Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Progettazione e Sviluppo di un Tool per la Generazione Automatica e il Monitoraggio di Scenari Simulati di Mobile Crowdsensing

Relatore:
Dott. Federico Montori

Presentata da:
Gabriele Evangelista

Sessione 2
Anno Accademico 2020/2021

01001001 01100110 01010111 01101001 01110011 01101000
01100101 01110011 01010111 01100101 01110010 01100101
01001000 01101111 01110010 01110011 01100101 01110011

Sommario

Il Mobile CrowdSensing (MCS) sta guadagnando una grandissima fama come paradigma per la raccolta di dati in contesti urbani. Nelle campagne di MCS, un gruppo di persone fornisce informazioni, rilevate grazie ai propri dispositivi mobili, utili per monitorare fenomeni ed eventi. Di solito, c'è bisogno di un gran numero di partecipanti per poter ottenere una quantità di dati adeguata. Una buona alternativa è l'utilizzo di simulatori, che permettono di evitare questo problema. CrowdSenSim 2.0 è il simulatore utilizzato in questo lavoro, pensato nello specifico per simulazioni in ambienti urbani.
Il mio contributo è stato quello di sviluppare un tool per scienziati che vogliono automatizzare tutte le fasi di simulazione (ad esempio lanciandone tante in sequenza) e visualizzare i risultati in modo user-friendly, evitando inoltre di sapere come funziona CrowdSenSim 2.0. Il tool, ovvero un'applicazione Android, permette di utilizzare il simulatore dovunque. Il lavoro è diviso in tre punti:

- Modifiche al simulatore CrowdSenSim 2.0 per renderlo completamente utilizzabile via linea di comando
- Creazione di un server che fornisce un numero di API con le quali interfacciarsi al simulatore da remoto
- Sviluppo di un'app Android con la quale, sfruttando le API create nel punto precedente, si può utilizzare il simulatore e visualizzare i risultati su mappe

Introduzione

Mobile crowdsensing (MCS) è diventato uno dei paradigmi più allettanti per il rilevamento di dati in contesti urbani smart, guadagnando molta popolarità negli ultimi anni. Nei sistemi di MCS un insieme di persone fornisce dati raccolti personalmente attraverso i propri dispositivi mobili per monitorare eventi. Gli smartphone, i tablet e i dispositivi wearable, quelli indossabili, sono ormai diffusissimi e sempre a portata di mano; inoltre dispongono di un gran numero di sensori. La mobilità umana garantisce una copertura più elevata e una migliore context-awareness di una rete di sensori tradizionale. Ciò li rende degli eccellenti strumenti per poter rilevare informazioni dall'ambiente circostante. Il problema principale è che, tipicamente, è necessario un gran numero di partecipanti per rendere una campagna di MCS utile. Spesso infatti non è fattibile reclutare un elevato numero di persone a causa dei costi da sostenere all'aumentare dei partecipanti e del tempo necessario per organizzare la campagna stessa. Inoltre un membro potrebbe essere restio nel condividere i propri dati, che spesso contengono dati sensibili come la posizione geografica. L'alternativa valida è rappresentata dai simulatori, che permettono, in tempi accettabili e a costi contenuti, di osservare e studiare le campagne di MCS e di impostare parametri per poterle configurare nel modo più utile, scegliendo ad esempio il numero di partecipanti.
Di seguito viene mostrata la struttura della tesi. Nel capitolo 2 viene presentato il simulatore CrowdSenSim 2.0, la sua architettura e il suo funzionamento, da cui è iniziato questo lavoro. Il capitolo 3 presenta le modifiche apportate ad esso e la creazione di un server che fornisce il simulatore co-

me servizio. CrowdSenSim 2.0 è stato reso utilizzabile interamente tramite linea di comando e sono state ottimizzate alcune operazioni, come la compilazione del codice tramite Makefile e il riutilizzo di mappe già scaricate. Del server vengono specificate la sua struttura e tutte le API che possono essere utilizzate per interagire con il simulatore da remoto. Nel capitolo 4 viene presentata CrowdSenSim App, l'applicazione sviluppata per dispositivi Android con la quale è possibile, sfruttando le API del server, utilizzare il simulatore in maniera user-friendly e agnostica. Il capitolo 5 si focalizza sull'analisi delle performance, in termini di tempo, di tutta la catena della simulazione composta dal download della mappa di una città; dalla generazione di eventi; dalla simulazione vera e propria e dalla visualizzazione dei risultati.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Stato dell'arte

1.1 Mobile CrowdSensing

Il termine *mobile crowdsensing* (MCS) è stato introdotto per la prima volta da Ganti *et al.* [7] per indicare un paradigma più generale del *mobile phone sensing* dal quale, secondo [5], si è evoluto. Guo *et al.* fornisce una definizione che evidenzia chiaramente la differenza: "Il mobile crowdsensing è un nuovo paradigma che permette a cittadini ordinari di contribuire nella raccolta dati rilevandoli o generandoli dai loro smartphone, che aggrega e unisce i dati per fornire un cloud specializzato nell'estrazione di dati di massa e per la distribuzione di un servizio incentrato sulle persone" [8].
Dunque, mentre il mobile phone sensing ha applicazioni incentrate su un individuo, il MCS ha come target un massa di persone. Un esempio tipico di mobile phone sensing sono le applicazioni di fitness, che utilizzano il GPS e altri sensori insallati sui cellurari degli utenti per rilevare le loro sessioni di attività fisica (come camminate, corse o attività ciclistiche).
Il MCS, unito all'Internet of Things e al cloud computing, è, invece, il candidato perfetto per la creazione di smart city all'avanguardia. Si può occupare di diversissimi campi, come la gestione intelligente del traffico (Nericell è un'applicazione che si occupa di ciò [12]), la rilevazione di parcheggi liberi (ParkSense li rileva utilizzando le scansioni WiFi degli smartphone [15]), il

monitoraggio del clima, della predizione di eventi (anche catastrofici), la localizzazione indoor, l'inquinamento dell'aria (come Hazewatch [16]) e tanti altri. L'impatto del fattore umano è notevole nel MCS, in quanto la mobilità e l'intelligenza di partecipanti umani garantisce una maggiore copertura e una migliore context awareness se comparata con le tradizionali reti di rivelamento [5]. In più, gli utenti si occupano personalmente dei propri dispositivi fornendo ad essi una carica periodica.
È possibile categorizzare il MCS in una struttura a quattro livelli, come mostrato in Figura 1.1:

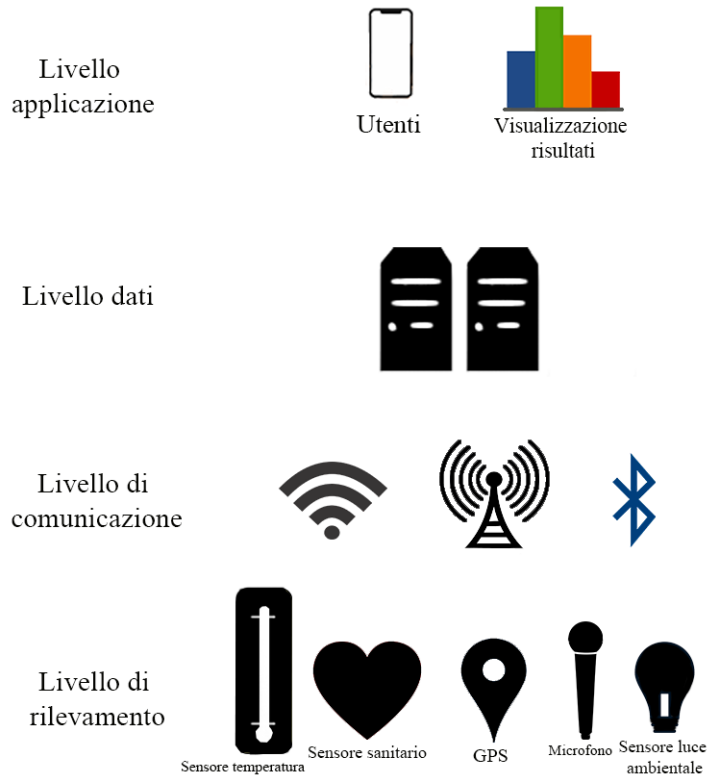


Figura 1.1: Architettura a quattro livelli del MCS

1. *Livello applicazione*: questo è il livello più alto della struttura e include gli utenti e le tecniche per reclutarli, le attività che essi svolgono, la

visualizzazione dei dati elaborati, la progettazione e l'organizzazione della campagna.

2. *Livello dati*: si occupa di immagazzinare ed elaborare i dati raccolti dal livello precedente e i protagonisti di questo livello sono i sistemi cloud.

3. *Livello di comunicazione*: comprende tutte le tecnologie e le infrastrutture che permettono la comunicazione dei dati rilevati, come il WiFi, le reti mobili, le onde radi o il Bluetooth. Possono essere applicate tecnologie per evitare la trasmissioni di dati duplicati e ridondanti.

4. *Livello di rilevamento*: è il cuore del MCS, formato da tutti i sensori che permettono di rilevare i dati. Possono essere diversissimi tra loro e spesso i più comodi sono quelli già installati sugli smartphone, come il GPS, l'accelerometro, sensori di temperatura, di luce ambientale o di prossimità, ma anche i più comuni microfoni e fotocamere. Possono essere inoltre connessi ai dispositivi mobili dei sensori specializzati e non comuni nell'utilizzo quotidiano, come sensori per le radiazioni, per la qualità dell'aria o per la presenza di glutine.

Si può inoltre distinguere il modo in cui gli utenti vengono coinvolti nella ricezione dei dati in due diverse categorie:

- *Opportunistic crowdsensing*: Ganti *et al.* lo definisce spiegando che *"il rilevamento è più autonomo e il coinvolgimento dell'utente è minimo (ad esempio, la continua rilevazione della posizione)"* [7]. L'utente quindi non ha alcuna attività specifica da eseguire, se non quella di installare l'applicazione sul dispositivo.

- *Participatory crowdsensing*: la sua definizione enfatizza la partecipazione esplicita degli utenti.

In realtà, in alcuni casi "mobile crowdsensing" e "participatory sensing" sono usati interscambiabilmente e, in altri, "mobile crowdsensing", "participatory sensing" e "opportunistic sensing" vengono utilizzati come sinonimi [5].

Sviluppare architetture che prevedano l'inserimento del fattore umano può però essere impegnativo, e non è l'unico problema di questo paradigma.
L'efficacia del MCS, come ribadito più volte in precedenza, dipende fortemente dalla quantità di partecipanti ma spesso questi sono riluttanti a collaborare per mancanza di incentivi appropriati [19]. Per esempio, la partecipazione a un'attività di rilevamento dati consumerà per forza molteplici risorse del dispositivo, come la batteria e la capacità di calcolo. In più, spesso i dati rilevati contengono informazioni di geo-localizzazione e condividerli suscita, negli utenti particolarmente sensibili all'importante tema della privacy, del disagio.
Questi fattori spesso impediscono di poter effettuare campagne di MCS in un vasto ambiente urbano con un alto numero di utenti. Ancora più difficile è iniziarle in un breve lasso di tempo. Vengono dunque eseguite simulazioni in un ambiente urbano realistico, impostando dati (come il numero di partecipanti o il tempo totale di simulazione) che vengono ritenuti adeguati, per poter sperimentare specifici aspetti di un sistema MCS (per esempio le tecniche per comunicare i dati o l'algoritmo per il reclutamento degli utenti) e analizzarne le performance.

1.2 Applicazioni mobili

Le applicazioni mobili sono programmi sviluppati per funzionare su dispositivi mobili come smartphone, tablet, wearable. Nel corso degli anni c'è stata una crescita esponenziale nel numero di app, che sono andate via via a specializzarsi in diverse categorie: da quelle di comunicazione, a quelle di intrattenimento, di istruzione, di personalizzazione del dispositivo, di incontri e, come visto nella sezione precedente, di fitness. Molte sono le app sviluppate con l'intento di agevolare il lavoro e la ricerca degli esperti del vasto mondo delle scienze, fornendo loro ambienti intuitivi e a portata di mano.
Moticon, ad esempio, ha sviluppato delle solette per scarpe completamente wireless con lo scopo di far studi clinici e ricerca riguardo alla performance

negli sport, oltre che permettere il monitoraggio di pazienti. Ha, in aggiunta, sviluppato la Moticon SCIENCE Mobile App con la quale è possibile vedere i dati rilevati dalle solette in tempo reale, come una heatmap di pressione esercitata dai piedi o la forza di reazione al suolo esercitata dagli stessi. In [2] vengono studiate, insieme alle solette Pedar-x, per determinare l'affidabilità e la qualità della misurazione della pressione del plantare e della forza di reazione.

Lig-Aikuma è un'applicazione di registrazione per la documentazione di linguaggi [4]. È stata sviluppata per registrare in modo semplice dati linguistici e genera automaticamente file contenenti dati di dialoghi in formato raw, oltre che metadati relativi all'oratore. Il suo obiettivo è supportare il progetto BULB (Breaking the Unwritten Language Barrier), nato per velocizzare l'analisi di linguaggi orali e non scritti della famiglia del Bantu. Questo progetto dipende dalla cooperazione di linguisti e scienziati informatici, impegnati nella creazione di tecnologie e competenze nelle aree del natural language processing, dell'automatic speech recognition e del machine translation [1]. L'app Lig-Aikuma permette quattro modalità di registrazione: registrazione libera (di un discorso); respeaking (di un discorso precedentemente registrato); traduzione (di un discorso precedentemente registrato); ricavare un discorso da un file di testo (eliciting) [4].

Corona Health è stata sviluppata in cooperazione con Robert Koch Institute (RKI), l'istituto governativo tedesco responsabile del controllo e della prevenzione di malattie. È basata su TrackYourHealth, una piattaforma modulare server-client per applicazioni mobile. Il suo obiettivo principale è quello di condurre diversi studi sugli effetti diretti e indiretti del COVID-19 e sulle contromisure da adottare per preservare la salute mentale e fisica della popolazione [18]. Gli utenti, dunque, oltre a ricevere feedback sulla loro salute, aiutano a riconoscere le necessità di miglioramento dei sistemi sanitari esistenti compilando questionari.

TYDR (Track Your Daily Routine) è un app nata per raccogliere dati per la ricerca di correlazioni tra i dati forniti da uno smartphone e la personalità

del proprietario in modo da predire personalità grazie ai dati ricevuti da dispositivi mobili [3]. L'app fa il tracking dei seguenti dati: posizione, meteo, sensore delle luci ambientali, accelerometro, attività fisiche, passi, sblocco/-blocco del telefono, inserimento e rimozione delle cuffiette, batteria e il suo caricamento, WiFi, Bluetooth, metadati delle chiamate, metadati della musica, metadati delle foto, metadati delle notifiche, utilizzo delle app e traffico delle app. In più utilizza dei questionari psicometrici standardizzati per ricevere dati demografici impossibili da rilevare in automatico.
Un'altra applicazione che rientra in questa categoria è CrowdSenSim App, l'app sviluppata per poter utilizzare il simulatore attraverso uno smartphone che verrà presentata in seguito.

1.3 Simulazione

Per simulazione si intende la progettazione e la realizzazione di un modello di un sistema in grado di imitare il comportamento del sistema reale. Questa deve consentire di valutare e prevedere lo svolgersi di eventi dinamici imposti da un analista. Per esempio, una simulazione di volo permette di prevedere il comportamento di un aeroplano in base ai comandi forniti dal pilota. È dunque importante analizzare la realtà e isolare le caratteristiche interessanti da includere nel modello. Altre caratteristiche fondamentali per una simulazione sono la sua complessità o semplicità, che devono essere adeguate alle necessità dell'analisi; la sua fedeltà al sistema reale per fornire dati utili; il tempo di simulazione, che spesso non coincide con quello che viene chiamato "wall-clock time", ovvero il tempo reale trascorso. Un modello può essere statico, in cui il suo stato non dipende dal tempo, o dinamico, nel quale invece il suo stato dipende da esso; deterministico, in cui non esiste alcun grado di casualità e dunque l'output per un determinato input è sempre lo stesso, o probabilistico, nel quale le variabili in input possono cambiare casualmente in base a una probabilità individuale e dunque, anche fissando un input, l'output è variabile.

Analizzando i vari simulatori esistenti utilizzati per una campagna di MCS, è evidente come non esista un simulatore che permetta di essere totalmente flessibile riguardo le proprietà elencate e che possa ricoprire qualsiasi variabile necessaria per essere completamente fedele alla realtà.
Network Simulator 3 (NS-3) è un simulatore di rete a eventi discreti progettato sia per la ricerca in ambito di networking, sia per l'istruzione. La sua architettura principale emula una vera e propria rete, incluso lo stack di protocolli ISO/OSI. In [17], viene utilizzato per fornire un esempio di rete di crowdsensing dove gli utenti riportano incidenti avvenuti nei servizi di trasporto sotterranei di Barcellona. I risultati mostrano che con 100 utenti riesce a rilevare il 29% degli incidenti avvenuti e la percentuale aumenta al 70% con 1200 utenti; i passeggeri dei servizi pubblici sotterranei, però, sono in media 1,07 milioni al giorno. Se si volesse eseguire una simulazione più vicina alla realtà bisognerebbe aumentare di diversi ordini di grandezza il numero di utenti e, considerando il livello di dettaglio dei dati comunicati, si creerebbe un'enorme mole di dati. Questo simulatore dunque è fortemente limitato sull'aspetto della scalabilità.
Objective Modular Network Testbed in C++ (OMNeT++) è un simulatore orientato agli oggetti di eventi modulari discreti che può essere usato per simulare, ad esempio, protocolli di comunicazione, reti di computer, sistemi distribuiti e multi-processore e giochi [9]. Offre inoltre una GUI con la quale è possibile seguire il flusso della simulazione e permette di lanciare simulazioni in parallelo. Soffre però dello stesso problema di NS-3, ovvero non è facilmente scalabile. Un importante fattore da tenere in considerazione, inoltre, è il tempo di esecuzione: nel caso sia eccessivo, e spesso lo è nei modelli accurati e dettagliati, rappresenta un grosso limite nell'utilizzo stesso del simulatore.
CubCarbon è un simulatore basato su simulazioni a eventi discreti e multi-agente [11] ma, a differenza dei precedenti, non implementa tutti i protocolli di rete. Offre una semplice interfaccia grafica per poter modellare reti utilizzando il framework OpenStreetMap (OSM). Si possono inserire, infatti, i sensori nell'esatta posizione desiderata, renderli mobili e possono essere in-

dividualmente configurabili con la propria command-line. Sempre in [11], viene mostrato come usarlo per simulare il diagramma dell'energia di una rete di sensori progettata su OSM. Anche se è particolarmente indicato per osservare il consumo energetico dei sensori, i modelli di traffico (ovvero i tipi di applicazioni, protocolli e servizi simulabili) non sono sufficienti e, secondo gli autori dell'articolo, il suo inserimento nel mercato è stato immaturo in termini di alcune feature che sono invece implementate in altri simulatori.

DIANEmu è un simulatore basato su eventi e simula solo protocolli di livello applicazione su ad hoc networks [10]. Queste sono reti in cui ogni nodo partecipa nel routing e nel forwarding dei dati per gli altri nodi, dunque la decisione di quale di questi si occupi del forwarding dei pacchetti è dinamica e basata sulle connessioni della rete e dall'algoritmo di routing in uso. DIANEmu permette di creare protocolli propri. Questi devono essere realizzati sotto forma di macchine a stati finiti e devono fornire funzioni pubbliche agli utenti e agli altri protocolli.

CrowdSenSim è un simulatore progettato per supportare un numero nell'ordine delle decine di migliaia di partecipanti che si muovono in un ampio ambiente urbano realistico [6]. Permette di visualizzare risultati inerenti al reclutamento dei partecipanti e al costo sostenuto per il rilevamento dati; in aggiunta include dei pattern di movimento per la mobilità di pedoni in ambienti urbani, ricevendo come input il layout della città su cui eseguire la simulazione. Non implementa il completo stack di rete ma fornisce un sufficiente livello di dettagli nei risultati mostrati. Il simulatore è fortemente incentrato sull'energia e implementa diversi algoritmi, per scenari partecipatori e opportunistici, con l'obiettivo di ridurre il consumo dell'energia del dispositivo [13]. Questo simulatore pecca nell'essere adattabile a molti casi d'uso tipici di una campagna MCS, tra cui quelli stateful (ovvero il momento in cui vengono rilevati gli eventi è importante e gli algoritmi dipendono dalla dipendenza che esiste tra i diversi tempi), e nell'essere poco flessibile nella creazione di eventi [14]. Molti di questi problemi sono stati risolti nella versione 2.0 del simulatore, che viene trattata nel capitolo successivo.

1.4 Motivazione

I simulatori precedentemente descritti permettono l'esecuzione di simulazioni avanzate, fornendo spesso dei metodi per poter personalizzare alcune feature. Senza un'ottima conoscenza dello specifico simulatore, però, il suo utilizzo diventa estremamente difficile e ciò può scoraggiare fortemente il suo impiego. In aggiunta, l'installazione dei suddetti simulatori è un'operazione per niente triviale o, comunque, lunga e noiosa: bisogna, dopo aver scaricato file di grossa dimesione, reperire le giuste librerie per il sistema target e poi compilarlo. Ovviamente, bisogna ripetere queste operazioni su tutti i sistemi su cui si vuole installare il simulatore.
Ho iniziato questo progetto per risolvere questi problemi: ho reso il simulatore CrowdSenSim 2.0 disponibile attraverso un servizio, comodamente utilizzabile attraverso un'app sviluppata per smartphone Android. Grazie a questa è possibile creare e iniziare simulazioni da remoto per poi visualizzare i risultati delle simulazioni su una mappa, il tutto in maniera user-friendly e senza sapere minimamente come sia fatto o come funzioni il simulatore. In questo modo, l'unica azione da compiere è l'installazione dell'app, senza doversi preoccupare in alcun modo di dover reperire il simulatore e usarlo direttamente. L'app non è stata sviluppata con l'intento di essere utilizzata da normali utenti, per permettere loro di rilevare dati, ma è stata realizzata per scienziati che hanno bisogno di lanciare in sequenza tante simulazioni e di analizzarne i risultati.

Capitolo 2

CrowdSenSim 2.0

In questo capitolo viene presentato il simulatore CrowdSenSim 2.0, sottolineando le novità rispetto alla sua versione precedente. Viene mostrata la sua architettura, specificando la funzione di ciascun modulo che lo compongono, e i miglioramenti apportati.

2.1 Architettura del simulatore

L'architettura del simulatore CrowdSenSim 2.0 è riportata in Figura 2.1. In rosso sono riportate le modifiche e le novità di questa versione che includono: un approccio stateful, di fondamentale importanza nelle campagne MCS; la generazione degli eventi è più flessibile in termini di granularità temporale e di altri parametri configurabili; l'utilizzo della codifica spaziale MGRS (Military Grid Reference System, uno standard di geocoordinate usate dai militari NATO al fine di localizzare punti sulla Terra); la generazione di cammini molto precisi percorsi dagli utenti simulati; il cambiamento dei cammini dinamico. Inoltre, il codice è stato sottoposto a un processo intensivo di refactoring e di pulizia.

Una simulazione viene svolta in questo modo. Dopo aver scelto la città in cui si deve svolgere la campagna di MCS, il simulatore genera un insieme di utenti che si muovo all'interno delle strade. Questi mandano i dati rilevati,

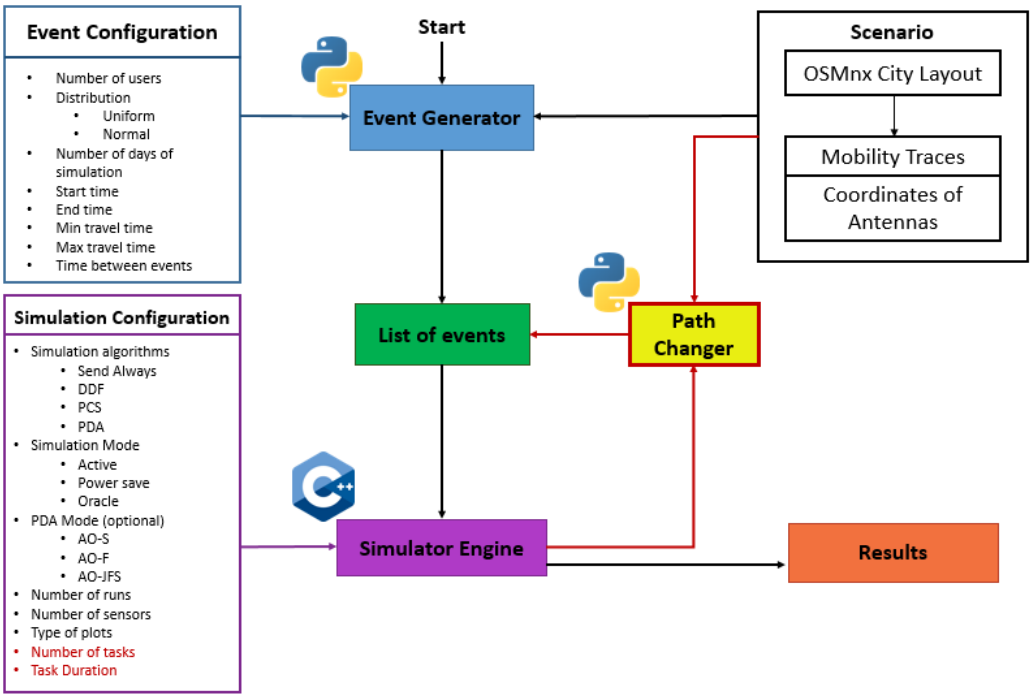


Figura 2.1: Architettura modulare di CrowdSenSim 2.0

ovvero la posizione o l'orario, alle antenne più vicine (che possono essere o degli access point WiFi o delle stazioni di rete cellulare). L'Event Generator, come dice il nome, genera gli eventi prendendo in input il layout della città, gli spostamenti degli utenti, le coordinate delle antenne e dei parametri presi da un file di configurazione. Successivamente, questa lista di eventi viene passata al Simulator Engine che si occupa di definire come gli utenti si comportano per ogni evento rilevato. Il modulo Path Changer permette di cambiare i cammini percorsi dagli utenti durante l'esecuzione, alterando la lista degli eventi: il Simulator Engine, dunque, riprende la sua esecuzione con la lista aggiornata.

Di seguito esploriamo i diversi moduli che compongono il simulatore.

2.2 Modulo City Layout

Il modulo City Layout permette di creare una rete di strade urbane sulle quali si muovono gli utenti simulati. Questi possono essere localizzati grazie a un insieme di coordinate, tra cui la longitudine, la latitudine e l'altitudine. Inoltre, CrowdSenSim 2.0 genera, per ogni evento, una coordinata MGRS che indica un'area quadrata sulla Terra. In base alla precisione della coordinata fornita, la lunghezza del lato del quadrato può essere di 10 km, 1 km, 100 m, 10 m o 1 m. Per creare il layout, la nuova versione del simulatore sfrutta il package Python open-source chiamato OSMnx[1]. Questo permette di scaricare layout di città sotto forma di grafi da OpenStreetMap[2].

2.3 Modulo Event Generator

Il modulo per la generazione di eventi è scritto completamente in Python. Questo permette di generare gli utenti in una determinata posizione e di muoverli all'interno dell'ambiente urbano. Gli utenti sono creati con una funzione di distribuzione spaziale e si muovono secondo diversi possibili modelli. La funzione di distribuzione può essere uniforme o casuale. Ogni pedone ha un determinato tempo di viaggio e il suo cammino è generato come una sequenza di eventi che vengono rilevati a intervalli di tempo regolari. Molte opzioni per la generazione di eventi sono state rese configurabili grazie al file di configurazione `Event_Generation_config.txt`:

- **Numero di utenti**: numero dei partecipanti della simulazione
- **Giorni di simulazione**: numero di giornate di simulazione
- **Distribuzione**: questa può essere normale o distribuita
- **Tempo di inizio della simulazione**: è possibile impostare l'ora e il minuto in cui la simulazione ha inizio

[1] `https://geoffboeing.com/2016/11/osmnx-python-street-networks/`
[2] `https://www.openstreetmap.org/`

- **Tempo di fine della simulazione**: come per quello di inizio è possibile impostare l'ora esatta di fine
- **Tempo minimo e tempo massimo di viaggio**: il tempo che ciascun utente ha per muoversi all'interno dell'ambiente urbano
- **Tempo tra gli eventi**: tempo che separa gli eventi
- **Numero di task**: per task si intendono eventi che non possono essere dedotti in anticipo. Per poter eseguire un task è sufficiente che l'utente si trovi nell'area di copertura del task stesso
- **Durata dei task**: tempo in cui è possibile eseguire un task

Al termine della generazione degli eventi viene creato un file che contiene le informazioni degli eventi generati, `UserMovementsList_0.txt`. Lo 0 indica il giorno della simulazione, dunque ne verranno creati tanti quanti i giorni indicati nella fase di creazione degli eventi. Ogni riga del file contiene i seguenti campi:

- **ID-User**: numero progressivo identificativo dell'utente
- **MGRS**: stringa MGRS con precisione di 1 metro, quella massima, dove avviene l'evento registrato
- **Day, Hour, Minute, Second**: giorno, ora, minuti e secondi in cui l'evento è stato rilevato
- **Lat, Long e Alt**: latitudine, longitudine e altitudine dell'evento
- **SecUsed**: secondi passati dall'inizio del percorso
- **Bearing**: valore dell'angolo α di Azimut tra il punto cardinale nord e il nodo della strada sulla quale si trova l'utente
- **Distance**: distanza percorsa fino al momento corrente

Gli ultimi tre valori non sono presenti nel primo evento di ciascun utente dal momento che vengono calcolati durante la creazione del percorso. Ciò significa che non sono presenti nel primo evento di ogni utente.

2.4 Modulo Simulator Engine

Il Simulator Engine è scritto in C++. Genera gli smartphone degli utenti da simulare con ciascuno una carica della batteria differente e, come si può vedere dalla Figura 2.1, prende come input la lista degli eventi generati nella fase precedente. A differenza della vecchia versione in cui gli eventi venivano eseguiti in ordine di user id (dunque gli eventi dell'utente 2 vengono eseguiti dopo tutti gli eventi dell'utente 1 e così via), la versione 2.0 del simulatore riordina gli eventi in ordine cronologico. La simulazione degli eventi avviene analizzandone uno alla volta e applicando tutti gli algoritmi specificati nel relativo file di configurazione chiamato `Simulation_config.txt`. In particolare, le opzioni configurabili in questo file (relativo per le simulazioni) sono le seguenti:

- **Tipo di grafico**: tipologia di grafico per la visualizzazione dei risultati. È possibile scegliere tra 7 opzioni
- **Somma dati sensori**: se l'opzione è inserita all'interno del grafico verranno sommati tutti i dati rilevati dai sensori dei cellulari
- **Tipo di antenna**: tipologia di tecnologia usata per trasmettere i dati ricavati dai sensori
- **Raggio heatmap**: alla fine della simulazione viene creata una pagina web in cui si può visualizzare una heatmap che analizza la distribuzione degli eventi letti. Questa opzione serve per impostare la dimensione in metri del raggio del cerchio mostrato nella heatmap
- **Numero di simulazioni**: valore che rappresenta quante simulazioni verranno eseguite

- **Numero di sensori**: quanti sensori sono stati assegnati a ciascuno smartphone
- **Lista di algoritmi**: quali algoritmi usare nella simulazione

È presente, inoltre, un ulteriore file di configurazione chiamato `PDA_config.txt`, il quale permette di configurare le opzioni per il Probabilistic Distributed Algorithm (PDA), uno degli algoritmi usati per il data collection:

- **Dimensione della finestra**: si specifica il numero di slot di tempo per rappresentare la dimensione della finestra
- **Dimensione time slot**: dimensione in secondi dello slot di tempo
- **Limite superiore e inferiore del SI**: valore massimo e minimo per l'indice di soddisfazione (SI)
- **Dati richiesti**: sensori presenti su ciascun dispositivo

Oltre al PDA, il Simulator Engine permette di usare anche l'algoritmo Piggyback CrowdSensing (PCS) e il Deterministic Distributed Algorithm (DDA). La nuova versione consente di applicarli tutti nella stessa esecuzione ed eseguirli nello stesso momento, generando risultati separati come se fossero stati eseguiti in diverse run del programma.

2.5 Modulo Path Changer

Questo nuovo modulo permette ai partecipanti di cambiare il proprio cammino durante l'esecuzione del programma a causa di particolari eventi. Gli utenti simulati hanno la capacità di prendere decisioni volontariamente, come muoversi verso un task dopo un evento specifico (ad esempio un incidente). Ovviamente, per permettere questa modifica dinamica del cammino degli utenti bisogna rompere il flusso pre-determinato della lista degli eventi. Dopo che un utente ha deciso di cambiare la traiettoria del proprio cammino,

la nuova lista di eventi generata viene passata di nuovo al Simulator Engine, il quale elimina tutti gli eventi da elaborare del pedone incriminato e li rimpiazza con quelli della nuova lista. L'ordine cronologico degli eventi della nuova lista viene comunque rispettata.

Capitolo 3

Modifiche e API

La Figura 3.1 rappresenta la struttura del lavoro svolto in questa tesi. Sono state apportate modifiche e miglioramenti al simulatore CrowdSenSim 2.0. È stato inoltre sviluppato un server che mette a disposizione delle API, le quali permettono di utilizzare il simulatore. Infine, è stata creata una applicazione per device mobili Android che, sfruttando le API del server, permette di utilizzare il simulatore da remoto, in maniera collaborativa, e di visualizzare i risultati.

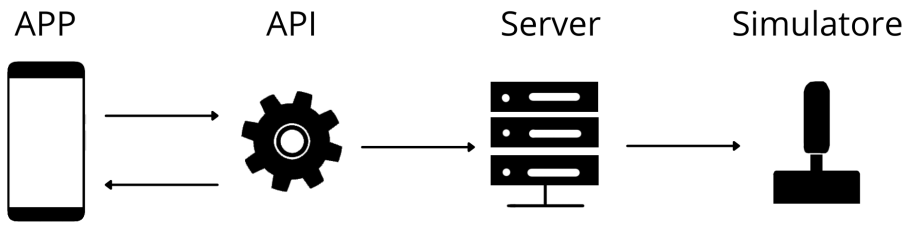


Figura 3.1: Architettura del sistema sviluppato

In questo capitolo vengono presentate le modifiche e le migliorie apportate al simulatore e la struttura del server che fornisce CrowdSenSim 2.0 come servizio, soffermandosi su tutte le API disponibili.

3.1 Modifiche al simulatore

3.1.1 Opzioni da terminale

Per prima cosa mi sono occupato di rendere il simulatore CrowdSenSim 2.0 completamente utilizzabile via linea di comando. Precedentemente, infatti, il simulatore stampava sul terminale dei menù e l'utente doveva scegliere, scrivendo un numero e premendo invio, quale azione compiere (ad esempio se creare una nuova lista di eventi o se iniziare una nuova simulazione). Ho modificato dunque il file `CrowdSenSim.cpp` aggiungendo tante opzioni quante sono le voci nei file di configurazioni del configuratore. Per poter fare il parsing degli argomenti passati tramite linea di comando al programma ho utilizzato la libreria `getopt.h`, sfruttando la sua funzione `getopt_long`. Questa è una variante della funzione `getopt` che permette di fare il parsing anche di argomenti con un lungo nome, come suggerisce il nome, facendoli iniziare con un doppio trattino. Finché la funzione `getopt_long` ritorna un valore diverso da -1 uno switch permette di eseguire l'azione corrispondente all'opzione passata. Di seguito vi è una lista di tutte le opzioni aggiunte con una spiegazione di quale valore si dovrebbe associare a ciascuna di esse.
Opzioni generiche:

- `-i`: inizia una nuova simulazione. Il valore deve essere l'indice della lista di eventi di cui si vuole far partire la simulazione. Non dovrebbe essere seguito da opzioni inerenti la creazione di eventi.
- `-g`: genera una nuova lista di eventi. Il valore deve essere il nome della città in cui viene generata la nuova lista di eventi.
- `-d`: elimina una lista di eventi. Il valore deve essere l'indice della lista che si vuole cancellare.

- `-r`: inizia più simulazioni. Il valore deve essere una lista di indici racchiusi tra virgolette, ad esempio `-r "70 71 72"`.

- `-s`: salva i valori passati alle opzioni. Se è inserita, i valori passati alle altre opzioni saranno salvati nei file di configurazione e usati nelle prossime simulazioni (se non verranno specificati diversi valori via terminale).

- `--id`: imposta un ID della simulazione. Questo ID viene utilizzato anche per creare una cartella all'interno del server, dentro la directory `simulations`, per salvare al suo interno i risultati della simulazione. Per chiarimenti, fare riferimento alla prossima sezione.

Opzioni relative alla creazione di eventi:

- `--n-users`: numero di utenti, da un minimo di 1 a qualsiasi numero.

- `--n-days`: numero di giorni, da un minimo di 1 a un massimo di 7.

- `--hour-start`: ora di inizio della simulazione, da un minimo di 0 a un massimo di 23.

- `--minute-start`: minuto di inizio della simulazione, da un minimo di 0 a un massimo di 59.

- `--hour-end`: ora di fine della simulazione, da un minimo di 0 a un massimo di 23.

- `--minute-end`: minuto di fine della simulazione, da un minimo di 0 a un massimo di 59.

- `--max-travel-time`: tempo massimo di viaggio in minuti, da 1 a un numero qualsiasi.

- `--min-travel-time`: tempo minimo di viaggio in minuti, da 1 a un numero qualsiasi.

- `--normal-distribution`: distribuzione degli eventi. Opzione senza valore, se è inserito la distribuzione è normale e non uniforme.
- `--time-between-samples`: tempo tra i vari sample.
- `--n-tasks`: numero di tasks, da un minimo di 1 a un qualsiasi numero.
- `--task-duration`: durata di un task, da un minimo di 1 a un qualsiasi numero.

Opzioni relative alle simulazioni:

- `-n`: nome della configurazione. I valori sono `JFS` o `JFSmia`.
- `--window-dimension`: numero di slot di tempo per la rappresentazione della dimensione della finestra per l'lgoritmo PDA. I valori vanno da un minimo di 1 a un qualsiasi numero.
- `--time-slot-dimension`: dimensione dello slot di tempo in secondi, da un minimo di 1 a un qualsiasi numero.
- `--lower-bound-si`: limite inferiore del SI (Satisfaction Index). Valori da 0.01 a qualsiasi numero float.
- `--upper-bound-si`: limite superiore del SI (Satisfaction Index). Valori da 0.01 a qualsiasi numero float.
- `--mgrs-precision`: precisione delle coordinate MGRS. Valori da 1 a 5.
- `--data-req-accelerometer`: quantitá di dati necessari da rilevare grazie all'accelerometro. Valori da 1 a un qualsiasi numero.
- `--data-req-gps`: quantitá di dati necessari da rilevare grazie al GPS. Valori da 1 a un qualsiasi numero.
- `--data-req-proximity`: quantitá di dati necessari da rilevare grazie al sensore di prossimità. Valori da 1 a un qualsiasi numero.

- `--graphic`: tipo di grafico da visualizzare per il risultato. I valori possibili sono: `0` per PDF SI; `1` per SI Time; `2` per PDF Data Sent; `3` per CDF SI; `4` per CDF Data Sent; `5` per Current Drain; qualsiasi altro numero per SI Time + PDF SI.

- `--no-sum-data`: opzione senza valore. Se inserito, i dati raccolti dai sensori non vengono sommati.

- `--antenna-type`: tipo di antenna. Valore di default: WiFi.

- `--heatmap-ray`: raggio della heatmap in metri. I valori vanno da 1 a un qualsiasi numero.

- `--n-simulations`: numero di simulazioni, da un minimo di 1 a un qualsiasi numero.

- `--n-sensors`: numero di sensori, da un minimo di 1 a un qualsiasi numero.

- `--dfc-config`: valore di default: PDA,JFSmia,AM,-1.

Per ciascuna di queste opzioni è stato aggiunto un controllo sul loro valore, ovvero se viene inserito un numero negativo o un numero fuori dal range accettato, il programma termina con un codice `RET_WRONG_ARGS` che ha come valore 3.

3.1.2 Miglioramenti

Sono state apportate diverse modifiche per migliorare la qualità del codice. Per prima cosa è stata aggiunta la possibilità di compilare il codice C++ del Simulator Engine attraverso lo strumento Make. Esso permette di non compilare l'intero programma ogni volta che si modifica una piccola parte del codice, infatti il Makefile compilerà automaticamente solo quei file in cui sono avvenute delle modifiche.
In questo caso sono stati aggiunti tre Makefile: il primo generico chiamato

appunto `Makefile`; il secondo incentrato sulla compilazione dei file di utility chiamato `utilsmake`; il terzo relativo ai file incentrati sull'algoritmo PDA chiamato `pdamake`. I target di `Makefile` sono `all`, che permette di compilare il Simulator Engine generando l'eseguibile chiamato `main-exe` e `clean`, per permettere la cancellazione di tutti i file oggetto. Di seguito si può vedere la regola applicata per la creazione dei file oggetto. Dal momento che alcuni file del Simulator Engine hanno diverse estensioni (`.cc` e `.cpp`) sono state create due regole indentiche per le diverse estensioni. Il target `clean` chiama il `clean` dei file `pdamake` e `utilsmake`, i quali hanno una struttura molto simile al Makefile generico, per poter cancellare tutti i file oggetto presenti.

```
%.o: %.cc
	$(CC) $(CFLAGS) -c -o $@ $<

%.o: %.cpp
	$(CC) $(CFLAGS) -c -o $@ $<

clean:
	rm -f *.o
	$(MAKE) -f pdamake clean
	$(MAKE) -f utilsmake clean
```

Il secondo miglioramento è stato quello di creare un file di requisiti per la parte scritta in Python, in modo da poter installare tutte le dipendenze con un unico comando, ovvero:

```
$> pip3 install -r requirements.txt
```

Le dipendenze sono le seguenti: mgrs (versione 1.4.0), folium (versione 0.12.1), geopy (versione 2.1.0), tqdm (versione 4.60.0), osmnx (versione 1.0.1), networkx (versione 2.5.1), numpy (versione 1.20.2) e matplotlib (versione 3.4.1). Il file `requirements.txt` è stato creato utilizzando il modulo `pipreqs`[1].
Un ulteriore modifica è stata apportata al modulo Event Generator. Pre-

[1] `https://pypi.org/project/pipreqs/`

cedentemente, ogni volta che si voleva creare una nuova lista di eventi, il modulo scaricava la mappa della città, la salvava in un file locale che veniva continuamente sovrascritto e la utilizzava. Ora invece, tutte le volte che si vuole creare una nuova lista di eventi, il modulo controlla prima se la mappa è già disponibile localmente. Se lo è, viene caricata attraverso la funzione `io.load_graphml` di `OSMnx`, altrimenti viene scaricata e salvata a parte con un suo nome (ad esempio `milano.graphml`).
Inoltre, sono stati inseriti all'interno del Simulator Engine dei print su terminale della percentuale di progresso della simulazione per poter sapere a quanto ammonta indicativamente.

3.2 Server

In questa sezione esploriamo il server che permette di utilizzare il simulatore da remoto, come è stato creato e quali API fornisce.
Il server è un web server Node[2], dunque scritto completamente in Javascript, disponibile all'indirizzo pubblico `http://130.136.2.167:3000/`.
È stato scelto di imporre l'autenticazione degli utenti per poter utilizzare il servizio, in modo da evitare che un malcapitato possa iniziare un numero esagerato di operazioni e causare un denial of service. Per questo, se si effettua una qualsiasi richiesta diversa da quella di login o di registrazione senza essere autenticato, il server risponde con un codice `403 Forbidden`. L'autenticazione implementata è basata su JSON Web Tokens (JWT)[3]. JWT è un metodo standard open (RFC 7519) che definisce un modo compatto e autonomo per trasmettere in modo sicuro informazioni tra diversi soggetti sotto forma di oggetti JSON.

3.2.1 Struttura del server

Vediamo ora la struttura, a livello di file e directory, del server:

[2] `https://nodejs.org/`
[3] `https://jwt.io/`

- `index.js`: è il file principale da cui si fa partire il server con il comando `$> node index.js`. Questo risponde alle varie chiamate HTTP e sfrutta il file `utils.js`.

- `utils.js`: è richiamato da `index.js` per poter gestire le richieste. Utilizza varie dipendenze per poter eseguire tutti i compiti richiesti, che vedremo di seguito.

- `users.json`: contiene gli utenti registrati al servizio.

- `.env`: contiene la variabile d'ambiente `ACCESS_TOKEN_SECRET`, una stringa da 128 caratteri utilizzata per l'autenticazione JWT.

- `pages`: directory che contiene le seguenti cartelle. Sono presenti le pagine HTML, con i relativi file CSS e Javascript, per poter utilizzare le funzioni del simulatore anche da un comune browser.

  - `html`
  - `css`
  - `js`
  - `swagger`
  - `simulations`: directory che contiene tutti i risultati delle simulazioni terminate, racchiusi all'interno di una cartella che ha per nome l'ID della simulazione.

3.2.2 Dipendenze

Il file `index.js` si appoggia interamente sul framework Express.js[4] per la progettazione della web application e delle API, che vediamo in 3.2.3.
Il file `utils.js` sfrutta diversi moduli node. Il primo è `jsonwebtoken`, utilizzato appunto per poter implementare l'autenticazione tramite JWT. Importa poi `bcrypt`, un modulo per poter cifrare dati utilizzato durante il salvataggio

[4] `https://expressjs.com/`

di un nuovo utente: la sua password non viene infatti salvata in chiaro ma in cypher-text, sfruttando anche la tecnica del sale. Il modulo `child_process` è utilizzato per poter iniziare l'esecuzione di un nuovo processo e per poter controllare il suo avanzamento, oltre che i flussi di standard output e di standard error. Il modulo `fs` viene utilizzato per poter accedere alle funzionalità del File System, come la creazione, l'eliminazione e l'elencazione di file. Viene utilizzato il modulo `dotenv` per poter utilizzare il file contente le variabili d'ambiente e il modulo `rimraf` per poter rimuovere una cartella in modo ricorsivo. È da notare che dalla versione 14 di Node, il modulo `fs` implementa nativamente la cancellazione ricorsiva, ma il server su cui è stato installato il servizio non aveva tale versione e dunque necessita di `rimraf`.

3.2.3 API

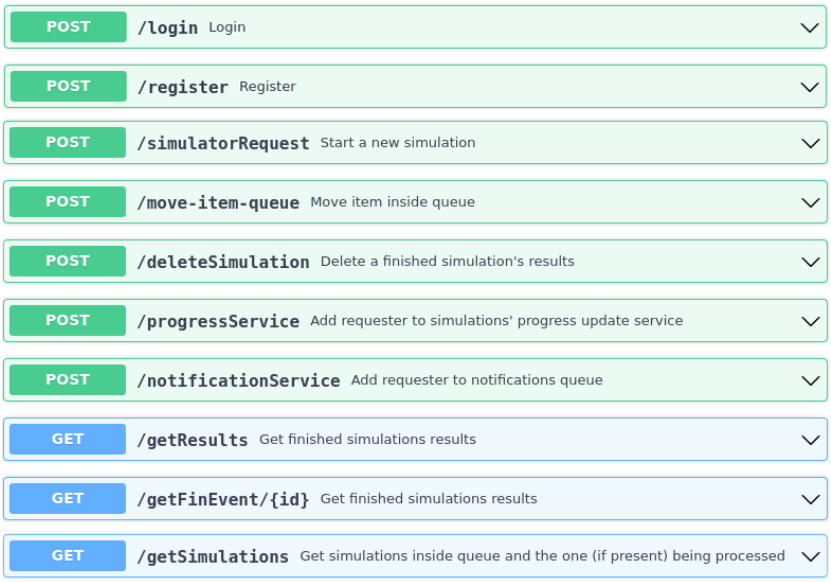


Figura 3.2: API fornite dal server

Analizziamo ora tutte le richieste che il server è in grado di soddisfare, iniziando dalle richieste di tipo POST. Viene specificato che tutte le richieste successive alla **/login**, per poter essere valide, hanno bisogno dell'hea-

der aggiuntivo chiamato `Authorization`. Il suo valore deve essere `Bearer [Token]`, come specificato dallo standard.

- **/register** : serve per registrare un nuovo utente. Viene gestita dalla funzione asincrona `registerRequest` all'interno di `utils.js`. Per effettuare tale richiesta, bisogna inserire nel body della richiesta l'username e la password dell'utente da creare. La funzione controlla se esiste già un utente, all'interno del file `users.json`, con lo stesso username. In caso affermativo, il server risponde con uno status code `400` e con un errore che ci informa del fatto che quell'username è già utilizzato; in caso negativo la funzione crea un hash della password seguita dal sale grazie al modulo `bcrypt`, viene inserito il nuovo utente all'interno del file `users.json` e il server ci risponde con uno status code `200`.

- **/login** : serve per poter effettuare il login. Questa viene gestita dalla funzione asincrona `loginRequest` che per prima cosa cerca se esiste l'utente con l'username passato. In caso negativo, ci viene informato ciò con uno status code `400`. In caso affermativo, calcola l'hash della password passata e controlla, sempre attraverso `bcrypt` che combacino. In caso negativo l'operazione non va a buon fine; in caso positivo, attraverso `jsonwebtoken`, viene creato e ci viene fornito un token firmato che ci permetterà di poter eseguire qualsiasi altra richiesta.

- **/simulatorRequest** : inzia una nuova simulazione. Per prima cosa controlla se è stato passata l'opzione `-i` o l'opzione `r`: in questo caso viene richiamata la funzione `simulatorRequest` che ci permette di gestire nuove simulazioni. Se non sono state inserite le precedenti opzioni, vuol dire che è stata inserita l'opzione `-d` o `-g`: in questo caso viene chiamata la funzione `simulatorListRequest`. La prima funzione crea un ID basato sulla data e l'ora della richiesta (generando dunque anche la cartella all'interno della directory `simulations` che, per ora, è vuota) e mette in coda la simulazione da eseguire. Se la coda è vuota, allora viene iniziata attraverso la funzione `simulation`, altrimenti

aspetta il suo turno. La seconda funzione, `simulatorListRequest`, aggiunge semplicemente in coda la richiesta senza creare l'ID (dato che non genera alcun risultato non essendo una simulazione in sé). Quando sarà il suo turno verrà gestita grazie alla funzione `listsHandler`.

- **/move-items-queue** : serve per spostare l'ordine delle simulazioni in coda, specificando l'indice di quella da spostare e la posizione in cui deve arrivare.

- **/deleteSimulation** : gestita dalla funzione `deleteSimulationRequest`, si occupa di eliminare utilizzando la funzione `rimraf` la cartella all'interno della directory `simulations` che ha per nome l'ID passato nel body della richiesta.

- **/progressService** : serve per aggiungere un utente alla coda ideata per aggiornare la barra del progresso della simulazione in esecuzione. Sfruttando la tecnica del long polling, il server non comunica la risposta attraverso la funzione `res.send`, che terminerebbe la comunicazione con l'utente, ma sfrutta la funzione `res.write` che la mantiene viva. Durante l'esecuzione di una simulazione, il modulo `process_child` (con il quale abbiamo lanciato il simulatore vero e proprio) ci permette di controllare lo standard output attraverso una callback: in caso venga stampata la stringa `PROGRESS [numero]%`, viene estrapolato il numero e mandato a tutti gli utenti ai quali viene notificato il progresso.

- **/notificationService** : si comporta in modo molto simile alla precedente: ha una sua coda dedicata, in cui vengono aggiunti gli utenti che mandano questa richiesta, e quando una `simulatorListRequest` termina, viene mandata una risposta a tutti gli utenti in coda con un tipo di evento terminato (per esempio, la generazione di una nuova lista di eventi).

Passiamo ora ad analizzare le richieste GET.

- **/getSimulations** : risponde con un array contente tutte le simulazioni in coda, precedute da quella in esecuzione (con il progresso aggiornato).
- **/getResults** : ritorna un array con tutti gli ID (e dunque i nomi delle cartelle) delle simulazioni terminate.
- **/getFinEvent/:id** : permette di eseguire il download, sfruttando la funzione `res.download`, del file `finEventSim.json` (all'interno della cartella `simulations/:id/`) generato alla fine di una simulazione e contenente tutti i risultati utili.

In seguito vengono elencate le richieste per poter utilizzare il simulatore anche da un comune browser. Viene specificato che, per poter accedere alle pagine (escluse quella presente nella radice e quella che mostra le API), bisogna prima effettuare il login.

- **/** : viene restituita una homepage dalla quale è possibile accedere alle diverse pagine fornite.
- **/api** : fornisce una pagina, creata attraverso l'utilizzo di Swagger Editor[5], per poter visualizzare tutte le API fornite dal server.
- **/log** : pagina che permette di effettuare il login e, successivamente, di poter visualizzare le successive pagine.
- **/create** : mostra la pagina con la quale si può creare una nuova lista di eventi inserendo tutte le opzioni viste nella sezione precedente.
- **/simulation** : visualizza la pagina che permette di iniziare una nuova simulazione.
- **/delete** : visualizza la pagina che permette di eliminare una lista di eventi.

[5] `https://editor.swagger.io/`

Capitolo 4

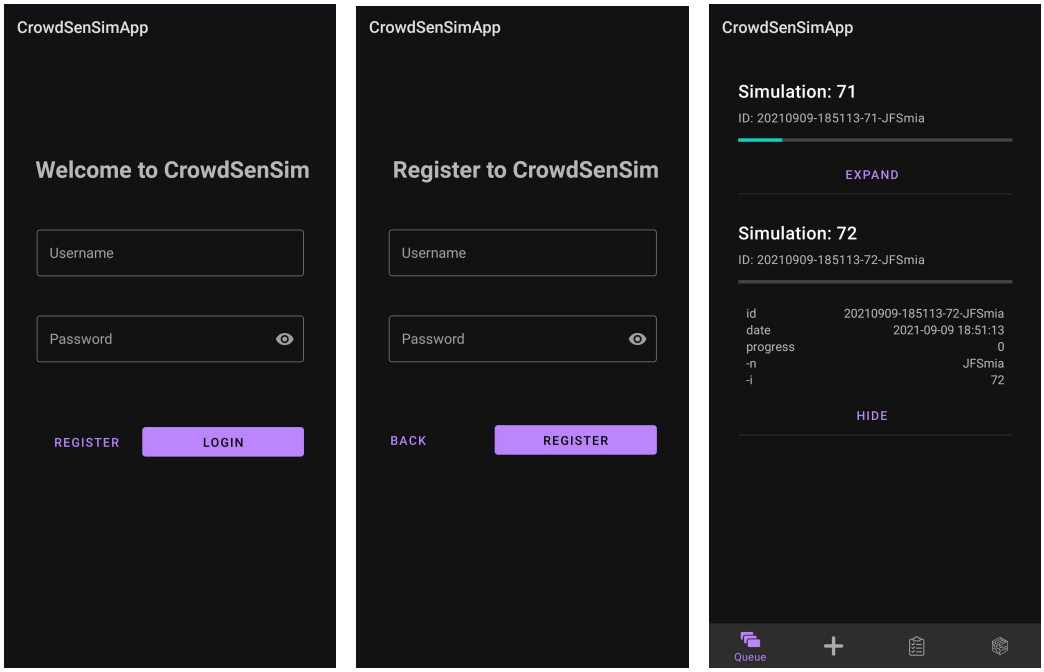
CrowdSenSim App

L'applicazione CrowdSenSim App è stata sviluppata in Java utilizzando Android Studio. La minima versione dell'SDK supportata è la 23, corrispondente ad Android Marshmellow (6.0).

4.1 Panoramica dell'applicazione

All'avvio, l'applicazione CrowdSenSim ci richiede di effettuare il login inserendo userame e password (Figura 4.1a). Nel caso non possedessimo i dati d'accesso, schiacciamo sul tasto *Register* che apre una nuova pagina. Qui possiamo creare un nuovo utente, per poi poter accedere alla homepage dell'app (Figura 4.1b).
Al primo accesso verrà richiesto il permesso di accedere allo storage del dispositivo, utilizzato per salvare i risultati delle simulazioni in locale.
La homepage presenta in basso un *Bottom Navigator* con quattro sezioni: *Queue*, *New simulation*, *List of events* e *Results*.

- **Queue**: in questa sezione (Figura 4.1c) possiamo visualizzare le simulazioni in coda, con una progress bar dinamica che ci mostra l'avanzamento di quella in esecuzione sul server. Ciascuna ha un tasto *Expand*: schiacciandolo, possiamo visualizzare tutte le opzioni inserite durante la fase di creazione. Ciascuna simulazione (tranne quella già



(a) Login (b) Register (c) Queue parte 1

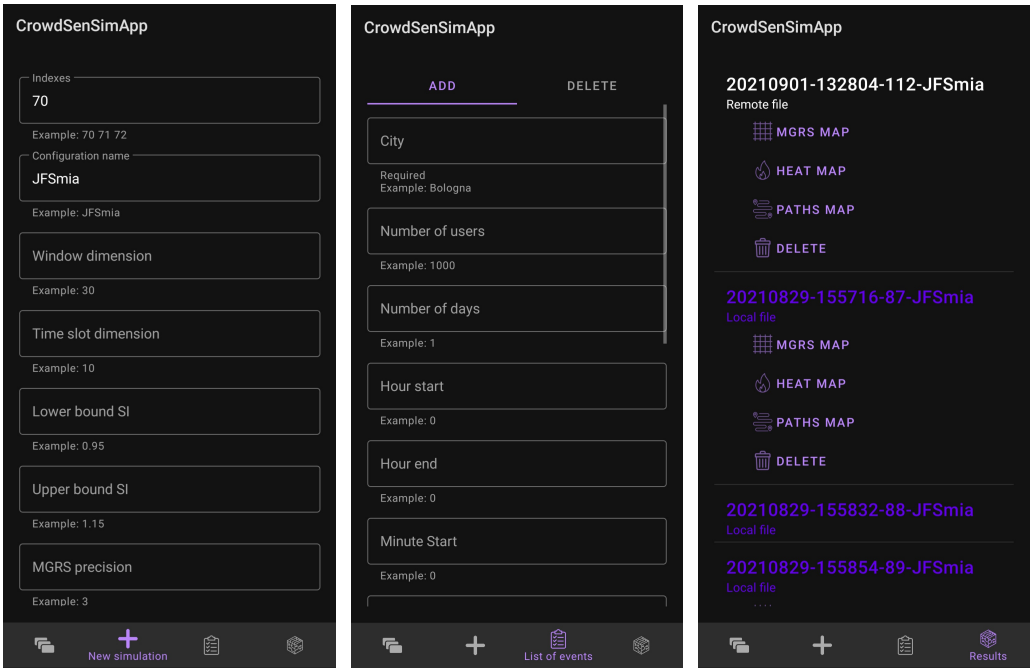
Figura 4.1: App screenshots parte 1

in esecuzione, ovvero la prima) può essere spostata all'interno della coda dinamicamente, cambiando così l'ordine delle simulazioni. Quando una simulazione è terminata, l'utente riceve una notifica. È possibile ottenere aggiornamenti manualmente facendo uno swipe verso il basso.

- **New simulation**: in questa sezione si può creare una nuova simulazione (Figura 4.2a). È richiesto l'indice della lista di eventi da simulare e il nome della configurazione dell'algoritmo. È possibile anche inserire una lista di indici (invece che solo uno) per inserire più simulazioni con le stesse opzioni ma con diversi indici. Tutte le altre opzioni sono facoltative e, se non diversamente specificate, verrano usati i valori di default riportati come esempio sotto ai campi di input.

- **Lists of events**: questa sezione (Figura 4.2b) è divisa in due tab, che

permettono di aggiungere o eliminare una lista di eventi. Nella prima tab, dedicata all'aggiunta di una nuova lista, è necessario inserire soltanto il nome della città, mentre tutte le altre opzioni sono facoltative (verrano usati i campi di esempio come valori di default); nella seconda è necessario inserire l'unico campo richiesto: l'indice della lista da eliminare. Una volta creata o eliminata un lista, l'utente riceverà una notifica.

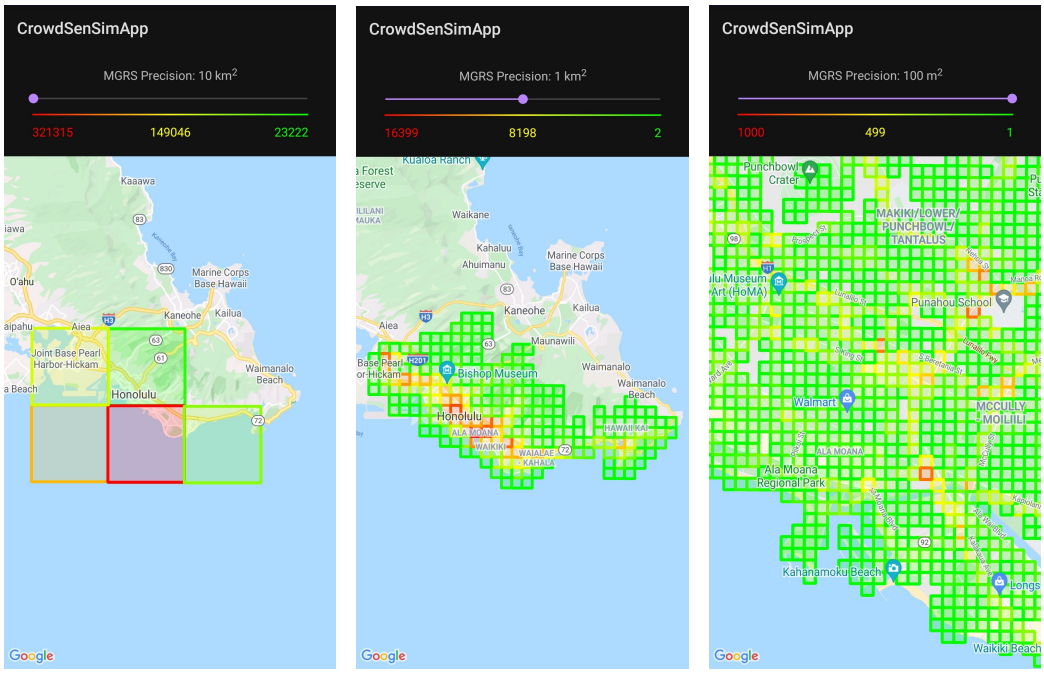


(a) New Simulation (b) List of events (c) Results

Figura 4.2: App screenshots parte 2

- **Results**: in questa sezione si possono visualizzare i risultati delle simulazioni (Figura 4.2c). Le simulazioni con il nome in bianco (e con la scritta *Remote file* appena sotto) sono file sul server. Cliccando su uno dei tasti per visualizzare le diverse mappe, partirà il download del file in locale, nella cartella `Download/CrowdSenSim` della memoria interna (importante: bisogna consentire i permessi di accesso alla memoria). Una volta effettuato il download, facendo un refresh con uno swipe

verso il basso, sarà visibile la stessa simulazione ma in viola, con il testo *Local file* dello stesso colore. Dopo aver ottenuto il file in locale, il comportamento dei tasti per visualizzare le mappe è uguale sia per il risultato remoto, sia per il risultato locale. Il tasto *Delete* invece, elimina il file dal server o dalla memoria del dispositivo in base a quale risultato fa riferimento. I tasti che permettono di visualizzare le mappe portano a delle nuove pagine. Dentro ciascuna di esse, all'inizio, viene visualizzato un dialog. Questo ci informa che i dati stanno venendo elaborati e poi, al termine, vengono visualizzati i risultati.



(a) MGRS precisione $10km^2$ (b) MGRS precisione $1km^2$ (c) MGRS precisione $100m^2$

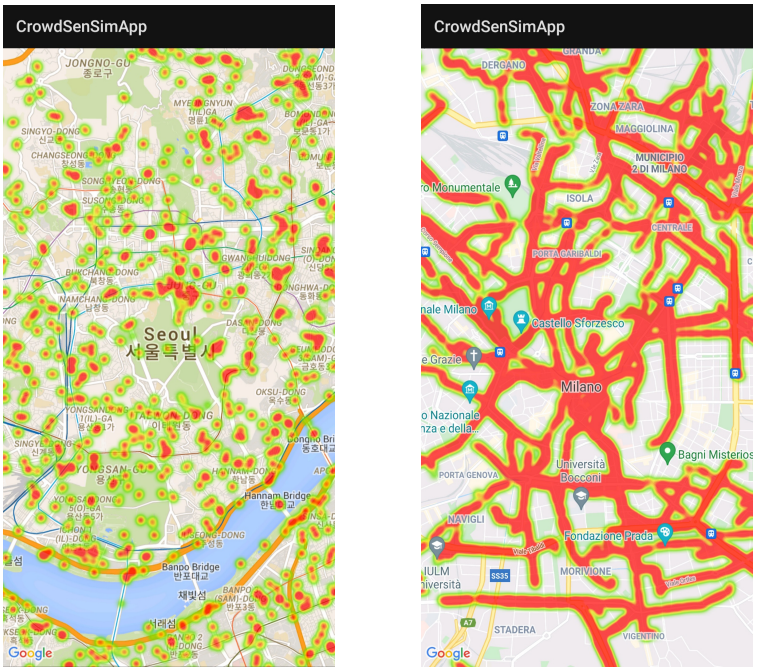
Figura 4.3: App screenshots parte 3

I risultati possono essere visualizzati in tre tipi diversi di mappe:

- **MGRS Map**: questa pagina permette di visualizzare una mappa su cui vengono disegnate aree MGRS. In alto è presente una barra con

la quale si può impostare la precisione delle coordinate. I valori possibili sono: $10km^2$ (Figura 4.3a), $1km^2$ (Figura 4.3b) e $100m^2$ (Figura 4.3c). Le aree avranno diversi colori: più sono rosse, più eventi sono avvenuti all'interno di quell'area. Appena sotto possiamo vedere una barra colorata che ci indica, al cambiare della precisione, a quanti eventi corrispondono i diversi colori.

- **Heatmap**: questa pagina permette di visualizzare una heatmap degli eventi su una mappa (Figure 4.4a e 4.4b).

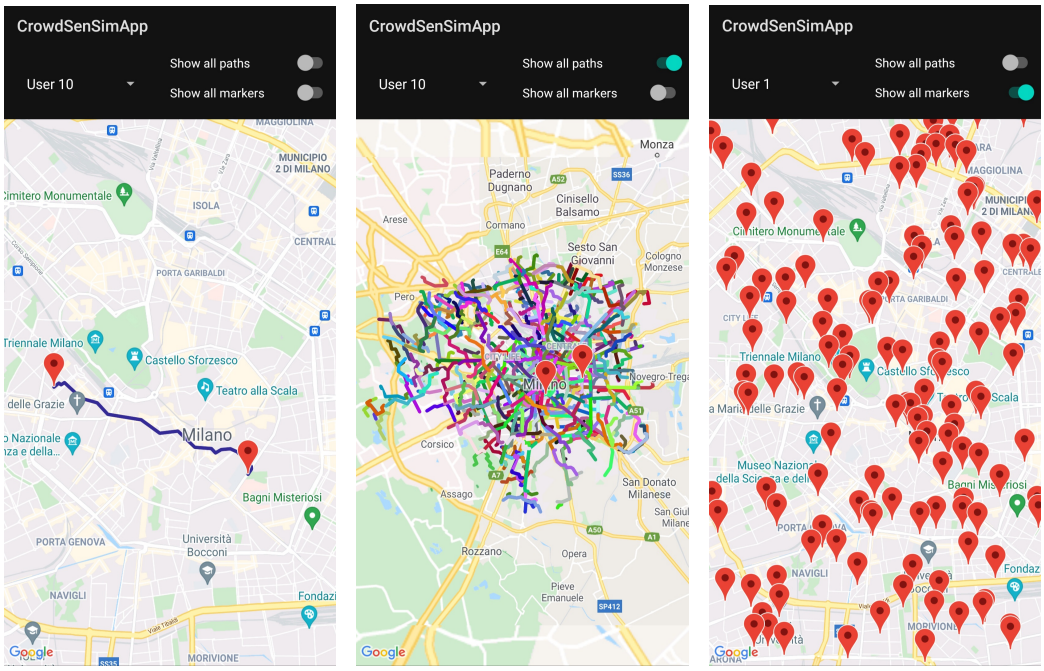


(a) Heatmap parte 1 (b) Heatmap parte 2

Figura 4.4: App screenshots parte 4

- **Paths Map**: questa pagina permette di visualizzare una mappa su cui vengono riportati tutti i percorsi degli utenti con due marker per segnare l'inizio e la fine di esso (Figura 4.5a). In alto vi è un selettore dal quale si può selezionare ogni singolo utente e visualizzare il suo percorso. Di fianco ad esso ci sono due switch: il primo permette

di visualizzare tutti i percorsi degli utenti (Figura 4.5b) e il secondo permette di disegnare tutti i marker di tutti i percorsi (Figura 4.5c).



(a) Paths Map parte 1 (b) Paths Map parte 2 (c) Paths Map parte 3

Figura 4.5: App screenshots parte 5

4.2 Dettagli di implementazione

In questa sezione vediamo le varie classi Java che compongono l'app.
La prima classe presentata è **Requester**: ciascuna richiesta al server, fatta eccezione per la richiesta di login e di registrazione di un nuovo utente, ha bisogno di un *access token* che il server ci fornisce al momento del login. Questo dato viene salvato nelle *Shared Preferences* e aggiunto in automatico a ogni richiesta dalla classe statica Requester che si occupa di esse. Questa utilizza la libreria ***OkHttp*** per mandare richieste al server. Ha un metodo *get* e un metodo *post* che permettono di eseguire le diverse richieste. Per tutte le ri-

chieste *post* si utilizza la tecnica del **long polling** che fa sembrare l'app una real-time. Questa tecnica prevede che il client faccia una richiesta normale al server sapendo però che questo potrebbe non rispondere immediatamente. Se il server non ha alcuna nuova informazione da mandare al client, mantiene aperta la richiesta e aspetta che l'informazione diventi disponibile. Quando succede risponde al client, che spesso, quando riceve la risposta, manda immediatamente una nuova richiesta al server ricominciando il ciclo. Inoltre il server può rispondere con una risposta "parziale", non completa, e dunque la connessione viene chiusa soltanto quando l'informazione è stata mandata nella sua totalità. Per poter ricevere riposte *chunked* il client OkHttp utilizza un *Interceptor*, che permette di analizzare risposte parziali e agire di conseguenza. Tutte le richieste *get*, invece, non hanno questo problema e non si appoggiano all'Interceptor.

FilesIO è un'altra classe statica. Questa permette di gestire i file dello storage del dispositivo. Permette di leggere e scrivere file, oltre che controllare se esistano già in memoria. In più gestisce anche il download dei file remoti con il ***Download Manager*** di Android. È utilizzata soprattutto nella sezione *Results* dell'app.

MonitorNewSimulationService è il servizio in foreground (dunque è visibile una sua notifica nello Status Bar) che si abilita all'entrata dell'applicazione. Permette di visualizzare una notifica quando una simulazione è terminata. È necessario usare un service dal momento che, senza di esso, una volta usciti dall'applicazione le notifiche non verrebbero visualizzate.

La sezione *Queue* è implementata attraverso **QueueFragment**, che contiene una recycler view. Vi sono dunque la ***QueueAdapter*** e un generico ***QueueItem*** per poter popolare e visualizzare la coda. La recycler view inoltre ha un ***ItemTouchHelper*** per permettere il riordinamento degli elementi all'interno di essa, tenendo premuto su una simulazione e rilasciandola nella posizione desiderata. Ovviamente non si può spostare la prima simulazione (perché è quella in esecuzione sul server) e non si può mettere una simulazione prima della prima.

La sezione *Results* è stata implementata in modo molto simile alla *Queue*, sfruttando la classe **ResultsFragment**. Adotta anch'essa una recycler view, dunque possiede un ***ResultsAdapter*** e un ***ResultsItem*** generico, ma non necessita dell'*ItemTouchHelper*.
La sezione *New Simulation* è stata realizzata con la classe **NewSimulationFragment**. Questa si occupa di prendere in input tutte le opzioni per la creazione di una nuova simulazione e di inviare la richiesta al server. Nel caso le richieste siano molteplici (inserendo all'interno del campo *Index* più indici) vengono lanciate diverse richieste separate con uno sleep per separarle. Questo per preservare l'ordine degli indici inseriti nonostante le richieste asincrone. Una volta arrivata/e la/e richiesta/e, il fragment ci ridireziona nel *QueueFragment*, ripristinando la vecchia coda (se c'erano già elementi) e aggiungendo quelle appena create attraverso un *Bundle*.
Per realizzare la sezione *List of events* è stata implementata la classe **EventsFragment** che contiene un ***TabLayoutMediator***. Questo crea le due tab (*Add* e *Delete*) e uno ***ScreenSlidePagerAdapter*** per permettere lo swipe orizzontale tra le due pagine. **AddListEventsFragment** e **DeleteListEventsFragment**, le classi delle due tab, sono molto simili alla classe *NewSimulationFragment* e si comportano più o meno nello stesso modo. Una volta creata o eliminata una lista, infatti, non si viene riportati nella sezione *Queue* ma viene mostrata una notifica dell'effettivo completamento dell'azione desiderata.
MapActivity è l'activity che permette di visualizzare i risultati su una mappa. In alto contiene un *FragmentContainer* vuoto, che verrà poi rimpiazzato da un *CommandsFragment* specifico (vedi in seguito) in base a quale bottone nella sezione *Results* viene premuto. Questo avviene grazie a uno switch, che analizza la stringa contenuta negli *Extra* dell'activity chiamata *mapType*. Sotto, ovviamente, vi è la mappa vuota. Per quanto riguarda i tipi di mappe specifiche, ciascuna ha la seguente struttura:

- Un **CommandsFragment**, la parte che viene visualizzata al di sopra della mappa e ci permette di eseguire azioni dinamiche su di essa. Que-

sto implementa la *OnMapReadyCallBack*, che viene chiamata quando la mappa è pronta per essere utilizzata.

- Un **Helper**, che si occupa di leggere i dati da visualizzare sulla mappa, prendendoli dal file locale, e di creare le strutture dati necessarie per visualizzarli correttamente. Per popolare le strutture dati, viene utilizzato l'oggetto ***JsonReader*** che permette di leggere stringhe JSON come flusso di token. Questo è particolarmente utile quando bisogna leggere file di grosse dimensioni, dato che se si utilizzasse l'oggetto ***JsonObject***, questo lo leggerebbe interamente riducendo le performance.

Passiamo ora a vedere come le varie mappe vengono realizzate. Di seguito è indicato un esempio di come è formato un file locale per fare più chiarezza nelle spiegazioni successive.

```
{
    "numberUsers": 1000,
    "paths": [
        "userId": "1",
        "lat": "44.91615",
        "long": "11.14576",
        "mgrs": "32TPQ6936575876"
    ]
}
```

La **HeatMap** contiene l'**HeatmapHelper** che legge semplicemente tutte le coordinate (indipendentemente dal numero dell'utente) e le aggiunge in una lista di coordinate. Una volta terminato, con essa crea un *HeatmapTileProvider*, che l'**HeatmapCommandsFragment** provvederà a impostare sulla mappa.
La **PathsMap** contiene il **PathsMapCommandsFragment** che per prima cosa legge dal file locale il numero di utenti, popolando così, con un *ArrayAdapter* di tipo stringa, lo spinner da cui si potrà selezionare ciascuno di

essi. Viene poi inizializzata la mappa con il **PathsMapHelper**. Questo ha diversi array: uno per le polyline (le strade che gli utenti percorrono), uno per le opzioni di visualizzazione di ciascuna polyline e uno in cui, in ciascuna posizione, vi è una coppia di Marker (che definiscono l'inizio e la fine del percorso). Ciclando sull'array *paths* dell'oggetto JSON, viene aggiunta in posizione $userId - 1$ (dato che il primo user ha come id 1, ma gli array iniziano dalla posizione 0) dell'array *polylineOptions* un nuovo oggetto LatLng (prendendo i dati *lat* e *long*) grazie al metodo *add* degli oggetti *PolylineOptions*. Una volta terminata la creazione delle opzioni crea per ciascuna un cammino, assegnandogli un colore random, e i relativi markers prendendo la prima e l'ultima coordinata (dato che gli eventi sono ordinati in tempo cronologico). Al cambio dell'utente con lo spinner, il *CommandsFragment* richiama il metodo *showPolyline* dell'*Helper* che si occupa di nascondere il sentiero visualizzato con i relativi markers e di mostrare quello desiderato dall'utente. Può inoltre richiamare i metodi *showAllPolylines* e *showAllMarkers* (e i rispettivi *removeAllPolylines* e *removeAllMarkers*) che permettono di mostrare (e rimuovere) tutti i percorsi degli utenti.

La **MGRSMap** contiene l'**MGRSCommandsFragment** che genera una seekbar con la quale è possibile impostare il livello di precisione delle coordinate. L'**MGRSMapHelper** contiene una *HashMap* per ciascun livello di precisione, così come una lista di oggetti *Polygon* per ciascuno. Le HashMap hanno come chiave la coordinata MGRS e come valore il numero di occorrenze della stessa. Ciclando sull'array *paths* dell'oggetto JSON, viene presa la coordinata MGRS e aggiunta di conseguenza a ciascuna delle precisioni. Se la HashMap la contiene già viene incrementato il suo valore, altrimenti la si inserisce normalmente con valore 1. Una volta terminato il ciclo, vengono generate le aree usando le coordinate MGRS (le HashMap ci assicurano che non esiste nessun doppione). Le aree vengono create usando la seguente tecnica: conoscendo la coordinata MGRS, questa viene convertita in un oggetto LatLng. La precisione, inoltre, fornisce il lato del quadrato dell'area. Dividendola per due, otteniamo un raggio. Viene quindi chiamato il meto-

do *getBoundsFromCenter* che ci calcola, utilizzando il centro e il raggio, le coordinate Nord-Est e Sud-Ovest, con le quali possiamo ricavare le due mancanti. A ciascuna area viene inoltre assegnato un colore dal rosso al verde. Più è rosso, più eventi sono avvenuti in quell'area. Un metodo semplice per assegnare il colore giusto è sfruttare il sistema di colori HSV (Hue Saturation Value), nel quale il verde ha una tonalità (hue) di 120 e il rosso un hue di 0. Assegnamo a $maxColorValue$ il numero più elevato di occorrenze di eventi in un livello di precisione, per esempio 100. Se dobbiamo aggiungere un'area con un valore *value* 100, questa deve avere il colore rosso (ovvero hue 0) dato che ha un numero di eventi molto alto (il massimo possibile in questo esempio). Costruiamo la seguente semplice proporzione per trovare il valore x, ovvero la tonalità da assegnare all'area:

$$value : maxColorValue = x : MaxHueValue$$

Dove $MaxHueValue$ è 120 (il verde, dato che è il valore hue massimo che vogliamo raggiungere). Dunque:

$$100 : 100 = x : 120$$

$$x = 120$$

Ma x dovrebbe essere 0, dal momento che lo hue del rosso è 0. Dunque assegnamo a x il valore $MaxValueHue - x$.
L'*MGRSCommandsFragment*, sfruttando l'*OnSeekBarChangeListener*, richiama il metodo *showPolygons* dell'*Helper* che si occupa di nascondere le aree della precedente precisione e di mostrare quelle desiderate dall'utente. Richiama inoltre i metodi *getMaxValue* e *getMinValue* per aggiornare i numeri visualizzati sotto la barra colorata.
Per trasformare una coordinata MGRS in un oggetto LatLng viene utilizzato il metodo *latLonFromMgrs*. Questo metodo fa parte di una libreria open-source sviluppata dalla NASA chiamata WorldWind[1].

[1] `https://worldwind.arc.nasa.gov/`

Capitolo 5

Risultati

Questo capitolo si focalizza sull'analizzare le performance in termini di tempo del simulatore e dell'applicazione sviluppata. Vengono mostrati grafici che mostrano il tempo trascorso dalla primissima operazione possibile, ovvero il download della mappa di una città, fino all'ultima operazione eseguita, ovvero la visualizzazione dei risultati. L'app è stata lanciata su uno smartphone Samsung S9+. Tutti i grafici mostrati di seguito sono stati realizzati sfruttando la libreria Javascript AmCharts[1]. Ciascun valore riportato è il risultato della media di tre misurazioni diverse per la stessa fase della simulazione.

5.1 Variazione del numero di utenti e delle città

In questa sezione osserviamo il variare dei tempi trascorsi al variare del numero di utenti partecipanti alla simulazione.
Sono stati utilizzati i seguenti colori: il verde si riferisce a una simulazione con un tempo di viaggio di 10 minuti, l'azzurro è relativo a un tempo di viaggio della durata di 20 minuti, il viola si riferisce a un tempo di viaggio di 30 minuti. Ciascun colore ha quattro diversi valori del campo *Value* del sistema

[1] `https://www.amcharts.com/javascript-charts/`

di colorazione HSV, che permette di ottenere una variazione via via più scura di esso. Ciascuna variazione indica una precisa fase della simulazione: la variante più chiara si riferisce al tempo di download della mappa (che ha lo stesso valore per tutti e tre i colori, dal momento che la variazione del tempo di viaggio non influisce sul tempo di scaricamento della mappa della città); la variazione leggermente più scura si riferisce alla fase di generazione della lista di eventi; la successiva riporta i tempi relativi alla simulazione vera e propria; la variazione più scura si riferisce ai tempi di visualizzazione dei risultati. Il tempo di viaggio è impostabile durante la creazione della lista di eventi inserendo entrambe le opzioni `--max-travel-time` e `--min-travel-time` con lo stesso valore desiderato. Tutte queste caratteristiche vengono osservate al variare degli utenti nei seguenti valori: 100, 200, 500, 1000, 2000, 5000, 10000. Si specifica inoltre che nei due grafici successivi (Figure 5.1 e 5.2) i tempi di visualizzazione si riferiscono esclusivamente alla visualizzazione su una Paths Map, ovvero la mappa che mostra i cammini degli utenti, perché è quella più onerosa da dover visualizzare (in termini di tempo). Analizziamo la variazione dei tempi al cambiare del tipo di mappa nella prossima sezione.

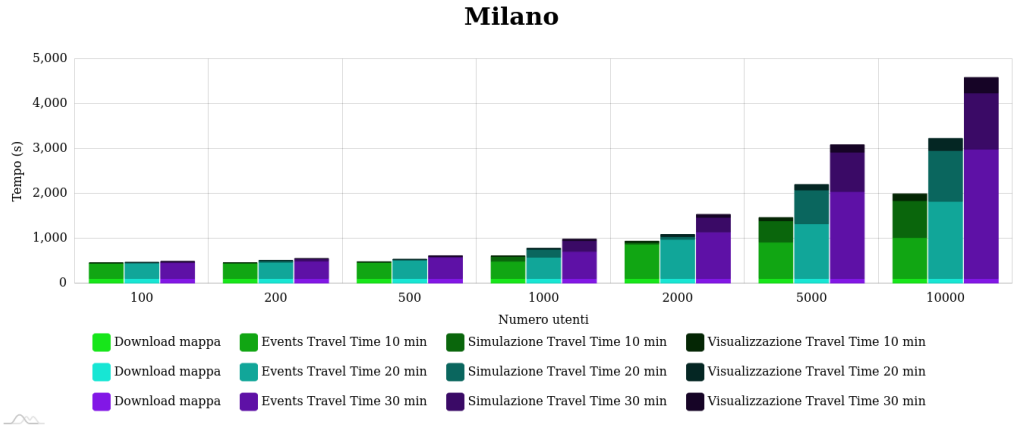


Figura 5.1: Tempi al variare degli utenti per Milano

Indichiamo con $T_{U,t}$ il tempo totale di simulazione con un numero U di utenti fissati e un travel time fissato t. La prima supposizione che potrebbe

sembrare ovvia è quella che, aumentando il numero di utenti, il tempo totale di simulazione aumenti. Osservando la Figura 5.1 concludiamo che la supposizione è vera, anche se la differenza di tempo totale di simulazione da 100 utenti a 200 (considerando il travel time di 10 minuti) è relativamente bassa: $T_{200,10} - T_{100,10} = 3s$. La differenza aumenta all'aumentare del travel time, infatti $T_{200,20} - T_{100,20} = 43s$ e $T_{200,30} - T_{100,30} = 66s$. La seconda supposizione abbastanza ovvia è che, fissando un numero di utenti U, il tempo totale di simulazione aumenti all'aumentare del travel time. Anche questa risulta vera. Il caso meno visibile sulla figura è quello con 100 utenti: $T_{100,20} - T_{100,10} = 16s$ e $T_{100,30} - T_{100,20} = 20s$.
Chiamiamo la differenza dei tempi totali di simulazione al variare del tempo di viaggio da 10 minuti a 20 $\Delta_{U,1}$ e la differenza dei tempi totali al variare del travel time da 20 minuti a 30 $\Delta_{U,2}$. Si può notare come, all'aumentare di U, $\Delta_{U,1}$ e $\Delta_{U,2}$ incrementino quasi proporzionalmente in modo abbastanza intuitivo.

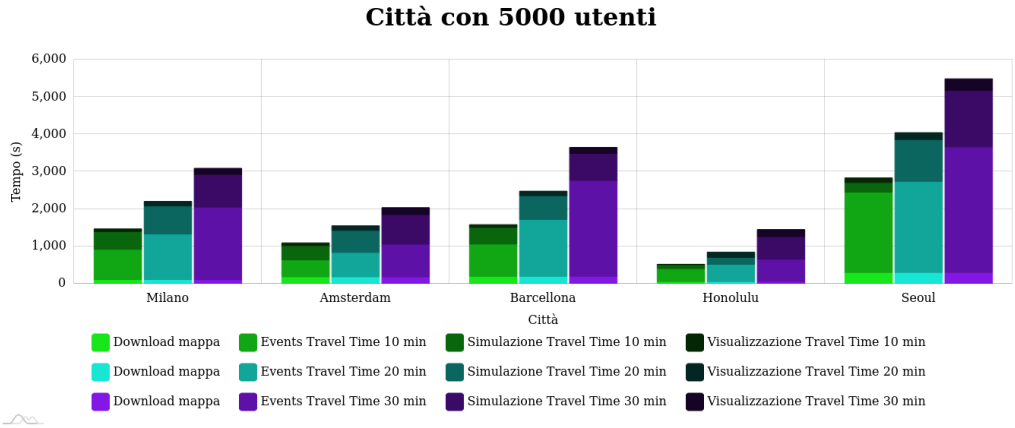


Figura 5.2: Tempi al variare della città con 5000 utenti ciascuna

Facendo riferimento alla Figura 5.2, tutte le osservazioni precedenti (escluse ovviamente quelle inenerenti al variare del numero degli utenti) sembrano valere anche per le altre città prese in analisi, ovvero Amsterdam (Paesi Bassi), Barcellona (Spagna), Honolulu (Hawaii) e Seoul (Corea del Sud). Possiamo

affermare che in generale la fase più onerosa a livello di tempo sia la seconda, ovvero la generazione della lista di eventi.
La mappa che ha richiesto più tempo per essere scaricata è stata quella di Seoul. Questo risultato era abbastanza prevedibile dal momento che si espande su un'area di 605,2 km^2 contro, ad esempio, i 181,8 km^2 di Milano.

Città	Superficie (km^2)	Numero nodi	Numero archi	Tempo (s)
Milano	181,8	183.321	423.700	111
Amsterdam	219,3	108.465	246.184	185
Barcellona	101,4	185.769	451.914	198
Honolulu	177,2	56.444	121.224	58
Seoul	605,2	340.969	794.220	302

Tabella 5.1: Città, superficie, numero di archi e nodi del grafo e tempo di download

Meno prevedibile era il tempo di download della mappa di Honolulu rispetto alle città europee. Come si può vedere dalla Tabella 5.1, la sua superficie è estesa quasi quanto quella di Milano, ma il suo tempo di scaricamento è circa la metà di quello della città italiana. Possiamo dire che generalmente il tempo di download dipende dal numero di nodi e di archi presenti nel grafo che rappresenta la città, anche se non proporzionalmente. Notiamo infatti che, nonostante Barcellona abbia un numero di nodi e di archi leggermente superiore a quello di Milano, per scaricare il grafo della città catalana ci sono voluti 87 secondi in più, mentre la differenza con il tempo di scaricamento del grafo di Amsterdam è molto minore (13 secondi) nonostante la differenza di nodi e di archi è molto più marcata (addirittura Barcellona ha quasi il doppio del numero di archi della città olandese).
La città con il tempo di generazione della lista degli eventi più alto, fissato a 5000 il numero di utenti e a 30 minuti il travel time, è Seoul con 3363 secondi (cioè quasi 55 minuti), seguita da Barcellona con 2567 secondi (quasi 42 minuti).

5.2 Visualizzazione dei risultati

In questa sezione vediamo la differenza dei tempi di visualizzazione al variare del tipo di mappa.
I colori scelti sono i seguenti: il rosso cremisi per la visualizzazione della mappa che mostra le aree MGRS, il rosa rubino per la heatmap e il verde filippino per la mappa che mostra i sentieri. Ciascun colore ha tre diverse varianti, ottenute usando la stessa tecnica precedente, per i tre diversi tempi di viaggio presi in analisi: più il colore è scuro, più è lungo il travel time.

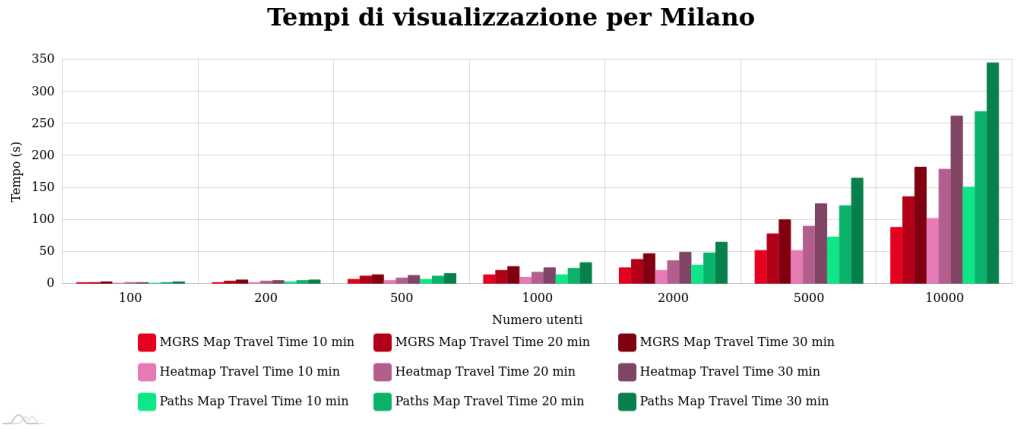


Figura 5.3: Tempi di visualizzazione al variare del tipo di mappa su Milano

Chiamiamo $M_{U,T}$, $H_{U,T}$ e $P_{U,T}$ rispettivamente il tempo di visualizzazione della mappa MGRS, Heatmap e Paths per un numero di utenti fissato U e un travel time, anch'esso fissato, T. Possiamo notare osservando la Figura 5.3 come, fissato un U, $M_{U,30} < H_{U,30} < P_{U,30}$.
La stessa relazione non vale per un tempo di viaggio di 20 minuti o di 10. Infatti se fissiamo u a 5000 o 10000 abbiamo che $M_{u,20} < H_{u,20} < P_{u,20}$ e $M_{u,10} < H_{u,10} < P_{u,10}$, mentre per tutti gli altri casi (u uguale a 100, 200, 500, 1000 o 2000) abbiamo che $H_{u,20} \leq M_{u,20} < P_{u,20}$ e $M_{u,10} \leq H_{u,10} < P_{u,10}$.

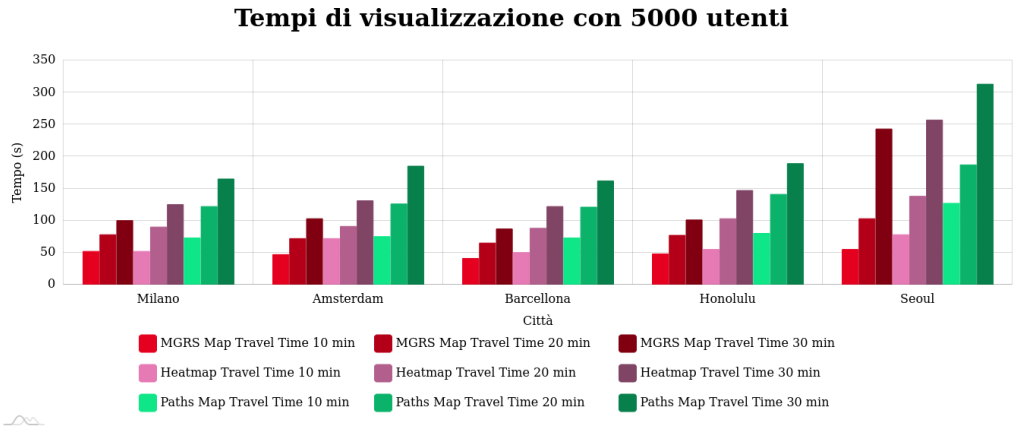


Figura 5.4: Tempi di visualizzazione al variare del tipo di mappa e alla città con 5000 utenti

Anche fissando gli utenti a un numero di 5000 e cambiando la città le considerazioni precedenti continuano a valere, come si può notare dalla Figura 5.4. Per visualizzare la Paths Map di Seoul, la città che richiede più tempo con 5000 utenti (considerano il travel time da 30 minuti), ci sono voluti 313 secondi (un po' più di 5 minuti).
È importante sottolineare che, in realtà, il tempo per la visualizzazione della mappa MGRS aumenterebbe di molto se si volessero mostrare le aree con tutti i livelli di precisione. È stato scelto infatti di mostrare soltanto i tre livelli di precisione "meno precisi", evitando di poter visualizzare le aree con il lato di 10 e 1 metro, anche perché ritenute poco significative.
La simulazione più lunga è stata quella avvenuta a Seoul con 5000 utenti e un travel time di 30 minuti per un totale di 5488 secondi, ovvero 1 ora e 31 minuti, suddivisi nel modo seguente: 302 secondi per il download della mappa (quasi 5 minuti); 3363 secondi (cioè quasi 55 minuti) per la generazione della lista degli eventi; 1510 secondi (circa 24 minuti) per la simulazione e 313 secondi (o quasi 5 minuti) per la visualizzazione della Paths Map. Il file JSON contenente i risultati di questa simulazione pesa 116,8 MB e occupa un totale di 4586123 linee. In realtà, il file relativo alla simulazione avvenuta a

Milano con 10000 utenti e con un tempo di viaggio di 30 minuti è più grande. Questo, infatti, pesa 196,8 MB e conta 7826309 linee, ma la sua simulazione è durata 4590 secondi (quasi 1 ora e 16 minuti). Da ciò possiamo concludere che il tempo totale di simulazione non dipende esclusivamente dal numero di utenti, ma anche e soprattutto dalla complessità del grafo della città scaricato con `OSMnx`.

Conclusioni

In questo lavoro è stato presentato CrowdSenSim 2.0, il simulatore stateful per lo sviluppo di sistemi MCS in ambienti urbani realistici. Consente l'osservazione di fenomeni come il consumo energetico e la raccolta dati, di generare mobilità degli utenti e offre vari algoritmi di raccolta dati. È stata descritta la sua architettura, composta da moduli scritti in C++ e in Python. Sono state poi elencate le modifiche apportate al simulatore: la possibilità di utilizzare completamente il simulatore via linea di comando, di poter compilare il codice del Simulator Engine grazie all'utility `make` e permettere il riutilizzo delle mappe delle città precedentemente scaricate durante la fase di generazione della lista di eventi. In seguito è stato presentato il web server Node, che permette di utilizzare CrowdSenSim 2.0 da remoto, e la sua struttura. È possibile utilizzare un comune browser per poter interagire con il simulatore grazie a una semplicissima web app. Questa sfrutta le API messe a disposizione dal server, le quali vengono elencate e descritte. È stata presenta poi CrowdSenSim App, l'applicazione per dispositivi mobili Android che permette di utilizzare il simulatore senza sapere come funziona in maniera user-friendly. Sfruttando le API messe a disposizione dal server, è possibile creare una nuova lista di eventi, far partire una nuova simulazione, eliminare una lista di eventi precedentemente creata e visualizzare i risultati su tre tipi di mappe diversi: la mappa che mostra i cammini percorsi da ciascun partecipante, la mappa che mostra una heatmap degli eventi e una mappa che mostra aree MGRS con una diversa colorazione in base al numero di eventi avvenuti all'interno dell'area. Successivamente vengono mostrati i

risultati delle simulazioni e le performance in termini di tempo dell'intera routine di simulazione. Vengono analizzati i tempi di download della mappa di una città, di generazione della lista di eventi, della simulazione vera e propria e della visualizzazione dei risultati su mappa. Per quest'ultima fase viene fatta un'analisi a parte, focalizzandosi su tutti e tre i diversi tipi di mappe.

Per visualizzare la mappa dei cammini (quella che richiede più tempo) con 5000 partecipanti e un tempo di viaggio di 30 minuti, facendo la media tra le cinque città prese in considerazione, ci vogliono 202,8 secondi, un po' più 3 minuti. La simulazione analizzata più onerosa, in termini di tempo, è stata quella avvenuta a Seoul con 5000 utenti e con un tempo di viaggio di 30 minuti, per un totale di 1 ora e 31 minuti.

Bibliografia

[1] Gilles Adda, Sebastian Stüker, Martine Adda-Decker, Odette Ambouroue, Laurent Besacier, David Blachon, Hélene Bonneau-Maynard, Pierre Godard, Fatima Hamlaoui, Dmitry Idiatov, et al. Breaking the unwritten language barrier: The bulb project. *Procedia Computer Science*, 81:8–14, 2016.

[2] Georgina Kate Barratt, Clint Bellenger, Eileen Yule Robertson, Jason Lane, and Robert George Crowther. Validation of plantar pressure and reaction force measured by moticon pressure sensor insoles on a concept2 rowing ergometer. *Sensors*, 21(7):2418, 2021.

[3] Felix Beierle, Vinh Thuy Tran, Mathias Allemand, Patrick Neff, Winfried Schlee, Thomas Probst, Rüdiger Pryss, and Johannes Zimmermann. Tydr–track your daily routine. android app for tracking smartphone sensor and usage data. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 72–75. IEEE, 2018.

[4] David Blachon, Elodie Gauthier, Laurent Besacier, Guy-Noël Kouarata, Martine Adda-Decker, and Annie Rialland. Parallel speech collection for under-resourced language studies using the lig-aikuma mobile device app. *Procedia Computer Science*, 81:61–66, 2016.

[5] Andrea Capponi, Claudio Fiandrino, Burak Kantarci, Luca Foschini, Dzmitry Kliazovich, and Pascal Bouvry. A survey on mobile cro-

wdsensing systems: Challenges, solutions, and opportunities. *IEEE communications surveys & tutorials*, 21(3):2419–2465, 2019.

[6] Claudio Fiandrino, Andrea Capponi, Giuseppe Cacciatore, Dzmitry Kliazovich, Ulrich Sorger, Pascal Bouvry, Burak Kantarci, Fabrizio Granelli, and Stefano Giordano. Crowdsensim: a simulation platform for mobile crowdsensing in realistic urban environments. *Ieee access*, 5:3490–3503, 2017.

[7] Raghu K Ganti, Fan Ye, and Hui Lei. Mobile crowdsensing: current state and future challenges. *IEEE communications Magazine*, 49(11):32–39, 2011.

[8] Bin Guo, Zhiwen Yu, Xingshe Zhou, and Daqing Zhang. From participatory sensing to mobile crowd sensing. In *2014 IEEE International Conference on Pervasive Computing and Communication Workshops (PERCOM WORKSHOPS)*, pages 593–598, 2014.

[9] S Imre, Cs Keszei, D Hollós, P Barta, and Cs Kujbus. Simulation environment for ad-hoc networks in omnet+.

[10] Michael Klein. *Dianemu: A java based generic simulation environment for distributed protocols*. Universität Karlsruhe, Fakultät für Informatik, 2003.

[11] Kamal Mehdi, Massinissa Lounis, Ahcène Bounceur, and Tahar Kechadi. Cupcarbon: A multi-agent and discrete event wireless sensor network design and simulation tool. In *7th International ICST Conference on Simulation Tools and Techniques, Lisbon, Portugal, 17-19 March 2014*, pages 126–131. Institute for Computer Science, Social Informatics and Telecommunications, 2014.

[12] Prashanth Mohan, Venkata N Padmanabhan, and Ramachandran Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile

smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 323–336, 2008.

[13] Federico Montori, Luca Bedogni, Claudio Fiandrino, Andrea Capponi, and Luciano Bononi. Performance evaluation of hybrid crowdsensing systems with stateful crowdsensim 2.0 simulator. *Computer Communications*, 161:225–237, 2020.

[14] Federico Montori, Emanuele Cortesi, Luca Bedogni, Andrea Capponi, Claudio Fiandrino, and Luciano Bononi. Crowdsensim 2.0: A stateful simulation platform for mobile crowdsensing in smart cities. In *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 289–296, 2019.

[15] Sarfraz Nawaz, Christos Efstratiou, and Cecilia Mascolo. Parksense: A smartphone based sensing system for on-street parking. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 75–86, 2013.

[16] Vijay Sivaraman, James Carrapetta, Ke Hu, and Blanca Gallego Luxan. Hazewatch: A participatory sensor system for monitoring air pollution in sydney. In *38th Annual IEEE Conference on Local Computer Networks-Workshops*, pages 56–64. IEEE, 2013.

[17] Cristian Tanas and Jordi Herrera-Joancomartí. Crowdsensing simulation using ns-3. In Jordi Nin and Daniel Villatoro, editors, *Citizen in Sensor Networks*, pages 47–58, Cham, 2014. Springer International Publishing.

[18] Carsten Vogel, Rüdiger Pryss, Johannes Schobel, Winfried Schlee, and Felix Beierle. Developing apps for researching the covid-19 pandemic with the trackyourhealth platform. *arXiv preprint arXiv:2103.13954*, 2021.

[19] Xinglin Zhang, Zheng Yang, Wei Sun, Yunhao Liu, Shaohua Tang, Kai Xing, and Xufei Mao. Incentives for mobile crowd sensing: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):54–67, 2015.

Ringraziamenti

Un grazie dal profondo del cuore va alla mia famiglia, senza la quale non avrei nemmeno potuto cominciare questo percorso e a cui sono totalmente riconoscente. Condivido con mia madre, mio padre e mio fratello questo traguardo e la mia felicità.
Sono fiero di ciò che sono e lo devo anche a loro.
Ringrazio inoltre il relatore Dott. Federico Montori, che mi ha accompagnato e guidato nella realizzazione di questo elaborato e per la grande disponibilità dimostratami in questi mesi di lavoro.