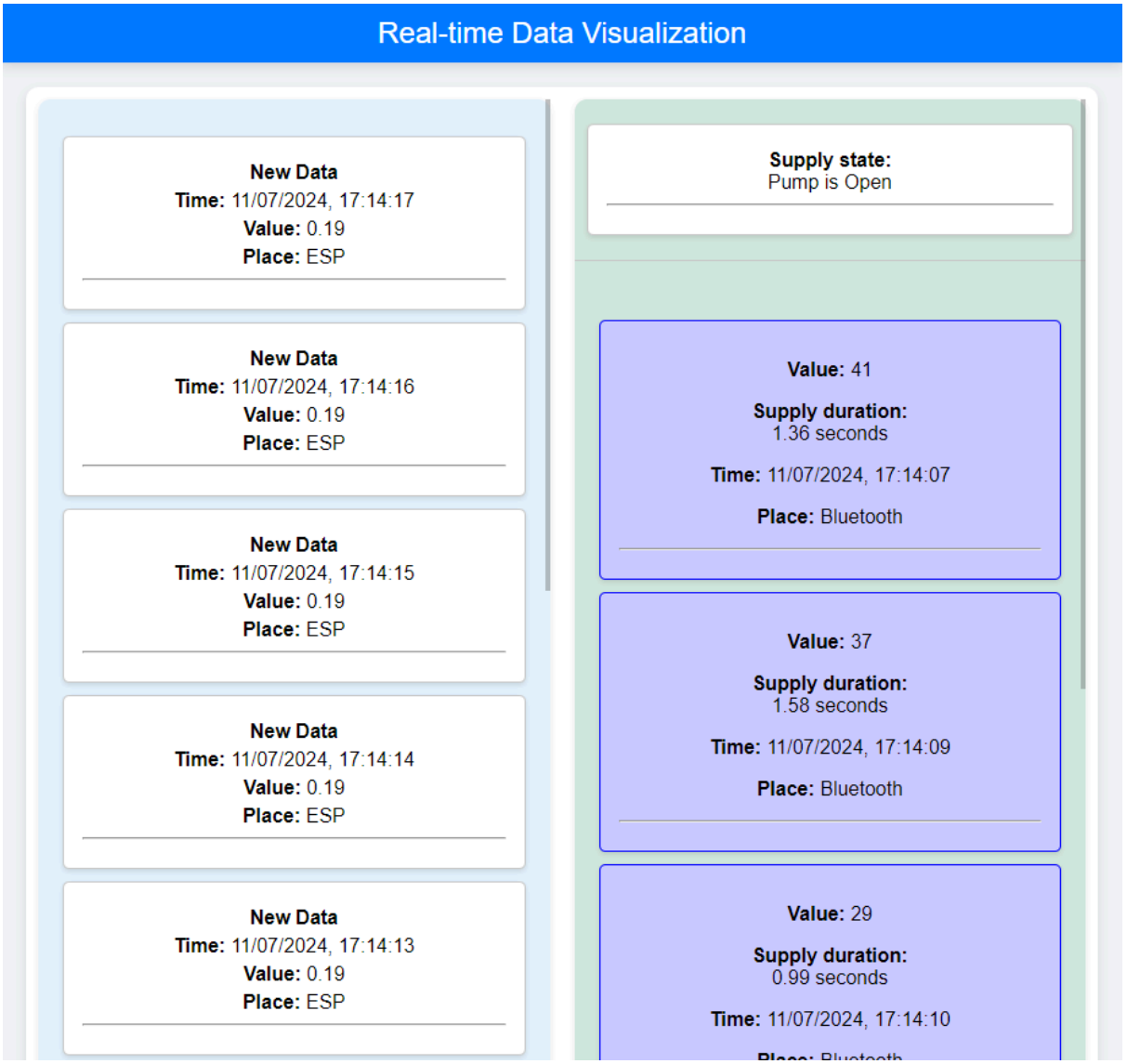
Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

Sistemi Embedded e Internet-Of-Things

Smart GreenHouse A.A. 2018-2019

Nicholas Ciarafoni
Giovanni Perini

Schermata pagina web, responsive per dispositivi desktop:



Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git

Sistemi Embedded e Internet-Of-Things

Smart GreenHouse A.A. 2018-2019

Nicholas Ciarafoni
Giovanni Perini

Schermata pagina web,responsive per dispositivi iPad mini:

Link repository GIT:
https://github.com/7VaGe/smartGreenHouse.git