Efficientamento energetico di una applicazione Android Open-Source: RoboDrink

Corso di Operating systems for mobile, cloud and IoT, Informatica

Prof. Porfirio Tramontana, Anno Accademico 2022/2023

DIETI - Università Degli Studi di Napoli Federico II

Erasmo Prosciutto, N86003546
Biagio Scotto di Covella N86003605

October 5, 2023

Il progetto completo con le ottimizzazioni apportate e i risultati ottenuti è disponibile al link: `https://github.com/biagioSc/Progetto-Operating_systems_for_mobile_cloud_and_IoT`

Contents

1 Introduzione

Le applicazioni mobile sono ormai parte integrante della nostra vita, fornendo una vasta gamma di servizi e funzionalità che semplificano molte attività quotidiane. Tuttavia, il continuo aumento della potenza di elaborazione e delle capacità grafiche nei dispositivi mobili ha portato a un consumo energetico significativo. Questo ha importanti implicazioni sia per gli utenti che per l'ambiente. Gli utenti desiderano dispositivi che abbiano una durata della batteria più lunga e che non si scarichino rapidamente durante l'uso, mentre l'aumento del consumo energetico ha un impatto negativo sul nostro pianeta, contribuendo al cambiamento climatico. Uno studio del 2013 ha riportato che il 18% delle applicazioni ha un feedback da parte degli utenti in relazione al consumo di energia [12].

Nei sistemi informatici in generale, l'efficientamento energetico è fondamentale per garantire un funzionamento sostenibile e redditizio. I data center, ad esempio, che ospitano una vasta gamma di servizi online, richiedono enormi quantità di energia per alimentare i Server e per il raffreddamento. Ridurre il consumo energetico di quest'ultimi non solo aiuta a risparmiare costi, ma contribuisce anche a ridurre l'impronta di carbonio dell'industria IT.

Vari sono gli approcci possibili per migliorare questi aspetti, tra cui l'ottimizzazione del codice, l'uso di hardware a basso consumo energetico e lo sviluppo di algoritmi efficienti. Inoltre, la consapevolezza degli sviluppatori e degli utenti sull'importanza di questo problema sta crescendo, spingendo verso soluzioni più sostenibili.

1.1 Obiettivi

Nelle evoluzioni di Android si è curato sempre più il consumo energetico e di risorse, come ad esempio la "Doze Mode", una modalità di risparmio energetico che si attiva quando il telefono è inattivo e che permette di applicare delle restrizioni come: ignorare i wake locks, sospendere l'accesso alla rete, ecc. In generale attualmente, una applicazione che non viene utilizzata in un certo arco temporale viene categorizzata e in base a questa classificazione avrà diversa priorità nell'utilizzo delle risorse.

Data l'importanza che ha l'efficientamento energetico nei dispositivi mobile, il nostro lavoro si propone di analizzare, ottimizzare e testare il cosumo energetico di una applicazione Android, utilizzando tecniche e strumenti allo stato dell'arte. Ci baseremo sulla letteratura scientifica per approfondire quali sono gli strumenti che permettono l'analisi di applicazioni mobile dal punto di vista dell'efficienza energetica, quali sono le differenze che contraddistinguono un applicativo efficiente da uno che non lo è, le migliorie che possono essere effettuate a vari livelli di astrazione, i risultati prima e dopo le ottimizzazioni.

1.2 Struttura

Il documento è strutturato come segue: nella sezione 2 viene data una panoramica sui principali studi e pratiche legate all'ottimizzazione energetica di applicazioni Android; nella sezione 3 sono descritti i principali tools utilizzati nelle analisi; nella sezione 4 sono indicate le best practices generali per l'ottimizzazione energetica; nella sezione 5 sono descritte le varie modalità di risparmio energetico offerte da Android; nella sezione 6 è descritta l'applicazione sulla quale verranno svolti i test; nella sezione 7 viene fatta una panoramica sulla configurazione necessaria pre-analisi; nella sezione 8 sono descritte le ottimizzazioni effettuate; nella sezione 9 sono descritti i risultati ottenuti dalle analisi effettuate e i relativi confronti; nella sezione 10 sono indicati tutti i passaggi per ripetere tutti i test svolti; infine nella sezione 11 traiamo le conclusioni sul lavoro svolto e sui risultati ottenuti.

2 Related work

Nella ricerca di soluzioni per l'efficientamento energetico delle applicazioni Android, è fondamentale considerare le conoscenze e gli studi precedenti in questo campo. Questa sezione fornirà una panoramica delle principali scoperte e pratiche legate all'ottimizzazione del consumo energetico delle applicazioni Android.

Numerose ricerche si sono concentrate sull'ottimizzazione del codice per ridurre il consumo energetico delle applicazioni Android. Ad esempio, lo studio condotto in [2] ha dimostrato che l'analisi e l'ottimizzazione del codice possono portare a significativi risparmi energetici.

Studi inerenti alla gestione efficiente delle risorse e ai "Memory-Leaks" come il [4] hanno evidenziato che la mancata chiusura corretta delle risorse, come cursori del database o connessioni di rete, può comportare un consumo eccessivo di energia. L'uso di blocchi "try-with-resources" in Java è stato raccomandato come pratica per garantire una corretta gestione delle risorse.

Un altro aspetto molto spesso sottovalutato è rappresentato dall'interfaccia grafica. Ricerche come la [11] hanno fornito linee guida dettagliate per la progettazione di interfacce utente a basso consumo energetico. L'adozione di tecniche come l'uso di ViewStub per il caricamento ritardato di elementi dell'interfaccia può ridurre il consumo energetico.

Come approfondito in [1], è estremamente importante utilizzare Wake Locks in modo intelligente per impedire che lo schermo si spenga durante l'utilizzo dell'app. Questa pratica di gestione della batteria può migliorare l'esperienza dell'utente e limitare gli sprechi energetici.

Ogni dispositivo inoltre, è quotidianamente connesso ad Internet ed è importante limitare le comunicazioni allo stretto necessario. Ricerche come la [8] hanno analizzato come molte applicazioni Android possono essere ottimizzate dal punto di vista energetico avendo una migliore gestione dei servizi di rete, specialmente quelli in background, molto spesso utilizzati senza che ve ne sia un reale bisogno.

In conclusione, un altro aspetto importante riguarda la gestione dei sensori, come ad esempio quelli utilizzati per i servizi di localizzazione. Studi come [6] hanno esaminato l'ottimizzazione energetica dei sensori GPS. L'attivazione e la disattivazione intelligente dei sensori in base alle esigenze dell'app possono ridurre significativamente il consumo energetico.

3 Tools

3.1 Android Studio Profiler

Oltre all'introduzione di tecniche per il risparmio energetico, nelle ultime versioni di Android sono stati messi a disposizione strumenti per il monitoraggio dell'utilizzo di risorse ed in particolare del consumo energetico, anche per gli emulatori. Parliamo di Android Studio Profiler, uno strumento molto potente che utilizzeremo per effettuare tutte le analisi del caso. Di seguito vi è una breve descrizione delle potenzialità di questo strumento.

- `Utilizzo della CPU`: il grafico mostrerà un profilo dell'utilizzo della CPU nel tempo. Si possono identificare picchi di utilizzo e trovare parti del codice che richiedono una quantità eccessiva di risorse CPU. Inoltre, è possibile vedere le trace dei metodi per individuare specifiche chiamate di funzioni che causano problemi.
- `Consumo energetico`: il grafico mostrerà come il consumo energetico varia nel tempo. Si possono individuare i picchi di consumo energetico e le relative cause, come servizi in background o chiamate di rete in eccesso.
- `Allocazione della memoria Java`: il grafico mostrerà come la memoria viene allocata e deallocata nel tempo. Si possono individuare eventuali aumenti anomali dell'allocazione di memoria e identificare le parti del codice responsabili.
- `Traffico di rete`: il grafico mostrerà le richieste di rete in entrata e in uscita, i tempi di risposta e la quantità di dati scambiati. Si possono identificare richieste in eccesso o inefficienze nella gestione delle chiamate di rete.

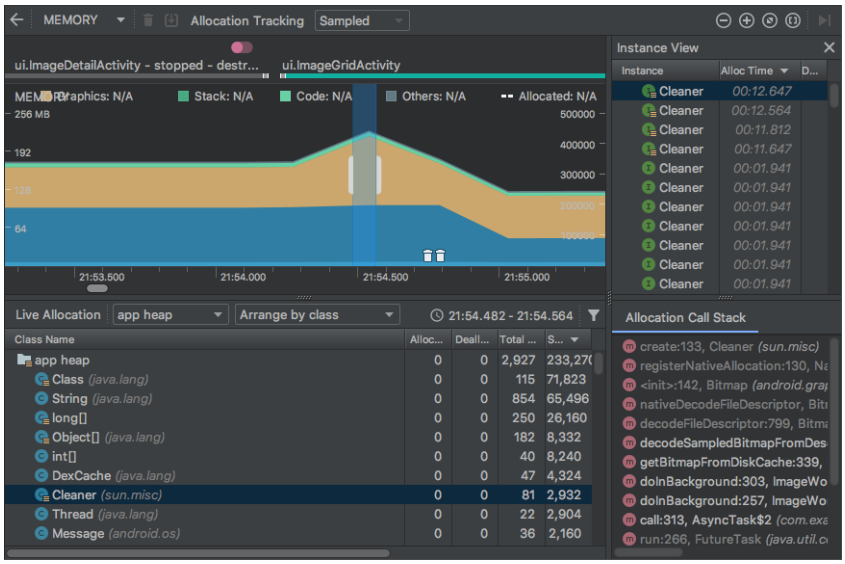

Figure 1: Android Studio Profiler: esempio di monitoraggio

3.2 Android Battery Historian

Android Battery Historian è uno strumento di analisi delle prestazioni sviluppato da Google per esaminare il consumo energetico delle applicazioni Android. Questo strumento è particolarmente utile per gli sviluppatori e i tester che vogliono comprendere come un'applicazione influenza il consumo della batteria di un dispositivo Android.

- `Raccolta dei dati:` Per utilizzare Android Battery Historian, è necessario raccogliere un registro (log) della batteria del dispositivo Android. Questo registro può essere acquisito da un dispositivo fisico o da un emulatore Android in esecuzione.
- `Formato del log:` Il registro della batteria deve essere nel formato bugreport di Android. E' possibile ottenerlo utilizzando il comando `adb bugreport` o utilizzando strumenti di raccolta dei dati delle prestazioni, come il Battery Historian Collector.

- `Caricamento dei dati:` Una volta ottenuto il registro della batteria, è possibile caricarlo nell'interfaccia web di Android Battery Historian. Questa interfaccia è accessibile attraverso un browser web.
- `Visualizzazione dei dati:` Android Battery Historian analizza il registro e genera una serie di grafici e report che mostrano il consumo energetico dell'applicazione nel tempo. Questi dati includono informazioni dettagliate sulle applicazioni in esecuzione, i wakelock, le connessioni di rete, i sensori attivati, il consumo CPU e altro ancora.
- `Identificazione dei problemi:` Con Android Battery Historian, è possibile identificare con precisione quali parti dell'applicazione contribuiscono in modo significativo al consumo energetico. Ad esempio, potresti scoprire che un wakelock non rilasciato è responsabile del consumo eccessivo della batteria e risolverlo per migliorare le prestazioni dell'app.
- `Strumento di diagnostica:` Oltre a mostrare i dati di consumo energetico, Android Battery Historian offre strumenti di diagnostica per individuare e risolvere i problemi relativi al consumo energetico nelle applicazioni Android.
- `Integrazione con Android Studio:` E' possibile integrare Android Battery Historian con Android Studio per semplificare il processo di raccolta dei dati di consumo energetico e di analisi delle prestazioni.
- `Uso avanzato:` Android Battery Historian può essere utilizzato per analizzare i dati di consumo energetico in modo più avanzato, come la creazione di profili di consumo energetico su misura per le app, il confronto delle prestazioni tra diverse versioni dell'app e altro ancora.

3.3 PowDroid

PowDroid è uno strumento basato su software progettato per misurare e profilare il consumo energetico delle applicazioni Android.

I dati estratti da PowDroid sono ottenuti dal sistema Android stesso e fungono da base di riferimento nel nostro approccio. Questo strumento sfrutta tool già esistenti nel framework Android, tra cui:

1. `Batterystats`: Un'utility inclusa nel framework Android che raccoglie dati grezzi sulla batteria di un dispositivo. Il comando corrispondente in Android Debug Bridge (ADB) è `adb shell batterystats`. Per evitare l'accumulo di dati tra le sessioni di test, è possibile pulire la cronologia utilizzando il comando `adb shell batterystats reset`. I dati risultanti sono memorizzati nel dispositivo in un file chiamato `data/local/tmp/battery.txt`.
2. `Bugreport`: Un altro strumento del framework Android che genera un file di report in formato ZIP sul dispositivo basato sui dati di `battery.txt` sopra menzionato. Il comando corrispondente è `adb bugreport [nome file].zip`. Il formato di un file "bugreport" è compatibile con Battery Historian.
3. `Battery Historian`: Battery Historian2 di Google converte il report di Bugreport in una visualizzazione basata sul web, che può essere visualizzata tramite un browser web. È possibile utilizzare uno script Go chiamato `local_history_parse.go` per convertire un file ZIP in un file CSV leggibile dall'utente. È importante notare che l'argomento `-summary=totalTime` viene utilizzato per produrre risultati basati sul tempo anziché sull'evoluzione della carica della batteria.
4. `Script di elaborazione energetica`: Sono stati sviluppati diversi script Python per gestire il flusso di lavoro di PowDroid. Tra questi, `PowDroid.py` è il programma principale, `split.py` suddivide un file CSV in base alle diverse metriche, e `Merge.py` unisce le metriche in sincronia con le finestre temporali.

4 Best Practices per l'ottimizzazione energetica

L'ottimizzazione energetica delle applicazioni Android è una componente essenziale nella progettazione e nello sviluppo di software per dispositivi mobili. Le applicazioni che consumano troppa energia possono avere un impatto negativo sull'esperienza dell'utente e sulla durata della batteria dei dispositivi, influenzando la loro adozione e la soddisfazione dell'utente.

In questo contesto, esamineremo una serie di pratiche e strategie per ridurre il consumo energetico delle applicazioni Android. Queste pratiche si basano su ricerche scientifiche e linee guida sviluppate dalla comunità degli sviluppatori Android ed hanno dimostrato di essere efficaci nell'ottimizzare il consumo energetico.

4.1 Ottimizzazione del Codice

- `Thread e Background Tasks`: Gestire attentamente l'utilizzo dei thread e dei processi in background. Eseguire operazioni intensive sul thread principale solo quando è necessario, altrimenti utilizzare thread separati o AsyncTask per mantenere l'interfaccia utente reattiva e risparmiare energia.

```
// Esempio di AsyncTask per operazioni in background
private class MyAsyncTask extends AsyncTask<Void, Void, Void> {
    @Override
    protected Void doInBackground(Void... params) {
        // Esegui il lavoro in background qui
        return null;
    }

    @Override
    protected void onPostExecute(Void result) {
        // Aggiorna l'interfaccia utente dopo il completamento del
            lavoro
    }
}
```

- `Risorse Efficienti`: Chiudere e rilasciare correttamente le risorse quando non sono più necessarie. Ad esempio, se si sta utilizzando un'istanza di InputStream, assicurarsi di chiuderla dopo l'utilizzo per evitare potenziali perdite di memoria e sprechi di risorse.

- `Ottimizzazione delle Query al Database`: Se l'app utilizza un database, ottimizzare le query in modo da limitare l'accesso frequente al database. Utilizzare le operazioni di indicizzazione e caching per ridurre la quantità di dati letti e scritti dal database.

```
// Esempio di query efficiente utilizzando SQLite
String query = "SELECT * FROM table WHERE column = ?";
Cursor cursor = database.rawQuery(query, new String[] { "value" });
// Gestisci il risultato della query
```

- `Eliminazione dei getter e setter`: Evidenze scientifiche dimostrano che l'omissione dei metodi `get/set` porta a un codice più veloce ed energeticamente più efficiente rispetto a un codice che li include. Esperimenti condotti hanno registrato una riduzione dei tempi di esecuzione compresa tra il 24% e il 30%, insieme a un calo dei consumi energetici dal 24% al 27%. [10].

- `Gestione delle strutture dati`: Secondo uno studio sperimentale [7], l'adozione di specifiche strutture dati del Java Collection Framework (JCF) può influenzare positivamente il consumo energetico complessivo. Quindi in una applicazione che fa un ampio uso di strutture dati come `HashMap` e `ArrayList`, è possibile sostituire `HashMap` con `SparseArray`, una collezione introdotta in Android. `SparseArray` mostra prestazioni superiori in termini di consumo energetico e utilizzo della memoria rispetto a `HashMap` [9]. Inoltre, studi come [3], rilevano che è possibile sostituire le classi del JCF che estendono la classe `Queue` con altre opzioni più efficienti dal punto di vista energetico. Ad esempio, è possibile sostituire gli `ArrayList` con la classe `LinkedBlockingDeque`.

- `Sostituzione del Binding dinamico con quello statico`: Ricerche hanno evidenziato che le chiamate di metodi con binding statico hanno un consumo energetico inferiore del 15% rispetto alle chiamate di metodi con binding dinamico. Tale efficienza è probabilmente attribuibile alla riduzione del carico di ricerca associato alle chiamate di metodi con binding dinamico [5].

4.2 Gestione delle Risorse

- `Cache Efficienti`: Implementare cache per archiviare temporaneamente dati o risorse grafiche frequentemente utilizzate. Questo ridurrà la necessità di recuperare dati da fonti esterne come una rete o un database, risparmiando energia.

- `Ottimizzazione delle Risorse Grafiche`: Utilizzare immagini con risoluzione appropriata e formati di immagine efficienti (come WebP) per ridurre le dimensioni dei file e il consumo di memoria. Adattare le risoluzioni delle immagini alle diverse densità di pixel dei dispositivi Android per evitare il ridimensionamento.

```
    <!-- Esempio di risorse grafiche per densita di pixel diverse -->
res/
    drawable-mdpi/
        my\textunderscore image.png
    drawable-hdpi/
        my\textunderscore image.png
    drawable-xhdpi/
        my\textunderscore image.png
```

- `Controllo delle Risorse in Background`: Ridurre al minimo l'uso delle risorse quando l'app è in background o non è visibile all'utente. Interrompere o ridurre l'attività di background come aggiornamenti, servizi e notifiche quando non sono necessari.

4.3 Ottimizzazione della GUI

- `Ottimizzazione dei Layout`: L'ottimizzazione dei layout è fondamentale poiché un layout complesso o mal progettato può richiedere notevoli risorse CPU e GPU per essere elaborato. L'utilizzo di layout semplici e leggeri riduce la quantità di calcoli necessari per il rendering, migliorando l'efficienza energetica e la risposta dell'app all'utente.

- `Adattatori Efficienti`: Gli adattatori efficienti, come RecyclerView o GridView, sono cruciali perché riducono il consumo di memoria e CPU durante la visualizzazione di elenchi o griglie di elementi. Questi adattatori riciclano le viste esistenti anziché crearne di nuove, contribuendo a mantenere bassi i costi di rendering e a migliorare la fluidità dell'app.

- `TextureView per Rendering`: L'utilizzo di TextureView invece di SurfaceView è consigliato perché TextureView offre un'implementazione basata su texture che può essere incorporata facilmente in un layout XML. Ciò consente un rendering più efficiente, poiché sfrutta al meglio le capacità hardware del dispositivo, riducendo il consumo energetico.

- `Rendering Hardware Accelerato`: Abilitare il rendering hardware accelerato è consigliato poiché sfrutta la potenza della GPU, riducendo il carico sulla CPU e migliorando l'efficienza del rendering grafico. Ciò si traduce in un minor consumo energetico durante l'esecuzione delle operazioni grafiche.

- `Ottimizzazione delle Animazioni`: L'uso di animazioni leggere e fluenti è fondamentale poiché le animazioni complesse o pesanti possono richiedere molte risorse di calcolo e grafiche. Le animazioni ben ottimizzate migliorano l'esperienza dell'utente e riducono il consumo energetico complessivo.

4.4 Gestione della Batteria

- `Wake Locks`: Evitare di mantenere il dispositivo in uno stato di "wake" più del necessario. Assicurarsi di rilasciare i Wake Locks non appena non sono più necessari tramite il metodo

release. Utilizzare i Wake Lock Parziali, che consentono al dispositivo di risparmiare energia mantenendo attiva solo la CPU, consentendo al display di spegnersi quando non è in uso

- `AlarmManager`: Se si ha necessità di eseguire attività periodiche o pianificate in background, è bene considerare l'utilizzo di AlarmManager anziché mantenere il dispositivo in uno stato di wake costante. Con AlarmManager, è possibile programmare il dispositivo per eseguire il lavoro in determinati intervalli e quindi rilasciare i "Wake Locks" dopo che l'attività è stata completata. In questo modo, il dispositivo può tornare in modalità di risparmio energetico tra le esecuzioni pianificate.

```
// Creazione di un Intent per l'attivita' da eseguire
Intent intent = new Intent(context, MyScheduledService.class);
PendingIntent pendingIntent = PendingIntent.getService(context, 0,
    intent, 0);

// Impostazione di un allarme periodico con AlarmManager
AlarmManager alarmManager = (AlarmManager) getSystemService(Context.
    ALARM\textunderscore SERVICE);
long intervalMillis = 60 * 1000; // Esempio: 1 minuto
long startTime = System.currentTimeMillis() + intervalMillis;
alarmManager.setInexactRepeating(AlarmManager.RTC\textunderscore
    WAKEUP, startTime, intervalMillis, pendingIntent);
```

4.5 Ottimizzazione della Rete

- `Comunicazione Efficiente`: Utilizzare protocolli di comunicazione efficienti come HTTP/2 o HTTP/3 che consentono il multiplexing delle richieste, riducendo così la latenza e il consumo di energia.

- `Limitazione delle Richieste`: Evitare richieste di rete eccessive. Ad esempio, utilizzare una politica di "backoff" per ritentare le richieste solo dopo un certo intervallo in caso di errori. Limitare il polling frequente dei dati se non è necessario.

- `Compresssione`: Comprimere i dati prima di inviarli dal Server all'app e decomprimerli sul dispositivo. L'utilizzo della compressione può ridurre notevolmente il consumo di dati e la velocità di trasmissione.

4.6 Gestione dei Sensori

- `Rilevamento della Posizione`: Utilizzare il rilevamento della posizione in modo intelligente. E' possibile ridurre la frequenza di aggiornamento della posizione o utilizzare servizi di geolocalizzazione in background solo quando l'applicazione è attiva. Utilizzare le API di geolocalizzazione in modo efficiente per ottenere una posizione approssimata quando possibile.

- `Uso del Sensore di Prossimità`: Utilizzare il sensore di prossimità per determinare se il dispositivo è vicino a un oggetto o all'orecchio dell'utente. Ad esempio, spegnere il display quando il dispositivo è vicino all'orecchio durante una chiamata telefonica per risparmiare energia.

- `Sensore di Luminosità`: Utilizzare il sensore di luminosità per regolare automaticamente la luminosità dello schermo in base alle condizioni ambientali. In questo modo, è possibile risparmiare energia riducendo la luminosità quando non è necessaria.

5 Modalità di risparmio energetico

Oltre alle pratiche sopra descritte, Android ha introdotto diverse funzionalità di risparmio energetico, progettate per ottimizzare l'uso delle risorse del sistema e minimizzare il consumo di energia delle applicazioni. Quest'ultime consentono ai dispositivi di gestire in modo intelligente l'attività delle applicazioni, limitando l'uso delle risorse allo stretto necessario.

5.1 Doze Mode

Il Doze Mode è una modalità di risparmio energetico introdotta in Android 6.0 (Marshmallow). Quando il dispositivo Android è inattivo e non viene utilizzato per un certo periodo di tempo, entra automaticamente in questa modalità. Durante il Doze Mode, il sistema operativo Android limita l'attività in background delle app per ridurre il consumo energetico. Le app vengono messe in uno stato di "sonno profondo" e vengono rallentate le attività in background come la sincronizzazione dei dati e gli aggiornamenti delle notifiche. Questa modalità è particolarmente efficace per estendere la durata della batteria quando il dispositivo è in stand-by, come quando non viene utilizzato durante la notte.

5.2 App Standby Buckets

Le App Standby Buckets sono state introdotte in Android 9 (Pie). Questa funzionalità classifica le app in diverse "secchielle" in base al loro utilizzo. Le app che vengono utilizzate attivamente vengono collocate in secchielle superiori, mentre quelle meno utilizzate o non utilizzate da molto tempo vengono spostate in secchielle inferiori. Le app nelle secchielle inferiori subiscono limitazioni nell'accesso alle risorse di sistema e alle notifiche in background per ridurre il consumo energetico.

5.3 App Background Restrictions

Questa funzionalità permette agli utenti di Android di impostare restrizioni specifiche per le app in background. Gli utenti possono limitare l'accesso delle app a risorse come la posizione GPS, i dati in background e l'uso della connessione Internet mentre sono in background. Questo aiuta a prevenire che le app consumino energia in modo eccessivo quando non sono in primo piano.

5.4 App Battery Optimization

L'ottimizzazione della batteria è una caratteristica di Android che permette di ottimizzare il consumo energetico delle app. Android identifica automaticamente le app che consumano energia in modo eccessivo e le mette in uno stato di ottimizzazione della batteria. Durante questa fase, il sistema limita l'attività in background delle app e riduce la loro frequenza di aggiornamento per conservare la carica della batteria.

6 Descrizione dell'applicativo

Dopo questa breve introduzione al campo dell'ottimizzazione energetica, ci sposteremo ora sulla descrizione dell'applicazione Open-Source che sarà oggetto dei nostri test. Quest'ultima, denominata "RoboDrink", è stata sviluppata nell'ambito del progetto di Laboratorio di Sistemi Operativi del CDL in Informatica della Federico II.

L'applicazione "RoboDrink" è stata sviluppata per simulare l'interazione tra un robot e gli utenti attraverso una "state machine" che gestisce le diverse fasi dell'interazione. Gli utenti possono accedere, registrarsi o utilizzare l'app come ospiti. Una volta autenticati, possono interagire con il robot, scegliere drink, partecipare a quiz e altro ancora. L'obiettivo finale è quello di fornire ai clienti una bevanda personalizzata basata sulle loro preferenze.

L'applicazione è strutturata in diversi stati, ognuno dei quali rappresenta una fase dell'interazione tra il robot e l'utente, tra cui: "Out of Sight", "New", "Welcoming", "Ordering", "Serving", "Interacting", "Farewelling" e "Gone". L'applicazione utilizza una architettura Client-Server basata sul protocollo TCP/IP. Il Server è stato implementato in C e utilizza PostgreSQL come database per gestire le informazioni sugli utenti e i drink preferiti. La comunicazione tra Client e Server avviene attraverso socket e il Server può gestire più Client contemporaneamente tramite l'uso di thread. Ogni stato dell'applicazione è gestito da funzioni specifiche nel Server, che comunicano con il Client utilizzando messaggi significativi. I messaggi vengono inviati dal Client al Server per richiedere operazioni specifiche, come l'accesso, la registrazione, la scelta di un drink, ecc. Il Server risponde con messaggi che indicano l'esito delle richieste dell'utente, ad esempio confermando un accesso riuscito o fornendo la descrizione di un drink selezionato.

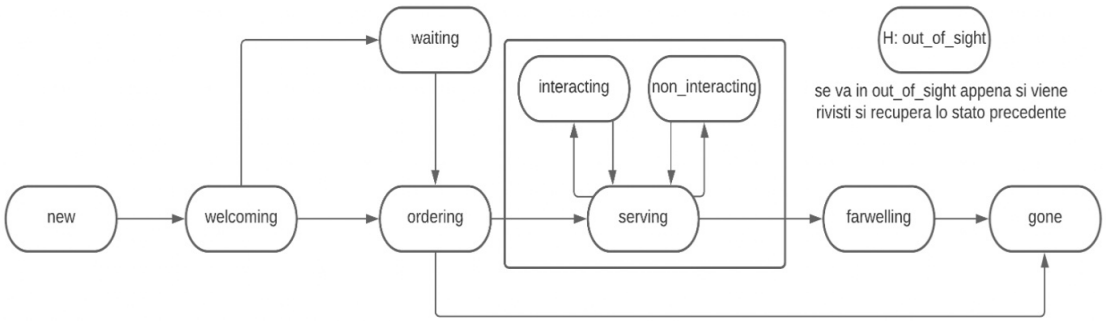

Figure 2: Descrizione degli stati dell'applicativo tramite NDFSM

7 Configurazione pre-analisi del consumo energetico

Per effettuare una analisi completa del consumo energetico dell' applicazione descritta nella sezione 6, usufruiremo dei tool illustrati nella sezione 3.

Di seguito è descritto il setup necessario per svolgere l'analisi.

7.1 Setup e dipendenze

- `Android Studio`
- `Android SDK platform tools`
- `Python 3.10 o versioni inferiori`
- `Pandas module`
- `Go runtime`

E' poi necessario inserire il comando *adb* (Android Debug Bridge) presente nell'SDK, all' environment PATH (Python e Go dovrebbero essere già stati inseriti tramite il processo di installazione; in caso contrario è necessario aggiungerli manualmente):

- `Procedura Windows`: Modificare la variabile d'ambiente PATH in Impostazioni di sistema avanzate, poi Proprietà del sistema, quindi Variabili d'ambiente, infine scegliere Percorso e modificarla aggiungendo il nuovo percorso (la posizione in cui è stato installato platform-tools).
- `Procedura Unix`: esportare ***PATH=/il/percorso/al/target/:$PATH*** nel file .profile o .bashrc. o .zshrc

Utilizzare il comando seguente per verificare la corretta esportazione:

```
1    adb --version
```

Successivamente, attivare il **Debug USB** sul dispositivo Android nelle opzioni sviluppatore. Sul telefono dovrebbe apparire una richiesta di autorizzazione al debug. Se il computer è un computer personale e sicuro, è necessario selezionare **Consenti sempre da questo computer**. Ripetere gli stessi passaggi per attivare il **Debug Wireless**. Nella sezione 10 verranno illustrati i passaggi per replicare gli esperimenti fatti.

7.2 Dispositivo di Test e pre-condizioni

Il dispositivo utilizzato per i Test è un Samsung Galaxy S10+, con sistema operativo Android 12, versione One UI 4.1.

Di seguito sono indicate le pre-condizioni:

1. `Modalità risparmio batteria disabilitata.`
2. `Timeout dello schermo disabilitato.`
3. `Livello di luminosità e suono impostato al 50%.`
4. `Applicazioni in esecuzione e servizi in background disattivati.`
5. `Durata media di tutti i test di circa 1 minuto e 30 secondi.`

7.3 Identificazione degli scenari

L'applicativo prevede diversi scenari descritti nella sezione 6 da un automa non deterministico. In un automa non deterministico una relazione di transizione può avere uno o più stati di destinazione. Non essendoci dunque un percorso univoco è necessario identificare i vari tipi di percorsi al fine di determinare con precisione il consumo energetico dell'applicazione e permettere una corretta gestione dei casi medi.

Provvediamo dunque ad evidenziare i punti decisionali che formeranno il rispettivo albero decisionale [Fig. 3] per poter successivamente elaborare vari tipi di scenari.

- `Punto decisionale 1`: l'utente è accolto nella schermata di welcoming, dove un bottone stabilirà il suo posizionamento in waiting (in caso vi siano almeno due utenti in ordering o serving) oppure in ordering.
- `Punto decisionale 2`: l'utente sta ordinando il suo drink scegliendo tra un suggerimento e una lista completa di bevande disponibili. Saranno le esigenze dell'utente a stabilire se continuare ed essere servito nello stato di serving oppure se uscire dall'applicazione, posizionandosi nello stato di gone.
- `Punto decisionale 3`: il drink dell'utente è in preparazione e quest'ultimo può scegliere se interagire con il robot tramite risposte a dei quiz, oppure se attendere in maniera passiva. I due rispettivi pulsanti posizioneranno l'utente nello stato di Interacting o di Non_Interacting.

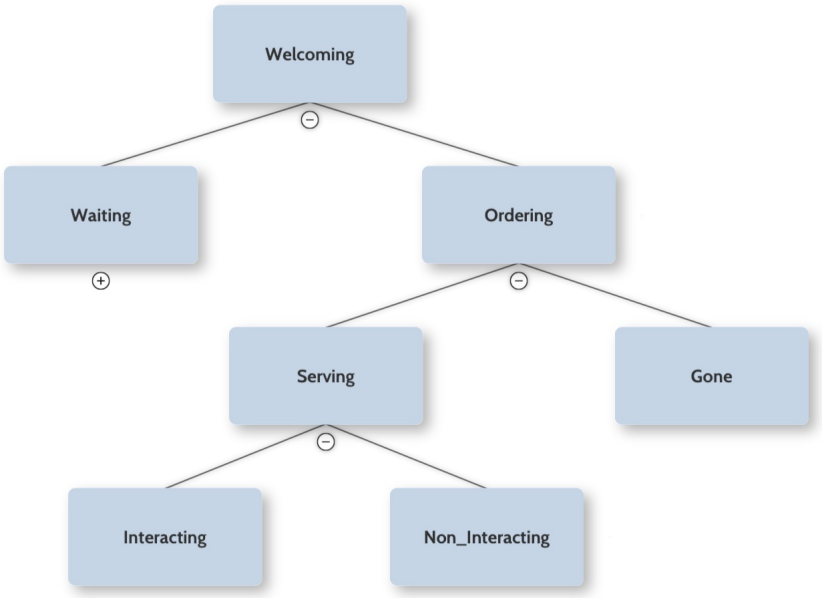

Figure 3: Albero dei punti decisionali descritti: Welcoming (punto 1), Ordering (punto 2) e Serving (punto 3).

Questi tre punti ci permettono di individuare i seguenti sei scenari:

- `Scenario 1`: new → welcoming → waiting → ordering → serving → interacting
- `Scenario 2`: new → welcoming → ordering → serving → interacting
- `Scenario 3`: new → welcoming → ordering → gone
- `Scenario 4`: new → welcoming → waiting → ordering → serving → non_interacting
- `Scenario 5`: new → welcoming → ordering → serving → non_interacting
- `Scenario 6`: new → welcoming → waiting → ordering → gone

8 Ottimizzazioni

Nel mondo complesso dell'analisi e dell'ottimizzazione, è naturale rivolgere l'attenzione verso le ottimizzazioni totali, ovvero quelle che apportano tipicamente il massimo miglioramento a un sistema nel suo complesso. Tuttavia, soffermarsi esclusivamente su questo tipo di ottimizzazione può tralasciare aspetti cruciali e opportunità nascoste.

Le ottimizzazioni parziali si concentrano su aree specifiche di un sistema. Anche se, presi singolarmente, questi miglioramenti potrebbero sembrare minimi o trascurabili, la loro somma può portare a significativi benefici complessivi. Effettuiamo dunque una analisi delle ottimizzazioni parziali per poi approfondire quelle totali e mostrare i dovuti confronti.

Le ottimizzazioni parziali saranno effettuate su due macro-categorie, così suddivise in base alle analisi delle problematiche rilevate:

1. `Gestione ottimale dei thread`.
2. `Gestione ottimale di oggetti/strutture e riutilizzo costruttivo del codice`.

Esamineremo in dettaglio gli sviluppi di ciascuna di queste componenti per determinare con maggiore precisione quali hanno effettivamente apportato benefici all'applicazione. Valuteremo inoltre, nella sezione 9, l'incidenza del contributo di ogni categoria nel raggiungimento del risultato finale, assegnando un peso espresso in percentuale.

Di seguito sono illustrati alcuni esempi di ottimizzazioni effettuate. Le modifiche più importanti hanno riguardato gli utilizzi impropri di `runOnUIThread` e la costante creazione di nuovi `Handler` e `OnTouchListeners`. Sono poi stati migliorati i tentativi di riconnessione, introdotte strutture dati più efficienti e ottimizzato il codice per i Timer dell'inattività. Inoltre sono state eliminate parti di codice duplicate e ridondanze che potevano naturalmente avere un impatto negativo a livello energetico.

8.1 Gestione ottimale dei thread

- **Rimossi alcuni usi di `Thread.sleep()`:** La chiamata a `Thread.sleep()` fa sì che il thread corrente si metta in attesa per un periodo di tempo specificato, consumando risorse inutilmente. Rimuovendo queste chiamate, si riduce il tempo in cui il dispositivo è attivo senza fare nulla, risparmiando energia.
- **Aumentato il tempo tra i tentativi di riconnessione al Server**: Tentare una riconnessione ogni volta che quest'ultima fallisce potrebbe portare a un consumo elevato di energia, specialmente se la connessione fallisce ripetutamente. Aumentando il tempo tra i tentativi o limitando il numero di tentativi, si riduce il consumo energetico.
- **Rimossi `runOnUiThread` superflui**:

```
int atIndex = user.indexOf("@");
if (atIndex != -1) {
    String username = user.substring(0, atIndex);
    runOnUiThread(() -> textViewLoggedIn.setText(username));
} else {
    runOnUiThread(() -> textViewLoggedIn.setText(user));
}
```

Listing 1: Codice precedente

```
int atIndex = user.indexOf("@");
if (atIndex != -1) {
    String username = user.substring(0, atIndex);
    textViewLoggedIn.setText(username);
} else {
    textViewLoggedIn.setText(user);
}
```

Listing 2: Codice ottimizzato

- **Limitati gli aggiornamenti dell'interfaccia utente**: L'aggiornamento frequente della UI, come il cambiamento di testo o l'animazione, può consumare energia, ed è opportuno porre un occhio di riguardo in questi termini.

8.2 Gestione ottimale di oggetti/strutture e riutilizzo costruttivo del codice

- **Rimozione di codice duplicato**: Ridurre la duplicazione del codice rende il programma più pulito e facile da mantenere. Un buon esempio è rappresentato dalla gestione dei topics.

```
if("Guest".equals(user)){
    String[] allTopics = {"Storia", "Attualita", "Sport", "Scienza", "
        Informatica", "Letteratura", "Musica", "Geografia"};
    selectedTopics = new String[2];
    Random random = new Random();
    for (int i = 0; i < 2; i++) {
        int randomIndex = random.nextInt(allTopics.length);
        selectedTopics[i] = allTopics[randomIndex];
    }
} else if("Nessuno".equals(selectedTopics[0])){
    String[] allTopics = {"Storia", "Attualita", "Sport", "Scienza", "
        Informatica", "Letteratura", "Musica", "Geografia"};
    selectedTopics = new String[2];
    Random random = new Random();
    for (int i = 0; i < 2; i++) {
        int randomIndex = random.nextInt(allTopics.length);
        selectedTopics[i] = allTopics[randomIndex];
    }
}
```

Listing 3: Codice precedente

```
if ("Guest".equals(user) || "Nessuno".equals(selectedTopics[0])) {
    selectRandomTopics();
}
```

Listing 4: Codice ottimizzato

```
private void selectRandomTopics() {
    String[] allTopics = {"Storia", "Attualita", "Sport", "Scienza", "
        Informatica", "Letteratura", "Musica", "Geografia"};
    for (int i = 0; i < 2; i++) {
        int randomIndex = random.nextInt(allTopics.length);
        selectedTopics[i] = allTopics[randomIndex];
    }
}
```

Listing 5: Codice ottimizzato

- **Rimossi tutti i callback `Handler` quando non sono più necessari**: Se si utilizzano `Handler` e `Runnable` per eseguire azioni dopo un ritardo, è importante rimuovere tutti i callback quando non sono più necessari. Questo previene l'esecuzione inutile di codice e riduce il consumo di energia.

- **Introdotte strutture dati efficienti**: Sono state inserite strutture `HashSet`, molto efficienti per operazioni quali inserimento, eliminazione e ricerca, che mediamente vengono svolte in tempo costante O(1).

```
1
2  private String[] selectedTopics;
3
4  private boolean isTopicSelected(String topic) {
5      for (String selectedTopic : selectedTopics) {
6          if (selectedTopic.equals(topic)) {
7              return true;
8          }
9      }
10     return false;
11 }
```

Listing 6: Codice precedente

```
1  private Set<String> selectedTopicsSet;
2
3  selectedTopicsSet = new HashSet<>(Arrays.asList(selectedTopics));
4
5  private boolean isTopicSelected(String topic) {
6      return selectedTopicsSet.contains(topic);
7  }
```

Listing 7: Codice ottimizzato

- **Utilizzo di una sola istanza di Handler**: Creare molte istanze di oggetti può essere costoso in termini di memoria e prestazioni. Utilizzando una sola istanza, si riduce l'overhead e si migliora l'efficienza energetica.

```
1  private void applyButtonAnimation(View v) {
2      v.startAnimation(buttonAnimation);
3      new Handler().postDelayed(v::clearAnimation, 100);
4  }
```

Listing 8: Codice precedente

```
1  private final Handler handler = new Handler();
2
3  private void applyButtonAnimation(View v) {
4      v.startAnimation(buttonAnimation);
5      handler.postDelayed(v::clearAnimation, 100);
6  }
```

Listing 9: Codice ottimizzato

- **Implementazione di un `View.OnTouchListener` centralizzato**: Evitando la creazione ripetuta di nuovi oggetti `OnTouchListener`, abbiamo ridotto il numero di oggetti che vengono creati. Questo riduce la pressione sul garbage collector, migliorando l'efficienza energetica.

```
private void setTouchListenerForAnimation(View view) {
    view.setOnTouchListener(new View.OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            if (event.getAction() == MotionEvent.ACTION_DOWN) {
                resetInactivityTimer();
                applyButtonAnimation(v);
            }
            return false;
        }
    });
}
```

Listing 10: Codice precedente

```
private final View.OnTouchListener animationTouchListener = new View.
    OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        if (event.getAction() == MotionEvent.ACTION_DOWN) {
            resetInactivityTimer();
            applyButtonAnimation(v);
        }
        return false;
    }
};
```

Listing 11: Codice ottimizzato

9 Risultati

Per ogni scenario sono state svolte un numero N di prove pari a 5, distinguendo ognuno di questi per tipologia di ottimizzazione: *NoOpt*[1], *OptP1*[2], *OptP2*[3], *Opt*[4].

Sebbene gli scenari totali dell'applicativo siano sei, al fine di rendere ripetibili i test effettuati, siamo stati costretti a limitarci agli scenari in cui non è presente lo stato di Waiting, stato che prevede risposte non determinabili a priori (e dunque variabili nel tempo) da parte del Server. Si prenda dunque in considerazione che i risultati ottenuti sono relativi a test effettuati con un stato in meno e sulla metà degli scenari, rispettivamente: scenario 2, scenario 3, scenario 5.

Si mette in evidenza inoltre che in tutti i test, l'utente utilizzato è già registrato alla piattaforma e passa direttamente per la fase di Log-In, che corrisponde allo stato di New.

Scenari	Durata (s)	Energia (J)
Scenario2NoOpt	133.93	178.59
Scenario3NoOpt	49.94	69.89
Scenario5NoOpt	100.46	125.16
Scenario2OptP1	136.63	161.58
Scenario3OptP1	55.07	62.77
Scenario5OptP1	101.38	111.79
Scenario2OptP2	135.48	162.15
Scenario3OptP2	50.51	61.66
Scenario5OptP2	102.39	116.07
Scenario2Opt	138.95	180.15
Scenario3Opt	39.94	54.23
Scenario5Opt	103.67	114.88

Table 1: Risultati di durata ed energia degli scenari

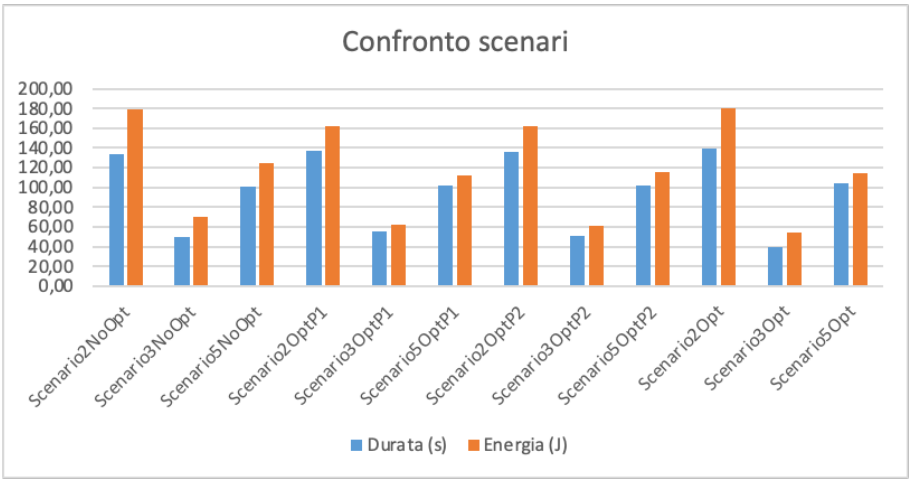

Figure 4: Grafico di confronto durata/energia degli scenari

[1]Scenari senza ottimizzazioni
[2]Scenari con la prima ottimizzazione parziale [8.1]
[3]Scenari con la seconda ottimizzazione parziale [8.2]
[4]Scenari con le ottimizzazioni totali

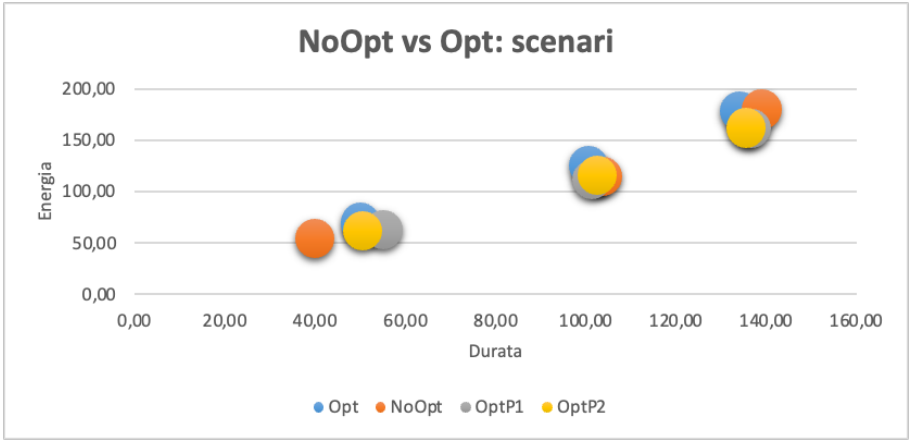

Figure 5: Diagramma di dispersione a bolle per i vari scenari

NoOpt	**Durata (s)**	**Energia (J)**	**Energia (J/Min)**
Media	94.78	124.54	1.31
Mediana	100.46	125.16	1.24
Dev. Stand.	34.52	44.37	/
OptP1	**Durata (s)**	**Energia (J)**	**Energia (J/Min)**
Media	97.69	112.05	1.14
Mediana	101.38	111.79	1.10
Dev. Stand.	33.40	40.34	/
OptP2	**Durata (s)**	**Energia (J)**	**Energia (J/Min)**
Media	96.13	113.29	1.17
Mediana	102.39	116.07	1.13
Dev. Stand.	34.97	41.07	/
Opt	**Durata (s)**	**Energia (J)**	**Energia (J/Min)**
Media	94.19	116.42	1.23
Mediana	103.67	114.88	1.10
Dev. Stand.	40.97	51.41	/

Table 2: Media, mediana e deviazione standard degli scenari

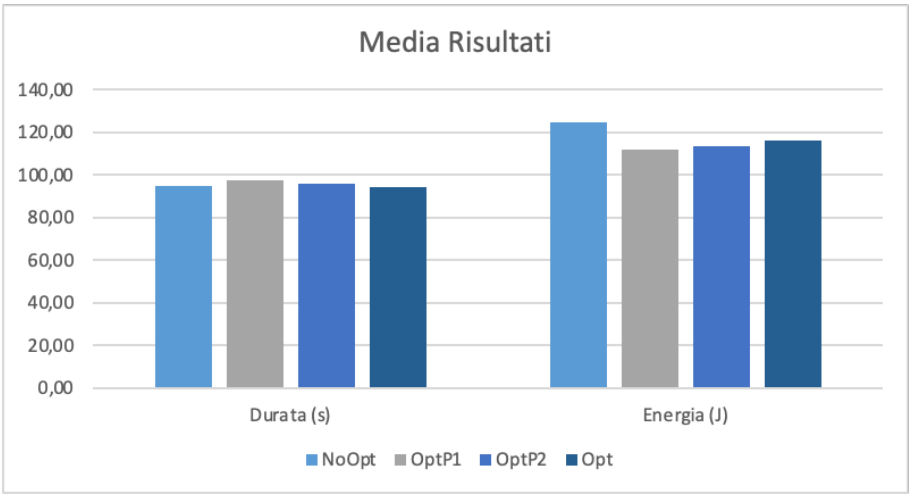

Figure 6: Media dei risultati totali di durata ed energia

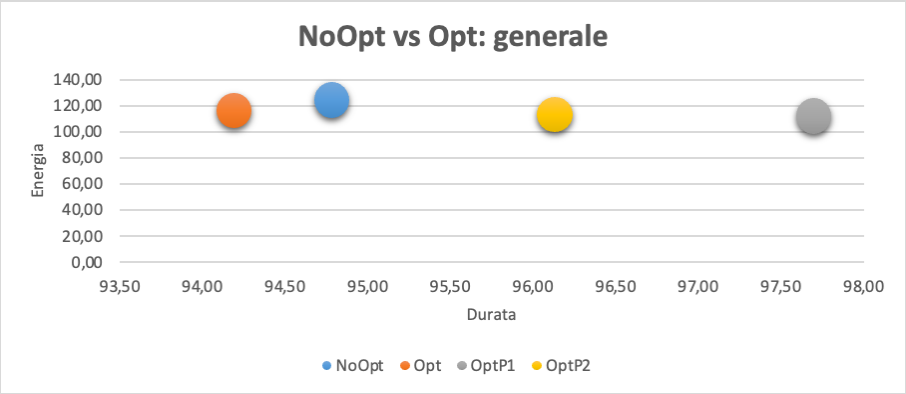

Figure 7: Diagramma di dispersione a bolle per i vari scenari

Per ulteriori grafici consultare il file risultatiElaborati.xlsx presente nella cartella del progetto principale.

Provvediamo ora a calcolare la variazione percentuale tra i vari scenari e le diverse metriche tramite la seguente formula in cui *Vecchio valore* rappresenta il valore iniziale e *Nuovo valore* rappresenta il valore dopo una certa modifica:

$$\textbf{Variazione percentuale} = \left(\frac{\text{Nuovo valore} - \text{Vecchio valore}}{\text{Vecchio valore}}\right) \times 100 \tag{1}$$

Iniziamo analizzando i cambiamenti nelle metriche di durata:

- Passando da NoOpt a OptP1, la durata è aumentata del 3.08%.
- Passando da NoOpt a OptP2, la durata è aumentata dell'1.42%.
- Passando da NoOpt a Opt, la durata è diminuita dello 0.62%.

E di energia:

- Passando da NoOpt a OptP1, l'energia consumata è diminuita del 10.04%.
- Passando da NoOpt a OptP2, l'energia consumata è diminuita del 9.03%.
- Passando da NoOpt a Opt, l'energia consumata è diminuita del 6.52%.

Ora verifichiamo quanto ha "pesato" ciascuna ottimizzazione parziale rispetto all'ottimizzazione totale.

Identificando con:

1. **var1**: la variazione da NoOpt a OptP1
2. **var2**: la variazione da NoOpt a OptP2

Scriveremo i contributi relativi di OptP1 e OptP2 come:

$$\textbf{Contributo rel. di OptP1} = \frac{\text{var1}}{\text{var1} + \text{var2}} \times 100 \tag{2}$$

$$\textbf{Contributo rel. di OptP2} = \frac{\text{var2}}{\text{var1} + \text{var2}} \times 100 \tag{3}$$

In termini di energia:

- OptP1 ha contribuito per il 52.63%.
- OptP2 ha contribuito per il 47.37%.

Entrambe le ottimizzazioni parziali, quando applicate separatamente, hanno aumentato la durata rispetto allo scenario senza ottimizzazioni (NoOpt). Tuttavia, quando combinate, hanno prodotto una leggera diminuzione nella durata. Questo suggerisce che le due ottimizzazioni, quando combinate, possono avere effetti sinergici che non sono evidenti quando sono applicate separatamente. Per quanto riguarda il consumo energetico, entrambe le ottimizzazioni parziali hanno prodotto riduzioni significative, con OptP1 che ha un leggero vantaggio su OptP2. Tuttavia, la combinazione delle due non ha prodotto una riduzione proporzionalmente equivalente, indicando di nuovo potenziali interazioni tra le due ottimizzazioni.

Evidenziamo inoltre che i valori di deviazione standard riportati nella Tabella 2 sono elevati in quanto derivano dalla media di diversi scenari in termini di durata ed energia. Dato che gli scenari considerati variano notevolmente in questi due aspetti, le deviazioni standard risultano ampie (come evidenziato nella 3). Per un maggior approfondimento si rimanda alle tabelle specifiche per ogni scenario contenute nel file "risultatiElaborati.xlsx".

Scenario2: NoOpt	**Durata (s)**	**Energia (J)**	**Energia (J/Min)**
Media	133.93	178.59	1.33
Mediana	134.85	188.34	1.40
Dev. Stand.	1.51	19.34	/
Scenario2: OptP1	**Durata (s)**	**Energia (J)**	**Energia (J/Min)**
Media	136.64	161.59	1.18
Mediana	136.57	136.57	1.15
Dev. Stand.	0.38	20.14	/
Scenario2: OptP2	**Durata (s)**	**Energia (J)**	**Energia (J/Min)**
Media	135.49	162.15	1.20
Mediana	136.31	150.05	1.10
Dev. Stand.	4.01	18.32	/
Scenario2: Opt	**Durata (s)**	**Energia (J)**	**Energia (J/Min)**
Media	138.95	180.15	1.30
Mediana	138.82	192.95	1.39
Dev. Stand.	0.72	24.14	/

Table 3: Media, mediana e deviazione standard dello scenario 2

In generale comunque, il miglioramento in termini energetici da una fase di scenari non ottimizzati a quella di scenari completamente ottimizzati è stato del 6.52%!

10 Replicazione degli esperimenti

Per replicare con successo gli esperimenti, è necessario seguire attentamente i seguenti passaggi:

1. **Installazione dell'APK sul Dispositivo**
   Prima di tutto, è necessario installare l'APK sul dispositivo per entrambe le versioni dell'applicazione (non ottimizzata e ottimizzata). Questo può essere fatto trasferendo il file APK sul dispositivo e poi aprendolo per avviare l'installazione. Assicurarsi di avere abilitato l'installazione da fonti sconosciute nelle impostazioni di sicurezza del dispositivo.

2. **Abilitazione dell'Opzione Sviluppatore**
   Per abilitare l'opzione sviluppatore sul tuo dispositivo Android:

   (a) Andare su `Impostazioni` > `Informazioni sul telefono` (o `Informazioni sul dispositivo`).

   (b) Trovare la voce `Numero build` e cliccare ripetutamente (circa 7 volte) fino a quando non compare un messaggio che indica che si è in modalità sviluppatore.

3. **Abilitazione del Debug USB**
   Dopo aver abilitato l'opzione sviluppatore:

   (a) Tornare al menu principale delle `Impostazioni` e selezionare `Opzioni sviluppatore`.

   (b) Cercare e attivare l'opzione `Debug USB`.

4. **Abilitazione del Debug Wireless e Annotazione dell'Indirizzo IP**
   Ancora nelle `Opzioni sviluppatore`:

   (a) Cercare e attivare l'opzione `Debug wireless`.

   (b) Una volta attivato, il dispositivo mostrerà un valore nel formato `ipaddress:port`. Annotare questo valore, poiché servirà per i passaggi successivi.

5. **Avvio dello Script Python**

   (a) Navigare nella directory dove si trova lo script `testApp.py`.

   (b) Avviare lo script eseguendo il comando `python testApp.py` nel terminale o prompt dei comandi.

   (c) Seguire attentamente i passaggi richiesti dallo script.

6. **Visualizzazione dei File CSV**
   Dopo aver eseguito lo script:

   (a) Navigare nella cartella `output` nella stessa directory dello script.

   (b) Qui si troveranno i file `.csv` creati.

Nota: Tutti i test sono stati eseguiti in modo automatizzato attraverso lo script Python, sfruttando i file di test creati dallo strumento Espresso Recorder [5] presente in Android Studio. Questa metodologia garantisce una uniformità assoluta nell'attuazione di ogni singolo test.

[5] Strumento di Capture and Replay il cui compito è quello di semplificare il processo di testing generando automaticamente Test Case.

11 Conclusioni

Dall'analisi degli scenari e delle ottimizzazioni presentate, possiamo trarre diverse conclusioni riguardo al comportamento energetico dell'applicazione "RoboDrink" e alle potenzialità delle ottimizzazioni proposte.

1. **Complessità degli scenari**: La descrizione dell'applicazione ha evidenziato una complessa serie di interazioni tra l'utente e il robot, strutturate attraverso diverse fasi gestite da una "state machine". Questa complessità rappresenta una sfida significativa quando si cerca di ottimizzare il comportamento energetico dell'applicazione.

2. **Ottimizzazioni parziali vs. totali**: L'analisi ha mostrato che l'ottimizzazione energetica non è sempre una questione di "tutto o niente". In alcuni casi, le ottimizzazioni parziali hanno portato a miglioramenti significativi da sole. Tuttavia, nel nostro caso la combinazione delle due ottimizzazioni parziali ha prodotto risultati leggermente peggiori rispetto alle singole ottimizzazioni, mantenendo comunque un ottimo miglioramento rispetto al codice di partenza.

3. **Trade-off tra durata ed energia**: La durata delle operazioni è una metrica importante, ma non sempre una durata minore significa un consumo energetico minore. In alcuni scenari, abbiamo notato che l'ottimizzazione della durata ha portato a un aumento del consumo energetico come nel caso del passaggio dallo scenario2OptP1 a allo scenario2OptP2, e viceversa come dallo scenario5OptP2 allo scenario5Opt. Questo sottolinea l'importanza di considerare entrambe le metriche quando si ottimizza un'applicazione.

4. **Importanza della ripetibilità dei test**: Gli scenari testati sono stati selezionati per garantire la ripetibilità dei test. Questo è fondamentale per ottenere risultati affidabili e comparabili, ma è importante notare che potrebbero esistere altri scenari reali che non sono stati considerati in questa analisi.

5. **Ottimizzazioni a livello di codice**: L'analisi del codice ha rivelato numerose aree di miglioramento, dalla rimozione di codice duplicato e inutilizzato, all'introduzione di strutture dati più efficienti e alla gestione ottimale dei thread. Queste ottimizzazioni, sebbene possano sembrare piccole singolarmente, possono avere un impatto significativo quando sommate. Nel nostro caso, applicando entrambe le ottimizzazioni contemporaneamente abbiamo avuto un miglioramento in termini di durata e un leggero peggioramento in termini di energia. Ma comunque in entrambe le ottimizzazioni apportate si è avuto un netto miglioramento in termini di energia rispetto al codice non ottimizzato.

In conclusione, il lavoro svolto su "RoboDrink" ha portato ad una diminuzione del consumo energetico di circa 6 punti percentuali e ci ha permesso di mostrare come con attenzione ai dettagli, metodo e strumenti adeguati è possibile realizzare applicazioni che non solo soddisfano le esigenze degli utenti, ma lo fanno in modo efficiente e cercando di rispettare l'ambiente.

References

[1] Faisal Alam, Preeti Ranjan Panda, Nikhil Tripathi, Namita Sharma, and Sanjiv Narayan. Energy optimization in android applications through wakelock placement. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4, 2014.

[2] Luis Cruz and Rui Abreu. Performance-based guidelines for energy efficient mobile applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 46–57, 2017.

[3] Stefanos Georgiou, Stamatia Rizou, and Diomidis Spinellis. Software development lifecycle for energy efficiency: Techniques and tools. Anno non specificato.

[4] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. Characterizing and detecting resource leaks in android applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 389–398, 2013.

[5] Ding Li and William G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Anno non specificato*, 2014.

[6] Jeongyeup Paek, Joongheon Kim, and Ramesh Govindan. Energy-efficient rate-adaptive gps-based positioning for smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, page 299–314, New York, NY, USA, 2010. Association for Computing Machinery.

[7] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The influence of the java collection framework on overall energy consumption. In *Anno non specificato.*

[8] Sanae Rosen, Ashkan Nikravesh, Yihua Guo, Z. Morley Mao, Feng Qian, and Subhabrata Sen. Revisiting network energy efficiency of mobile apps: Performance in the wild. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, page 339–345, New York, NY, USA, 2015. Association for Computing Machinery.

[9] Ruben Saborido, Rodrigo Morales, Foutse Khomh, Yann-Gael Gu, and Giuliano Antoniol. Getting the most from map data structures in android. In *Anno non specificato.*

[10] A. R. Tonini, L. M. Fischer, J. C. B. d Mattos, and L. B. d Brisolara. Analysis and evaluation of the android best practices impact on the efficiency of mobile applications. In *Proceedings of the 2013 III Brazilian Symposium on Computing Systems*, 2013.

[11] K.S. Vallerio, Lin Zhong, and N.K. Jha. Energy-efficient graphical user interface design. *IEEE Transactions on Mobile Computing*, 5(7):846–859, 2006.

[12] Claas Wilke, Sebastian Richly, Sebastian Götz, Christian Piechnick, and Uwe Aßmann. Energy consumption and efficiency in mobile applications: A user feedback study. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 134–141, 2013.