Cellular Connectivity and Noise Map

Relazione

Xia · Tian Cheng
Matricola: 0000975129
Email: tiancheng.xia@studio.unibo.it

Anno accademico
2022 – 2023

Corso di Laboratorio di applicazioni mobili
Alma Mater Studiorum · Università di Bologna

Indice

1 Introduzione

1.1 Feature implementate

Di seguito sono elencate le feature implementate:

- Mappa partizionata in aree non sovrapposte con ridimensionamento automatico delle celle in base al livello dello zoom (Figura 1).
- Range della qualità delle misurazioni calcolato automaticamente, con possibilità di scegliere il numero di classi da creare (Figura 2).
- Misurazione di Wi-Fi, LTE, Bluetooth e suono con le seguenti modalità:
  - Attiva: su comando dell'utente.
  - Passiva: dopo un determinato intervallo temporale o durante il movimento.
  - In background: durante il movimento.
- Filtro di ricerca per alcune tipologie di misurazioni (Wi-Fi e Bluetooth)
- Notifiche per aree prive di misurazioni recenti.
- Esportazione su file e importazione delle misurazioni (Figura 3).

1.2 Screenshot applicazione

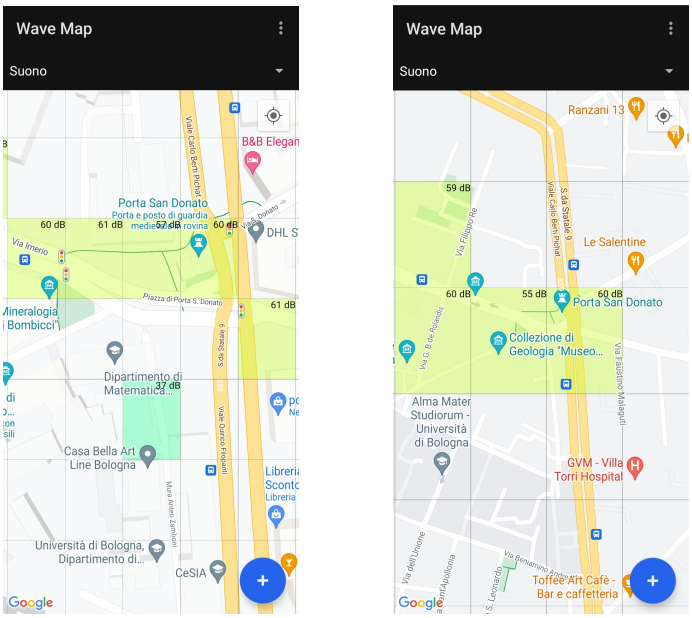
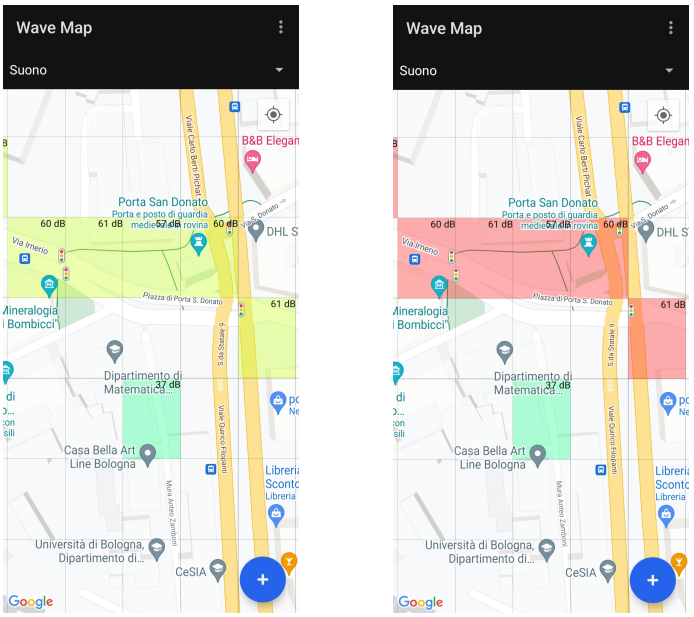


Figura 1: Mappa con celle ridimensionate in base al livello di zoom



Suddivisione in 3 range Suddivisione in 2 range

Figura 2: Range misurazioni calcolati algoritmicamente

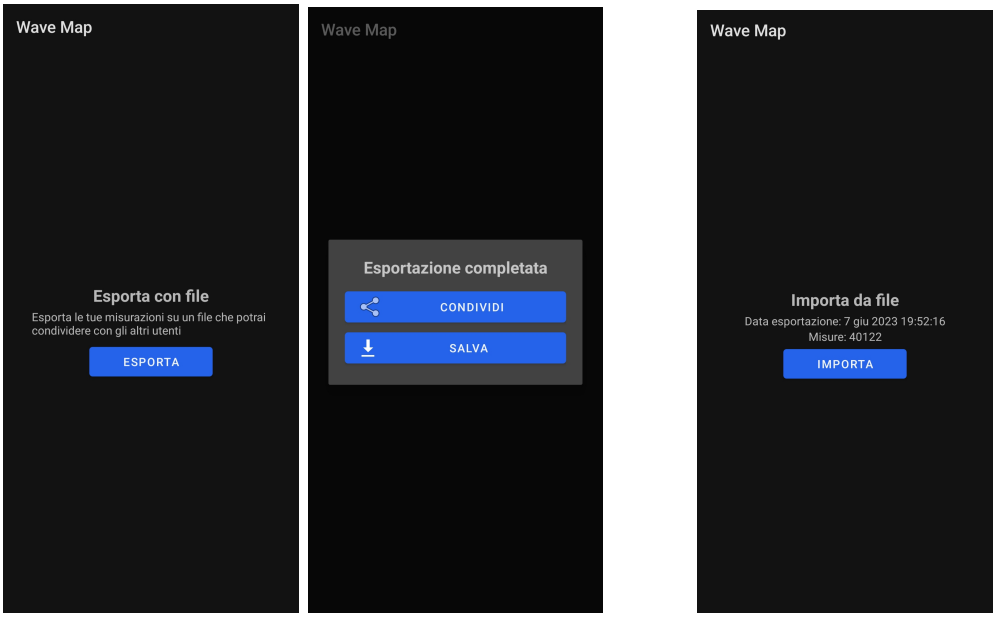


Figura 3: Esportazione e importazione da file

2 Scelte progettuali

2.1 Informazioni generali

Il progetto è stato sviluppato come applicazione nativa utilizzando Kotlin. Come pattern architetturale è stato scelto l'approccio Model-View-ViewModel, mentre per l'interfacciamento con il database locale è stata utilizzata la libreria `Room`.
Le operazioni asincrone sono state principalmente implementate tramite le `coroutine` e per favorire un codice più "lineare", quando possibile, sono state trasformate le funzioni con callback in funzioni `suspend` utilizzando come wrapper `suspendCoroutine`.

2.1.1 Organizzazione package

In Tabella 1 è descritta l'organizzazione dei package:

`db`	Classi che implementano la struttura e le operazioni sulle tabelle del database.
`dialogs`	Classi per istanziare i dialog utilizzati nell'applicazione.
`notifications`	Classi per istanziare le notifiche utilizzate nell'applicazione.
`measures`	Classi che implementano le operazioni per effettuare le misurazioni.
`services`	Classi che implementano i servizi in background.
`ui`	`Activity`, `Fragment` e `ViewModel` dell'applicazione.
`utilities`	Metodi e variabili di utilità generale.

Tabella 1: Organizzazione package

2.2 Mappa

Per la mappa è stato utilizzato *Google Maps* e l'implementazione è contenuta nel fragment `WaveHeatMapFragment`.

2.2.1 Generazione cella

Una cella della mappa rappresenta la misurazione di un'area quadrata[1] e la dimensione di quest'ultima scala automaticamente in base al livello dello zoom.
Una cella è descritta dalle coordinate del vertice superiore sinistro (nord-ovest) e a partire da questa vengono calcolate le coordinate degli altri convertendo la dimensione della cella (in metri) in un offset da applicare a latitudine e longitudine. Per questioni estetiche, gli offset sono approssimati in modo tale che tutte le righe siano allineate verticalmente (Figura 4).

[1] Esclusa la zona equatoriale, le celle appariranno rettangolari

Figura 4: Offset calcolati in maniera precisa (sinistra) e approssimata (destra)

Il colore di una cella è determinato dal valore e dal tipo di misurazione, e viene impostato in formato HSV modificando il valore della tinta (hue). Per ciascuna tipologia di misurazione sono specificati gli estremi del range di qualità e il numero di intervalli in cui suddividerlo. La tinta è quindi determinata collocando il valore della misurazione in uno dei sotto-intervalli del range e selezionando come valore finale l'estremo inferiore (Algoritmo 1).

Algoritmo 1: Tinta di una cella

```
1 fun getHue(measure, measure_range=[a, b], hue_range=[0, 150], n_ranges)
2 |   hue ← measure scalata da measure_range a hue_range
3 |   hue ← hue discretizzata in hue_range suddiviso in n_ranges valori
4 |   return hue
5 end
```

Infine, per ogni cella è presente un'etichetta contenente il valore della misurazione. Poiché *Google Maps* non permette di inserire esplicitamente del testo nella mappa, l'implementazione prevede di generare l'etichetta come un'immagine che viene poi impostata come icona di un marker. Inoltre, in caso di necessità, la dimensione del testo viene scalata per far in modo che rientri nei limiti della cella.

2.2.2 Generazione griglia

La griglia è composta da celle generate relativamente ad una posizione di riferimento. In particolare, in fase di inizializzazione viene designata come cella di riferimento quella che pone la posizione dell'utente al centro e in base a questa è possibile determinare la posizione di tutte le altre celle della mappa.
Nello specifico, date delle coordinate $(\texttt{pos}_{\texttt{lat}}, \texttt{pos}_{\texttt{lon}})$, per determinare la cella che la contiene si calcola il numero di celle da saltare rispetto a quella di riferimento:

$$\texttt{to\_skip\_tiles}_{\texttt{lat}} = \lceil \frac{\texttt{pos}_{\texttt{lat}} - \texttt{center\_top\_left}_{\texttt{lat}}}{\texttt{latitudeOffset(tile\_length\_meters)}} \rceil$$

$$\texttt{to\_skip\_tiles}_{\texttt{lon}} = \lfloor \frac{\texttt{pos}_{\texttt{lon}} - \texttt{center\_top\_left}_{\texttt{lon}}}{\texttt{longitudeOffset(tile\_length\_meters)}} \rfloor$$

Le coordinate del vertice superiore sinistro della cella che contiene $(\texttt{pos}_{\texttt{lat}}, \texttt{pos}_{\texttt{lon}})$ sono quindi:

$$\texttt{tile}_{\texttt{lat}} = \texttt{center\_top\_left}_{\texttt{lat}} + (\texttt{to\_skip\_tiles}_{\texttt{lat}} \cdot \texttt{latitudeOffset(tile\_length\_meters)})$$

$\texttt{tile}_{\texttt{lon}} = \texttt{center\_top\_left}_{\texttt{lon}} + (\texttt{to\_skip\_tiles}_{\texttt{lon}} \cdot \texttt{longitudeOffset(tile\_length\_meters)})$

Con questo approccio, ogni volta che viene spostata la visuale della mappa, la griglia viene generata iterando a partire dalle coordinate dell'angolo nord-ovest visibile dello schermo, fino a raggiungere l'angolo sud-est.
In aggiunta, per maggiore efficienza, si tiene traccia delle celle generate in modo da evitare di dover ridisegnare l'intera area visibile ad ogni movimento della mappa. Questo meccanismo viene resettato quando viene cambiato il livello di zoom, in quanto tutte le celle già presenti diventano obsolete e vengono necessariamente cancellate.

2.3 Raccolta dei dati

2.3.1 Struttura e memorizzazione delle misurazioni

Una misurazione è descritta dall'interfaccia `WaveMeasure` e contiene il valore della misurazione, un timestamp, la posizione e un flag per indicare se si tratta di una misurazione propria o ottenuta tramite condivisione. In aggiunta, è presente un campo per informazioni aggiuntive utile per distinguere alcune tipologie di misurazioni (es. per Wi-Fi e Bluetooth viene salvato il BSSID).
L'interfaccia `WaveMeasure` viene quindi utilizzata per implementare la classe `MeasureTable` che descrive la tabella del database dedicata per memorizzare le misurazioni. Tutte le misurazioni sono salvate nella stessa tabella e sono differenziate da un campo (`type`) aggiunto in fase di salvataggio nel database. Inoltre, ciascuna misura è identificata univocamente da un UUID, utile anche per differenziare le misurazioni provenienti da altri utenti.
In aggiunta, una seconda tabella descritta da `BSSIDTable` contiene la mappatura da BSSID a SSID.

2.3.2 Sampler

Per la raccolta dei dati è stato introdotto il concetto di *sampler* per gestire in maniera modulare le misurazioni. Nello specifico, un *sampler* è descritto dalla classe astratta `WaveSampler` e richiede l'implementazione dei seguenti metodi:

- `sample` per prendere una nuova misurazione.
- `store` per il salvataggio dei dati nel database.
- `retrieve` per la ricerca dei dati note le coordinate dei vertici di una cella della mappa.

Inoltre, sono esposte le seguenti funzioni ausiliarie:

- `average` richiama `retrieve` e restituisce la media dei valori.
- `sampleAndStore` richiama in sequenza `sample` e `store`.

Per maggiore flessibilità, le misure vengono sempre intese come liste di `WaveMeasure`. Ciò permette di gestire misurazioni che per loro natura non generano un'unica misurazione (es. Wi-Fi e Bluetooth).
A partire da `WaveSampler` sono quindi implementati i *sampler* per:

- Wi-Fi (`WiFiSampler`):
  - Ottiene la potenza della rete al quale il dispositivo è attualmente connesso tramite il servizio di sistema `ConnectivityManager`. Per versioni inferiori alla API 29, viene invece utilizzato il `WifiManager`.
  - Misura la potenza delle reti circostanti registrando un `BroadcastReceiver` con filtro `WifiManager.SCAN_RESULTS_AVAILABLE_ACTION` e richiedendo una scansione completa attraverso il `WifiManager`.
- Bluetooth (`BluetoothSampler`):
  - Misura la potenza dei dispositivi accoppiati mediante il `BluetoothManager`.
  - Misura la potenza dei dispositivi circostanti registrando un `BroadcastReceiver` con filtro `BluetoothDevice.ACTION_FOUND` e richiedendo al `BluetoothManager` una scansione completa.
- LTE (`LTESampler`): ottiene la potenza del segnale LTE tramite il `TelephonyManager`.
- Suono (`NoiseSampler`): viene fatta la media di una serie di campionature effettuate utilizzando un `MediaRecorder`.

2.4 App principale

2.4.1 ViewModel dei *sampler*

Per ciascun *sampler* è stato definito un ViewModel dedicato contenete le operazioni e le informazioni per ciascun tipo di misurazione. In particolare, è stata definita una gerarchia di ViewModel come rappresentato in Figura 5.

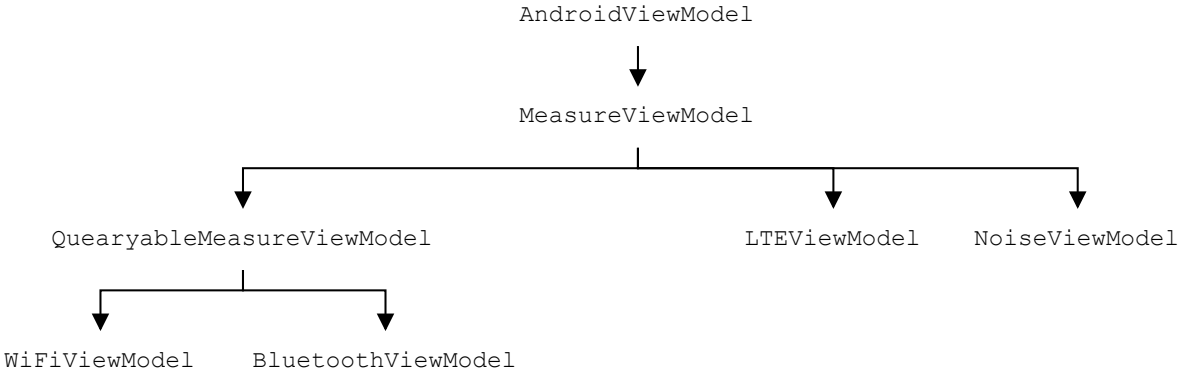


Figura 5: Gerarchia dei ViewModel

La gerarchia ha come radice `AndroidViewModel` in quanto fornisce il `Context` dell'applicazione necessario per i *sampler*.
La classe astratta `MeasureViewModel` fornisce tutte le informazioni utili per una tipologia di misurazione. Specificamente, contiene l'istanza del `WaveSampler` e i parametri quali: il range di qualità, il numero di intervalli, il numero di misure passate da considerare e un

flag per indicare se caricare anche le misure condivise. Vengono inoltre esposti i metodi per effettuare una nuova misurazione, ottenere la media delle misurazioni di una casella e caricare le impostazioni specifiche per il tipo di misura. Da `MeasureViewModel` sono implementati `LTEViewModel` e `NoiseViewModel`.
`MeasureViewModel` viene poi estesa in un'ulteriore classe astratta `QuearyableMeasureViewModel` per rappresentare un tipo di misura a cui è possibile sottoporre un filtro di ricerca. È infatti richiesta l'implementazione di un metodo per elencare le possibili opzioni di ricerca e per cambiare la ricerca. Da questo tipo di ViewModel sono implementati `WiFiViewModel` e `BluetoothViewModel`, in entrambi i casi la ricerca si basa sul BSSID.

2.4.2 MainViewModel

Il ViewModel `MainViewModel` implementa la logica principale dell'applicazione. Contiene un'istanza di tutti i ViewModel dei *sampler* (in un vettore) e ad ogni istante ne considera uno come attivo (identificato dall'indice del vettore). Inoltre, per ogni *sampler* tiene traccia dello stato (in misurazione e non) e garantisce mutua esclusione per ogni singolo ViewModel in modo tale da evitare di avviare più misurazioni della stessa tipologia.
Attraverso dei `LiveData`, `MainViewModel` comunica alla view eventi come: l'inizio e la fine di una misurazione, la necessità di aggiornare la mappa ed eventuali situazioni di errore.
Inoltre, `MainViewModel` gestisce le scansioni periodiche (se attivate nelle impostazioni) utilizzando un `Handler` e il metodo `postDelayed`.
Bisogna infine notare che tutte le operazioni asincrone sono legate allo scope `viewModelScope` in modo tale che, in caso la view venisse ricreata, le operazioni in corso non vengono interrotte.

2.4.3 MainActivity

La `MainActivity` è il punto di entrata dell'applicazione e comprende la mappa e i comandi dell'utente. Oltre a istanziare e interfacciarsi al `MainViewModel`, gestisce la richiesta dei permessi e gli eventi riguardanti i cambiamenti di casella provenienti da `WaveHeatMapFragment`.
Inoltre, si occupa di avviare e fermare i servizi in background durante la fase di creazione e rimozione del proprio ciclo di vita.

2.5 Impostazioni

Le impostazioni sono implementate con la libreria `Preferences`[2].
Il punto di entrata è implementato nell'activity `SettingsActivity` che carica il fragment contenente le impostazioni principali descritte in `main_preferences.xml`.
In aggiunta, ciascun *sampler* ha una propria pagina dedicata contenente impostazioni specifiche. Poiché tutti i *sampler* hanno le stesse proprietà nelle impostazioni, è stata creata come classe radice `MeasureSettingsFragment` che estende `AppCompatActivity` e genera dinamicamente i vari campi della pagina, senza avere la necessità di descriverli separatamente in un file `xml`. Quindi per ciascun *sampler* è implementata una classe che eredita `MeasureSettingsFragment` inizializzata con i parametri specifici necessari e

[2] `https://developer.android.com/develop/ui/views/components/settings/organize-your-settings`

quest'ultima viene utilizzata come destinazione del collegamento presente nella pagina principale delle impostazioni.

2.6 Servizi in background

La classe `BackgroundScanService` estende `Service` e implementa le funzionalità dell'applicazione attive in background. Durante la creazione, è necessario specificare nell'`Intent` i sotto-servizi richiesti (scansione in background e notificazione di aree prive di misurazioni recenti).
In fase di avvio del servizio, viene inizializzato un `FusedLocationProviderClient` impostato per fornire periodicamente la posizione del dispositivo. In particolare, gli aggiornamenti vengono notificati dopo che è stata percorsa una distanza minima dalla posizione precedente e hanno una priorità impostata a `PRIORITY_BALANCED_POWER_ACCURACY` per preservare la batteria del dispositivo.
Le operazioni effettuate alla ricezione di un aggiornamento della posizione sono (assumendo che siano abilitate nelle impostazioni):

1. Verifica se l'area è coperta da misurazioni recenti, in caso negativo viene inviata una notifica. Per evitare di inviare notifiche troppo frequentemente, viene tenuto traccia del momento in cui è stata inviata quella più recente in modo tale da poter ignorare le notifiche successive se dovessero essere create a distanza troppo ravvicinata.

2. Effettua una misurazione completa utilizzando tutti i *sampler*.

Nel caso in cui sia necessario effettuare misurazioni in background, il servizio viene avviato come *foreground service*. Ciò è necessario in quanto l'accesso al microfono non è consentito per servizi in background; inoltre, in questo modo si ha una maggiore trasparenza nei confronti dell'utente che è informato tramite una notifica sul fatto che l'applicazione stia effettuando delle scansioni anche quando l'applicazione non è attivamente in uso.

2.7 Condivisione dati

Le operazioni per l'esportazione e l'importazione dei dati sono implementate come metodi statici della classe `ShareMeasures`.
La condivisione dei dati avviene tramite la creazione di un file contenente le misurazioni. Per maggiore interoperabilità con eventuali estensioni, i dati vengono salvati in formato JSON e per tale scopo viene utilizzata la libreria `Gson`.
Per facilitare l'estensione delle funzionalità di condivisione, la navigazione tra i fragment dedicati alla condivisione è stato implementato utilizzando Navigation component.

2.7.1 Esportazione

L'esportazione è gestita dal fragment `FileExportFragment` e ViewModel `FileExportViewModel`.
Durante la fase di esportazione, tutte le misurazioni e tutti i BSSID salvati nel database vengono convertiti in una stringa JSON, assieme ad alcuni metadati. Il risultato è quindi temporaneamente salvato nella cache dell'applicazione.
L'utente ha quindi la possibilità di salvare il risultato in un file locale nella cartella *Downloads* oppure di condividerlo attraverso un `Intent` di tipo `ACTION_SEND`.

2.7.2 Importazione

L'importazione è gestita dall'activity `ImportActivity` e ViewModel `ImportViewModel`.
`ImportActivity` ha come `intent-filter` azioni del tipo `VIEW` e `SEND`, e quindi permette di selezionare l'azione di importazione quando l'utente apre o condivide un file.
Una volta aperto e validato un file di misurazioni, l'importazione consiste nell'inserire nel database tutte le misurazioni presenti marcandole come misure ottenute per condivisione. Per evitare duplicati, l'UUID viene usato come discriminante e vengono quindi ignorate tutte le misurazioni già presenti.
Analogamente, la tabella dei BSSID viene importata escludendo le righe già note.

3 Problemi noti

3.1 Scansione Wi-Fi

3.2 Servizio in background

3.3 Importazione