Relazione distributed systems

Analisi del progetto

L'obiettivo iniziale del progetto consisteva nell'effettuare un porting dell'applicazione Openstack Lightning-rod verso i browser web grazie all'utilizzo di WebAssembly e del framework Pyodide.

WebAssembly (WASM) rappresenta una tecnologia relativamente recente che consente l'esecuzione di codice nativo all'interno del browser, rendendo dunque possibile aggirare, almeno in teoria, limiti di portabilità e interoperabilità del software verso diverse tipologie di dispositivi e sistemi operativi ospitanti. A livello implementativo, il codice viene precompilato in formato "bytecode" ed eseguito durante il runtime da un interprete WebAssembly all'interno del browser web, comportando una maggiore flessibilità in quanto il bytecode non dipende dall'architettura del sistema ospitante ma allo stesso tempo vengono minimizzati gli overhead a runtime in quanto il codice non è completamente interpretato.

Tale meccanismo viene sfruttato da Pyodide, un framework in grado di permettere l'esecuzione di un ambiente Python all'interno del browser senza la necessità di un server backend o che un interprete sia installato localmente sulla piattaforma. Questo si traduce nella necessità di ospitare un interprete Python nell'ambiente WebAssembly, pertanto l'interprete CPython, opportunamente precompilato in bytecode, viene eseguito all'interno dell'ambiente consentendo l'esecuzione degli script nel contesto web.

Sulla base di quanto detto precedentemente, effettuare il porting dell'agente IoT/Cloud Lightning-rod da un ambiente Linux a un ambiente OS agnostic e compatibile con architetture mobili come Android significherebbe riuscire a sfruttare a pieno le potenzialità dell'agente scavalcando le differenze architetturali e d'ambiente. Va tuttavia evidenziato come Pyodide sia un framework relativamente recente e che WebAssembly, per via delle sue potenzialità in ambiente web, è basato sulla filosofia di security by design, pertanto il codice WASM viene eseguito in un ambiente sandboxed all'interno della macchina host, risultando nell'impossibilità di eseguire alcuni comandi e operazioni.

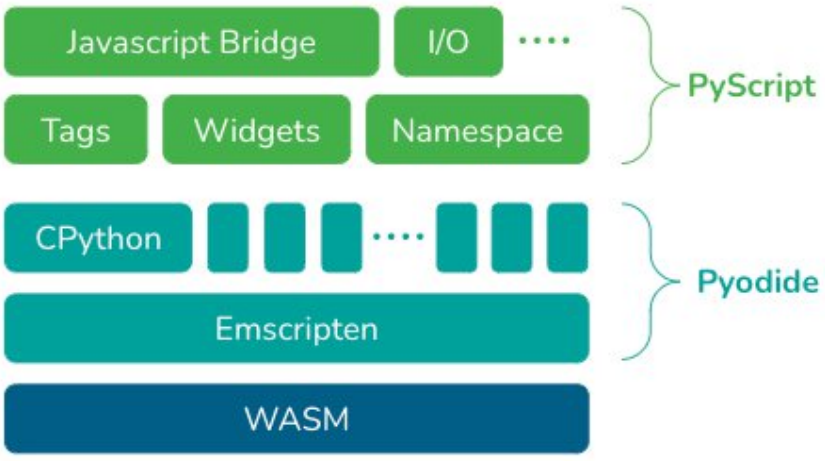


Figura 1 Stack WASM, Pyodide e PyScript

Alcune di queste limitazioni possono essere superate attraverso il consenso esplicito dell'utente, come ad esempio l'accesso ad alcune interfacce del sistema operativo, tuttavia a causa dei motivi precedentemente elencati, alcune funzionalità o moduli Python quali Multiprocessing, Threading e WebSockets non possono essere eseguiti. Tali moduli sono essenziali per l'esecuzione (almeno parziale) dell'agente, pertanto finché tali limitazioni non verranno superate, un porting dell'agente su WebAssembly sembra impossibile.

Una via alternativa al fine di effettuare un porting dell'agente per altri sistemi operativi e dispositivi mobili potrebbe essere quella di utilizzare dei framework Python OS agnostic che permettono di ricompilare il codice ad alto livello al fine di essere eseguito su ogni piattaforma supportata. Alcuni nomi noti all'interno di questa categoria sono Kivy[1] e Beeware[2], i quali permettono di descrivere ad alto livello l'interfaccia grafica e il comportamento dell'app e gestire separatamente, spesso in maniera del tutto automatizzata, il deploy verso diverse piattaforme. In particolare Beeware permette di utilizzare la toolchain del framework per esportare l'applicazione per ambienti desktop, dispositivi mobili e perfino ambienti web, seppur ancora questa parte dello sviluppo sia ancora in alpha e con le medesime limitazioni sopra citate. Per tali motivazioni BeeWare sarà il framework di riferimento nei paragrafi successivi, in quanto utilizzato anche in previsione di possibili sviluppi futuri.

Nei paragrafi seguenti verrà effettuato un riepilogo dello sviluppo, focalizzandosi sulle problematiche affrontate e nelle tecniche adottate al fine di risolverle. Il codice di riferimento è disponibile sulla pagina GitHub[3] del progetto, insieme al fork di Lightning-rod, per consentire un confronto diretto tra i codici sorgente dei due agenti.

Sviluppo dell'applicazione

Nelle fasi iniziali dello sviluppo, l'obiettivo primario consisteva nel rielaborare il codice Python originale di Lighting-rod al fine di ottenere un modulo indipendente dal sistema sottostante, permettendo, inoltre, l'esecuzione dell'agente come parte integrante di un sistema più complesso e non più come un applicazione standalone. Questo ha comportato una riscrittura di alcune parti del sorgente facenti riferimento percorsi assoluti sulla macchina ospitante e funzioni presenti nativamente solo all'interno di sistemi Linux come comandi shell e l'autenticazione PAM sostituita in favore di una più classica autenticazione username-password.

Allo scopo di minimizzare le modifiche al sorgente, si è adottato un approccio che prevedeva di utilizzare il modulo Python "multiprocessing" allo scopo di eseguire l'applicazione e l'agente su processi separati, mantenendo inalterate le meccaniche di sincronizzazione tra le diverse componenti di Lightning-rod. Questo approccio si è rivelato essere efficace, risultando in un esecuzione regolare in ambiente desktop. Tuttavia, non si può affermare lo stesso per i dispositivi mobili a causa di politiche di sicurezza più severe all'interno di Android e iOS.

Analizzando attentamente la documentazione del progetto "Python-for-Android" (lo strumento utilizzato dal framework Kivy e che utilizza un approccio simile alla pipeline di BeeWare), emergono diverse limitazioni. In particolare, è impossibile eseguire subprocess al di fuori del

[1] https://kivy.org/

[2] https://beeware.org/

[3] https://github.com/Hikari1026/LightningRodApp

contesto dell'applicazione, come ad esempio il comando "ifconfig". Inoltre, all'interno dell'applicazione non vi è un interprete Python, in quanto il codice viene convertito in Java e compilato, il che rende impossibile istanziare nuovi processi evocando l'interprete "sys.executable". Inoltre, va notato che il modulo multiprocessing non è supportato, ma è possibile "sostituirlo" utilizzando i "Service" nativi di Android. Un Service è inteso come un operazione asincrona senza UI che viene chiamata da altre attività e verrà eseguita nel thread ospitante. Tale approccio non risulta essere conforme con il caso d'uso e, comunque, prevederebbe l'introduzione di codice specifico per la piattaforma, andando contro il proposito originario del progetto di rendere l'applicazione OS agnostic.

Accantonata l'idea di utilizzare l'approccio "multiprocessing" ai fini di una maggiore compatibilità, l'unica opzione rimasta era quella di adattare il codice per l'esecuzione multithread. Questo ha inevitabilmente comportato una massiccia riscrittura di diverse componenti del codice sorgente e delle primitive di sincronizzazione tra i thread dell'agente e l'applicazione contenitore. Una volta completata la migrazione, l'applicazione risulta essere perfettamente funzionante e non si evidenziano differenze significative nelle prestazioni durante l'utilizzo, anche quando utilizzata su piattaforma Android.

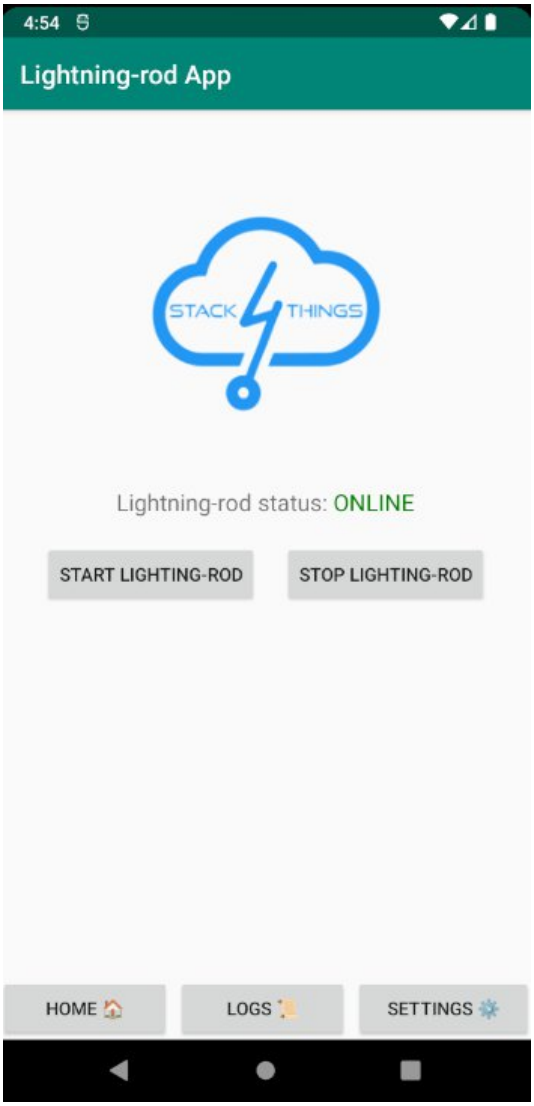


Figura 2 **Schermata iniziale dell'applicazione**

La configurazione di Lightning-rod è stata ibridizzata, permettendo la modifica dei JSON di configurazione e delle credenziali d'accesso all'interno dell'app, tuttavia la configurazione standard mediante server Flask, il quale fornisce un interfaccia web per configurare Lightning-rod, è stata mantenuta, permettendo a qualunque dispositivo sulla rete locale di connettersi tramite browser web alla pagina di configurazione.

Figura 3 Interfaccia web di configurazione

Componenti aggiuntivi

Lightning-rod è pensato per essere eseguito all'interno di un ambiente Linux in cui sono già preinstallate le dipendenze WSTUN e Nginx. WSTUN è un applicazione JavaScript contenente una serie di strumenti per stabilire tunnel TCP all'interno di una connessione WebSocket e WebSocketSecure, permettendo di aggirare eventuali firewall e reti private. Nginx invece è un

software open-source utilizzato per creare e gestire un web server, permettendo dunque di esporre servizi al di fuori della macchina ospitante attraverso la connessione TCP.

Tali dipendenze sono elencate tra le dipendenze di Lightning-rod, così come le relative istruzioni necessarie per l'installazione. All'interno dell'applicazione, i due software vengono gestiti mediante regolari comandi Bash per compiere le operazioni richieste. Allo scopo di risolvere le dipendenze esterne dell'agente, una possibile soluzione sarebbe quella di includere all'interno dell'applicazione i file binari di Node.js e Nginx opportunamente compilati per la piattaforma di riferimento, tuttavia, sebbene questo approccio si sia rivelato funzionante in ambiente desktop, ancora una volta le politiche di sicurezza di Android ne impediscono la portabilità.

A partire dalla versione 10 di Android, è stata introdotta una restrizione che impedisce alle applicazioni di eseguire i file all'interno della home directory in quanto questo costituirebbe una violazione del W^X ("write xor execute"), ovvero un principio di sicurezza secondo il quale ogni spazio di memoria può essere scrivibile o eseguibile, ma non entrambi contemporaneamente. Di conseguenza, è diventato impossibile eseguire file binari o codice al di fuori del contenuto dell'APK. Per tale motivo, le componenti dell'agente responsabili della gestione dei servizi web sono state temporaneamente escluse dal pacchetto, almeno fino a quando tali dipendenze non potranno essere integrate in modo compatibile con le specifiche di sicurezza di Android.

Plugins

Lightning-rod, attraverso la connessione WAMP, consente l'esecuzione di chiamate RPC all'interno del dispositivo. Il codice eseguibile "a distanza" è denominato Plugin, il quale può essere iniettato all'interno del dispositivo per poi richiederne l'esecuzione remota.

L'esecuzione dei plugin all'interno dell'applicazione consente di eseguire codice specifico alla tipologia di utilizzo dell'applicazione e, soprattutto, specifico all'architettura utilizzata. Un importante aggiunta all'interno dell'agente è la capacità di comunicare con il sistema sottostante tramite comandi shell se l'applicazione viene eseguita in ambiente desktop o mediante l'accesso all'SDK della piattaforma se eseguita su dispositivi mobili. I Plugin sono supportati dai backend Rubicon-ObjC[4] e Chaquopy[5], rispettivamente per iOS e Android; il loro compito è fungere da ponte tra l'ambiente Python nel quale viene eseguito il plugin stesso —e il sistema sottostante che mette a disposizione delle API native per accedere alle funzioni e alle periferiche del dispositivo.

Alcune funzionalità, come ad esempio il GPS, richiedono permessi speciali per essere utilizzate. L'utente deve autorizzare manualmente l'accesso alle periferiche sensibili tramite impostazione o tramite relativo popup durante l'esecuzione del Plugin, inoltre ogni permesso deve essere esplicitato nel AndroidManifest.xml come mostrato in Figura 4.

[4] https://github.com/beeware/rubicon-objc

[5] https://chaquo.com/chaquopy/

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-feature android:name="android.hardware.sensor.gyroscope" android:required="true" />
    <uses-feature android:name="android.hardware.sensor.proximity" android:required="true" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/formal_name"
        android:networkSecurityConfig="@xml/network_security_config"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme.Launcher">
      <!-- https://developer.android.com/guide/topics/resources/runtime-changes#HandlingTheChange -->
      <activity
            android:configChanges="orientation|screenSize|screenLayout|keyboardHidden"
            android:name="org.beeware.android.MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Figura 4 **Esempio di AndroidManifest.xml**

All'interno del repository sono inclusi alcuni esempi di Plugin che illustrano in dettaglio come utilizzare Chaquopy. Tuttavia si consiglia di consultare la documentazione dell'SDK e del backend di riferimento per una migliore comprensione. Di seguito è riportato un frammento di codice di un Plugin che utilizza Chaquopy per accedere ai dati dell'accelerometro.

```python
class PythonSensorEventListener(dynamic_proxy(SensorEventListener)):
    def __init__(self, event):
        super().__init__()
        self.event = event
        self.accel_results = None

    @Override(jvoid, [SensorEvent])
    def onSensorChanged (self, sensorEvent):
        self.accel_results = {
            "Sensor_name" : sensorEvent.sensor.getName(),
            "Accuracy" : sensorEvent.accuracy,
            "Timestamp" : sensorEvent.timestamp,
            "Distance" : sensorEvent.values[0]
        }
        self.event.set()

    @Override(jvoid, [Sensor, jint])
    def onAccuracyChanged(self, sensor, i):
        pass
```

Figura 5 **Frammento di codice Python con API Java di Chaquopy**

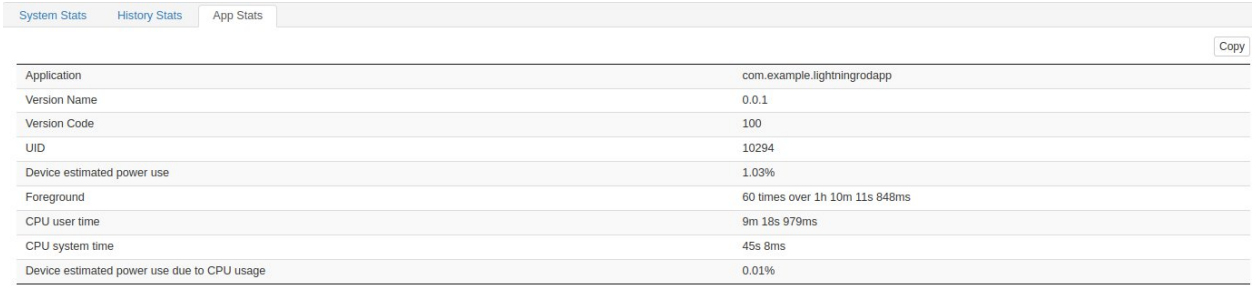
Consumi rilevati

Dato il carattere IoT dell'applicazione, sviluppata principalmente per dispositivi mobili, è essenziale effettuare un approfondita analisi dei consumi energetici. Tuttavia, stimare con precisione tali consumi può rivelarsi complesso a causa di numerosi fattori esterni che influenzano le misurazioni, rendendo quindi queste poco più che stime indicative. Tra questi fattori esterni rientrano l'hardware del dispositivo (come CPU, memoria disponibile e capacità della batteria) e le varie attività eseguite all'interno dell'applicazione.

Al fine di fornire dati il più oggettivi possibile, riducendo qualsiasi possibile bias causato da fattori esterni, è stato utilizzato uno strumento di monitoraggio messo a disposizione da Google, il quale utilizza i log di sistema per monitorare lo stato delle applicazioni e i relativi consumi. Tale strumento prende il nome di Battery Historian e permette di analizzare i consumi di rete ed energetici, filtrando i risultati per singola applicazione; questo permette di avere un idea più precisa sui possibili consumi in relazione al tipo di utilizzo e hardware di esecuzione.

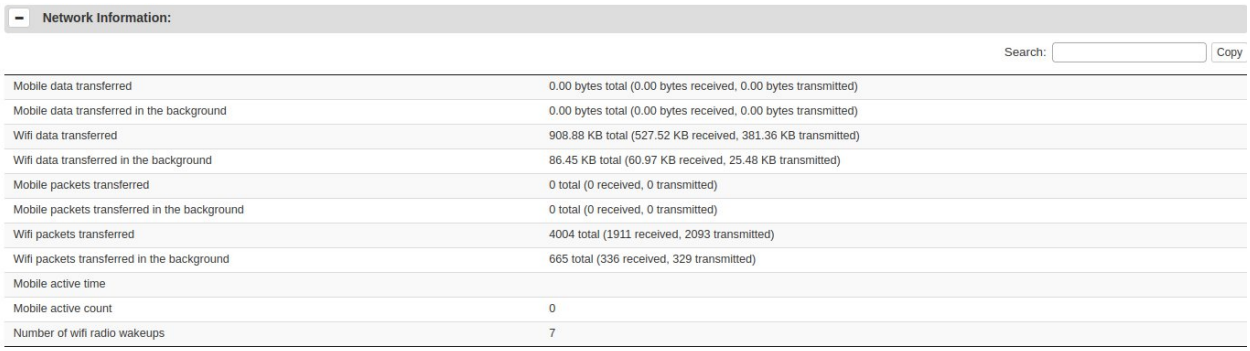
Figura 6 Grafico timeline

Un primo test effettuato può essere considerato come uno stress test dell'applicazione: il tempo di osservazione del test è stato di circa un ora, durante la quale diversi Plugin sono stati iniettati e frequentemente eseguiti all'interno dell'applicazione. Tali Plugin avevano il compito di stressare il sistema eseguendo computazioni numeriche, accedendo ai dati dell'accelerometro e del GPS. L'esecuzione delle RPC è stata frequente nel tempo di osservazione, inoltre il dispositivo è stato sbloccato diverse volte per controllare lo stato dell'applicazione e i log generati.



Application	com.example.lightningrodapp
Version Name	0.0.1
Version Code	100
UID	10294
Device estimated power use	1.03%
Foreground	60 times over 1h 10m 11s 848ms
CPU user time	9m 18s 979ms
CPU system time	45s 8ms
Device estimated power use due to CPU usage	0.01%

Figura 7 Statistiche energetiche

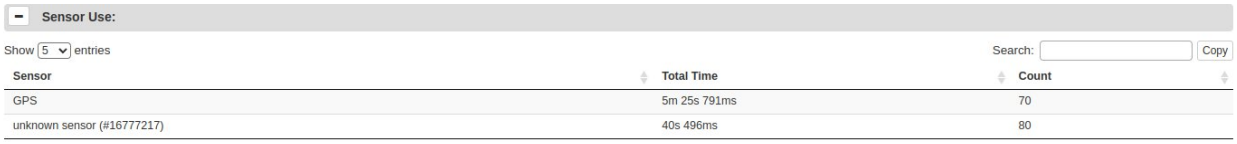


Network Information:

Search: Copy

Mobile data transferred	0.00 bytes total (0.00 bytes received, 0.00 bytes transmitted)
Mobile data transferred in the background	0.00 bytes total (0.00 bytes received, 0.00 bytes transmitted)
Wifi data transferred	908.88 KB total (527.52 KB received, 381.36 KB transmitted)
Wifi data transferred in the background	86.45 KB total (60.97 KB received, 25.48 KB transmitted)
Mobile packets transferred	0 total (0 received, 0 transmitted)
Mobile packets transferred in the background	0 total (0 received, 0 transmitted)
Wifi packets transferred	4004 total (1911 received, 2093 transmitted)
Wifi packets transferred in the background	665 total (336 received, 329 transmitted)
Mobile active time	
Mobile active count	0
Number of wifi radio wakeups	7

Figura 8 **Statistiche di rete**



Sensor Use:

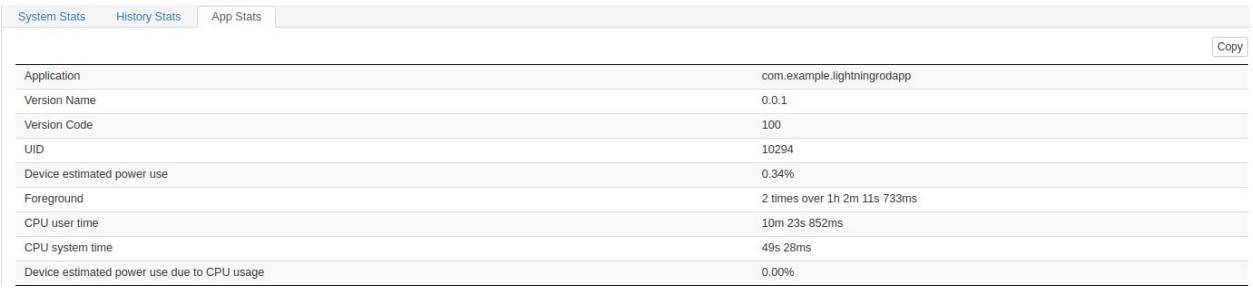
Show 5 entries Search: Copy

Sensor	Total Time	Count
GPS	5m 25s 791ms	70
unknown sensor (#16777217)	40s 496ms	80

Figura 9 **Sensori utilizzati**

Com'è possibile osservare dalle immagini, i consumi sono più che contenuti considerando che il test effettuato fosse uno stress test. Dai risultati ottenuti è possibile affermare che il carico computazionale sulla CPU cresce linearmente con il numero e la tipologia di Plugin eseguito, inoltre che l'accesso alle periferiche del dispositivo non costituisce un vero costo energetico in quanto tali dati sono periodicamente aggiornati a un livello inferiore all'applicazione.

Come previsto, i consumi reali sul dispositivo dipendono principalmente dallo stato delle periferiche e non dai consumi effettivi dell'applicazione, pertanto è ragionevole supporre che le prestazioni energetiche possano essere ulteriormente migliorate riducendo il numero di RPC eseguite o semplicemente utilizzando il telefono a schermo bloccato. Un ulteriore test è stato eseguito con le suddette specifiche, il quale ha portato ai risultati previsti.



System Stats | History Stats | App Stats

Copy

Application	com.example.lightningrodapp
Version Name	0.0.1
Version Code	100
UID	10294
Device estimated power use	0.34%
Foreground	2 times over 1h 2m 11s 733ms
CPU user time	10m 23s 852ms
CPU system time	49s 28ms
Device estimated power use due to CPU usage	0.00%

Figura 10 **Statistiche energetiche durante il mild test**

Riguardo al consumo energetico va comunque effettuata la precisazione che l'applicazione, affinché continui a funzionare a schermo bloccato, deve essere esclusa dalle politiche di risparmio energetico del dispositivo. Tale operazione deve essere obbligatoriamente effettuata manualmente dall'utente, il quale dovrà disabilitare le ottimizzazioni energetiche sull'applicazione oltre che quelle relative al traffico dati quando lo schermo è bloccato. Inoltre, a seconda della versione di Android e delle sue varianti, tali ottimizzazioni possono essere più o meno aggressive su diversi aspetti, pertanto sarà responsabilità dell'utente disabilitarle.

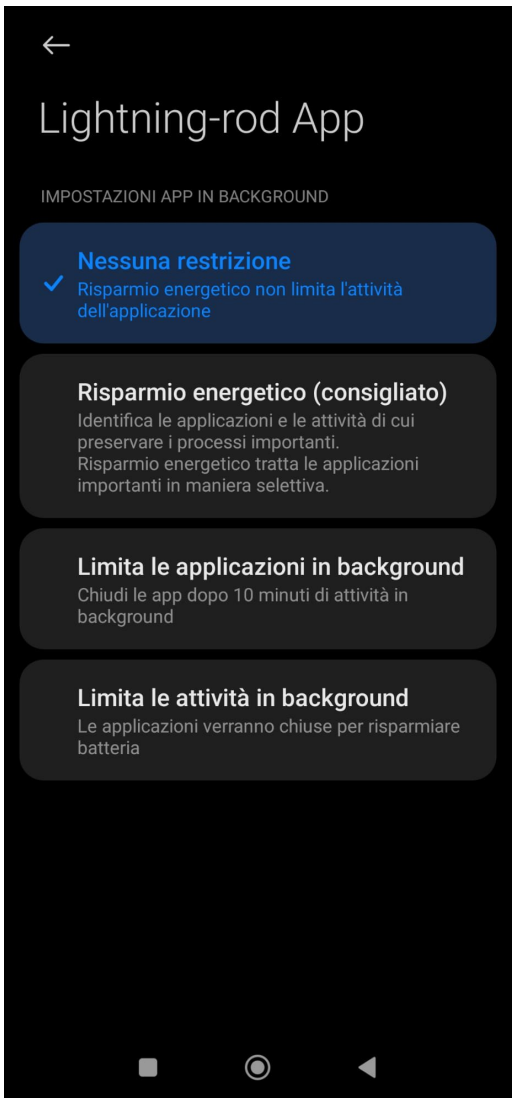


Figura 11 Esempio di schermata di ottimizzazione energetica dell'applicazione

Sviluppi futuri

L'obiettivo primario di questo progetto era quello di fornire una base di sviluppo per Lightning-rod su diverse piattaforme. L'applicazione risulta funzionante nelle features sviluppate e testate, tuttavia vi è sicuramente spazio per eventuali migliorie. Alcuni possibili sviluppi futuri riguardano sicuramente l'implementazione delle features mancanti per problemi di compatibilità, un miglioramento generale dell'interfaccia grafica anche in vista delle future versioni di Beeware e ovviamente la possibilità di esportare l'applicazione su web qualora le problematiche precedentemente elencate vengano risolte nelle future versioni di WebAssembly.