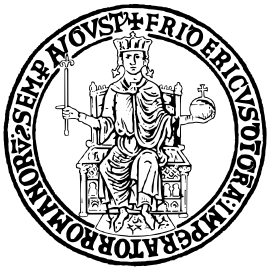
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Implementazione e validazione di un sistema On-Board per l'ausilio al conducente su veicoli connessi tramite Bluetooth Low Energy e App Flutter

Relatore
Prof. Porfirio Tramontana

Correlatori
Dott. Gennaro Galante
Ing. Enrico Landolfi

Candidata
Valeria Moriello
N86/2139

Anno Accademico 2022–2023

Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Implementazione e validazione di un sistema On-Board per l'ausilio al conducente su veicoli connessi tramite Bluetooth Low Energy e App Flutter

Relatore
Prof. Porfirio Tramontana

Correlatori
Dott. Gennaro Galante
Ing. Enrico Landolfi

Candidata
Valeria Moriello
N86/2139

Anno Accademico 2022–2023

A te che mi hai dato il coraggio di intraprendere questo percorso di studi.
Lo dedico a te papà, che hai messo i miei sogni prima di ogni altra cosa.

Indice

Introduzione

Nel dinamico e in continua evoluzione scenario dei trasporti connessi, la ricerca e lo sviluppo di soluzioni innovative emergono come elementi fondamentali per ottimizzare l'esperienza di guida e garantire elevati standard di sicurezza stradale. Questa tesi si propone di esplorare e implementare un sistema On-Board, basato su Bluetooth Low Energy (BLE) e implementato tramite un'applicazione Flutter, sviluppata in collaborazione con *NetCom Group S.p.A.*.

Contestualizzazione del Problema

L'origine di questo progetto risiedeva in un obiettivo iniziale: sviluppare un'interfaccia per la visualizzazione di dati provenienti da una Raspberry Pi 3B+. Tuttavia, la complessità del contesto e le esigenze emergenti hanno determinato una ridefinizione completa del progetto. L'attuale versione dell'applicazione rappresenta una transizione strategica dal Bluetooth convenzionale a una più avanzata e efficiente connessione Bluetooth Low Energy (BLE). È importante notare che, pur mantenendo la Raspberry Pi come riferimento nel contesto della tesi, nella realtà operativa è parte di un'On-Board Unit (OBU), elemento chiave nella trasformazione del sistema.

Obiettivi della Ricerca

Il cuore di questa ricerca è costituito dalla progettazione e dall'implementazione di un sistema versatile ed efficiente, sfruttando appieno le potenzialità del Bluetooth Low Energy (BLE) e nella realizzazione di un'app creata mediante *Flutter*. La migrazione dal Bluetooth tradizionale al BLE ha presentato significative sfide, affrontate con un approccio analitico e una progettazione attenta. Inoltre, particolare enfasi è dedicata all'organizzazione del codice in *Flutter*, strutturato per massimizzare la manutenibilità e la scalabilità attraverso l'adozione di una *clean architecture* e del design pattern denominato *Cubit*.
Attraverso l'analisi approfondita dei requisiti, studi di fattibilità, progettazione del sistema e rigorosi test di comunicazione, questa tesi si propone di fornire una base solida per il successo dell'applicazione nel suo contesto operativo. L'obiettivo finale è contribuire all'evoluzione delle tecnologie di ausilio al conducente, offrendo una soluzione avanzata e adattabile alle esigenze in continua evoluzione del settore automotive.

–1–
Bluetooth e Bluetooth Low Energy

Contents: **1.1 Panoramica Bluetooth.** 1.1.1 Storia e sviluppo del Bluetooth – 1.1.2 Caratteristiche principali del Bluetooth – 1.1.3 Applicazioni comuni del Bluetooth. **1.2 Evoluzione verso Bluetooth Low Energy (BLE).** 1.2.1 Motivazioni e vantaggi. **1.3 Specifiche tecniche del Bluetooth Low Energy.** 1.3.1 Introduzione a Generic Profiles – 1.3.2 UART over Bluetooth Low Energy: emulazione di una porta UART attraverso BLE – 1.3.3 UART over Bluetooth Low Energy: funzionamento.

1.1 Panoramica Bluetooth

Il Bluetooth è una tecnologia di comunicazione wireless a corto raggio sviluppata per consentire lo scambio di dati tra dispositivi vicini. Introdotta nel 1994 da Ericsson, questa tecnologia è diventata un elemento chiave nella connettività di dispositivi mobili e dispositivi elettronici.

Figura 1.1: Logo Bluetooth

1.1.1 Storia e sviluppo del Bluetooth

Il Bluetooth ha una storia di sviluppo che risale ai primi anni '90 quando la necessità di creare una connessione wireless più semplice tra dispositivi è emersa. *Ericsson*, una società di telecomunicazioni svedese, fu pioniera nello sviluppo di questa tecnologia, che doveva essere un'alternativa senza fili alle connessioni a infrarossi. Nel 1998, il primo standard Bluetooth fu ufficialmente presentato e da allora ha subito diversi aggiornamenti per migliorare le prestazioni e l'efficienza energetica.

1.1.2 Caratteristiche principali del Bluetooth

Il Bluetooth opera nella banda di frequenza *ISM (Industrial, Scientific, Medical)* a 2,4 GHz e utilizza la tecnologia di spread spectrum per evitare interferenze con altre reti wireless. La tecnologia Bluetooth consente la comunicazione punto a punto o multipunto, consentendo a un dispositivo di connettersi a più dispositivi contemporaneamente. Inoltre, offre una gamma di servizi, tra cui trasmissione di dati, trasmissione vocale, e connessioni di rete.

1.1.3 Applicazioni comuni del Bluetooth

Il Bluetooth è onnipresente in una vasta gamma di dispositivi e applicazioni. Alcuni esempi comuni includono:

- **auricolari Bluetooth e dispositivi audio** che consentono la trasmissione senza fili di audio da sorgenti come smartphone o computer a dispositivi di riproduzione audio;
- **dispositivi di localizzazione** che permettono di tracciare la posizione di oggetti o persone in ambienti chiusi;
- **automobili connesse** in cui il Bluetooth è spesso integrato nei veicoli per consentire la connessione di telefoni cellulari e lo streaming audio;
- **dispositivi medici** che utilizzano tale tecnologia per la trasmissione di dati da dispositivi di monitoraggio a dispositivi di visualizzazione o archiviazione dati.

Questa breve panoramica del Bluetooth offre una visione generale della storia, delle caratteristiche principali e delle applicazioni comuni di questa tecnologia di comunicazione wireless. Nei paragrafi successivi, ci concentreremo sulla sua evoluzione verso la versione a basso consumo energetico, il **Bluetooth Low Energy (BLE)**.

1.2 Evoluzione verso Bluetooth Low Energy (BLE)

Con il progresso incessante delle tecnologie wireless, il Bluetooth Low Energy (BLE) è emerso come una pietra miliare nell'evoluzione della connettività senza fili. Questa sezione si focalizza sull'importante transizione da Bluetooth a BLE, esplorando le ragioni

dietro questo cambiamento e mettendo in luce le caratteristiche distintive che hanno reso il BLE una scelta preferita in molteplici contesti, inclusi veicoli connessi e applicazioni mobile avanzate.

1.2.1 Motivazioni e vantaggi

L'introduzione del Bluetooth Low Energy è stata guidata da una crescente richiesta di connettività wireless efficiente, soprattutto in scenari in cui la conservazione energetica è cruciale. In confronto al Bluetooth tradizionale, il BLE ha introdotto significative ottimizzazioni, rendendolo particolarmente adatto per applicazioni con basso consumo energetico. Questa transizione è stata motivata da diverse esigenze, tra cui:

- **il consumo energetico ridotto** è delle caratteristiche distintive del BLE è la sua capacità di operare con un consumo energetico notevolmente inferiore rispetto alle versioni precedenti di Bluetooth. Questo rende il BLE ideale per dispositivi alimentati a batteria, promuovendo la durata della batteria in applicazioni critiche;

- **la comunicazione a basso volume di dati** rende il BLE adatto per dispositivi come sensori IoT, dispositivi medici e applicazioni in ambito automobilistico;

- **le connessioni veloci e disconnessioni automatiche** supportati da tale tecnologia quando non è richiesta una connessione attiva. Questo contribuisce a ottimizzare ulteriormente il consumo energetico;

1.3 Specifiche tecniche del Bluetooth Low Energy

Il Bluetooth Low Energy (BLE) è caratterizzato da specifiche tecniche che lo rendono un'opzione ideale per una vasta gamma di applicazioni, dall'*Internet of Things (IoT)* agli ambienti automobilistici e oltre. Questa sezione si propone di esaminare le specifiche tecniche chiave del BLE, fornendo una panoramica approfondita delle sue capacità e limitazioni.

Inizialmente noto come *Bluetooth Smart*, è una tecnologia sviluppata appositamente dal gruppo Bluetooth *SIG* per soddisfare le esigenze di applicazioni che richiedono una trasmissione wireless con un consumo energetico inferiore rispetto al Bluetooth tradizionale, mantenendo al contempo un bit rate più elevato. Il progetto ha avuto origine nel 2001 e ha visto la sua commercializzazione nel 2006 sotto il nome di *Wibree* da parte di *Nokia*. Nel 2007, il marchio è stato integrato nelle specifiche Bluetooth. L'inclusione ufficiale nel Bluetooth versione 4.0 è avvenuta all'inizio del 2010, con i primi dispositivi che implementavano questa caratteristica che sono stati lanciati nel 2011. Attualmente, il Bluetooth Low Energy si basa sulle specifiche del Bluetooth 4.1, rilasciate nel dicembre 2013.

Figura 1.2: Logo Bluetooth Smart

1.3.1 Introduzione a Generic Profiles

i *Generic Profiles* rappresentano un insieme standardizzato di funzionalità e comportamenti che consentono ai dispositivi di comunicare in modo coerente e interoperabile. Questi profili forniscono una struttura comune per definire le capacità e le caratteristiche di dispositivi specifici, garantendo una maggiore flessibilità e adattabilità nell'implementazione di servizi e applicazioni.
Due componenti chiave che guidano l'implementazione dei profili BLE sono il **Generic Access Profile (GAP)** e il **Generic Attribute Profile (GATT)**. Ciascuno di essi svolge un ruolo cruciale nella definizione delle interazioni tra dispositivi, dalla gestione della connessione alla trasmissione di dati specifici.
Il *GAP* assume la responsabilità della gestione della connessione e della fase di *advertising*, rendendo il dispositivo visibile all'ambiente circostante e regolando le interazioni con altri dispositivi. Nell'ambito del *GAP*, vengono definiti due ruoli chiave che possono essere assunti dai dispositivi connessi, contribuendo alla dinamica della comunicazione BLE:

- **Peripheral**: questo ruolo designa un dispositivo visibile agli altri attraverso la fase di *advertising*, consentendo la ricezione di connessioni in ingresso (*slave*). Tuttavia, un dispositivo con questo ruolo non può iniziare autonomamente una connessione. Tipicamente, i dispositivi di sensoristica ricoprono questo ruolo, fornendo dati a dispositivi centrali.

- **Central**: in contrasto, un dispositivo con il ruolo di Central è in grado di cercare dispositivi visibili e di iniziare una connessione *(master)*. Non può, tuttavia, accettare connessioni in ingresso. Questo ruolo è spesso assegnato a smartphone, tablet o altri dispositivi che si connettono ai sensori per acquisire dati.

Un aspetto cruciale da notare è che due dispositivi che ricoprono lo stesso ruolo non possono stabilire una connessione diretta tra di loro. Inoltre, un dispositivo *Peripheral* ha la capacità di connettersi a un solo dispositivo *Central* alla volta. È interessante notare che è anche possibile creare una topologia di tipo broadcast, consentendo l'invio di dati direttamente nel pacchetto di *advertising*, rendendoli visibili a tutti gli altri dispositivi nelle vicinanze.
Sono così definite due diverse topologie che rappresentano due modalità di comunicazione distintive:

- **Topologia connessa** nella quale i dispositivi BLE stabiliscono connessioni dirette uno-a-uno. Ad esempio, un dispositivo Central può connettersi a uno specifico dispositivo Peripheral. Questa connessione bidirezionale permette lo scambio affidabile e continuo di dati tra i dispositivi connessi.

- **Topologia broadcast** nella quale i dati vengono inviati in modo unidirezionale e non richiedono connessioni dirette. Un dispositivo BLE trasmette i dati attraverso il pacchetto di advertising, rendendoli visibili a tutti gli altri dispositivi nelle vicinanze.

Di seguito vengono mostrate le due possibili topologie.

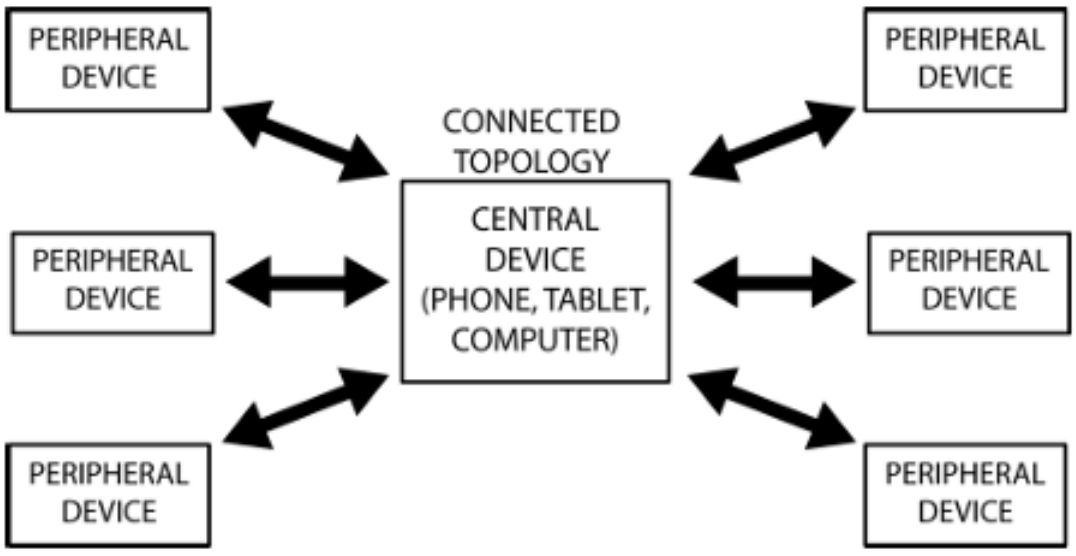


Figura 1.3: Topologia connessa

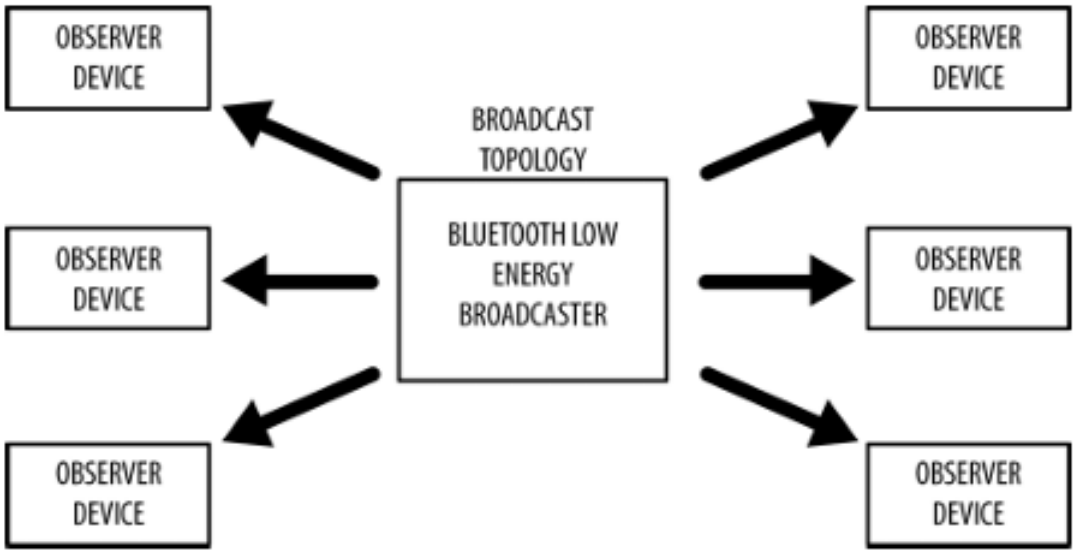


Figura 1.4: Topologia broadcast

Il **Generic Attribute Profile (GATT)**, d'altra parte, definisce la struttura e il protocollo di comunicazione per la trasmissione di dati tra dispositivi connessi.
Il *GATT* introduce il concetto di **Attributes** (Attributi) organizzati in una struttura gerarchica che include **Services** (Servizi) e **Characteristics** (Caratteristiche). I servizi rappresentano le funzionalità di alto livello offerte dal dispositivo, mentre le caratteristiche contengono i dati effettivi e le informazioni sui dati associati a ciascun servizio. In particolare:

- i **servizi** raggruppano caratteristiche correlate all'interno di un dispositivo BLE e definiscono un insieme logico di funzionalità. Ogni servizio è identificato da un **Universal Unique Identifier (UUID)** e può contenere una o più caratteristiche. I servizi semplificano la struttura di un dispositivo, organizzando le funzionalità correlate in gruppi coerenti;

- le **caratteristiche** rappresentano singoli elementi di dati o operazioni all'interno di un servizio BLE. Ciascuna caratteristica è identificata da un *UUID* e può essere di tipo *read* o *write*, determinando se è possibile leggere o scrivere i dati associati. Le caratteristiche sono fondamentali per la trasmissione e la ricezione di dati tra dispositivi BLE.

Inoltre è possibile definire dei ***GATT* characteristic descriptors**, più comunemente detti **descriptors** (descrittori). Essi sono metadati associati alle caratteristiche. Forniscono informazioni aggiuntive o dettagli di una specifica caratteristica e ne definiscono il comportamento o le proprietà. In altre parole, un descrittore fornisce ulteriori informazioni

su come interpretare o utilizzare una particolare caratteristica associata a un dispositivo BLE.

In breve, il *GATT* facilita la definizione di servizi e caratteristiche standardizzati, consentendo a dispositivi diversi di comunicare in modo comprensibile e coeso. Questi profili *generic* facilitano l'implementazione di applicazioni personalizzate e servizi su piattaforme BLE, contribuendo all'interoperabilità tra dispositivi di marche diverse.

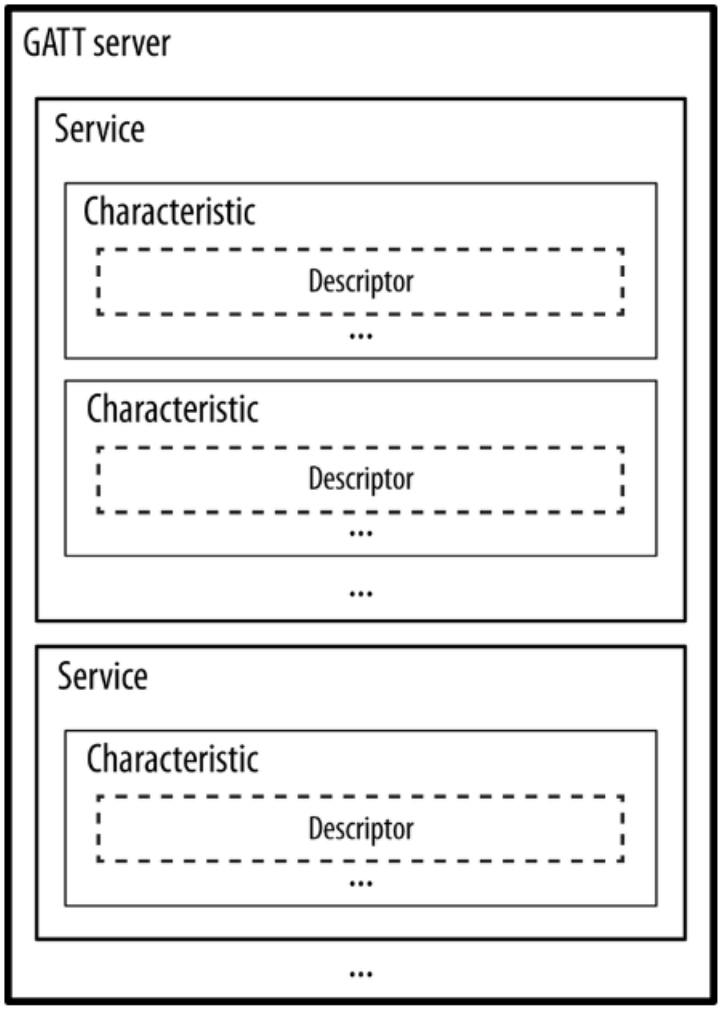


Figura 1.5: Gerarchia GATT

Come qualsiasi altro protocollo o profilo nelle specifiche Bluetooth, il *GATT* inizia definendo i ruoli che i dispositivi interagenti possono assumere:

- **Client**: invia richieste a un server e riceve risposte (e aggiornamenti inizializzati dal server) da esso. Il client *GATT* non conosce nulla in anticipo riguardo agli attributi del server, quindi deve prima informarsi sulla presenza e sulla natura di quegli attributi eseguendo la scoperta del servizio. Dopo aver completato la scoperta del

servizio, può quindi iniziare a leggere e scrivere gli attributi trovati nel server, oltre a ricevere aggiornamenti inizializzati dal server.

- **Server**: riceve richieste da un client e invia risposte; è responsabile di memorizzare e rendere disponibili i dati dell'utente al client, organizzati in attributi. Ogni dispositivo BLE venduto deve includere almeno un server *GATT* di base che può rispondere alle richieste del client, anche se solo per restituire una risposta di errore.

1.3.2 UART over Bluetooth Low Energy: emulazione di una porta UART attraverso BLE

Il protocollo **Universal Asynchronous Receiver/Transmitter (UART)** è uno dei protocolli più diffusi per la comunicazione con dispositivi computerizzati attraverso una porta seriale. Si tratta di una tecnologia ampiamente utilizzata per la trasmissione di dati asincrona tra dispositivi e rappresenta uno standard consolidato per le comunicazioni seriali.
Quando si parla di *UART* in un contesto moderno, ci si riferisce spesso all'emulazione di una porta seriale attraverso la tecnologia Bluetooth Low Energy. In questa transizione, l'*UART* tradizionale, utilizzato per la comunicazione su porta seriale, viene adattato e implementato su una connessione BLE. Questa evoluzione consente di sfruttare le caratteristiche avanzate e il basso consumo energetico offerti dalla tecnologia BLE.
L'emulazione di *UART* mediante un server *GATT* rappresenta un passo chiave in questa transizione. Mentre il *GATT* è comunemente associato alla gestione di attributi e servizi BLE, l'implementazione di un servizio *UART* personalizzato consente di emulare il comportamento di una porta seriale attraverso il framework *GATT*.

In questa prospettiva, è cruciale comprendere che questa emulazione di *UART* non fa parte dei profili definiti da *Bluetooth Special Interest Group (SIG)*. Si tratta di un servizio definito su misura, ampiamente accettato dalla comunità. Questo servizio personalizzato, sebbene non ufficialmente standardizzato, ha guadagnato una vasta adozione nel settore BLE. Hardware provider come *Nordic*, *Adafruit*, *mbed* e molti altri supportano questa emulazione, rendendola una scelta popolare per la comunicazione seriale su piattaforme BLE.
L'emulazione di *UART* su BLE offre numerosi vantaggi, tra cui l'estensibilità, la flessibilità e l'efficienza energetica. Questa tecnologia è particolarmente adatta per applicazioni *IoT*, sensori wireless e dispositivi indossabili che richiedono comunicazioni affidabili e a basso consumo energetico, aprendo la strada a una nuova era di connettività intelligente e sostenibile.

1.3.3 UART over Bluetooth Low Energy: funzionamento

L'implementazione di una connessione Bluetooth Low Energy rappresenta il punto di partenza cruciale per abilitare la comunicazione wireless tra dispositivi. Questa sezione

esplorerà il processo di inizializzazione della connessione BLE, concentrandosi sui ruoli distinti assunti dai dispositivi centrali e periferici.

Inizializzazione della Connessione BLE
La connessione BLE è il primo passo fondamentale. Un dispositivo centrale (solitamente il client) e un dispositivo periferico (solitamente il server) stabiliscono una connessione BLE utilizzando le specifiche del serve *GATT* per il profilo di comunicazione.

Ruoli di client e server
I ruoli di client e server possono essere adattati in base alle esigenze specifiche dell'applicazione. Il dispositivo centrale, agendo come client *GATT*, invierà richieste al dispositivo periferico (server *GATT*) per leggere o scrivere le caratteristiche di emulazione *UART*.

Caratteristiche *GATT*
Il server *GATT* è il cuore di questa emulazione. Le caratteristiche *GATT* vengono utilizzate per emulare i flussi di dati di una comunicazione *UART*. Due caratteristiche principali sono coinvolte:

- **Caratteristica di trasmissione (TX)**. Questa caratteristica è utilizzata dal dispositivo periferico per inviare dati al dispositivo centrale. È l'equivalente della trasmissione di dati in una connessione seriale tradizionale.
- **Caratteristica di ricezione (RX)**. Questa caratteristica è utilizzata dal dispositivo periferico per ricevere dati dal dispositivo centrale. Rappresenta il canale attraverso il quale vengono ricevuti i dati inviati dal dispositivo centrale.

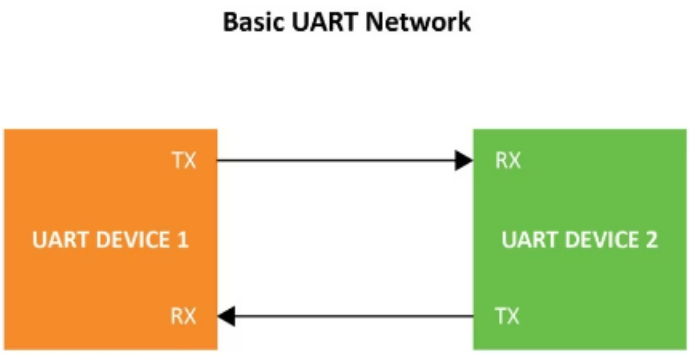


Figura 1.6: Caratteristiche TX ed RX: rappresentazione dei canali su cui avvengono gli scambi asincroni dei messaggi

Comunicazione asincrona

Un elemento chiave della porta *UART* è la sua natura asincrona e questa caratteristica è mantenuta nella sua emulazione su BLE. Il dispositivo centrale può scrivere dati sulla caratteristica TX del servizio *GATT* del dispositivo periferico, e viceversa. Questo consente una comunicazione bidirezionale tra i dispositivi.

Controllo di flusso

Per gestire la comunicazione asincrona, è possibile implementare meccanismi di controllo del flusso. Questi meccanismi possono prevenire la perdita di dati o gli overflow di buffer, assicurando una comunicazione affidabile tra i dispositivi.

–2–
Flutter e Design Pattern

Contents: **2.1 Introduzione a Flutter. 2.2 Strati architetturali.** 2.2.1 Anatomia di un'app Flutter – 2.2.2 Widget. **2.3 Clean Architecture.** 2.3.1 BLoC (Business Logic Component) – 2.3.2 Cubit.

Lo sviluppo mobile, guidato dalla continua evoluzione dei dispositivi e dei sistemi operativi come *iOS* e *Android*, ha presentato la sfida di mantenere app native su diverse piattaforme senza una codebase condivisa. In risposta a questa esigenza, varie proposte di soluzioni sono emerse, cercando di superare le limitazioni di ottimizzazione delle caratteristiche specifiche delle piattaforme.

Nel contesto di queste sfide, *Google* ha introdotto **Flutter**, un toolkit UI cross-platform che ha rapidamente guadagnato popolarità. Lanciato in beta nel gennaio 2018, *Flutter* ha raggiunto la versione 1.0 entro dicembre dello stesso anno. Con *Flutter*, si concretizza l'idea di sviluppare app multipiattaforma senza la necessità di una codebase separata per ciascun sistema operativo.

2.1 Introduzione a Flutter

Flutter è un framework open-source sviluppato da *Google*, progettato per la creazione efficiente di app cross-platform. La peculiarità distintiva di *Flutter* risiede nell'adozione del linguaggio di programmazione *Dart*. Questo framework offre agli sviluppatori un ambiente coeso e potente per la costruzione di interfacce utente moderne e ad alte prestazioni.

Figura 2.1: Logo Flutter

L'obiettivo chiave di Flutter è consentire la creazione efficiente di nuove app, caratterizzata da:

- una fase di sviluppo rapida che sfrutta funzionalità come l'**hot reload**, eliminando la necessità di ricompilare il codice;
- interfacce utente espressive e flessibili con un set di widget componibili, librerie per animazioni e un'architettura stratificata ed estensibile;
- performance prossime a quelle native;
- una codebase unica per applicazioni destinate ad *Android* e *iOS*.

A differenza di molti framework ibridi, *Flutter* si caratterizza per l'assenza di *WebView* (componente di sistema preinstallato di Google che consente alle app per Android di mostrare contenuti web) e *widget OEM (Original Equipment Manufacturer*: componenti grafici di interfaccia utente previsti dal sistema operativo sottostante), elementi comuni in molte applicazioni.
Nelle app native tradizionali, quando si crea un *widget*, esso viene scelto tra quelli disponibili nella libreria *OEM* del dispositivo. Tuttavia, con *Flutter*, questo approccio viene superato: non è più necessario un passaggio intermedio. Quando un'app nativa sviluppa un *widget*, *Flutter* lo seleziona direttamente tra le sue risorse, eliminando la complessità aggiunta dai *WebView* e dai *widget OEM*.

Al contrario di altri framework che potrebbero fare affidamento su *WebView* per la visualizzazione di contenuti web all'interno delle app o utilizzare *widget OEM* predefiniti del sistema operativo, *Flutter* abbraccia un modello diverso. Il framework si avvale di un proprio motore di rendering ad alte prestazioni basato su *Skia*, una libreria open source dedicata alla grafica 2D. In questo modo, Flutter si interfaccia direttamente con le SDK native di Android e iOS, garantendo un controllo più diretto sull'aspetto e le prestazioni delle interfacce utente, senza dover fare affidamento su *WebView* o *widget OEM* predefiniti.

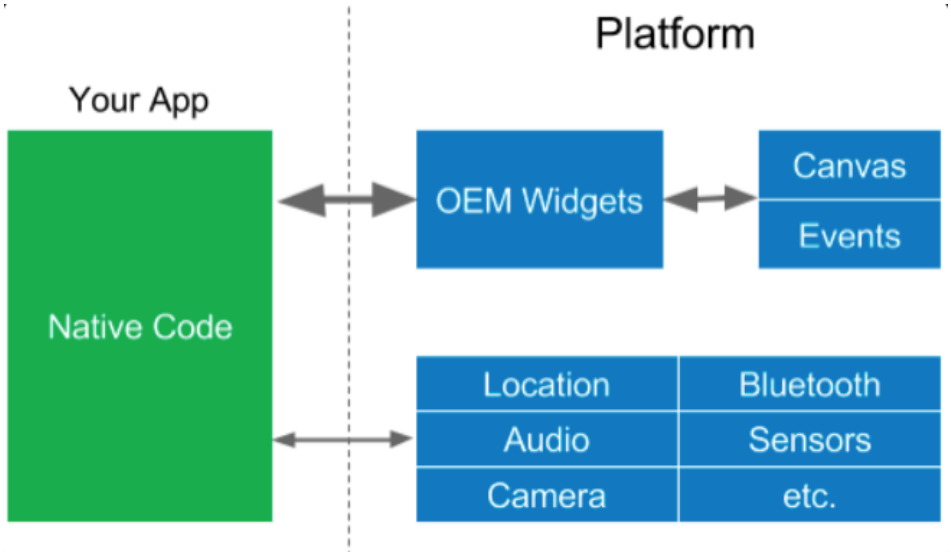


Figura 2.2: Interazione tra un'app nativa e la piattaforma di destinazione

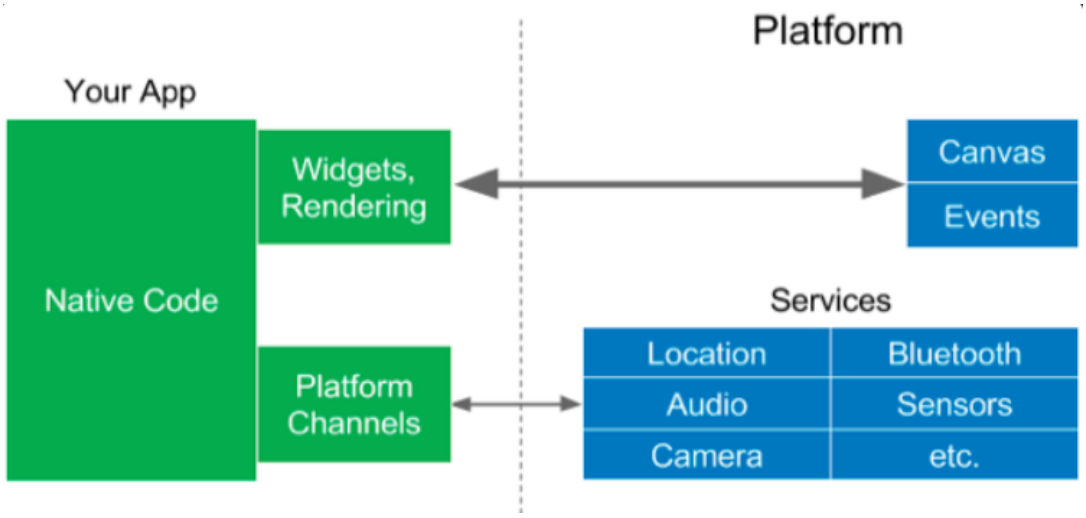


Figura 2.3: Interazione tra un'app Flutter e la piattaforma di destinazione

Durante lo sviluppo, le app Flutter eseguono in una VM, offrendo un *hot reload* di stato senza la necessità di una ricompilazione completa. Per il rilascio, le app Flutter vengono compilate direttamente in codice macchina, sia per *Intel x64* o istruzioni *ARM*, o in *JavaScript* se si mira alla piattaforma web. Il framework è open source, con una licenza *BSD* permissiva, e gode di un ecosistema vibrante di pacchetti di terze parti che integrano la funzionalità della libreria di base.

2.2 Strati architetturali

Flutter è progettato come un sistema stratificato ed estensibile, composto da una serie di librerie indipendenti che dipendono ognuna dallo strato sottostante. Nessuno degli strati ha accesso privilegiato al livello inferiore, e ogni parte del framework è progettata per essere opzionale e sostituibile. L'architettura di *Flutter* si compone di tre macro blocchi principali composti a loro volta da API e librerie che caratterizzano ogni strato.

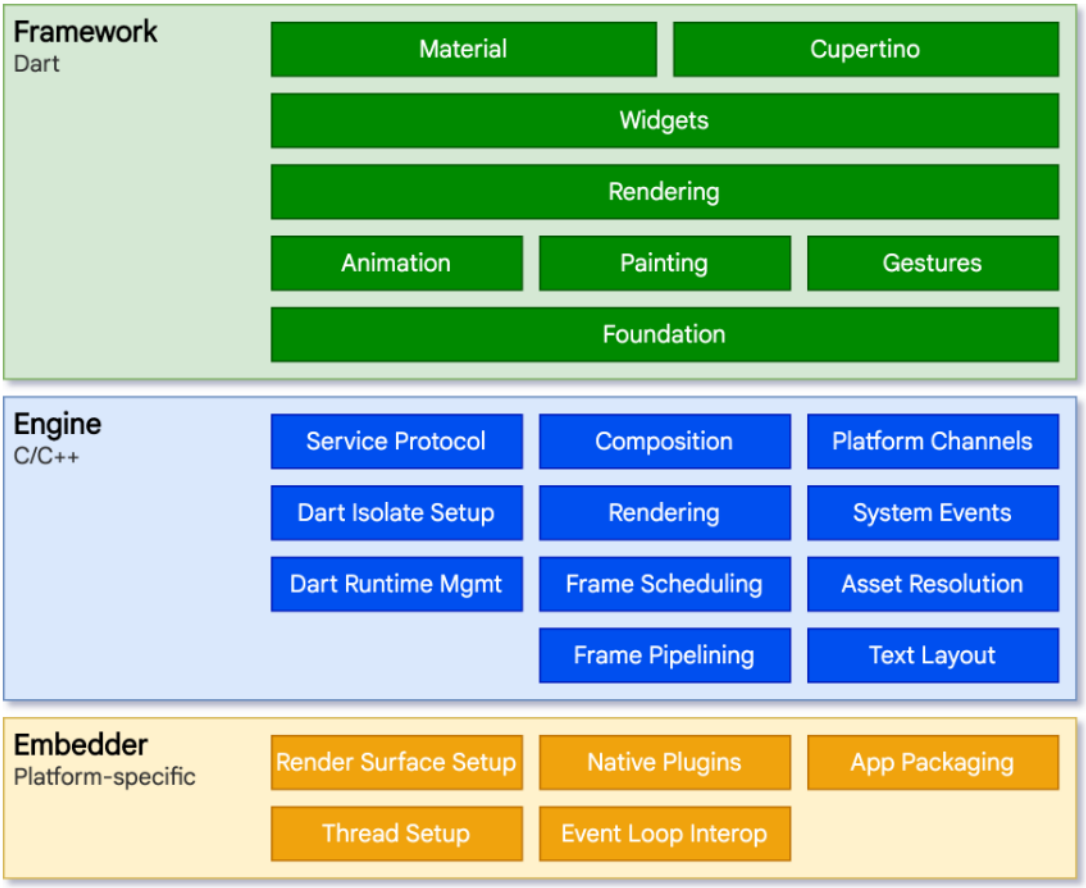


Figura 2.4: Strati architetturali di Flutter

- **Framework**
  Lo strato *Framework* costituisce il fulcro dell'architettura di *Flutter*, rappresentando un elemento cruciale per gli sviluppatori, fornendo un vasto arsenale di librerie e pacchetti essenziali per la costruzione di applicazioni.

  In questo strato, si delineano diversi livelli dedicati ad aspetti fondamentali come animazioni, definizioni di gesture e la creazione di *widget*. Particolarmente rilevante è il livello *Widgets*, che si erge come protagonista nello sviluppo applicativo. Questo livello consente la configurazione di *Material* e *Cupertino layers*, offrendo la possibi-

lità di creazione di componenti grafici coerenti con gli stili distintivi di *Android* e *iOS*. Inoltre, offre la flessibilità di definire *widget* personalizzati.

- **Engine**
  Il *core* di *Flutter* è il **Flutter Engine**, scritto in C ma principalmente in C++, che supporta le primitive necessarie per tutte le applicazioni *Flutter*. Il motore è responsabile della rasterizzazione (conversione di un'immagine bidimensionale descritta da vettori, in un'immagine raster o bitmap, ovvero formata da pixel) di scene composte ogni volta che è necessaria la creazione di un nuovo frame. Fornisce l'implementazione a basso livello dell'API principale di *Flutter*, compresi la grafica (tramite *Impeller* su *iOS* e in arrivo su *Android*, e *Skia* su altre piattaforme), il layout del testo, l'input o output di file e rete, il supporto per l'accessibilità, l'architettura dei plugin, una runtime *Dart* e una *toolchain* di compilazione. L'*engine* è esposto al framework *Flutter* attraverso *dart:ui*, che incapsula il codice C++ sottostante in classi Dart.

- **Embedder**
  Questo strato rappresenta il fondamento dell'architettura di *Flutter* e svolge un ruolo cruciale nell'*Engine* di *Flutter*. In questo strato, gli *embedder*, personalizzati per le diverse piattaforme, sono definiti con l'obiettivo di sincronizzare il processo di rendering con il toolkit nativo dello schermo e di gestire gli eventi di input. Per raggiungere questo scopo, gli *embedder* interagiscono con il layer di *Engine* utilizzando API C/C++ di basso livello, che, tuttavia, rimangono esclusive all'interno dell'architettura.

  Nel caso in cui uno sviluppatore debba implementare comportamenti specifici, ha a disposizione *API* di integrazione di alto livello progettate per le piattaforme *Android* e *iOS*.

  In aggiunta, questo strato include una *Shell* dedicata alla *Dart VM*, progettata per ogni piattaforma. La *Shell* offre un accesso alle API native della piattaforma di riferimento, implementando codice specifico come la gestione della comunicazione con gli *Input Method Editor (IME)* e la gestione degli eventi del ciclo di vita dell'app, adattandosi al sistema operativo di interesse.

2.2.1 Anatomia di un'app Flutter

Il diagramma sottostante offre un'ampia panoramica dei componenti che costituiscono una tipica applicazione generata attraverso il comando *flutter create*. Questa rappresentazione visiva delinea la disposizione degli elementi all'interno di uno stack, evidenzando la posizione fondamentale del *Flutter Engine*, delimitando le interfacce delle API e identificando le *repository* che ospitano le singole parti. Attraverso questa mappa strutturale, emerge una chiara comprensione delle relazioni tra gli elementi costitutivi dell'ecosistema *Flutter*.

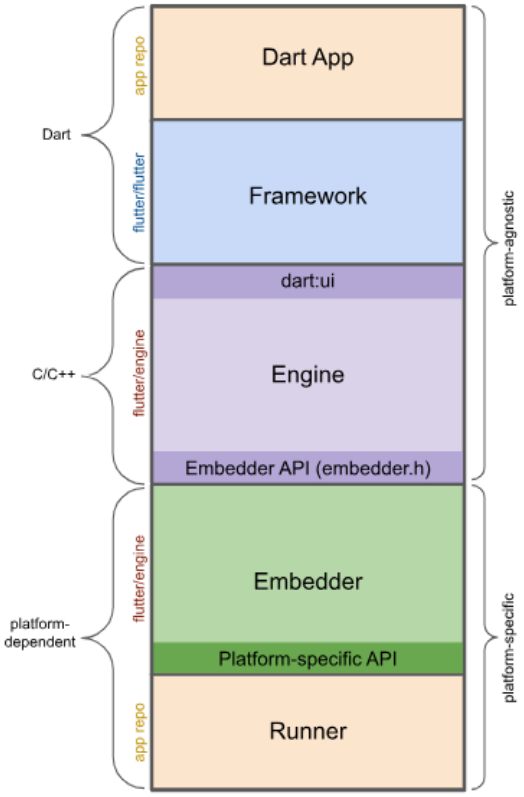


Figura 2.5: Anatomia di un'app Flutter

Di seguito, vengono presentati i principali attori coinvolti nella costruzione di un'app *Flutter*, delineando il contributo distintivo di ciascuno:

- **Dart app**
  Insieme di *widget* tra loro composti per creare l'interfaccia utente desiderata. Implementa la logica di business. Di proprietà dello sviluppatore dell'app.

- **Framework**
  Fornisce un'API di alto livello per la creazione di app di qualità (*widget*, test hit, rilevamento dei gesti, accessibilità, input di testo). Compone l'albero dei *widget* dell'app in una scena.

- **Engine**
  Responsabile della rasterizzazione di scene composte. Implementa a livello basso le *API* principali di *Flutter* (grafica, layout del testo, runtime *Dart*). Espone funzionalità al framework tramite l'API *dart:ui*. Si integra con una piattaforma specifica attraverso l'*API Embedder* del motore.

- **Embedder**
  Coordinato con il sistema operativo per accedere a servizi come superfici di ren-

dering, accessibilità e input. Gestisce il ciclo degli eventi. Espone un'*API* specifica della piattaforma per integrare l'*Embedder* nelle app.

- **Runner**
  Compone le parti esposte dall'*API* specifica della piattaforma dell'*Embedder* in un pacchetto di app eseguibile sulla piattaforma di destinazione. Parte del modello di app generato da *flutter create*, di proprietà dello sviluppatore dell'app.

2.2.2 Widget

In *Flutter*, il concetto di **widget** rappresenta un pilastro fondamentale che va oltre la mera rappresentazione visiva degli elementi dell'interfaccia utente. I *widget* agiscono come blocchi di costruzione essenziali che plasmano l'intera struttura dell'applicazione, estendendo la loro influenza a ogni aspetto, dai componenti strutturali come bottoni e menu, a dettagli stilistici come il font, sino agli intricati dettagli del layout, come il padding, e persino alla gestione di gesti attraverso widget appositamente dedicati.

La peculiare natura dei *widget* risiede nella loro capacità di formare una struttura gerarchica, in cui ciascun *widget* ne annida all'interno degli altri ed ereditano le caratteristiche del *widget* genitore. Tale gerarchia favorisce una gestione flessibile e organizzata di interazioni e stili, promuovendo una netta separazione delle responsabilità.

La composizione si configura come principio cardine nella progettazione dei *widget*. Spesso, esso si compone attraverso l'assemblaggio di unità più piccole e specializzate, ciascuna responsabile di un aspetto specifico. Ad esempio, il versatile *widget Container* può accogliere una varietà di *widget* figli, ciascuno contribuendo al layout, allo stile, al posizionamento e alle dimensioni degli elementi al suo interno.

I *widget* si dividono principalmente in due categorie principali: **Stateless** e **Stateful**. I *widget* **Stateless** sono statici e non tengono traccia dello stato interno, ideali per rappresentare elementi immutabili come testi o icone. D'altra parte, gli **Stateful** mantengono uno stato interno che può cambiare durante il ciclo di vita dell'applicazione, rendendoli adatti per gestire componenti dinamiche come form e interazioni utente.

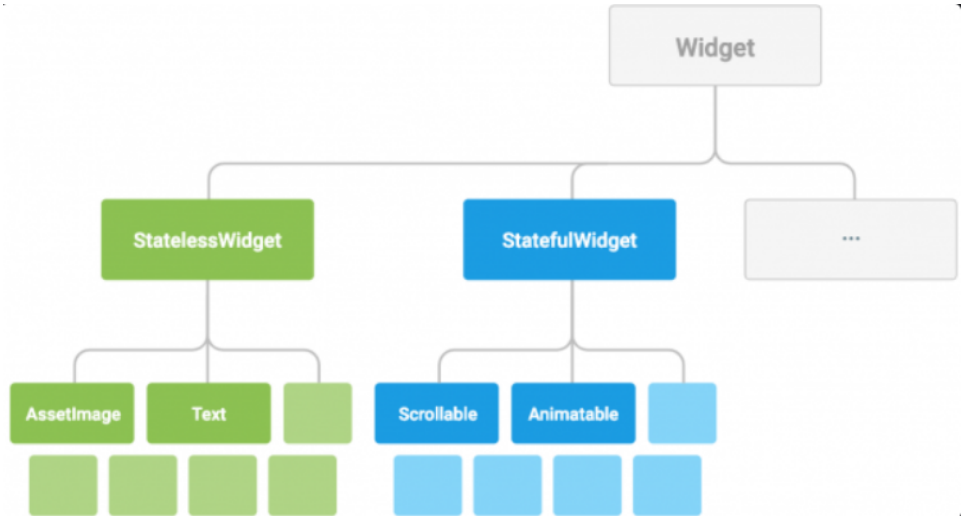


Figura 2.6: Rappresentazione di Stateless e Stateful widget

La struttura gerarchica dei *widget* è stata saggiamente concepita con una profondità relativamente bassa ma una spaziatura notevole. Questo approccio mira a massimizzare le possibilità di combinazione, permettendo agli sviluppatori di crearne dei nuovi, sia essi *Stateless* o *Stateful*, con un'ampia gamma di comportamenti e aspetti.

In conclusione, i Stateful non si limitano a essere semplici elementi visivi; essi costituiscono gli elementi costruttivi fondamentali che conferiscono flessibilità e modularità allo sviluppo in *Flutter*. Sono gli strumenti che abilitano gli sviluppatori a creare interfacce utente dinamiche, responsive e ricche di funzionalità in modo efficiente e organizzato.

2.3 Clean Architecture

Nelle fasi iniziali della loro carriera, alcuni sviluppatori software si concentrano principalmente sull'acquisizione rapida di nuove competenze per creare funzionalità operative. Spesso non prestano molta attenzione a scrivere codice pulito o a seguire un'architettura ben definita per migliorare le prestazioni dell'applicazione. Quando si utilizza un framework, alcuni sviluppatori adottano comunemente il modello *MVC (Model View Controller)*, che suddivide l'applicazione in modelli, viste e controller. Sebbene questo possa funzionare bene per app semplici, emergono problemi quando è necessario espandere l'applicazione con nuove funzionalità o aggiornare librerie. In questi casi, la dipendenza e l'interconnessione del codice rendono difficile apportare modifiche senza coinvolgere numerose parti del sistema. Pertanto, è essenziale progettare un'architettura appropriata prima di iniziare lo sviluppo dell'applicazione.

La ***Clean Architecture*** è una filosofia di progettazione del software che consente agli sviluppatori di separare il software in diversi strati, rendendolo più manutenibile e testabile. **Robert C. Martin** (1952 - informatico statunitense noto anche con lo pseudonimo

di *Uncle Bob*) ha introdotto tale filosofia, per fornire agli sviluppatori un modo di organizzare il codice che rispetti i principi *SOLID*, da lui stesso ideati, e consenta un alto disaccoppiamento delle varie parti del software.

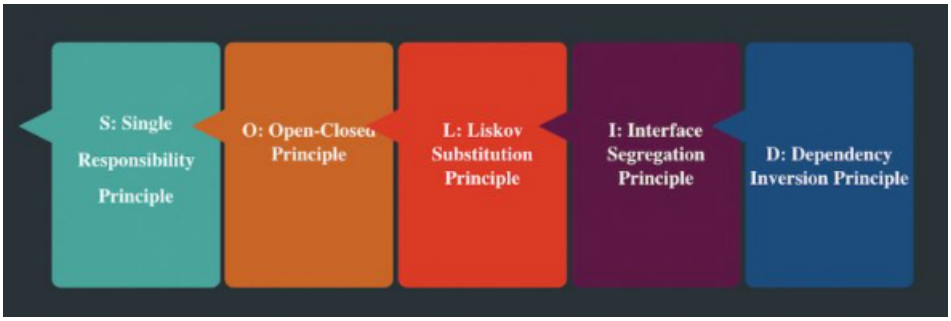


Figura 2.7: I cinque principi SOLID

La *Clean Architecture* in *Flutter* offre notevoli vantaggi sfruttando le potenti capacità di UI del framework e una solida base architetturale. Ciò si traduce in un'applicazione più gestibile, scalabile e facilmente testabile. Questa strategia consente agli sviluppatori di creare codice modulare, contribuendo complessivamente a un'architettura *Flutter* ottimizzata.

Anche se la navigazione attraverso l'implementazione della *Clean Architecture* può risultare inizialmente impegnativa, i benefici successivi, come il miglioramento della qualità del codice e uno sviluppo più efficiente, giustificano gli sforzi dedicati.

Un modello architettonico consigliato per progetti *Flutter*, ispirato ai principi di *Robert C. Martin*, è illustrato di seguito.

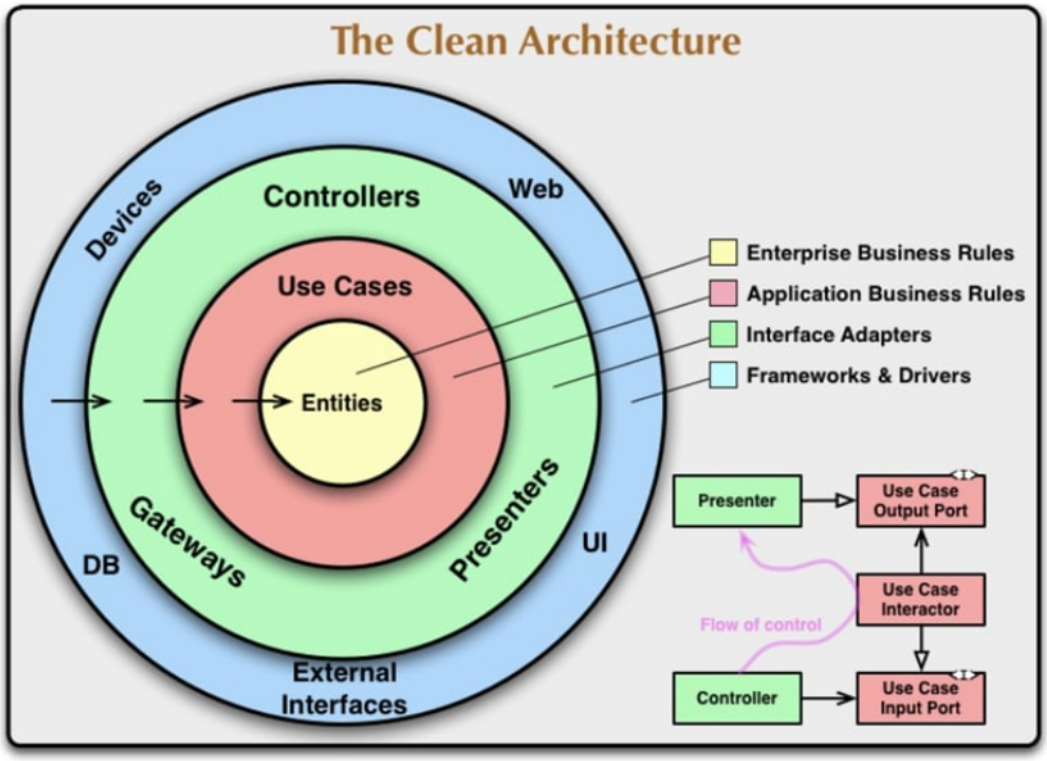


Figura 2.8: Diagramma della Clean Architecture

Tuttavia, è importante notare che questa architettura richiede una personalizzazione adeguata prima di poter essere applicata efficacemente nello sviluppo di un'app *Flutter*. Ciò che però è importante notare è la concezione di suddividere l'applicazione in strati, nel rispetto della *Dependency Rule* di *Uncle Bob*, il quale afferma:

"Nessun elemento nell'anello interno può avere alcuna conoscenza di ciò che si trova nell'anello esterno."

Nel dettaglio, viene analizzato ogni strato:

- **Frameworks & Drivers**

  Il livello più esterno della *Clean Architecture* in *Flutter* comprende gli strumenti di alto livello che gestiscono l'interazione dell'utente e l'ambiente di esecuzione, come l'interfaccia utente di *Flutter*, servizi HTTP, driver di database e widget UI.

- **Interface Adapter**

  Lo strato *Interface Adapter* trasforma i dati tra il livello più esterno e quelli interni. Per un'app *Flutter* tipica, questo strato includerà elementi come l'albero dei *widget* e i modelli necessari per la rappresentazione dell'interfaccia utente.

- **Application Business Rules**

Questo strato, noto anche come *Use Case layer*, contiene le regole di business specifiche dell'applicazione. È qui che viene implementata la funzionalità prevista dal software senza alcuna influenza da parte di entità esterne. Questa incapsulazione delle regole di business consente modifiche all'operatività del sistema senza influenzare i cicli di vita delle entità.

- **Enterprise Business Rules**

  Il livello più interno, rappresenta gli oggetti di business dell'applicazione. Queste regole sono cruciali per l'attività aziendale e verosimilmente potrebbero essere condivise tra diversi sistemi in un'azienda. Ad esempio, se esiste una regola aziendale che vieta di divulgare lo stipendio, un'entità 'dipendente' potrebbe avere una variabile 'stipendio' come privata.

2.3.1 BLoC (Business Logic Component)

Se la *Clean Architecture* stabilisce una chiara separazione delle responsabilità all'interno di un'applicazione e mira a rendere il codice più manutenibile, scalabile e testabile, organizzandola in diversi strati o cerchi concentrici che includono l'interfaccia utente (UI), gli adattatori delle interfacce, le regole di business e le entità aziendali, il design pattern **BLoC** è una metodologia specifica per gestire lo stato e la logica di business in un'app *Flutter*.

Quando viene usato *BLoC*, ci sono due importanti cose da tenere in considerazione: **eventi** e **stati**. Ciò significa che quando un'evento viene inviato possiamo ricevere uno o più stati e questi stati sono inviati in uno **stream**.

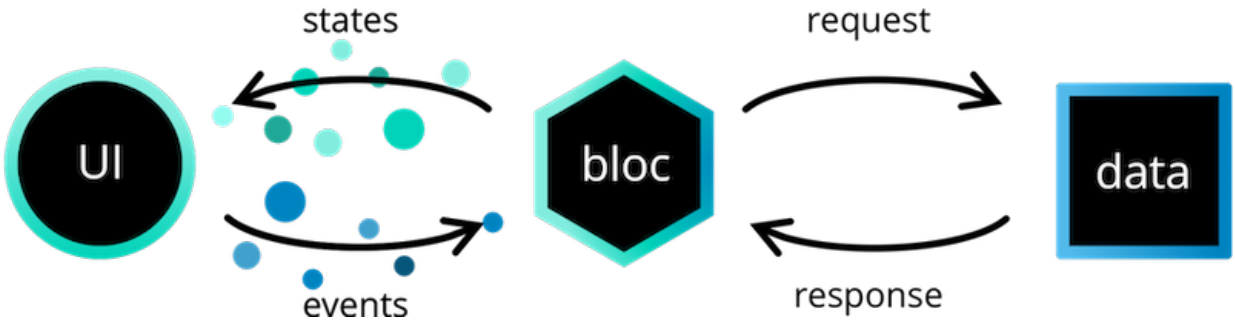


Figura 2.9: Architettura BLoC

Ci sono diversi concetti fondamentali da comprendere nell'uso del pattern *BLoC* in *Flutter*:

- **Eventi**: rappresentano le attività dell'utente o altre azioni che possono modificare lo stato dell'applicazione. Gli eventi sono generalmente rappresentati come classi dati semplici.

- **Bloc**: è una classe che riceve eventi, li elabora e produce un nuovo stato. È responsabile del controllo dello stato dell'applicazione e della risposta agli input dell'utente.

- **Stato**: rappresenta la condizione attuale dell'applicazione ed è generalmente rappresentato come una classe dati immutabile.

- **Stream**: in contesto *BLoC*, uno stream è una raccolta di eventi asincroni che possono essere monitorati per eventuali modifiche. Gli stream vengono utilizzati in *BLoC* per descrivere lo stato dell'applicazione in un dato momento.

- **Sink**: è un controller di Stream utilizzato per inviare eventi a uno stream. Nel contesto *BLoC*, un Sink è usato per inviare eventi al Bloc per l'elaborazione.

- **StreamController**: è utilizzato per costruire e gestire gli stream di eventi inviati al Bloc.

- **BlocBuilder**: un *widget* fornito dal pacchetto *flutter_bloc*, il *BlocBuilder* connette il *Bloc* all'interfaccia utente. Ascolta le modifiche nello stato del *Bloc* e ricostruisce l'UI di conseguenza.

- **BlocProvider**: Il pacchetto *flutter_bloc* include un *widget* chiamato BlocProvider che aggiunge un Bloc all'albero dei widget. Assicura che il Bloc venga creato solo una volta ed è accessibile a tutti i widget nel sottoalbero.

Da notare che, *BLoC* si riferisce al design pattern, mentre *Bloc* si riferisce a un'istanza di una classe che implementa tale pattern.

2.3.2 Cubit

In ambiente *Flutter*, il concetto di **Cubit** emerge come un'alternativa leggera e semplificata per la gestione dello stato dell'applicazione. La parola *Cubit* è una contrazione di *Business Logic* (Logica di Business) e questo design pattern si concentra specificamente sulla gestione di uno stato reattivo all'interno di un'app *Flutter*.

A differenza di *BLoC*, *Cubit* è concepito per essere più snello e si basa sulla semplicità concettuale. La sua principale responsabilità è gestire la logica di business e mantenere lo stato dell'applicazione in maniera chiara e comprensibile.

In particolar modo, ciò che caratterizza un *Cubit* è che si possono inviare solo **stati** e per innescarli, è necessaria una chiamata ad una **funzione**.

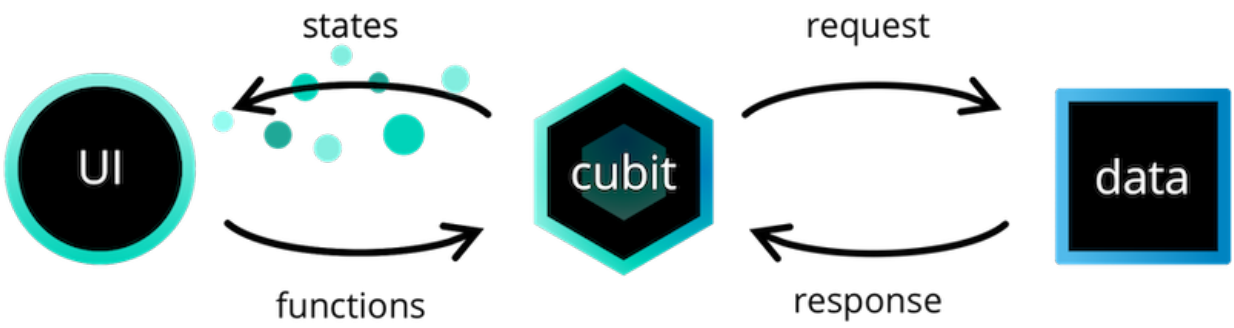


Figura 2.10: Architettura Cubit

Le principali caratteristiche di *Cubit*, che lo spingono ad essere un design pattern utilizzato di frequente, includono:

- **Gestione dello stato**
  Cubit gestisce lo stato dell'applicazione. Invece di utilizzare eventi per innescare transizioni di stato, come avviene in BLoC, Cubit riceve chiamate di funzioni per rappresentare uno stato specifico.

- **Reattività**
  Una volta che uno stato è cambiato, tutti i *widget* interessati vengono automaticamente informati e aggiornati. Questa reattività semplifica la sincronizzazione tra la logica di business e l'interfaccia utente.

- **Semplicità**
  Cubit è progettato per essere più semplice rispetto a BLoC. Si concentra su uno scambio più diretto di informazioni tra componenti senza la complessità aggiunta degli eventi.

- **Composizione**
  Cubit supporta la composizione, consentendo la creazione di *cubits* più piccoli e specializzati che possono essere combinati per gestire funzionalità più complesse.

- **Testabilità**
  La gestione semplificata dello stato rende più agevole il testing di unità per assicurarsi che l'applicazione funzioni come previsto in diversi scenari.

Ciò che stabilisce una diversità tra un *Bloc* e un *Cubit* è che il primo è una classe più avanzata che si basa sugli **eventi** per innescare cambiamenti di **stato** anziché sulle **funzioni**. A differenza dei *Cubit*, invece di chiamare una funzione su un Bloc ed emettere direttamente un nuovo stato, i Bloc ricevono eventi e convertono gli eventi in ingresso in stati in uscita.

–3–
Raspberry Pi 3 Model B+ nel contesto automotive

3.1 Introduzione alla Raspberry Pi 3 B+

La **Raspberry Pi 3 B+** fa parte della rinomata famiglia di microcomputer sviluppata dalla *Raspberry Pi Foundation*. Questo modello, reso disponibile nel marzo 2018, ha suscitato notevole interesse grazie alle sue caratteristiche avanzate e alle capacità versatile offerte. Creato dalla Raspberry Pi Foundation, un'organizzazione non-profit, questo microcomputer ha aperto nuove prospettive nell'ambito dell'informatica embedded, diventando una soluzione accessibile e potente per una varietà di applicazioni.

Figura 3.1: Raspberry Pi 3 Model B+

3.1.1 Specifiche tecniche

La *Raspberry Pi 3 B+* si distingue per una serie di specifiche tecniche che ne fanno una piattaforma di calcolo versatile e potente. Al cuore di questo microcomputer troviamo il processore *Broadcom BCM2837B0*, un *Quad-core Cortex-A53* a 64 bit, operante a una frequenza di clock di 1.4 GHz. Accoppiato con 1 GB di RAM LPDDR2 SDRAM, questo processore offre prestazioni notevoli per una vasta gamma di applicazioni, dalla gestione di sistemi embedded complessi alla prototipazione di progetti innovativi.

Dal punto di vista grafico, la *Raspberry Pi 3 B+* è dotata di una *GPU VideoCore IV* a doppio nucleo, capace di gestire l'uscita video Full HD (1080p) attraverso l'interfaccia HDMI. Questa combinazione di potenza di elaborazione e capacità grafiche rende la *Raspberry Pi 3 B+* adatta a progetti che richiedono la gestione fluida di contenuti multimediali e l'elaborazione di dati visivi.

La connettività è un punto di forza di questo microcomputer, con supporto sia per reti wireless (802.11.b/g/n/ac a 2.4GHz e 5GHz) che per connessioni Bluetooth 4.2 e Bluetooth Low Energy (BLE). Quattro porte USB 2.0 offrono ulteriore flessibilità per l'aggiunta di dispositivi periferici, mentre la porta Ethernet Gigabit consente connessioni cablate ad alta velocità.

La presenza di 40 pin *GPIO (General Purpose Input/Output)* rende la *Raspberry Pi 3 B+* adatta per progetti di elettronica e automazione, consentendo l'interfacciamento con una varietà di sensori e attuatori. Inoltre, con l'ampia gamma di ambienti di sviluppo supportati, tra cui *Python, C, C++, Java, e Scratch*, la *Raspberry Pi 3 B+* si presenta come una piattaforma di sviluppo aperta e accessibile, ideale per chiunque desideri esplorare il mondo dell'informatica embedded. La combinazione di queste specifiche tecniche la rende un potente strumento adatto a soddisfare le esigenze di progetti diversificati, dal controllo veicolo alla telemetria, dall'automazione domestica all'educazione informatica.

Per maggiore chiarezza, di seguito vengono elencate tutte le specifiche tecniche.

Processore

- **Modello**: Broadcom BCM2837B0
- **Architettura**: Cortex-A53 a 64 bit
- **Core**: Quad-core
- **Frequenza di clock**: 1.4 GHz

Memoria

- **RAM**: 1 GB LPDDR2 SDRAM

Grafica

- **GPU**: VideoCore IV a doppio nucleo
- **Uscita video**: HDMI, compatibilità con video Full HD (1080p)

Connettività

- **Wireless**: IEEE 802.11.b/g/n/ac (2.4GHz e 5GHz)
- **Bluetooth**: Bluetooth 4.2, Bluetooth Low Energy (BLE)

Porte e slot

- **USB**: 4 porte USB 2.0
- **Ethernet**: 10/100/1000 Mbps (Gigabit Ethernet)
- **Slot MicroSD**: supporta schede di memoria fino a 32 GB o superiore

Alimentazione

- **Tensione di alimentazione**: 5V tramite micro-USB
- **Consumo energetico**: variabile, in genere tra 2.5W e 6.7W
- **GPIO** (General Purpose Input/Output)
- **Pin GPIO**: 40 pin GPIO (General Purpose Input/Output)
- **Interfacce**: I2C, SPI, UART, PWM, etc.

Audio

- **Uscita audio**: jack da 3.5mm
- **Ingresso audio**: tramite HDMI o USB

Dimensioni

- **Dimensioni**: 85.6mm x 56.5mm
- **Peso**: circa 45 grammi

Supporto video

- **Decodifica video**: H.264 a 1080p30, H.265 a 1080p30, MPEG-4 a 1080p30
- **Encode Video**: H.264 a 1080p30

Sistema operativo

- **Sistema operativo supportato**: Raspberry Pi OS (precedentemente noto come Raspbian) e altri sistemi operativi basati su Linux

3.2 BlueZ

La *Raspberry Pi 3 B+* svolge il ruolo di server per l'invio dei dati e questa funzionalità è resa possibile grazie all'utilizzo del software **BlueZ** per la gestione delle connessioni Bluetooth. *BlueZ* è un pacchetto di stack Bluetooth open-source che fornisce un'implementazione completa delle specifiche Bluetooth. Il microcomputer utilizza *BlueZ* per configurare e gestire un server *GATT (Generic Attribute Profile)* che consente la comunicazione Bluetooth Low Energy.

3.2.1 Architettura

L'architettura di *BlueZ* è una componente cruciale nel panorama dell'implementazione Bluetooth su sistemi *Linux*. Esso, sviluppato e mantenuto dalla comunità open-source, svolge un ruolo fondamentale nella gestione delle comunicazioni Bluetooth e offre un'ampia gamma di funzionalità attraverso un'architettura ben progettata.

BlueZ opera come un *daemon* noto con il nome di *bluetoothd*, che funge da cuore pulsante del sistema. Il *daemon* è responsabile della gestione e della fornitura di supporto per il *Generic Access Profile (GAP)* e il *Generic Attribute Profile (GATT)*, elementi chiave per la comunicazione Bluetooth.

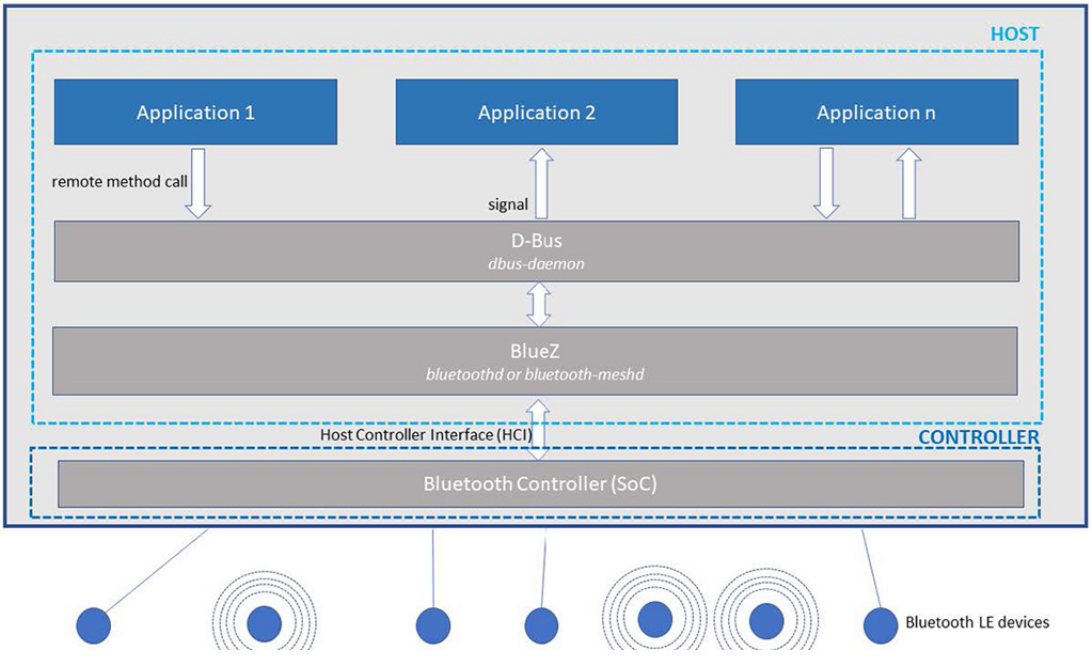


Figura 3.2: Architettura di BlueZ

La figura sopra illustra la struttura di base dell'architettura di *BlueZ*. Il *daemon bluetoothd* interagisce con le applicazioni attraverso un intermediario di *IPC (Inter-Process*

Communication) noto come **D-Bus**, un servizio di sistema ampiamente utilizzato su *Linux* per facilitare la comunicazione tra processi.

Le applicazioni, che operano come processi indipendenti, fanno chiamate di comunicazione tra processi (*IPC*) alle API di *BlueZ* attraverso *D-Bus*. Tale meccanismo di comunicazione consente la trasmissione di messaggi e segnali tra le applicazioni e il *daemon* Bluetooth.
L'architettura modulare di *BlueZ* permette al sistema di adattarsi a una vasta gamma di situazioni e requisiti.

3.2.2 Configurazione del Server GATT

Il *server GATT* consente ai dispositivi client di accedere a caratteristiche specifiche, rappresentate come *attributi GATT*. La configurazione di *BlueZ* sulla *Raspberry Pi 3 B+* coinvolge la definizione di servizi e caratteristiche personalizzate per gestire l'invio dei dati.
La procedura include la definizione di *servizi GATT* che descrivono le funzionalità offerte dal server e la configurazione delle caratteristiche corrispondenti per la trasmissione dei dati.

3.2.3 Interfaccia con BlueZ tramite Linea di Comando

L'interazione con *BlueZ* può avvenire attraverso la linea di comando o mediante script personalizzati. Questo permette una flessibilità notevole nella configurazione del *server GATT* in base alle esigenze di qualsiasi progetto. La *Raspberry Pi 3 B+* può essere programmata anche per avviare automaticamente il *server GATT* all'accensione, garantendo la disponibilità continua del servizio.

3.3 Applicazioni nel Contesto Automotive

La *Raspberry Pi 3 B+* non è soltanto un microcomputer versatile e accessibile, ma ha trovato applicazioni significative anche nel settore automobilistico, rivelandosi una soluzione potente e flessibile per diverse esigenze in questo contesto. La sua presenza in questo settore evidenzia la sua capacità di adattarsi a una vasta gamma di applicazioni, contribuendo all'innovazione e alla modernizzazione dei sistemi automobilistici.

3.3.1 Embedded Systems e Controllo Veicolo

Una delle principali aree di applicazione della *Raspberry Pi* nell'ambito automobilistico riguarda la sua integrazione come componente di sistemi embedded. Grazie alle sue dimensioni compatte e alle prestazioni affidabili, essa trova impiego nel controllo di varie funzionalità del veicolo, come il sistema di infotainment, la gestione termica, la sicurezza e altro ancora.

3.3.2 Sensori e Telemetria

L'uso della *Raspberry Pi* si estende anche all'acquisizione e all'elaborazione dei dati provenienti da una vasta gamma di sensori presenti nei veicoli moderni. Questi possono includere sensori di temperatura, accelerometri, sensori di prossimità e altro ancora. L'elaborazione dei dati avviene sul microcomputer, che può successivamente trasmettere informazioni rilevanti attraverso connessioni wireless o cablate.

3.3.3 Sviluppo e Prototipazione

La facilità di sviluppo e prototipazione offerta dalla *Raspberry Pi* è un fattore cruciale nel settore automobilistico. Consente agli ingegneri e ai progettisti di testare e implementare rapidamente nuove soluzioni, riducendo i tempi di sviluppo e fornendo spazio per l'innovazione in un settore in continua evoluzione.

3.3.4 Soluzioni di Connessione e Comunicazione

Inoltre, la *Raspberry Pi* può essere configurata per gestire connessioni e comunicazioni avanzate. Ad esempio, può essere utilizzata come parte di sistemi di connettività avanzata per veicoli, consentendo la comunicazione veicolo a veicolo (*V2V*) e la comunicazione veicolo-infrastruttura (*V2I*), contribuendo così allo sviluppo di veicoli intelligenti e connessi.

–4–
Software Development

4.1 Contesto del progetto

Il progetto si colloca all'interno di un contesto di innovazione, affrontando le sfide della mobilità urbana attraverso la realizzazione di un sistema avanzato di comunicazione tra autoveicoli. In un'epoca in cui la tecnologia guida la trasformazione dei trasporti, l'obiettivo primario è facilitare la condivisione tempestiva e intelligente di informazioni tra veicoli, con l'intento di migliorare la sicurezza stradale e ottimizzare l'efficienza del traffico.

Il fulcro di questa innovazione risiede nelle *On Board Unit (OBU)*, dispositivi messi a disposizione dalla *Unex Corporation*, installati in ciascun veicolo partecipante. Le *OBU* operano come nodi intelligenti, acquisendo dati rilevanti sul veicolo e l'ambiente circostante e trasmettendo queste informazioni agli altri autoveicoli presenti nella rete.

Un elemento chiave di questo progetto è la tecnologia *Vehicular-to-Everything (V2X)*, un concetto che abbraccia varie forme di comunicazione, inclusa *Vehicle-to-Vehicle (V2V)* e la *Vehicle-to-Infrastructure (V2I)*. Questa visione crea una rete collaborativa in cui veicoli e infrastrutture condividono dati in tempo reale, sfruttando la potenza della connettività per migliorare la sicurezza stradale, ottimizzare il flusso del traffico e promuovere soluzioni di mobilità intelligenti.

Attraverso la *V2X*, le *OBU* diventano agenti attivi in un ecosistema intelligente, consentendo una comunicazione bidirezionale con altri veicoli e le infrastrutture stradali. Questa comunicazione non solo fornisce informazioni sullo stato del traffico e rischi potenziali, ma apre la strada all'implementazione di soluzioni di guida collaborative, come il coordinamento ottimizzato negli incroci e l'avviso tempestivo di emergenze stradali.

In sintesi, *NetCom Group* non solo si propone di implementare la tecnologia *V2X* attraverso le *OBU*, ma aspira a ridefinire il panorama della mobilità urbana, trasformando ogni veicolo in un nodo consapevole e proattivo di una rete di comunicazione avanzata. L'obiettivo finale è contribuire a creare una città più sicura, efficiente e con un sistema di trasporti all'avanguardia.

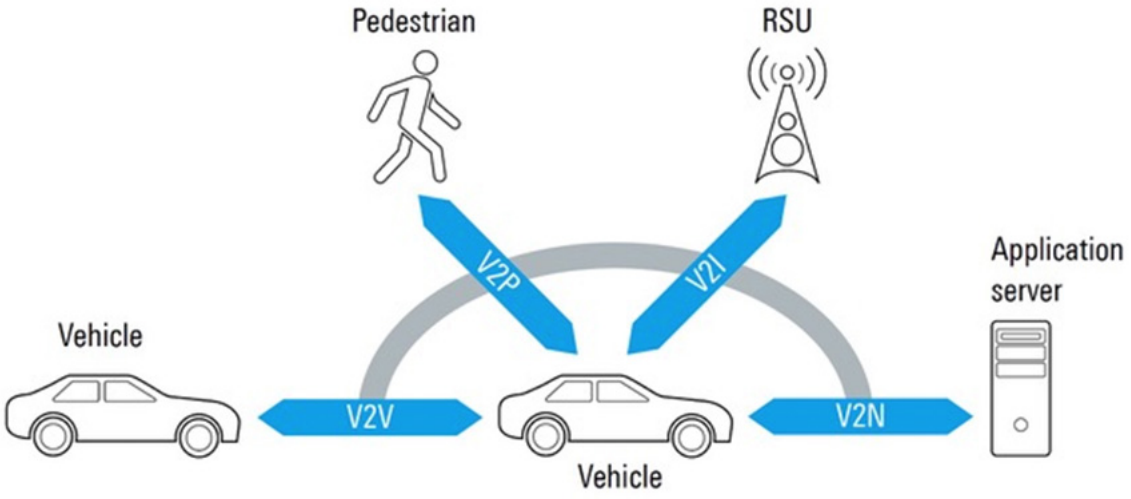


Figura 4.1: Rappresentazione V2X

4.2 Obiettivi principali e studio di fattibilità

Il progetto ha avuto inizio con la richiesta da parte dell'azienda di sviluppare un'interfaccia per un'app *Android* esistente, la quale riceve dati da una *Raspberry Pi 3 Model B+* sulla quale è in esecuzione uno script *Python*, responsabile dell'invio di tali dati. Questa fase iniziale ha fornito le basi del progetto, focalizzandosi sull'aspetto visuale e funzionale dell'applicazione.
È importante notare che, nonostante la fase iniziale abbia coinvolto la *Raspberry Pi 3 Model B+*, attualmente il contesto progettuale prevede la sua sostituzione con la più avanzata *Raspberry Pi 4*. Le caratteristiche potenziate di quest'ultima offrono nuove opportunità e prestazioni migliorate, contribuendo significativamente all'avanzamento del progetto, specialmente nell'ambito delle *On Board Unit (OBU)*.

Un punto fondamentale è rappresentato dalla necessità di rendere l'applicazione cross-platform. Inizialmente, la richiesta si concentrava sulla realizzazione dell'interfaccia, ma il primo studio di fattibilità ha assunto un ruolo determinante quando è stato deciso di adottare il framework non solo come supporto per l'interfaccia, ma come strumento per lo sviluppo completo dell'applicazione. Questa decisione ha ridefinito il percorso del progetto, portando a una soluzione integrata e versatile che ha ottimizzato il processo di sviluppo su diverse piattaforme.
Successivamente, è emersa una nuova esigenza. Inizialmente, la comunicazione dell'app preesistente avveniva tramite *Bluetooth*, utilizzando il protocollo *RFCOMM*. Tuttavia, per migliorare le prestazioni e adattarsi alle moderne esigenze di comunicazione, l'azienda ha richiesto la migrazione dell'app *Bluetooth* a *Bluetooth Low Energy (BLE)*. Tale richiesta ha concentrato lo studio e lo sviluppo nella realizzazione di una nuova app, sviluppata

in *Flutter*, che replicasse il comportamento dell'app preesistente su *Android*. L'elemento chiave di questa evoluzione è stato il passaggio dal *Bluetooth* al *BLE*, con l'obiettivo di garantire una comunicazione più efficiente.
Di conseguenza nasce una nuovo obiettivo, ovvero creare uno script *Python*, sostituendo quello precedentemente utilizzato, che fosse in grado di stabilire una comunicazione *BLE* anziché *Bluetooth*. Questo ha comportato l'adozione di un nuovo protocollo rispetto a *RFCOMM*, adeguandosi alla natura più efficiente e orientata alle basse potenze del *Bluetooth Low Energy*. Questa evoluzione tecnologica ha contribuito a modernizzare il sistema di comunicazione, rendendolo più adatto alle esigenze attuali e future del progetto.

4.3 Analisi dei requisiti

L'analisi dei requisiti è un passo fondamentale nella definizione e comprensione delle funzionalità necessarie per il corretto funzionamento del software. Nel contesto del progetto in esame, il focus è sull'implementazione di un'applicazione in *Flutter* e di uno script *Python* per la gestione della comunicazione Bluetooth Low Energy tra veicoli.

4.3.1 Requisiti funzionali

L'app deve presentare in modo chiaro e comprensibile i dati del veicolo, tra cui la velocità, il comportamento di guida, la velocità del vento, l'inclinazione della strada, la distanza dal veicolo frontale, la presenza di pedoni in strada e la richiesta di priorità da parte di un autoveicolo d'emergenza. Nel dettaglio i dati sono i seguenti:

- **velocità del vento** indica la velocità del vento in km/h;
- **pendenza** è l'inclinazione della strada espressa in gradi;
- **velocità** è la velocità del veicolo espressa in km/h;
- **distanza relativa** è la distanza dal veicolo frontale espressa in metri;
- **comportamento dell'utente alla guida** è una stringa che rappresenta il comportamento dell'utente alla guida;
- **collisione** riporta il rischio di una collisione:
  0: indica nessun rischio;
  1: rischio di collisione;
- **pedone vulnerabile** indica la presenza di un pedone in strada:
  0: nessun pedone in strada;
  1: pedone in strada;

- **veicolo d'emergenza** indica la ricezione di una richiesta di priorità da parte di un autoveicolo d'emergenza:
  0: nessuna richiesta di priorità;
  1: presenza di una richiesta di priorità.

Lo script *Python* deve generare dati casuali e continui, impacchettati in un messaggio JSON, per simulare situazioni di traffico realistiche, contribuendo alla validazione della corretta ricezione dei dati da parte dell'app.

4.3.2 Requisiti non funzionali

Cross-Platform
L'applicazione deve essere sviluppata in *Flutter* per garantire la compatibilità cross-platform, permettendo l'esecuzione su dispositivi *Android* e *iOS*.

Efficienza della Comunicazione
Lo script Python deve essere progettato per effettuare la comunicazione Bluetooth Low Energy in modo efficiente, garantendo una trasmissione affidabile dei dati tra il veicolo e l'app.

4.4 Progettazione

La sezione di progettazione costituisce il fulcro dell'applicazione di comunicazione tra veicoli. Verrà esplorata la progettazione dettagliata sia dello script *Python* eseguito su *Raspberry Pi 3 Model B+*, che dell'applicazione mobile sviluppata con *Flutter*.
L'obiettivo principale di questa fase è tradurre i requisiti delineati nell'analisi in una struttura progettuale robusta e coerente. Verrà analizzata l'architettura del sistema e le scelte tecnologiche fondamentali adottate per garantire il corretto funzionamento dell'applicazione e la comunicazione efficace tra i componenti coinvolti.

Nel dettaglio, si presterà attenzione a due aspetti chiave:

1. **Progettazione dello script Python per Raspberry Pi 3 Model B+**
   - Si analizzerà come lo script *Python* è stato strutturato per gestire la comunicazione Bluetooth Low Energy e la generazione di dati simulati da parte della Raspberry Pi.
   - Sarà esaminata la configurazione del server *GATT (Generic Attribute Profile)* per garantire una comunicazione affidabile tra la *Raspberry Pi* e l'app mobile.
2. **Progettazione dell'app Flutter**
   - Sarà valutata l'architettura adottata, basata sul pattern di gestione degli stati *Cubit*, e si analizzerà come questa abbia guidato l'organizzazione del codice e la gestione degli eventi nell'applicazione.

- Si esplorerà la struttura delle schermate, il flusso di navigazione e la gestione della connessione Bluetooth all'interno dell'app.

Attraverso questa disamina dettagliata, verrà fornita una visione chiara delle scelte progettuali adottate, evidenziando le soluzioni tecniche che hanno contribuito al successo del progetto. La progettazione di entrambi i componenti, l'app *Flutter* e lo script *Python* su *Raspberry Pi*, sarà presentata in modo coerente e integrato, riflettendo la sinergia tra le due parti fondamentali del sistema di comunicazione veicolare.

4.4.1 Architettura dello script Python su Raspberry Pi 3 Model B+

L'architettura dello script `uart_peripheral.py`, implementato in *Python*, abilita la *Raspberry Pi 3 B+* a presentarsi come una periferica BLE, consentendo all'app *Flutter* di rilevarla. Il microcomputer trasmette dati utilizzando il protocollo *GATT*, emulando specificamente una porta *UART* per la comunicazione di questi dati. Di conseguenza, l'app *Flutter* agisce come un client in grado di ricevere le informazioni relative al veicolo, le quali vengono trasmesse in formato *JSON*.

4.4.2 Flusso dei dati

Lo script avvia un server *GATT* utilizzando la libreria *BlueZ* per fornire un'interfaccia Bluetooth. Questo server *GATT* contiene una caratteristica *UART* personalizzata.

Periodicamente, la caratteristica *UART* emula la trasmissione di dati del veicolo. I dati del veicolo sono generati casualmente, rappresentando informazioni come la presenza di collisioni, il comportamento di guida, la velocità del vento, l'inclinazione di eventuali salite, la velocità del veicolo, le distanze tra due veicoli, la presenza di utenti vulnerabili sulla strada e veicoli di emergenza.

I dati del veicolo vengono organizzati in una struttura di tipo dizionario *Python*. Questo dizionario viene quindi convertito in una stringa *JSON* utilizzando la libreria `json`.

La stringa *JSON* viene convertita in una lista di byte. Essa rappresenta il valore della caratteristica *UART* e viene inviata attraverso la chiamata alla funzione `PropertiesChanged`, aggiornando così il valore della caratteristica.

Il processo di generazione e trasmissione dei dati viene ripetuto ciclicamente, consentendo la trasmissione continua delle informazioni del veicolo sulla *Tx Characteristic*.

4.4.3 Modello dei Dati (lato server)

I dati del veicolo trasmessi seguono una struttura *JSON*. Nello specifico i dati rappresentano:

- **wind_speed** indica la velocità del vento in km/h;
- **slope** è l'inclinazione della strada espressa in gradi;
- **speed** è la velocità del veicolo espressa in km/h;
- **relative_distances** è la distanza dal veicolo frontale espressa in metri;
- **driving_behaviour** è una stringa che rappresenta il comportamento dell'utente alla guida;
- **collision** riporta il rischio di una collisione:
  0: indica nessun rischio;
  1: rischio di collisione;
- **vulnerable_road_user** indica la presenza di un pedone in strada:
  0: nessun pedone in strada;
  1: pedone in strada;
- **emergency_vehicle** indica la ricezione di una richiesta di priorità da parte di un autoveicolo d'emergenza:
  0: nessuna richiesta di priorità;
  1: presenza di una richiesta di priorità.

L'architettura dello script facilita la configurazione della Raspberry Pi come periferica BLE, consentendo la sua rilevabilità da parte di un dispositivo client, come un'app *Flutter*. Ciò avviene attraverso l'utilizzo di un server *GATT* che abilita la trasmissione di dati sulla porta *UART* emulata. Grazie a questa configurazione, il client può ricevere i dati del veicolo in formato *JSON*, favorendo l'integrazione di tali dati nell'applicazione.

4.4.4 Architettura dell'app Flutter

L'architettura dell'app è stata progettata seguendo principi di modularità e facilità di manutenzione, con un'enfasi particolare sulla gestione dello stato attraverso il **design pattern Cubit**. Questo pattern offre un approccio dichiarativo e reattivo alla gestione dello stato dell'applicazione, consentendo una chiara separazione tra la logica di business e la presentazione dell'interfaccia utente.

La struttura del progetto è organizzata in modo gerarchico, riflettendo una suddivisione logica delle responsabilità.
Di seguito, una panoramica delle principali cartelle e file:

Lib

- `main.dart`: questo file rappresenta il punto di ingresso dell'applicazione, inizializzando il *widget* principale e avviando l'esecuzione dell'app.

Core

- `ids.dart`: questo file contiene gli **UUIDs** essenziali per la comunicazione Bluetooth Low Energy tramite il server *GATT*. Gli *UUIDs* identificano in modo univoco i servizi e le caratteristiche BLE necessarie per la corretta trasmissione e ricezione dei dati attraverso la porta *UART*. La corretta gestione degli UUIDs è fondamentale per stabilire e mantenere una connessione BLE affidabile tra dispositivi.

Presentation

- `homepage` e `dashboard`: queste cartelle rappresentano le due principali sezioni dell'applicazione, ciascuna contenente la propria logica di presentazione. Ogni sezione è ulteriormente suddivisa in *blocs* e *views*.

- `Blocs`: contiene i *Cubits* che gestiscono lo stato e la logica di business per le rispettive sezioni.

- `Views`: qui sono presenti le interfacce utente di ciascuna sezione. Le *views* interagiscono con i *Cubits* per riflettere dinamicamente lo stato dell'applicazione.

Questa suddivisione ben definita delle cartelle e dei file mira a garantire una struttura chiara e organizzata, facilitando lo sviluppo, il testing e la manutenzione del codice. La scelta di implementare il *design pattern Cubit* contribuisce a una gestione efficace dello stato, offrendo una visione pulita e comprensibile della logica applicativa.

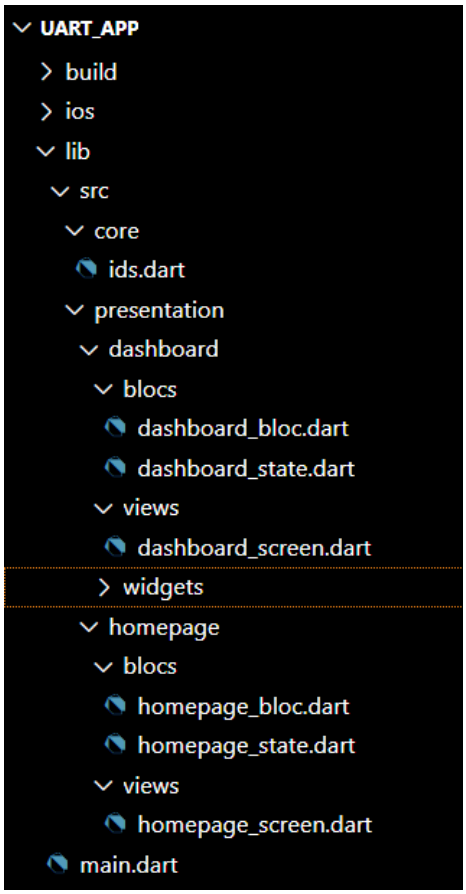


Figura 4.2: Suddivisione delle cartelle secondo design pattern Cubit

4.4.5 Modello dei Dati (lato client)

Il modello dei dati nell'app *Flutter* è progettato per gestire le informazioni ricevute sotto forma di stringa *JSON*. La struttura dei dati segue un formato specifico, che viene decodificato e interpretato dall'app per visualizzare le informazioni in modo comprensibile per l'utente.

Questo modello dei dati fornisce una struttura organizzata per la visualizzazione e l'interpretazione delle informazioni cruciali relative al veicolo e all'ambiente circostante. La gestione di questa struttura avviene nell'app *Flutter*, che decodifica la stringa *JSON* e presenta le informazioni all'utente attraverso l'interfaccia utente dell'app.

4.4.6 Gestione Bluetooth Low Energy (BLE)

L'applicazione utilizza la libreria **flutter_reactive_ble** per semplificare la gestione della connessione BLE. Questa libreria fornisce un'interfaccia reattiva che consente una facile integrazione e gestione degli eventi BLE. Gli aspetti chiave della gestione BLE includono:

1. **Inizializzazione del Bluetooth**: la libreria inizializza il modulo Bluetooth e verifica la presenza della funzionalità BLE sul dispositivo.

2. **Scansione dei dispositivi BLE**: l'app effettua la scansione per individuare i dispositivi BLE disponibili, focalizzandosi su quelli che supportano il servizio e la caratteristica *UART*.

3. **Connessione e disconnessione**: l'app gestisce i processi di connessione e disconnessione con il dispositivo *Raspberry Pi*, gestendo gli eventi corrispondenti in modo reattivo.

4. **Trasmissione e ricezione dei dati**: Utilizzando la libreria, l'app invia e riceve dati attraverso il canale BLE in modo semplificato. Il formato dei dati è rappresentato in formato *JSON*.

4.4.7 Considerazioni UI/UX

Date le specifiche del progetto, l'attenzione principale durante la progettazione dell'interfaccia utente non è stata posta su aspetti estetici o avanzati dettagli di design. Invece, l'obiettivo predominante è stato garantire un'interfaccia funzionale e pratica che consentisse a futuri sviluppatori di modificarla in base a qualsiasi esigenza.
Il design dell'interfaccia utente è stato guidato dalla priorità delle funzionalità chiave legate alla connessione BLE e alla visualizzazione dei dati trasmessi. L'aspetto visivo è stato mantenuto in modo essenziale, concentrando gli sforzi sulle operazioni di base e sulla chiarezza delle informazioni presentate all'utente.

4.5 Sviluppo

Questa sezione delinea il processo di sviluppo, suddiviso tra il lato server *Python* su *Raspberry Pi* e l'app mobile *Flutter*.
Il percorso inizierà esplorando la programmazione *Python* su *Raspberry Pi 3 B+*, concentrandosi sulla logica di comunicazione BLE e sulla gestione dei dati scambiati con il dispositivo esterno. Successivamente, sarà affrontato lo sviluppo dell'app mobile mediante il framework *Flutter*, analizzando il codice, la gestione dello stato e le interfacce utente.

4.5.1 Raspberry Pi 3 Model B+

Inizializzazione dell'ambiente di sviluppo

Il software lato *Raspberry* è stato sviluppato utilizzando il linguaggio di programmazione *Python*. L'utilizzo di *Python* offre vantaggi in termini di chiarezza del codice e facilità di sviluppo. Per la gestione della comunicazione Bluetooth Low Energy e dell'interfaccia *D-Bus*, sono state utilizzate le librerie *BlueZ* e *DBus*.

Prima di avviare lo sviluppo, è stata necessaria una configurazione appropriata dell'ambiente sulla *Raspberry Pi*. Questo include:

- Installazione delle librerie *BlueZ* e *DBus* mediante l'utilizzo del package manager della distribuzione *Linux* utilizzata sulla *Raspberry Pi*. In particolare è importante che la versione di *BlueZ* sia almeno la 5.43.

```
sudo apt-get install bluez dbus
```

Figura 4.3: Screen relativo al comando per la corretta installazione di DBus

- Verificare, se necessario, la configurazione dei permessi per l'accesso all'adattatore BLE.
- In più, la *Raspberry Pi* può essere configurata per eseguire lo script *Python* all'avvio, se necessario.

Questi passaggi sono essenziali per garantire che l'ambiente di sviluppo sia pronto per l'implementazione del software di comunicazione BLE.

Definizione degli UUID e delle Caratteristiche

Uno degli aspetti fondamentali dello sviluppo è stata la scelta degli *UUID* per identificare il servizio *GATT* e le relative caratteristiche. Gli *UUID* scelti sono conformi agli standard BLE e sono stati selezionati in modo che corrispondessero agli *UUIDs* previsti dall'applicazione *nRF Toolbox*, messa a disposizione dalla *Nordic Semiconductor*, con cui successivamente sono stati effettuati test relativi al corretto funzionamento dello script.

```
UART_SERVICE_UUID =            '6e400001-b5a3-f393-e0a9-e50e24dcca9e'
UART_RX_CHARACTERISTIC_UUID =  '6e400002-b5a3-f393-e0a9-e50e24dcca9e'
UART_TX_CHARACTERISTIC_UUID =  '6e400003-b5a3-f393-e0a9-e50e24dcca9e'
```

Figura 4.4: UUIDs dello script Python

Le classi `TxCharacteristic`, `RxCharacteristic` e `UartService` sono state implementate per rappresentare le caratteristiche BLE e il servizio *GATT* associato. La caratteristica di trasmissione (`TxCharacteristic`) è responsabile dell'invio periodico dei dati del veicolo, mentre la caratteristica di ricezione (`RxCharacteristic`) gestisce la ricezione di dati sulla porta *UART* virtuale.

All'interno della classe `UartService`, le caratteristiche `TxCharacteristic` e `RxCharacteristic` sono collegate al servizio *GATT*. Questo collegamento è essenziale per garantire che il servizio comprenda tutte le funzionalità necessarie per la comunicazione BLE.

Generazione e invio dei dati casuali

Una parte fondamentale dello sviluppo è stata la creazione di dati casuali relativi al veicolo per simulare informazioni reali. La classe `TxCharacteristic` implementa il metodo `send_veicolo_data()` che genera dati casuali come la presenza di una collisione, il comportamento di guida, la velocità del vento, l'angolo di pendenza, la velocità del veicolo, le distanze relative, la presenza di utenti vulnerabili sulla strada e veicoli di emergenza.

La caratteristica di trasmissione è stata configurata per inviare periodicamente i dati del veicolo. Il metodo `StartNotify()` è responsabile dell'inizio della notifica, mentre il metodo `StopNotify()` ne gestisce la terminazione.

Questa configurazione assicura che la *Raspberry Pi* invii dati casuali ed emulati dell'automobile all'applicazione *Flutter* ogni secondo, attraverso la caratteristica di trasmissione.

```
class TxCharacteristic(Characteristic):
    def __init__(self, bus, index, service):
        Characteristic.__init__(self, bus, index, UART_TX_CHARACTERISTIC_UUID,
                                ['notify'], service)
        self.notifying = False
        self.timeout_id = None

    def send_veicolo_data(self):
        if not self.notifying:
            return

        # Genera dati del veicolo casuali
        collision = int(numpy.random.choice(numpy.arange(0, 2, 1), p=[0.95, 0.05]))
        driving_behaviour = str(random.randrange(0, 100)) + "%"
        wind_speed = random.randrange(0, 100)  # km/h
        slope = random.randrange(0, 50)  # °
        speed = random.randrange(0, 130)  # km/h
        relative_distances = random.randrange(1, 1000)  # m
        vulnerable_road_user = int(numpy.random.choice(numpy.arange(0, 2, 1), p=[0.95, 0.05]))
        emergency_vehicle = int(numpy.random.choice(numpy.arange(0, 2, 1), p=[0.95, 0.05]))

        # Costruisci la stringa JSON con i dati del veicolo
        veicolo_data = {
            "collision": collision,
            "driving_behaviour": driving_behaviour,
            "wind_speed": wind_speed,
            "slope": slope,
            "speed": speed,
            "relative_distances": relative_distances,
            "vulnerable_road_user": vulnerable_road_user,
            "emergency_vehicle": emergency_vehicle
        }

        # Converti il dizionario in una stringa JSON
        veicolo_json = json.dumps(veicolo_data)

        # Converti la stringa JSON in una lista di byte
        value = [dbus.Byte(c.encode()) for c in veicolo_json]

        # Invia i dati del veicolo
        self.PropertiesChanged(GATT_CHRC_IFACE, {'Value': value}, [])

        # Ripeti la chiamata ogni secondo
        GLib.timeout_add_seconds(1, self.send_veicolo_data)

    def StartNotify(self):
        if not self.notifying:
            self.notifying = True
            self.PropertiesChanged(GATT_CHRC_IFACE, {'Notifying': True}, [])
            self.timeout_id = GLib.timeout_add_seconds(1, self.send_veicolo_data)

    def StopNotify(self):
        if self.notifying:
            self.notifying = False
            self.PropertiesChanged(GATT_CHRC_IFACE, {'Notifying': False}, [])
            if self.timeout_id is not None:
                GLib.source_remove(self.timeout_id)
                self.timeout_id = None
```

Figura 4.5: Classe TxCharacteristic

Gestione della comunicazione BLE

Per abilitare la comunicazione Bluetooth Low Energy sulla *Raspberry Pi*, il software interagisce con il sistema *BlueZ* utilizzando *DBus*. *BlueZ* è il *middleware* Bluetooth - software che si colloca tra il sistema operativo e le applicazioni, agendo come uno strato intermedio che facilita la comunicazione e l'interoperabilità tra diversi componenti software - per sistemi *Linux* che gestisce le operazioni Bluetooth e *DBus* è un sistema di comunicazione interprocesso che consente l'interazione tra il software e *BlueZ*.

Le classi `UartApplication` e `UartAdvertisement` sono utilizzate per registrare l'applicazione BLE e l'annuncio BLE rispettivamente, facendo uso delle interfacce *DBus*. In particolare:

- **UartApplication**: si occupa di definire e gestire il servizio *GATT* (`UartService`) e di rendere disponibili le caratteristiche associate. L'applicazione BLE registra il proprio servizio *GATT* presso il gestore *GATT* di *BlueZ*, in modo che altri dispositivi possano scoprirlo e interagire con le sue caratteristiche.

- **UartAdvertisement**: configura e gestisce l'annuncio BLE, che include informazioni come l'*UUID* del servizio *GATT* offerto, ovvero il servizio *UART* e il nome locale del dispositivo.

Il metodo `RegisterApplication` registra l'applicazione presso il gestore *GATT*, mentre il metodo `RegisterAdvertisement` registra l'annuncio BLE presso il gestore dell'annuncio. Queste operazioni sono fondamentali per rendere disponibile il servizio *GATT* e annunciarlo agli altri dispositivi BLE circostanti.

```
service_manager.RegisterApplication(app.get_path(), {},
                                    reply_handler=register_app_cb,
                                    error_handler=register_app_error_cb)
ad_manager.RegisterAdvertisement(adv.get_path(), {},
                                 reply_handler=register_ad_cb,
                                 error_handler=register_ad_error_cb)
```

Figura 4.6: Metodi RegisterApplication e RegisterAdvertisement

Integrazione con il loop principale

Per garantire che il programma rimanga attivo durante l'esecuzione e che le operazioni BLE siano gestite correttamente, è stata implementata una sezione dedicata al loop principale. Esso consente al software di continuare a eseguire le operazioni necessarie, come l'invio periodico dei dati BLE, la gestione degli annunci e la ricezione dei dati sulla caratteristica RX.

```
try:
    mainloop.run()
except KeyboardInterrupt:
    adv.Release()
```

Figura 4.7: Loop principale

Il loop principale è avviato utilizzando il metodo `mainloop.run()`. Fa in modo che il programma rimanga in esecuzione finché non viene interrotto da un'eccezione di tipo `KeyboardInterrupt` (ad esempio, quando l'utente preme *Ctrl+C*). Alla ricezione di questa eccezione, l'annuncio BLE viene rilasciato attraverso il metodo `adv.Release()`, garantendo una chiusura pulita del programma.

Test e verifica

Per garantire il corretto funzionamento del software, sono state utilizzate diverse metodologie di test. Ciò include test unitari per verificare singoli componenti e test di interoperabilità per garantire la corretta comunicazione.

È stata data particolare attenzione ai test di **interoperabilità** con l'applicazione *nRF-Toolbox*. Verificare che il software *Raspberry Pi* sia in grado di comunicare con successo con un'applicazione esterna, ha aiutato a garantire che il protocollo BLE implementato sia conforme agli standard e che la trasmissione e la ricezione di dati avvengano senza problemi.

Figura 4.8: nRF Toolbox per BLE messo a disposizione dalla Nodic Semiconductor

Di seguito sono presentati degli screenshot che illustrano il corretto funzionamento dello script su Raspberry Pi 3 B+ e la corretta ricezione dei dati attraverso l'app *RF Toolbox*.

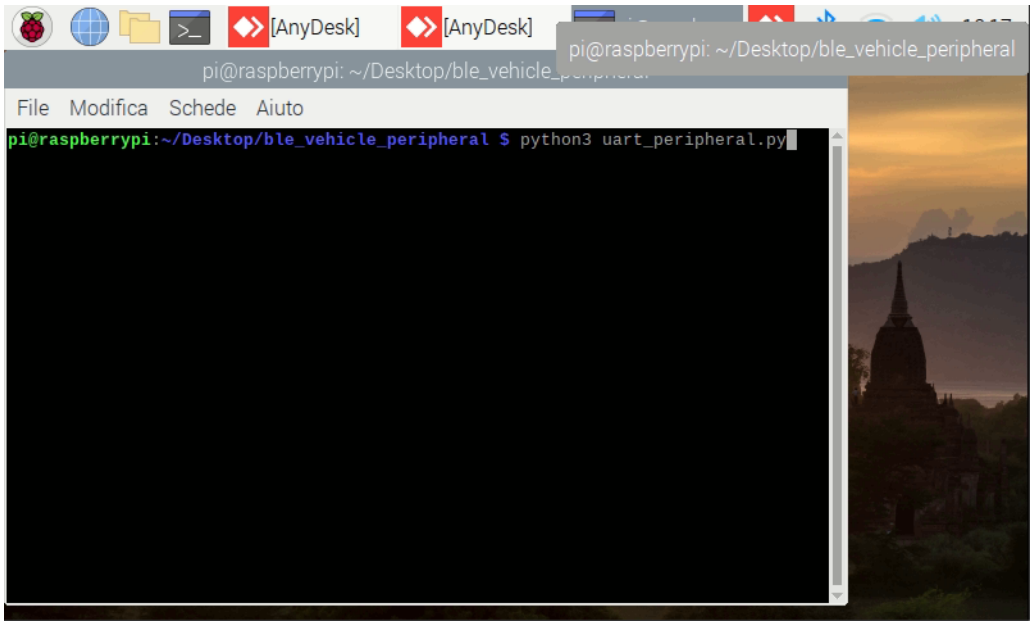


Figura 4.9: Comando per avviare lo script

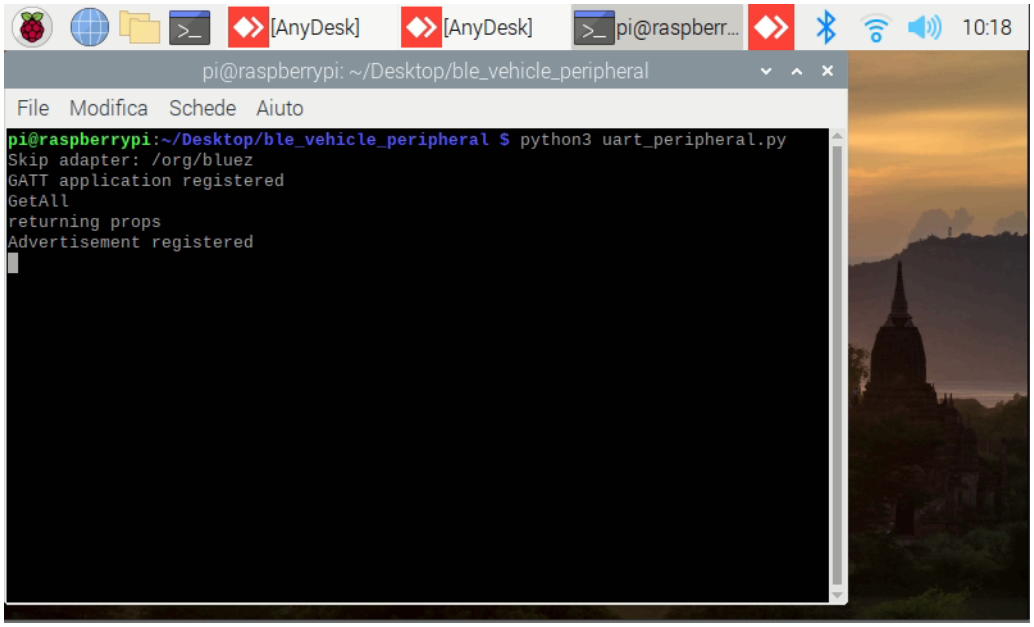


Figura 4.10: Server GATT correttamente registrato, il dispositivo è pronto per essere rilevato e sta inviando i dati relativi al veicolo

Figura 4.11: Screenshot relativo all'applicazione nRFToolbox, che riceve correttamente i dati casuali generati dalla Raspberry Pi 3 B+

4.5.2 Mobile Flutter App

Inizializzazione dell'ambiente di sviluppo

Prima di esplorare il codice, è importante comprendere l'ambiente di sviluppo utilizzato per la creazione dell'app mediante il framework *Flutter*. In questa sezione, verranno forniti dettagli sull'*IDE*, la versione di *Flutter* e altre configurazioni pertinenti.

Versioni delle Piattaforme

- **Flutter SDK**: Versione 3.13.7

- **Dart SDK**: Versione 3.1.3

IDE (Integrated Development Environment)

- **IDE Utilizzato**: Visual Studio Code
- **Estensioni Relevanti**
  Bloc: utilizzata per facilitare la scrittura di codice basato su BLoC (Business Logic Component) nell'app Flutter.
  Awesome Flutter Snippets: estensione che fornisce snippet utili per velocizzare la scrittura del codice Flutter.

Inizializzazione del progetto Flutter: pubspec.yaml

Il progetto *Flutter* è stato inizializzato utilizzando il comando `flutter create` all'interno del terminale di *Visual Studio Code*. La struttura di base del progetto include il file `pubspec.yaml` per la gestione delle dipendenze e le configurazioni di progetto.

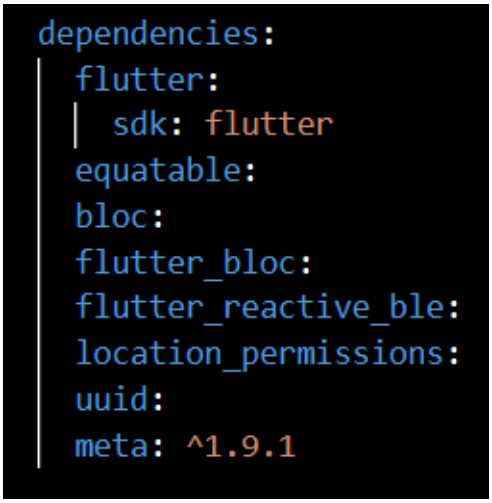


Figura 4.12: Screenshot relativo alle librerie utilizzate presenti in pubspec.yaml

Le librerie adottate sono le seguenti:

- **Equatable**: è una libreria *Flutter* che semplifica la gestione delle classi immutabili. Fornisce un modo facile per implementare il metodo `==` e `hashCode` in modo consistente, aiutando a evitare errori comuni legati alla comparazione di oggetti.
- **flutter_bloc**: fornisce gli strumenti necessari per implementare e gestire facilmente i *Cubit*.
- **flutter_reactive_ble**: questa libreria semplifica l'interazione con i dispositivi Bluetooth Low Energy (BLE) in *Flutter*. Fornisce metodi per la scansione dei dispositivi, la connessione e la comunicazione con i dispositivi BLE.

- **location_permissions**: è una libreria che semplifica la gestione dei permessi di posizione in *Flutter*. Aiuta a richiedere e verificare lo stato dei permessi di posizione, che è spesso necessario per le applicazioni che coinvolgono la geolocalizzazione.
- **uuid**: è una libreria per la generazione e la manipolazione di identificatori univoci in *Flutter*.

Configurazione delle Autorizzazioni: AndroidManifest.xml

Nel corso dello sviluppo dell'app *Flutter* per la gestione delle comunicazioni Bluetooth e BLE, è fondamentale comprendere il ruolo dei permessi, specialmente nel contesto del sistema operativo *Android*. Infatti, *Android* richiede autorizzazioni specifiche per consentire all'app di accedere e utilizzare le funzionalità Bluetooth e di localizzazione. In questo contesto, è essenziale esplorare i permessi inseriti nel file `AndroidManifest.xml`, considerando che lo sviluppo è avvenuto su un dispositivo *Android*. Di seguito, verranno analizzati brevemente ogni permesso e la sua importanza per il funzionamento dell'app.

```
<uses-permission
    android:name="android.permission.BLUETOOTH"
    android:maxSdkVersion="30" />
<uses-permission android:name="android.permission.BLUETOOTH_SCAN"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE"/>
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

Figura 4.13: Screenshot relativo ai permessi presenti all'interno dell'AndroidManifest.xml

- `android.permission.BLUETOOTH`: autorizza l'app ad accedere al modulo Bluetooth del dispositivo, consentendo la comunicazione con dispositivi Bluetooth nelle vicinanze.
- `android.permission.BLUETOOTH_SCAN`: abilita la scansione di dispositivi Bluetooth nelle vicinanze; operazione fondamentale per individuare e connettersi a dispositivi disponibili.
- `android.permission.BLUETOOTH_ADVERTISE`: Consente all'app di pubblicare annunci Bluetooth a supporto di funzionalità di broadcasting o condivisione di dati tramite Bluetooth Low Energy (BLE).
- `android.permission.BLUETOOTH_CONNECT`: autorizza la connessione a dispositivi Bluetooth, permettendo all'app di stabilire connessioni con altri dispositivi abilitati Bluetooth.

- **`android.permission.BLUETOOTH_ADMIN`**: concede all'app il permesso di eseguire operazioni amministrative Bluetooth, come attivare o disattivare il Bluetooth del dispositivo.

- **`android.permission.ACCESS_FINE_LOCATION`**: fornisce l'accesso preciso alla posizione del dispositivo, requisito comune per operazioni Bluetooth avanzate e localizzazione accurata di dispositivi nelle vicinanze.

- **`android.permission.ACCESS_COARSE_LOCATION`**: abilita l'accesso approssimativo alla posizione del dispositivo, spesso richiesto insieme ad `ACCESS_FINE_LOCATION` per determinate operazioni Bluetooth.

Linserimento dei permessi, all'interno dell'`AndroidManifest.xml`, è fondamentale per garantire che l'applicazione possa interagire con successo con le funzionalità Bluetooth del dispositivo, offrendo un'esperienza utente affidabile e sicura su piattaforma *Android*.

UUIDs: ids.dart

Il file **`ids.dart`** definisce gli *UUIDs* necessari per la comunicazione attraverso la porta *UART*. Gli identificatori sono univoci per i servizi e le caratteristiche Bluetooth. `UART_UUID` rappresenta il servizio *UART*; `UART_RX` e `UART_TX` rappresentano rispettivamente le caratteristiche di ricezione e trasmissione. Gli *UUIDs* devono essere coerenti con quelli utilizzati nello script Python in esecuzione sulla *Raspberry Pi*, altrimenti non si riuscirebbe a garantire una corretta comunicazione BLE tra i dispositivi.

```dart
import 'package:flutter_reactive_ble/flutter_reactive_ble.dart';

Uuid UART_UUID = Uuid.parse("6E400001-B5A3-F393-E0A9-E50E24DCCA9E");
Uuid UART_RX = Uuid.parse("6E400002-B5A3-F393-E0A9-E50E24DCCA9E");
Uuid UART_TX = Uuid.parse("6E400003-B5A3-F393-E0A9-E50E24DCCA9E");
```

Figura 4.14: Screenshot relativo agli UUIDs presenti in ids.dart

Entry Point: main.dart

Il file `main.dart` rappresenta l'entry point dell'app *Flutter*, definendo la struttura generale dell'applicazione.

```dart
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'UART App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ), // ThemeData
      home: const HomeConnector(),
    ); // MaterialApp
  }
}
```

Figura 4.15: main.dart

- **`main()`**: è la funzione principale all'avvio dell'app;
- **`MyApp`**: è una classe `StatelessWidget` che rappresenta l'intera applicazione. Imposta il titolo dell'app e il tema generale. Il *widget* principale dell'app è `HomeConnector`
- **`HomeConnector`**: questo *widget* è responsabile di inizializzare e fornire un `HomepageCubit` alla gerarchia di *widget*.

Gestione dello stato e della logica di business: homepage_bloc.dart e dashboard_bloc.dart

Entrambi i file **`homepage_bloc.dart`** e **`dashboard_bloc.dart`** contengono classi *Cubit* che gestiscono il flusso di dati e gli stati associati alle rispettive schermate dell'app. La prima classe si occupa principalmente della gestione dei dispositivi BLE e della scansione, mentre la seconda gestisce la connessione e la ricezione dei dati dal dispositivo BLE.

Il file **`homepage_bloc.dart`** contiene la definizione del `HomepageCubit`, classe *Cubit* del pacchetto *bloc* utilizzata per gestire lo stato della schermata principale dell'app.

```dart
class HomepageCubit extends Cubit<HomepageState> {
  final FlutterReactiveBle flutterReactiveBle = FlutterReactiveBle();
  List<DiscoveredDevice> _foundBleUARTDevices = [];
```

Figura 4.16: La classe homepageCubit presente all'interno di homepage_bloc.dart

Attributi principali:

- `flutterReactiveBle`: inizializza un'istanza di `FlutterReactiveBle` utilizzata per interagire con la libreria `flutter_reactive_ble`;
- `_foundBleUARTDevices`: una lista di dispositivi BLE trovati durante la scansione.

Metodi principali:

- `startScan()`: metodo che avvia la scansione dei dispositivi BLE con il servizio `UART_UUID`;
- `stopScan()`: metodo che interrompe la scansione dei dispositivi BLE;
- `_showErrorDialog(BuildContext context, String errorMessage)`: metodo privato per visualizzare un dialogo di errore;
- `_showSuccessDialog(BuildContext context)`: metodo privato per visualizzare un dialogo di successo;
- `disconnect()`: metodo che gestisce la disconnessione del dispositivo BLE.

Il file **`dashboard_bloc.dart`** contiene la definizione del `DashboardCubit`, che è una classe *Cubit* del pacchetto *bloc* utilizzata per gestire lo stato della schermata dashboard dell'app.

```dart
class DashboardCubit extends Cubit<DashboardState> {

  DashboardCubit() : super(DashboardInitial());

  final FlutterReactiveBle flutterReactiveBle = FlutterReactiveBle();
```

Figura 4.17: La classe dashboardCubit presente all'interno di dashboard_bloc.dart

Metodi Principali:

- `onConnectDevice(DiscoveredDevice device)`: metodo che gestisce la connessione a un dispositivo BLE e avvia la ricezione dei dati.
- `onNewReceivedData(List<int> data)`: metodo che gestisce la ricezione di nuovi dati dal dispositivo BLE e aggiorna lo stato;
- `_sendData(String dataToSendText)`: metodo privato per inviare dati alla periferica BLE.

Definizione degli stati delle schermate: homepage_state.dart e dashboard_state.dart

I file **`homepage_state`** e **`dashboard_state`** definiscono le classi di stato associate alla schermata principale e alla dashboard. Ciascuna classe rappresenta uno specifico stato dell'applicazione, gestendo le informazioni necessarie durante le diverse fasi dell'esecuzione.

Il file **`homepage_state.dart`** definisce lo stato associato alla schermata principale dell'applicazione *Flutter*. In particolare, introduce le seguenti classi:

- `HomepageState` è una classe astratta estesa dalle diverse rappresentazioni dello stato della schermata principale. Ha un metodo chiamato `get` che restituisce una lista di oggetti che identificano in modo univoco lo stato, tipico delle classi che estendono `Equatable`. In questa dichiarazione, si stanno definendo le proprietà `props`, elenco di oggetti utilizzato per determinare l'uguaglianza tra istanze della classe. La parte `=> []` indica che, di default, l'elenco delle proprietà è vuoto;
- `HomepageInitial` è una classe che rappresenta lo stato iniziale della schermata principale. Come attributo ha un'istanza di `FlutterReactiveBle`;
- `HomepageLoading` rappresenta lo stato di caricamento durante la scansione dei dispositivi BLE;
- `HomepageError` rappresenta lo stato di errore con un messaggio di errore associato;
- `HomepageDeviceConnected` rappresenta lo stato associato alla connessione a un dispositivo BLE specifico;
- `HomepageListDevices` rappresenta lo stato con una lista di dispositivi BLE trovati durante la scansione.

```dart
abstract class HomepageState extends Equatable {
  const HomepageState();

  @override
  List<Object?> get props => [];
}
```

Figura 4.18: La classe homepageState presente all'interno di homepage_state.dart

Il file **`dashboard_state.dart`** definisce lo stato associato alla schermata del dashboard dell'applicazione *Flutter*. Introduce le seguenti classi:

- `DashboardState` è una classe astratta estesa dalle diverse rappresentazioni dello stato della schermata del dashboard; è dotata di un getter proprio come `HomepageState` che restituisce una lista di oggetti che identificano in modo univoco lo stato;

- `DashboardInitial` rappresenta lo stato iniziale della schermata della dashboard;
- `DashboardLoading` rappresenta lo stato di caricamento durante la connessione a un dispositivo BLE o la ricezione di dati;
- `DashboardConnected`: rappresenta uno stato associato a una connessione stabilita con un dispositivo BLE, mostrando la schermata relativa ai dati ricevuti;
- `DashboardDisconnecting` rappresenta uno stato durante la disconnessione da un dispositivo BLE;
- `DashboardDisconnected` rappresenta uno stato di disconnessione da un dispositivo BLE;
- `DashboardError` rappresenta uno stato di errore con un messaggio di errore associato.

```dart
abstract class DashboardState extends Equatable {
  const DashboardState();

  @override
  List<Object?> get props => [];
}
```

Figura 4.19: La classe dashboardState presente all'interno di dashboard_state.dart

Implementazioni delle interfacce utente: homepage_screen.dart e dashboard_screen.dart

I file **`homepage_screen.dart`** e **`dashboard_screen.dart`** contengono i *widget* e le classi necessari per gestire la UI e la logica delle schermate principale e della dashboard. Utilizzano i *Cubit* `HomepageCubit` e `DashboardCubit` per gestire lo stato dell'applicazione e interagire con i dispositivi BLE.

Il file **`homepage_screen.dart`** rappresenta la schermata principale dell'app *Flutter*. Includende diverse classi e *widget* e gestisce l'aspetto visivo e la logica della schermata iniziale. Di seguito, una descrizione delle principali componenti:

- `HomeConnector`: un *widget* di connessione che utilizza `BlocProvider` per fornire uno stato del *Cubit* `HomepageCubit` all'interno della gerarchia dei `widget`. Come attributo principale ha `discoveredDevice`, che corrisponde ad essere il dispositivo BLE rilevato durante la scansione;
- `_HomePageBody`: *widget* principale che rappresenta il corpo della schermata principale. Come attributi principali ha un `BlocBuilder` che ricostruisce la UI in base

allo stato del *Cubit* `HomepageCubit`; `ElevatedButton` è un pulsante che avvia la scansione dei dispositivi BLE;

- `_buildDeviceList`: metodo che costruisce la lista dei dispositivi BLE rilevati durante la scansione. Come principale attributo ha `ListView.builder`, ovvero un *widget* che costruisce dinamicamente una lista di *widget* di tipo `ListTile` per ciascun dispositivo;
- `_showErrorDialog e _showSuccessDialog`: metodi che visualizzano dialoghi rispettivamente di errore o successo.

```dart
class HomeConnector extends StatelessWidget {
  const HomeConnector({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (_) => HomepageCubit(),
      child: const HomePageBody(),
    ); // BlocProvider
  }
}
```

Figura 4.20: La classe HomeConnector che estende StatelessWidget, presente all'interno di homepage_screen.dart

Il file **`dashboard_screen.dart`** rappresenta la schermata della dashboard dell'app *Flutter*. Includendo diverse classi e `widget`, gestisce l'aspetto visivo e la logica della schermata della dashboard. Di seguito, una descrizione delle principali componenti:

- `DashboardConnector`: un *widget* di connessione che utilizza `BlocProvider` per fornire uno stato del *Cubit* `DashboardCubit` all'interno della gerarchia dei *widget*. Come principale attributo ritroviamo `discoveredDevice` che rappresenta il dispositivo BLE con cui si è interessati a stabilire una connessione;
- `_DashboardBody`: *widget* principale che rappresenta il corpo della schermata della dashboard. Come attributi principali troviamo `BlocBuilder` che ricostruisce la Ui in base allo stato del *Cubit* `DashboardCubit`; `Column` è un *widget* di colonna che organizza i componenti della Ui verticalmente;
- `_DashboardConnectedWidget`: *widget* che visualizza le informazioni quando il dispositivo è connesso. Come attributo ha un *widget* `Text` che mostra lo stato della connessione e i dai ricevuti.

```dart
class DashboardConnector extends StatelessWidget {
  const DashboardConnector({Key? key, required this.discoveredDevice})
      : super(key: key);

  final DiscoveredDevice discoveredDevice;

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (context) => DashboardCubit()..onConnectDevice(discoveredDevice),
      child: const _DashboardBody(),
    ); // BlocProvider
  }
}
```

Figura 4.21: La classe DashboardConnector che estende StatelessWidget, presente all'interno di dashboard_screen.dart

Illustrazioni dell'interfaccia utente

In questa sezione, verranno presentati alcuni screenshot significativi dell'interfaccia utente dell'applicazione. Gli screenshot forniscono una panoramica visiva delle schermate principali e delle funzionalità chiave implementate nell'app, ovvero l'inizio di scannerizzazione dei dispositivi, la rivelazione del server *GATT* attivo della *Raspberry Pi*, la connessione al server e la corretta trasmissione dei dati generati casualmente.

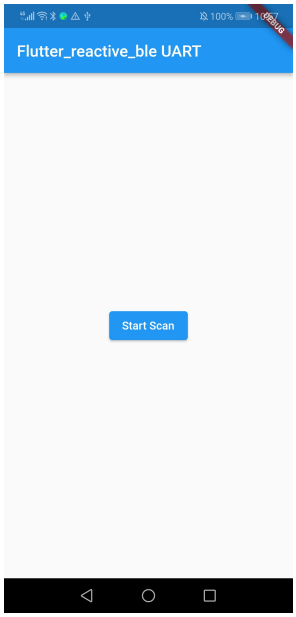


Figura 4.22: Pulsante per iniziare la scannerizzazione dei dispositivi

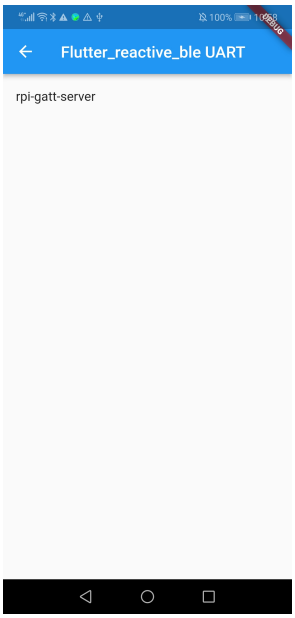


Figura 4.23: Lista dei dispositivi BLE che supportano la caratteristica UART

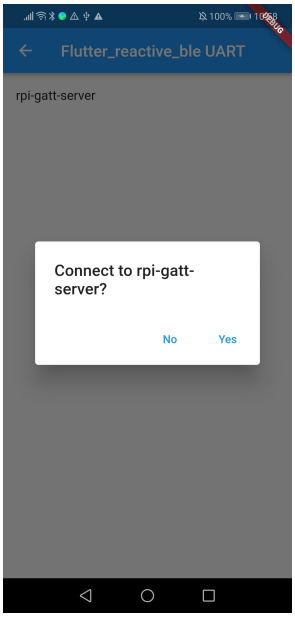


Figura 4.24: Finestra di dialogo per richiedere la connessione al dispositivo

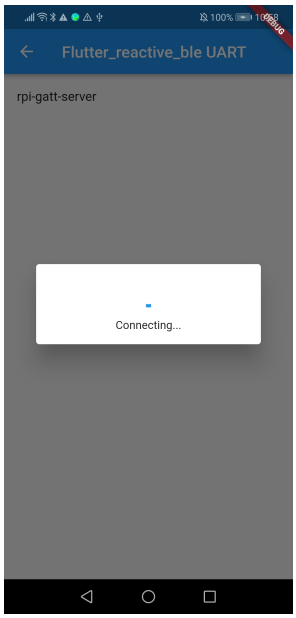


Figura 4.25: Schermata di caricamento, in attesa di connessione e prelievo dei dati dalla Raspberry

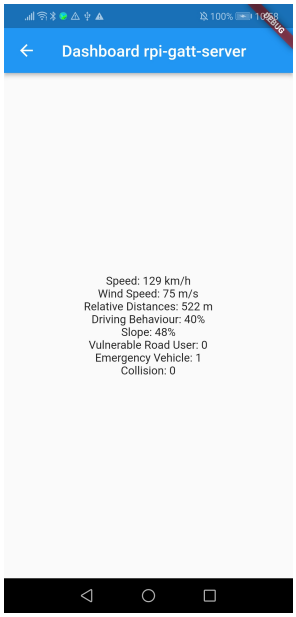


Figura 4.26: Visualizzazione dei dati

–5–
Conclusioni e sviluppi futuri

5.1 Conclusioni

In questo lavoro di tesi è stata affrontata l'implementazione di una soluzione di comunicazione Bluetooth Low Energy (BLE) tra una *Raspberry Pi* e un'applicazione mobile sviluppata con il framework *Flutter*. L'obiettivo principale era stabilire una connessione stabile tra i dispositivi, consentendo la trasmissione affidabile di dati.

5.1.1 Risultati e contributi

L'implementazione ha avuto successo nell'instaurare una connessione BLE tra la *Raspberry Pi* e l'app mobile realizzata con *Flutter*. La comunicazione avviene attraverso un servizio *GATT* e le relative caratteristiche *UART*. La soluzione ha superato con successo i test di interoperabilità, garantendo una corretta comunicazione tra i dispositivi.

Questo lavoro ha contribuito a comprendere gli aspetti chiave dell'implementazione BLE, tra cui la gestione della comunicazione attraverso il middleware *BlueZ* sulla *Raspberry Pi* e l'interazione con la libreria `flutter_reactive_ble` su *Flutter*. La strutturazione del software, basata sul design pattern *BLoC*, in particolare con l'uso di *Cubit*, ha fornito un'architettura modulare e manutenibile.

5.1.2 Considerazioni sull'apprendimento

L'esperienza di sviluppo di questa soluzione ha consentito una maggiore comprensione delle tecnologie Bluetooth Low Energy e delle peculiarità legate all'implementazione su dispositivi embedded come la *Raspberry Pi*. Inoltre, l'utilizzo del framework *Flutter* ha permesso di esplorare le sfide specifiche legate allo sviluppo di interfacce utente cross-platform.

5.2 Sviluppi futuri

Nonostante il successo nell'implementazione della soluzione, ci sono opportunità di miglioramento che possono arricchire e potenziare la soluzione attuale.

5.2.1 Validazione su più dispositivi e ambienti

La soluzione dovrebbe essere validata su una gamma più ampia di dispositivi e ambienti operativi per garantire la sua affidabilità e compatibilità universali. La considerazione di scenari reali e test su diversi dispositivi contribuiranno a consolidare ulteriormente la robustezza della soluzione.

5.2.2 Studio approfondito sull'interfaccia utente

Un'ulteriore area di sviluppo futuro potrebbe concentrarsi sull'ottimizzazione dell'interfaccia utente per garantire una visualizzazione chiara e non invasiva dei dati. Alcuni aspetti da considerare includono:

- **Progettazione di Interfacce Intuitive**

  Effettuare uno studio approfondito sull'usabilità dell'applicazione, focalizzandosi sulla progettazione di interfacce intuitive e facili da navigare. La semplificazione della presentazione dei dati può contribuire a evitare sovraccarichi informativi e garantire che il conducente possa accedere rapidamente alle informazioni più rilevanti.

- **Personalizzazione dell'interfaccia**

  Consentire al conducente di personalizzare l'interfaccia in base alle proprie preferenze potrebbe essere un ulteriore passo avanti. Questo potrebbe includere la possibilità di ridimensionare o riorganizzare gli elementi dell'interfaccia, garantendo una visualizzazione su misura per le esigenze specifiche di ciascun utente.

5.2.3 Implementazione di funzionalità aggiuntive

Il presente progetto rappresenta solo l'inizio di un percorso di sviluppo più ampio e ambizioso, lasciando spazio a numerosi potenziali miglioramenti e ampliamenti delle funzionalità. Attualmente, l'applicazione si concentra sulla trasmissione di dati veicolari attraverso la tecnologia Bluetooth Low Energy (BLE) e offre funzionalità di base per la scansione e la connessione a dispositivi compatibili.

Tuttavia, con una visione orientata al futuro, sono possibili diverse estensioni e miglioramenti che possono arricchire l'esperienza dell'utente e rendere l'applicazione più completa e versatile.

Barra di navigazione: bottom bar

Per migliorare l'esperienza dell'utente, si potrebbe considerare l'implementazione di una *bottom bar* nell'app mobile. Questa barra potrebbe fornire un accesso rapido a funzionalità aggiuntive, consentendo al conducente di accedere facilmente a informazioni di interesse, per rendere la navigazione semplice e facilmente gestibile.

Posizione del veicolo su mappa

Una funzionalità che potrebbe essere aggiunta all'interno della *bottom bar* è la visualizzazione del veicolo su mappa.
Integrare una mappa interattiva che visualizzi la posizione corrente del veicolo potrebbe essere estremamente utile per il conducente. Infatti, si potrebbero fornire informazioni sulla posizione in tempo reale, non solo migliorando la navigazione, ma anche consentendo al conducente di essere consapevole degli eventuali pericoli lungo il percorso e prendere decisioni informate per garantire la sicurezza durante il viaggio.

Referenze

[1] Kevin Townsend, Carles Cufı, Robert Davidson et al.
Getting started with Bluetooth low energy: tools and techniques for low-power networking. 'Reilly Media, Inc.", 2014.

[2] Martin Woolley.
The Bluetooth Technology for Linux Developers Study Guide.
https://www.bluetooth.com/blog/the-bluetooth-for-linux-developers-study-guide/.

[3] *Building user interfaces with Flutter*.
https://docs.flutter.dev/ui.

[4] *Flutter Clean Architecture [1]: An Overview Project Structure*.
https://dev.to/marwamejri/flutter-clean-architecture-1-an-overview-project-structure-4bhf.

[5] *bloc 8.1.2*.
https://pub.dev/packages/bloc.

[6] *flutter_reactive_ble 5.2.0*.
https://pub.dev/packages/flutter_reactive_ble.

[7] *Raspberry Pi 3 Model B+*.
https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/.

[8] *Autorizzazioni Bluetooth*.
https://developer.android.com/develop/connectivity/bluetooth/bt-permissions?hl=it.

Elenco delle figure

Ringraziamenti

È da tempo che mi immagino il giorno in cui avrei scritto i ringraziamenti sulla mia tesi. Quasi non mi sembra vero, eppure quel giorno è arrivato.

Voglio ringraziare il Prof. Di Martino, per avermi spronata a continuare questo percorso di studi. Fino a luglio 2023 non avrei mai immaginato di continuare anche con la specialistica e voglio ringraziarlo di cuore, perché è la scelta migliore che potessi fare.

Voglio ringraziare il Prof. Tramontana, il mio relatore, che mi ha sostenuta durante tutto il periodo del tirocinio, assistendomi in maniera eccelsa.

Voglio ringraziare la Netcom Group per avermi dato la possibilità di effettuare il tirocinio presso la loro azienda e accolta come un membro della loro famiglia.

Ringrazio il Dott. Francesco Altiero per avermi supportata e sopportata tutto il mese di luglio, inviandogli mattina, pomeriggio e sera esercizi di Algoritmi e Strutture Dati che ha corretto anche di Sabato e Domenica. Per lo meno ne è valsa la pena!

Ringrazio i ragazzi incontrati e conosciuti in questo primo semestre della specialistica: Alfredo, Fabiana, Fabio, Ferdinando, Francesco, Francesco Jr., Giulia, Mariano, Mario e Pasquale. (Sì Alfredo, siete in ordine alfabetico).
Voi mi avete assistita e motivata in ogni modo possibile durante questo semestre. Siete il motivo per cui nonostante le lezioni cominciassero alle 8:30 del mattino, io fossi sempre felice, nonostante l'ansia, nonostante la stanchezza. Siete delle bellissime persone e spero di conoscervi sempre di più!

Voglio ringraziare Simone, il mio ragazzo, migliore amico, compagno di avventure. Ci siamo incontrati per caso, per passioni comuni e ci siamo innamorati dal primo sguardo. Ne abbiamo passate veramente parecchie in questi quattro anni e non vedo l'ora di assistere a cosa ci metterà davanti il nostro futuro. Se sono riuscita a raggiungere questo traguardo è sicuramente anche grazie a te. Ti amo e... KIPITON!

Ringrazio i ragazzi di Smash:
Andrea, Antonio, Eugenio, Raffaele.

Abbiamo questa passione in comune che ci lega, che ci unisce sempre di più e io con voi mi diverto più che mai. A modo vostro, in contesti diversi, avete sempre fatto il tifo per me, ciò mi ha dato la carica per andare avanti in questo percorso.

Voglio ringraziare Gennaro Picone, per me sei come un fratello. Mi hai accolta in casa tua, abbiamo studiato esami insieme, in generale mi hai aiutata ogni qual volta pensassi fosse necessario e... mi hai incatenato il frigo di casa.

Ringrazio Federico Salzano, una delle prime persone che ho conosciuto ad Informatica. Mi hai aiutata con i primi esercizi di programmazione e nonostante non ci vediamo spesso, il nostro rapporto di amicizia va avanti da anni.

Ringrazio le donne splendide della mia famiglia, simbolo di determinazione e forza: mia nonna Nunzia, mia madre Tiziana, mia sorella Caterina e ultima, ma non per importanza, Tizianina mia nipote. Siete sempre state al mio fianco sempre e nonostante abbia fatto qualche scelta opinabile nella mia vita, mi avete sempre ricordato che la cosa più importante è il legame indissolubile che abbiamo.

Dedico un ringraziamento speciale alla mia dolce mamma. Hai dovuto affrontare una vita piena di difficoltà. Troppe cose ti sono state portate via troppo presto. Io però voglio ricordarti che sei la mia regina. Spero che tu ora possa essere fiera di me e so che lo sei sempre stata, ma so quanto fosse importante anche per te questa laurea, perché una delle tue priorità è sempre stata quella di volermi vedere felice e realizzata (ci sto ancora lavorando, ma siamo sul percorso giusto). Hai deciso di supportarmi fino alla fine e siccome non è scontanto ti voglio dire: grazie, ti amo.

Gli ultimi ringraziamenti sono dedicati a due persone che in questo giorno così importante per me, non ci potranno essere. La loro scomparsa ha segnato per sempre la mia vita:
nonno Spartaco e mio padre Maurizio.

Nonno, abbiamo vissuto in casa insieme e per me sei stato come un secondo papà. Nell'anima ho stampato tutto ciò che hai fatto per me. Eri un nonno sensazionale e la tua fama precedeva il tuo nome tra i miei amici. Abbiamo fatto un sacco di corse per arrivare in orario a scuola e all'università, fino a quando ho preso la patente. Ci siamo divertiti tantissimo e io custodirò per sempre nel cuore, la danza che abbiamo fatto al compleanno dei miei 18 anni. Grazie.

Papà, non so neanche da dove cominciare, se volessi ringraziarti per ogni cosa che hai fatto per me, probabilmente dovrei scrivere una tesi a parte.
Cerco di essere forte tutti i giorni, ma ammetto che a volte, nonostante il tempo che passa, mi sembra come se ci avessi lasciato ieri. So quanto ti fidassi di me e ti prometto con tutto

il cuore, che darò sempre il massimo. Non ho alcun dubbio sul fatto che, da qualche parte, starai festeggiando per il traguardo di tua figlia. Ti amo.