Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

WISHLIST: PROGETTAZIONE ED IMPLEMENTAZIONE DI UNA PIATTAFORMA DI MOBILE SOCIAL NETWORK PER DISPOSITIVI ANDROID

Relatore:
Chiar.mo Prof.
MARCO DI FELICE

Presentata da:
LORENZO GIGLI

Sessione III
Anno Accademico 2013/2014

Ai miei genitori
ai miei fratelli Stefano e Antonio
ai miei nonni (anche quelli che non ci sono più)

Introduzione

A: Che bella borsa che hai! Dove l'hai presa?
B: Non lo so, me l'hanno regalata C e D. È piaciuta molto anche a tua sorella
A: Potrei regalargliela, visto che tra poco compie gli anni
B: Perchè non chiami C e glielo chiedi?
...telefonata...
A: Loro l'hanno presa da PQR, ma ha detto D che ce l'hanno anche da RST
B: Aspetta, da RST hanno le svendite, ti conviene guardare lì
A: Ok, grazie. Allora ci vado a vedere

Situazioni simili a quella realisticamente rappresentata nel *dialogo*, sono state da tutti noi più volte sperimentate. La constatazione di questa diffusa consuetudine a condividere le informazioni su un oggetto che si vorrebbe ricevere, regalare o acquistare per sè, ha fornito l'idea per la realizzazione di questo progetto: trasferire tutto ciò su una applicazione, la cui naturale vocazione appare quella di attualizzarsi nella tipologia "mobile social network". Pertanto il mio lavoro è consistito nella realizzazione di una piattaforma social per creare, gestire, condividere un insieme di liste organizzate di oggetti che l'utente desidera acquistare o ricevere: nasce così Wishlist.

Si tratta di un'applicazione complessa che presenta diverse peculiarità non riscontrabili (quanto meno tutte insieme) su altri prodotti di questo genere presenti sul mercato.

Dall'analisi dei comportamenti di blogger e commentatori tecnici sui social network "vetrina" emerge una frequente tendenza ad utilizzarli in modi

per i quali non offrono servizi specifici. La condivisione di un oggetto è una delle pratiche più consuete, il che dimostra un diffuso interesse verso questo argomento e, al tempo stesso, la mancanza di uno strumento completo ed efficiente per questo scopo: difficilmente si va oltre la pubblicazione di un url o di una singola immagine. Da queste premesse nasce il progetto di costruire uno strumento finalizzato specificamente alla condivisione strutturata e verticale di informazioni su un oggetto attraverso un social network dedicato. L'utilizzo della applicazione da parte degli utenti permette di moltiplicare, virtualmente all'infinito, le informazioni sull'oggetto e al tempo stesso di amplificare esponenzialmente la connotazione sociale derivante dalla condivisione di tali informazioni e delle opinioni al riguardo: da una semplice lista di oggetti, sono arrivato a qualcosa di più.

In questo elaborato sono esposte le varie tappe di studio e di lavoro svolto per la realizzazione di questa applicazione.

In una prima parte (capitoli 1 e 2), viene esaminato lo stato dell'arte. Il primo capitolo contiene un'analisi del mondo delle applicazioni mobili, riguardante sia gli aspetti "storici", sia la diffusione attuale, sia le previsioni per i futuri sviluppi. Nel secondo capitolo vengono esaminati i mobile social network, descrivendo i diversi tipi e le architetture su cui essi si basano. Questo settore ha raggiunto un'enorme diffusione e la domanda è ancora in forte espansione: la trasposizione dei social netowrk (nati in origine per dispositivi fissi) alle tecnologie mobile sembra infatti avere soddisfatto una basilare necessità dell'utenza, tanto da dilagare in pochi anni, raggiungendo una diffusione elevatissima.

La seconda parte dell'elaborato (capitoli 3, 4 e 5) riguarda in maniera specifica l'applicazione sviluppata. Nel terzo capitolo (progettazione) viene presentata l'idea iniziale che ha dato origine a questo progetto e i requisiti tecnici e funzionali. Ho utilizzato un'architettura di tipo client-server: sviluppando interamente l'applicazione Android, e ristrutturando e ampliando server e database già presentati nel progetto di Basi di Dati. Il quarto capitolo (implementazione) espone nel dettaglio le soluzioni adottate riguardo

le problematiche più specifiche e pertinenti ad applicazioni di questo tipo: notifiche push, utilizzo offline e sincronizzazione. L'implementazione di questi aspetti ha richiesto un impegno particolare, sia per le difficoltà tecniche, sia perchè rappresentano un punto di forza dell'applicazione, aumentando fortemente il coinvolgimento dell'utente e l'usabilità. Di particolare rilievo appaiono le notifiche geolocalizzate, indispensabili per portare l'applicazione ai livelli più elevati. Il quinto capitolo (validazione) presenta un'analisi particolareggiata sull'impatto determinato dalle notifiche push sulla frequenza di utilizzo delle applicazioni; viene inoltre riportata un'analisi delle prestazioni legate alle operazioni principali della fase implementativa; infine mostra le schermate finali dell'applicazione realizzata.

Indice

Introduzione **i**

I Stato dell'Arte **1**

1 Applicazioni Mobili **3**

1.1 Espansione 4

1.2 Sistemi operativi 8

1.2.1 Android 10

1.2.2 iOS 14

1.2.3 Windows Phone 14

2 Mobile Social Network **17**

2.1 Componenti 19

2.1.1 Clients 19

2.1.2 Infrastrutture 19

2.1.3 Fornitori di contenuti 19

2.2 Categorie di applicazioni 20

2.2.1 Network generalisti 20

2.2.2 Network verticali 20

2.2.3 Network sensoriali 21

2.3 Architetture 22

2.3.1 Architettura centralizzata 22

2.3.2 Architettura distribuita 23

2.3.3 Architettura ibrida 24
2.4 Analisi dei competitors 26
2.4.1 Wishlist (Beans) 26
2.4.2 Fashiolista (Fashiolista) 27

II Wishlist 29

3 Progettazione 31
3.1 L'idea 31
3.2 Requisiti 32
3.2.1 Requisiti Funzionali (RF) 32
3.2.2 Requisiti di Interfaccia utente (RUI) 33
3.2.3 Requisiti Tecnici (RT) 34
3.3 Architettura 34
3.3.1 Server 35
3.3.2 Client 35

4 Implementazione 37
4.1 Server 37
4.1.1 Tecnologie utilizzate 37
4.2 Architettura dell'applicazione 39
4.2.1 Architettura delle applicazioni Android 39
4.2.2 Tecnologie utilizzate 41
4.2.3 Fragment Oriented 43
4.3 Notifiche push 44
4.3.1 Google Cloud Messaging (GCM) 44
4.3.2 Amazon Simple Notification Service (SNS) 46
4.3.3 Implementazione e flussi 47
4.3.4 Servizio di tracking in background 56
4.4 Utilizzo offline e sincronizzazione 58
4.4.1 Persister 59
4.4.2 SyncHelper 62

4.4.3 Database 64

5 Validazione 69

5.1 L'importanza delle notifiche push 69

5.2 Analisi delle prestazioni 73

5.3 Schermate 76

Conclusioni 81

A Server API 83

Bibliografia 85

Elenco delle figure

1.1 Numero di applicazioni disponibili sui principali app stores (luglio 2014) 4

1.2 Numero di downloads gratuiti e a pagamento, effettuati attraverso i vari app stores, a livello mondiale, dal 2011 al 2017 (in miliardi) 5

1.3 Tempo medio dedicato rispettivamente ai dispositivi mobili o alla tv dal 2012 al 2014, relativo agli utenti degli Stati Uniti . 6

1.4 Aumento del tempo speso sui media digitali dal giugno 2013 al giugno 2014, relativo agli utenti degli Stati Uniti 7

1.5 Numero di smartphone venduti a livello mondiale, suddivisi per sistema operativo dal 2009 al 2014 (in milioni di unità) . . 9

1.6 Quote di mercato dei vari sistemi operativi mobili 2011-2014 . 10

1.7 Dpi dei dispositivi Android (2 Febbraio 2015) 11

1.8 Versioni di Android (2 Febbraio 2015) 12

1.9 Frammentazione dei dispositivi Android 13

1.10 Funzionamento dei Google Play Services 13

2.1 Intersezione 18

2.2 Architettura centralizzata 23

2.3 Architettura distribuita 24

2.4 Architettura ibrida 25

2.5 Schermate dell'applicazione Wishlist 26

2.6 Schermate dell'applicazione Fashiolista 27

4.1 Architettura di GCM 45
4.2 Architettura di Amazon SNS 46
4.3 Architettura di Amazon SNS 47
4.4 Flussi delle notifiche push 48
4.5 Architettura della persistenza e sincronizzazione 59

5.1 Incremento del numero di lanci dell'applicazione con le notifiche push abilitate 70
5.2 Numero di volte in cui l'applicazione viene utilizzata 71
5.3 Tasso di utilizzo dell'applicazione nei mesi successivi 72
5.4 Consumo energetico per la ricezione di una notifica push . . . 73
5.5 Consumo energetico del servizio di tracking 74
5.6 Consumo energetico per l'aggiunta di un nuovo oggetto 75
5.7 Consumo energetico del GPS per l'aggiunta di un nuovo oggetto 75
5.8 Schermate iniziali, da sinistra verso destra: welcome, registrazione, login. 76
5.9 Schermata del menu laterale 76
5.10 Schermate principali, da sinistra verso destra e dall'alto al basso: lista delle wishlists, lista degli oggetti di una wishlist, dettaglio dell'oggetto, commenti di un oggetto, likes di un oggetto, galleria. 77
5.11 Schermate di creazione, da sinistra verso destra e dall'alto al basso: creazione di una wishlist, creazione di un oggetto, selezione delle immagini, selezione del luogo, selezione della categoria. 78
5.12 Schermate social, da sinistra verso destra e dall'alto al basso: esplora, attività, mappa degli oggetti, profilo utente, alcune notifiche. 79

Elenco delle tabelle

A.1 Server API 85

Elenco listati di codice

4.1 Controllo dell'APK dei Google Play services 48

4.2 Metodo per il controllo del token di GCM 49

4.3 Metodo per effettuare la registrazione ai server di GCM 50

4.4 Creazione dell'endpoint su Amazon SNS 51

4.5 Chiamata della funzione che pubblica una notifica push 52

4.6 Implementazione della funzione che pubblica le notifiche push in Node.js 53

4.7 Gestione del messaggio di GCM sull'applicazione 54

4.8 Classe NotifyCommand e relativi metodi 55

4.9 Caricamento del fragment corrispondente alla notifica 56

4.10 Configurazione degli aggiornamenti di localizzazione 58

4.11 Metodo getAllWishlists del Persister 60

4.12 Metodo createWishlist del Persister 61

4.13 Metodo performSync del SyncHelper 62

4.14 Metodo syncDeletedWishlists del SyncHelper 63

4.15 Metodo finish del SyncHelper 64

4.16 Query per la creazione del database SQLite 65

4.17 Metodo create di WishlistDataSource 66

4.18 Metodo getUnsync di WishlistDataSource 66

4.19 Metodo getNewNegativeId di DataSource 67

Parte I

Stato dell'Arte

Capitolo 1

Applicazioni Mobili

Le "applicazioni mobili" sono programmi per computer destinati e idonei all'utilizzo su "dispositivi mobili". Con questo termine si intendono tutti quegli apparecchi elettronici che, per caratteristiche fisiche (peso e dimensioni) e funzionali, sono in grado di seguire l'utente nei suoi vari spostamenti e di operare in qualunque ambiente: al chiuso, all'aperto, su mezzi di trasporto.

Le applicazioni mobili sono rese disponibili dagli "app store", piattaforme digitali di distribuzione, solitamente gestiti dal proprietario del sistema operativo.

L'origine e l'evoluzione delle applicazioni sono intrinsecamente legate alla nascita e allo sviluppo dei dispositivi mobili e degli app store: queste tre tecnologie sono infatti strettamente interdipendenti. Le applicazioni sono il carburante per il funzionamento, e quindi il successo, dei dispositivi mobili: uno smartphone senza applicazioni è come una bellissima automobile superaccessoriata, ma col serbatoio vuoto; allo stesso modo un'applicazione non avrebbe motivo di esistere se non fosse supportata da un dispositivo utile al suo impiego. Senza il distributore tuttavia il carburante non arriverebbe mai al serbatoio: ecco perchè diventa indispensabile la funzione svolta dagli app store.

1.1 Espansione

Per quanto già da molti anni fossero disponibili tecnologie mobili (il termine ”Smartphone” è usato per la prima volta da Ericsson nel 1997) è l'avvento del primo IPhone nel 2007 che conferisce un'improvviso impulso allo sviluppo di questo settore. Pertanto consideriamo questo evento come un punto di partenza per l'enorme e inarrestabile espansione del mercato delle applicazioni mobili.

Da allora, con la comparsa di altri sistemi operativi e relativi stores, e con lo sviluppo di un numero esorbitante di nuove applicazioni, si è assistito a un incremento continuo e inarrestabile di questo mercato, tanto da poter considerare attualmente l'avvento delle applicazioni una vera e propria rivoluzione del ”mobile”. L'enorme dimensione del ”fenomeno apps” si può dedurre dalla figura 1.1, che riporta dati aggiornati al 2014 e mostra come il numero di applicazioni disponibili sia attualmente dell'ordine di milioni, con un netto predominio degli stores di Google e Apple.

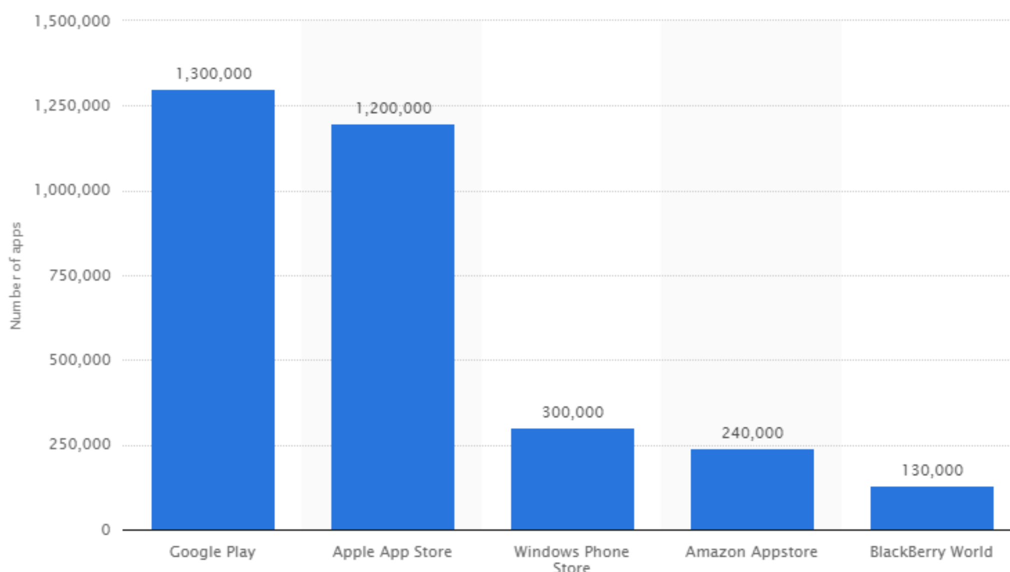

Figura 1.1: Numero di applicazioni disponibili sui principali app stores (luglio 2014)

fonte: `http://www.statista.com/statistics/276623`

Tra i motivi di questo enorme successo sono sicuramente da annoverare la semplicità di utilizzo, anche da parte di utenti non esperti, e il costo bassissimo o nullo; infatti la grande maggioranza delle applicazioni è scaricabile gratuitamente e presenta eventualmente un costo (comunque sempre minimo) solo per versioni avanzate (pro) o pagamenti "in-app". La decisione degli sviluppatori e delle aziende di pubblicare prevalentemente applicazioni gratuite, si è rivelata una scelta vincente. Il grafico 1.2 riporta il numero totale di applicazioni scaricate a livello mondiale dal 2011 al 2014, con una previsione di continuo incremento fino al 2017.

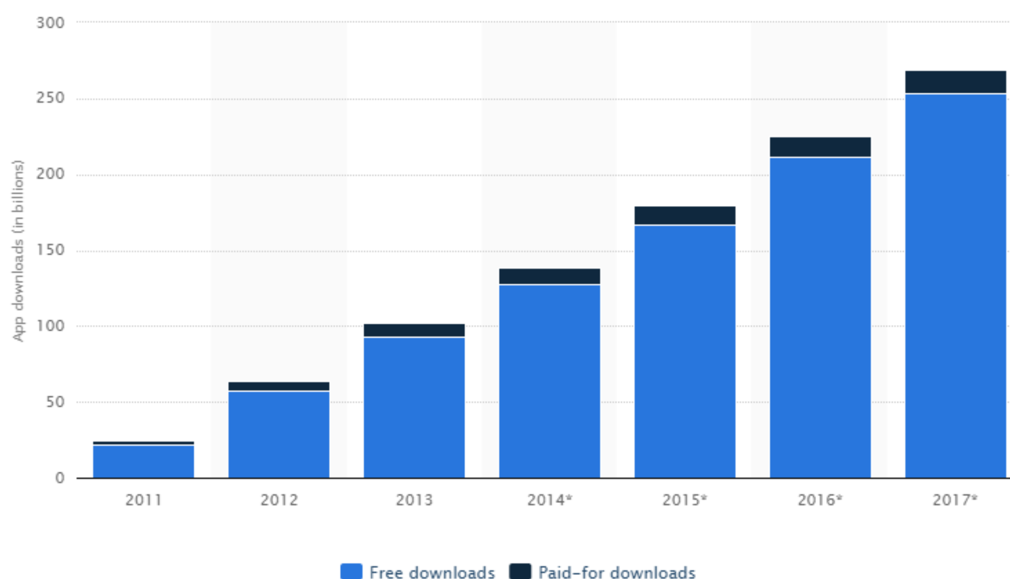

Figura 1.2: Numero di downloads gratuiti e a pagamento, effettuati attraverso i vari app stores, a livello mondiale, dal 2011 al 2017 (in miliardi) fonte: `http://www.statista.com/statistics/271644`

Nonostante si potessero avanzare dubbi che la crescita iniziale rappresentasse solo il boom di una nuova tecnologia, numerosi studi,[1][2] basati su metodi di previsione statistica, concordano con l'ulteriore espansione di questo mercato nei prossimi anni.

[1] Flu14a.
[2] Gar13.

Una conferma di questa tendenza viene anche da alcuni studi riguardanti i consumatori degli Stati Uniti: questo mercato, essendo il più avanzato e maturo, può essere considerato un esempio delle prossime tendenze negli altri paesi. Queste ricerche hanno considerato il tempo speso giornalmente sulle applicazioni mobili, sulla tv e su altri dispositivi digitali.

Il tempo dedicato alle applicazioni mobili risulta in aumento, con un fortissimo salto tra il 2012 e il 2013, probabilmente dovuto alla progressiva diffusione dei tablets e all'aumento delle dimensioni degli schermi. Nella seconda metà del 2014 si raggiunge un tempo speso di quasi tre ore, superando così quello dedicato alla tv.[3]

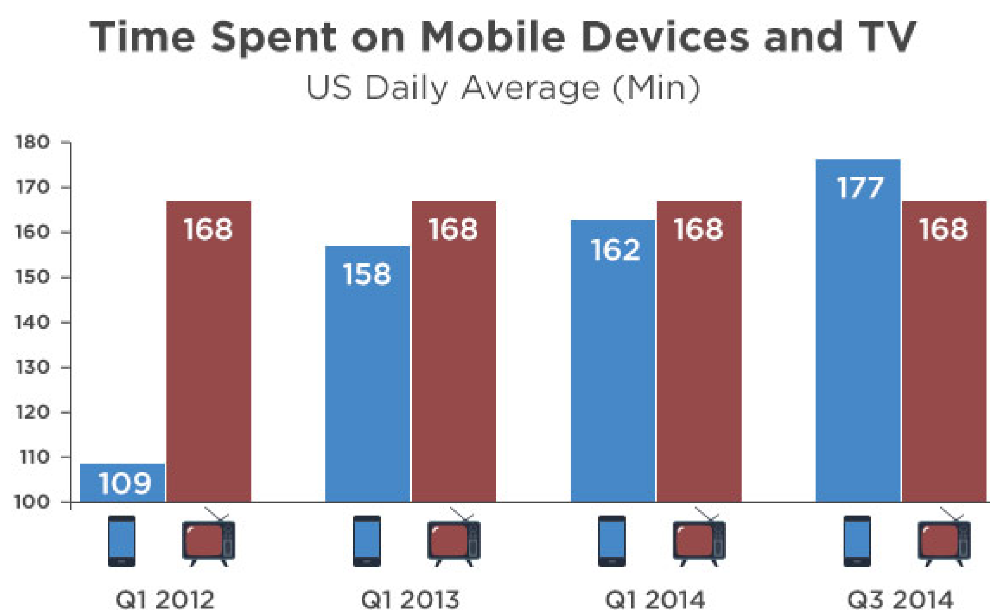

Figura 1.3: Tempo medio dedicato rispettivamente ai dispositivi mobili o alla tv dal 2012 al 2014, relativo agli utenti degli Stati Uniti
fonte: `http://www.flurry.com/sites/default/files/blog-images/mobile_vs_tv_1_v1b-1.jpg`

[3] Flu14b.

Nel grafico 1.4 (anch'esso riferito a utenti degli Stati Uniti) viene rappresentato il tempo speso sui media digitali da giugno 2013 a giugno 2014. Il tempo relativo ai computer desktop non risulta in aumento (+1%, non statisticamente significativo), mentre il tempo speso sulle applicazioni e sui browser mobili si è accresciuto rispettivamente del +52% e del +17%. Anche in questo settore si determina così, come accaduto anche per la tv, il sorpasso del mobile rispetto al fisso.[4]

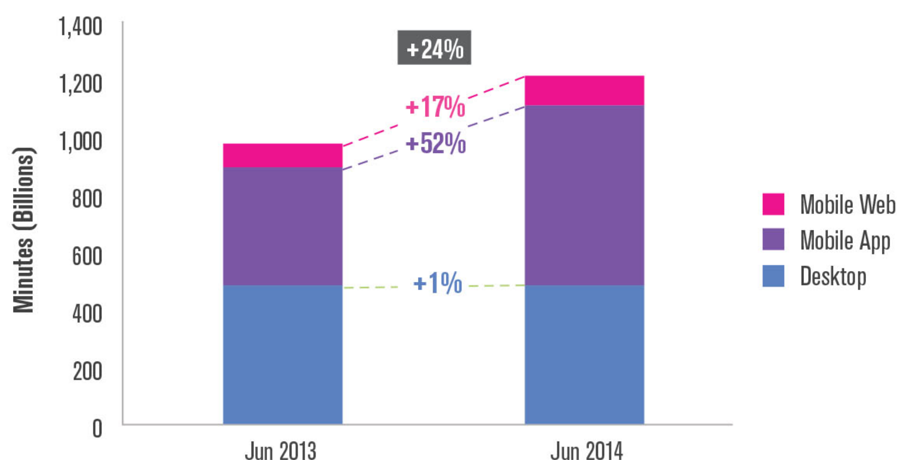

Figura 1.4: Aumento del tempo speso sui media digitali dal giugno 2013 al giugno 2014, relativo agli utenti degli Stati Uniti
fonte: `http://www.comscore.com/ita/Insights/Presentations-and-Whitepapers/2014/The-US-Mobile-App-Report`

[4]com14.

1.2 Sistemi operativi

Un sistema operativo per dispositivi mobili, in inglese "mobile OS", è un sistema operativo che controlla un dispositivo mobile con lo stesso principio con cui Mac OS, Unix, Linux o Windows controllano un desktop computer oppure un laptop. Tuttavia affronta problematiche legate intrinsecamente alla natura del dispositivo mobile, più critiche rispetto ad un desktop o un laptop. Tra le tante basti considerare la limitatezza delle risorse (memoria, CPU), l'assenza di alimentazione esterna, le differenti tecnologie per l'accesso a Internet (WiFi, GPRS, HSDPA, ecc.), i nuovi metodi d'immissione (touchscreen, minitastiere), le ridotte dimensioni del display.[5]

Tali difficoltà sono state adeguatamente risolte (pur essendo in continuo miglioramento), tanto che la diffusione di queste tecnologie ha raggiunto negli ultimi anni numeri elevatissimi.

Il grafico 1.5 riporta dati sul mercato globale degli smartphone dal 2009 al 2014, prendendo in esame un periodo che va dagli albori di questa tecnologia fino a dati recentissimi: è immediatamente visualizzabile come il numero sia in costante e sensibile aumento. Considerando la distribuzione dei diversi sistemi operativi mobili, si nota come l'aumento sia legato principalemente ad Android e, in misura molto inferiore ad iOS. Tra i primi sistemi operativi, alcuni risultano praticamente estinti (Symbian e Bada), mentre Windows Phone (Microsoft), seppure con piccoli numeri, mostra una presenza costante o in leggero aumento. Per una comprensione più immediata della dimensione del fenomeno riportiamo i dati numerici relativi al 2010, con 296,65 milioni, e al 2013, con 967,83 milioni. I dati del 2014 (al momento di questa pubblicazione incompleti), sono ora disponibili[6] riportando un numero dell'ordine di 1,2 miliardi di dispositivi, pari a circa 1/6 della popolazione mondiale.

[5]Wik15.
[6]Gar14.

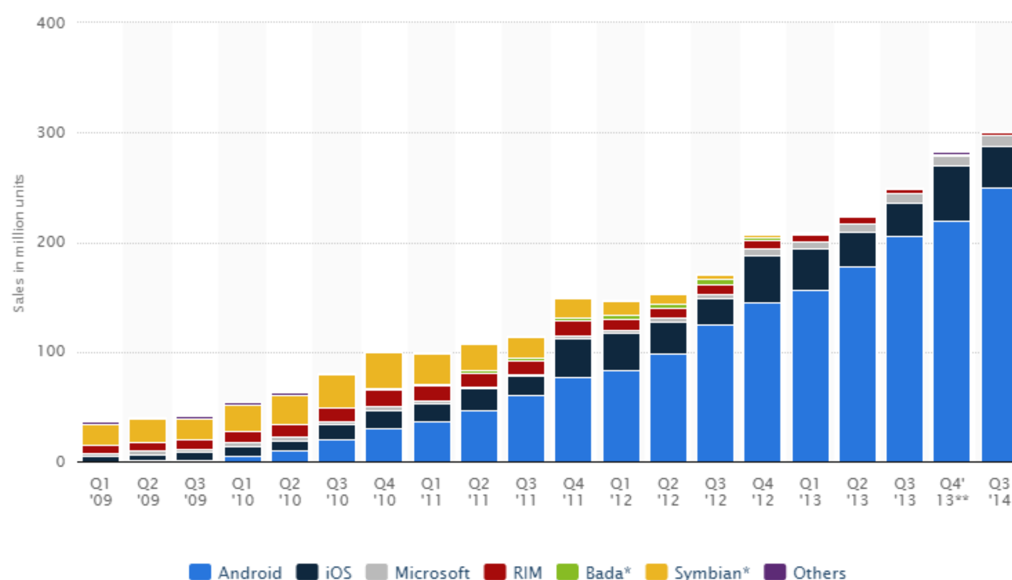

Figura 1.5: Numero di smartphone venduti a livello mondiale, suddivisi per sistema operativo dal 2009 al 2014 (in milioni di unità)
fonte: `http://www.statista.com/statistics/266219`

Il grafico 1.6 riporta in valori percentuali le quote di mercato relative ai diversi sistemi operativi in base al numero di unità spedite dal 2011 al 2014. Rispetto al grafico precedente, si può notare con più immediatezza l'aumento del distacco tra Android e tutti gli altri sistemi: procedendo nel tempo la "forbice" si apre. Questo divario è facilmente comprensibile se si pensa che Android, essendo "open source", ha portato molte aziende di elettronica (Samsung, LG, HTC, Huawei, ecc.) ad adottarlo come sistema operativo per i propri dispositivi, non solo di tipo mobile ma anche Smart TV, elettrodomestici, consoles, auto, ecc. Al contrario iOS e Windows Phone sono sistemi "chiusi", progettati esclusivamente per i dispositivi prodotti dalle aziende proprietarie del sistema operativo stesso. È da sottolineare tuttavia come i numeri relativi alla diffusione dei vari sistemi operativi, non abbiano un riscontro diretto nel loro valore economico, ma non è questa la sede per una disamina di questo tipo.

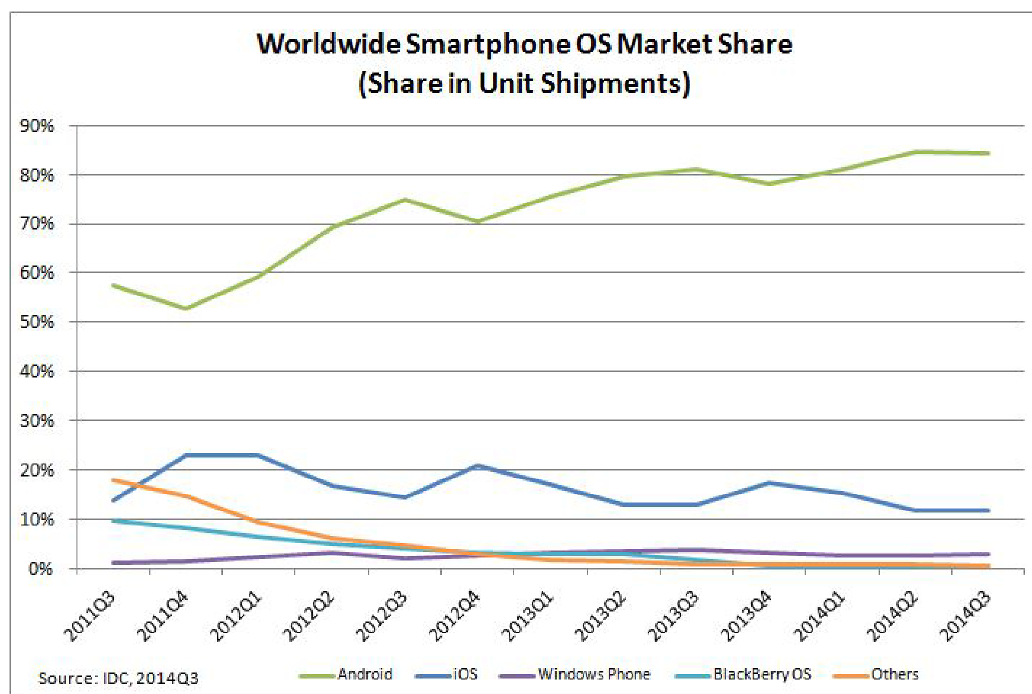

Figura 1.6: Quote di mercato dei vari sistemi operativi mobili 2011-2014
fonte: `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`

Di seguito prenderò in esame più dettagliatamente i tre sistemi più diffusi: Android, iOS e Windows Phone.

1.2.1 Android

Le origini di Android risalgono all'ottobre 2003, quando Andy Rubin, Rich Miner, Nick Sears e Chris White fondano la *Android Inc.*, una società finalizzata allo sviluppo di un sistema operativo per dispositivi mobili che permettesse di ottenere, secondo le parole di Rubin, "smarter mobile devices that are more aware of its owner's location and preferences". Nel 2005 la *Android Inc.* viene acquistata da *Google*, intenzionata ad entrare nel mercato mobile: con le ampie risorse ora disponibili, il team di Rubin mette a punto la nuova tecnologia. Il 5 novembre 2007 viene presentata da OHA (*Open Handset Alliance*, un consorzio di aziende che include anche *Google*) la prima versione del sistema operativo, basato su kernel Linux. Da allora vengono

rilasciate versioni successive (12 macro release) fino al novembre 2014 in cui viene lanciata la 5.0 col nome di *Lollipop*. Il primo dispositivo equipaggiato con Android che venne lanciato sul mercato fu l'HTC Dream, il 22 ottobre del 2008. Da allora il successo di questo sistema operativo è andato sempre crescendo, fino ad occupare attualmente la posizione leader nel mercato.

Uno dei maggiori punti di forza, nonchè uno degli aspetti più apprezzati di Android è rappresentato dalla sua licenza open source (Apache 2.0). La pubblicazione del codice sorgente ha permesso agli sviluppatori di avere una migliore conoscenza di quello che succede in background sul sistema e di contribuire in maniera attiva al progetto stesso. Questo ha reso Android estremamente flessibile, permettendo alla nuove tecnologie di integrarsi efficacemente e velocemente al suo interno, diventando così fortemente appetibile per i vari produttori di dispositivi.

La frammentazione di Android

Questo fenomeno è all'origine della "frammentazione" che, pur rappresentando un indubbio vantaggio dal punto di vista commerciale, è anche il più grosso problema della piattaforma. Android infatti è presente su un numero enorme di dispositivi, caratterizzati da concetti e form factor sostanzialmente diversi che vanno a coprire parecchie nicchie di mercato.

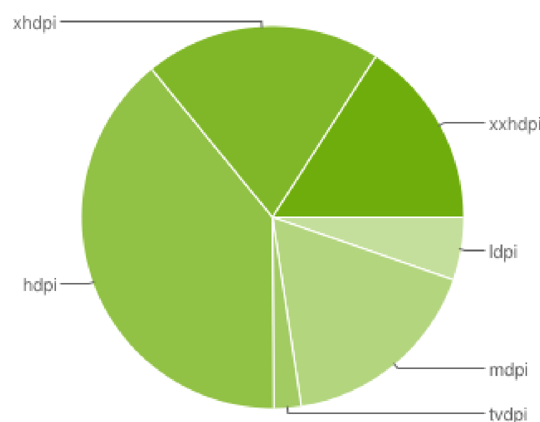

Figura 1.7: Dpi dei dispositivi Android (2 Febbraio 2015)
fonte: `https://developer.android.com/about/dashboards/index.html`

Avere una piattaforma operativa frammentata, però, obbliga lo sviluppatore ad un conseguente lavoro aggiuntivo, per testare e ottimizzare le proprie applicazioni su un numero effettivamente molto corposo di prodotti di terze parti. Generalmente i produttori di dispositivi mobili operano piccole modifiche sulla versione rilasciata da Google per personalizzarla in base al proprio brand o per esigenze tecniche. All'uscita di una nuova versione occorre un ulteriore lavoro di adeguamento, che non sempre avviene in tempi immediati (a volte, in caso di dispositivi considerati ormai superati, il produttore decide di non effettuare l'aggiornamento). Come si può vedere dalla figura 1.8, infatti, solo una piccola percentuale di dispositivi è equipaggiata con la versione più recente del sistema operativo, mentre la grande maggioranza resta bloccata per tempo indefinito alle release precedenti.

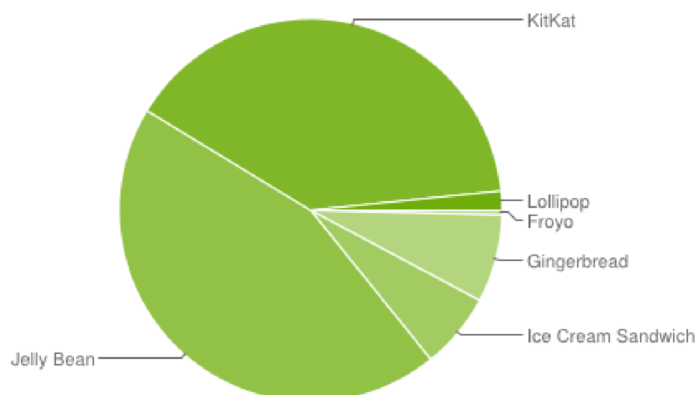

Figura 1.8: Versioni di Android (2 Febbraio 2015)
fonte: `https://developer.android.com/about/dashboards/index.html`

Sfruttando i dati raccolti dagli smartphone, OpenSignal è riuscita a creare un quadro del numero di smartphone differenti in circolazione. Secondo le stime della società, nel 2014 sono presenti 18.796 modelli di dispositivi Android, un numero in netto aumento rispetto agli 11.868 dell'anno precedente. I dati di OpenSignal si basano sui 682.000 dispositivi su cui è stata installata l'applicazione. Il numero è naturalmente destinato ad aumentare nel corso del tempo.[7]

[7]Ope14.

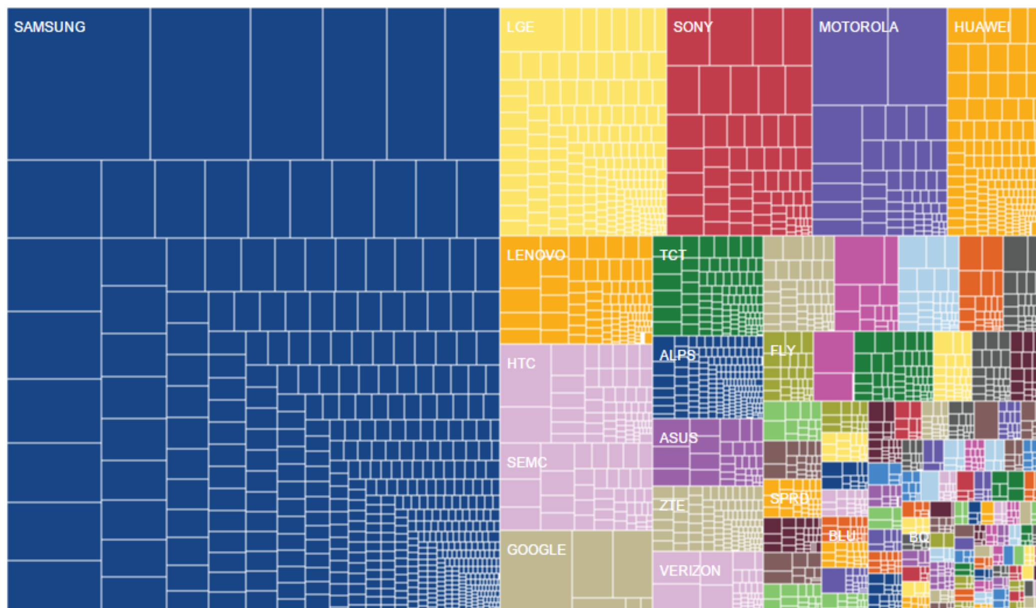

Figura 1.9: Frammentazione dei dispositivi Android
fonte: `http://opensignal.com/reports/2014/android-fragmentation`

Per affrontare il problema della frammentazione, Google ho introdotto nel 2012 i "Google Play Services"; questa tecnologia permette allo sviluppatore di utilizzare le nuove APIs dei servizi di Google senza preoccuparsi del supporto del dispositivo. L'APK dei servizi è aggiornata automaticamente tramite il Google Play Store, la client library invece, attraverso l'SDK Manager.

Figura 1.10: Funzionamento dei Google Play Services
fonte: `https://developer.android.com/images/play-services-diagram.png`

1.2.2 iOS

iOS è il sistema operativo mobile sviluppato da Apple.
La prima versione (1.0, senza una denominazione specifica) entra in commercio come sistema operativo del primo iPhone il 29 giugno 2007.
In seguito il sistema viene denominato "iPhone OS" e ne sono poi pubblicate successive versioni (parallelamente all'uscita sul mercato dei nuovi modelli di iPhone), con aggiunta di varie funzioni tra cui particolarmente rilevante la disponibilità dell'App Store (iPhone 3G, 11 luglio 2008).
Nel 2010, con l'uscita dell'iPhone 4, il sistema viene ribattezzato iOS. Poco dopo, con la versione 4.2.1 il sistema non è più dedicato esclusivamente all'iPhone, ma è idoneo all'utilizzo su altri dispositivi quali iPod touch e iPad.
Da allora si sono susseguite diverse versione con l'aggiunta di numerose funzionalità e API, miglioramenti dell'interfaccia grafica, fino alla più recente iOS 8 del giugno 2014.

1.2.3 Windows Phone

Windows Phone è un insieme di sistemi operativi per smartphone sviluppati da Microsoft.
L'azienda inizia a lavorare ai sistemi operativi mobili nel 2004 con un progetto chiamato *Photon* che viene però abbandonato poichè lo sviluppo appariva troppo lento. Nel 2008 Microsoft ritorna sull'argomento riorganizzando le modalià di lavoro e ricerca e nel 2009 pubblica Windows Mobile 6.5, destinato ai dispositivi mobili Pocket PC, Smartphone, Portable Media Center.
Successivamente le scelte aziendali portano alla decisione di dedicare questa tecnologia esclusivamente agli smartphone e inizia lo sviluppo di Windows Phone.
La prima versione, Windows Phone 7, viene presentata al Mobile World Congress il 15 febbraio 2010 e nell'ottobre dello stesso anno viene lanciata sul mercato. È seguita nel ottobre 2012 dal lancio di Windows Phone 8 e, nel 2014, da Windows Phone 8.1 .

Windows Phone arriva sul mercato molto in ritardo rispetto agli altri principali sistemi operativi mobili (Android e iOS), per cui inizialmente l'insufficiente disponibilità di applicazioni determina uno scarso successo della piattaforma. Questo divario si è assottigliato nel corso degli ultimi due anni con l'arrivo di quasi tutte le principali applicazioni, tuttavia permane una sensibile differenza rispetto ai competitors all'interno del mercato di questo settore.

Capitolo 2

Mobile Social Network

Una rete sociale (social network) consiste in un qualsiasi gruppo di individui connessi tra loro da diversi legami sociali (valori condivisi, aiuto reciproco, scambio di beni ecc.).[1] L'utilizzo di questo termine nell'ambito dell'informazione e delle comunicazioni tecnologiche viene riferito a quelle piattaforme che forniscono servizi per lo scambio di dati e informazioni tra gli utenti. L'utilizzo di tali piattaforme arricchisce continuamente il servizio di nuove informazioni e relazioni. Il sistema da parte sua, attraverso l'analisi dei dati forniti dagli utenti, modifica e migliora il servizio stesso.[2]

La nozione di social network e la sua applicazione negli utlimi anni hanno attirato molte attenzioni da parte di ricercatori in vari campi (informatici, economisti, psicologi ecc.): il social network, infatti, in quanto sistema che struttura le interconnessioni tra i diversi utenti, rappresenta un valido strumento per il loro studio e la loro comprensione.

I social network, accessibili inizialmente soltanto da dispositivi fissi, hanno in seguito trovato un'ottimale collocazione sui dispositivi mobili e relative applicazioni. Infatti oggi la mobilità fisica delle persone è in tutto il mondo parte integrante del comune modo di vivere. Essa comporta uno spostamento casuale di grandi numeri di individui. L'avvento dei dispositivi e delle appli-

[1] S W94.
[2] N D06.

cazioni mobili si è rivelato quindi uno strumento indispensabile per portare i social network alla portata di soggetti dinamici, ai fini di fornire servizi di comunicazione e informazione efficienti in relazione alla effettiva posizione dell'individuo in ogni momento.

L'idea di "mobile social network" è nata dalla combinazione di concetti appartenenti a due diverse scienze: le reti sociali descritte e studiate dalla sociologia e le reti di comunicazione mobile del mondo informatico. Con questo termine si indica quindi un sistema di comunicazione mobile che coinvolga le relazioni sociali degli utenti.

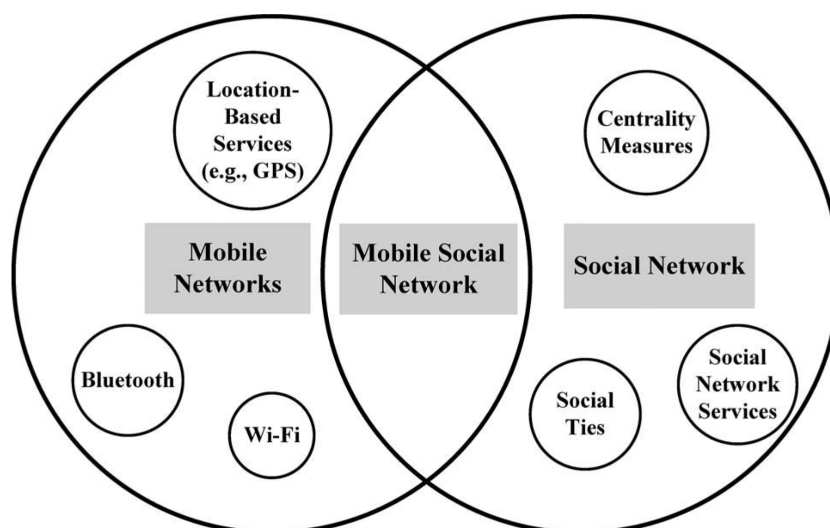

Figura 2.1: Intersezione
fonte: Applications, Architectures, and Protocol Design Issues for Mobile Social Networks: A Survey

In un network di questo tipo gli utenti possono accedere, condividere, distribuire dati in un contesto mobile, sfruttando le loro relazioni sociali. Grazie all'ubiquitaria disponibilità di dispositivi mobili (smartphones, tables, ecc.), un mobile social network può avvantaggiarsi grandemente delle interazioni e degli spostamenti fisici degli utenti per realizzare efficienti ed effettivi servizi di consegna dati.[3]

[3]NH11.

2.1 Componenti

Le componenti che costituiscono un mobile social network sono tre: clients, infrastrutture di rete, fornitori di contenuti.

2.1.1 Clients

I dispositivi mobili sono l'interfaccia dei mobile social network, rappresentando l'endpoint di ingresso e uscita del contesto sensoriale dell'utente. Essi gli consentono di trasferire la sua presenza in forma virtuale verso il fornitore di contenuti o direttamente verso altri client, oppure di percepire e osservare i contesti di altri utenti. Possono avere diverse interfacce di comunicazione di rete e attraverso queste consentono l'interazione con l'infrastruttura del sistema.

2.1.2 Infrastrutture

Questo termine si riferisce all'insieme dell'hardware e del software che permettono il trasferimento di dati da una fonte (es. fornitore di contenuti) a una destinazione (es. client). L'infrastruttura di rete sulla quale si basa il sistema dipende dall'architettura utilizzata. In caso di architettura centralizzata essa è gestita dall'operatore telefonico o di rete Wi-Fi locale e pertanto non è controllabile da parte del gestore del social network. Solo in alcuni casi particolari questi due elementi coincidono (es. in alcune città degli stati uniti, Google si pone allo stesso tempo come fornitore di contenuti e come ISP attraverso la sua rete di fibra ottica). Nel caso di una architettura distribuita, dipende persino dai diversi modelli dei dispositivi stessi, ciò che ne evidenzia un'importante criticità.

2.1.3 Fornitori di contenuti

Il fornitore di contenuti è l'elemento centrale della piattaforma client-server ed è quello che si occupa di raccogliere i dati inviati dai dispositivi

attraverso l'infrastruttura di rete, elaborarli e reinviarli ad altri dispositivi. Questo è vero per una architettura centralizzata, che sfrutta eventualmente sistemi cloud, ma è altresì vero anche per architetture distribuite, dove questo ruolo viene coperto dal middleware di comunicazione integrato nell'applicazione sul dispositivo.

2.2 Categorie di applicazioni

Le categorie di applicazioni mobili sono da ricondurre a tre tipi fondamentali: network generalisti, network verticali, network sensoriali.

2.2.1 Network generalisti

Si definiscono social network generalisti quei servizi che non trattano un argomento specifico, ma sono finalizzati esclusivamente a connettere i propri utenti, permettendo loro lo scambio di vari tipi di dati senza uno scopo preciso. Le iniziali versioni accessibili solo attraverso i browser (es. Facebook e Twitter) si sono in seguito allargate al mercato mobile creando una o più applicazioni proprie. A queste si aggiungono i network generalisti che fin dalla loro origine si sono proposti con entrambe le versioni (es. Google+). Le versioni mobile non solo mantengono le stesse funzionalità, ma possono aggiungerne di nuove tramite l'uso dei sensori. Alcuni studi hanno dimostrato come queste applicazioni native riescano a portare la "user experience" dell'utente a livelli persino superiori rispetto alla versione web, aumentando in maniera sensibile la soddisfazione e l'utilizzo.[4]

2.2.2 Network verticali

I network verticali sono social network dedicati ad una particolare finalità condivisa. Tra i principali troviamo quelli di e-Health, quali PatientsLikeMe e CaringBridge del mercato americano, ma sono molti gli esempi di ambiti

[4] Sap10.

totalmente diversi, quali ad esempio myBrickset, social network mobile per collezionisti di Lego, o BoardGameGeek, indirizzato agli appassionati di giochi da tavolo. Altro esempio in crescita di questo tipo di network è quello della triade "Apple Game Center", "Google Play Games" e "Xbox Live" (ai quali si stanno aggiungendo lentamente piattaforme esterne), tutti con la stessa finalità di interconnettere utenti di giochi mobile, tutti con la stessa modalità di engagement degli utenti, tutti con la stessa tipologia di target, ognuno con la caratteristica di essere esclusivo per la propria piattaforma.

2.2.3 Network sensoriali

I network sensoriali sono social network dedicati al rilievo e alla comunicazione di dati sensoriali. La diffusione di sensori di vario genere (GPS, accelerometri, sensori di temperatura ecc.) apre gradualmente nuove opportunità per la creazione di applicazioni che sfruttino quei dati in modo nuovo. I primi sensori utilizzati e attualmente più diffusi sono sicuramente GPS e fotocamera, che hanno già dimostrato ampiamente le loro potenzialità di mercato: applicazioni quali Foursquare hanno aperto la strada ai servizi location-based, così come applicazioni quali Instagram hanno spinto l'acceleratore sugli aspetti di condivisione delle foto. Attualmente il campo si sta aprendo a nuovi esempi di network sensoriali quali Snapchat, Vine, Uber, ecc. Gli aspetti sociali di queste reti, consentono ad esempio di sfruttare le informazioni sulla posizione del dispositivo mobile per costruire una rete locale di interazioni fisiche attorno ad un preciso punto nello spazio. Premessa inevitabile è la diffusione delle specifiche applicazioni, non esistendo ancora un protocollo condiviso di localizzazione. È necessario quindi raggiungere una massa critica di utenza che renda vantaggioso l'uso della piattaforma per i suoi utenti.

L'utilizzo di sistemi di localizzazione in combinazione con i social network introduce una serie di problemi legati alla privacy e alla sicurezza personale degli utenti, se non trattati con la dovuta cautela.

2.3 Architetture

Per quanto riguarda i social network mobile è possibile ricondurli a tre principali architetture, ognuna delle quali con peculiari tipologie di connessione e di interconnessione tra gli utenti, con conseguenti effetti sulla user experience delle applicazioni.

2.3.1 Architettura centralizzata

L'architettura centralizzata è quella più diffusa, essendo anche la più semplice dal punto di vista della progettazione e implementazione. Si basa sul classico pattern client-server, nel quale il dispositivo mobile è il client e l'infrastruttura server è il fornitore dei contenuti e dei serivizi associati all'applicazione. I servizi sono totalmente dipendenti dal server e prevedono la gestione di componenti tipici di un social network, quali ad esempio gestione lista di amici,condivisione di contenuti.

Un indubbio vantaggio di questo approccio è la semplicità di costruzione di eventuali client specifici per diverse piattaforme (es. Android, iOS, Windows Phone, ecc.): l'utilizzo delle API definite sul server come interfaccia comune rende più facile e veloce la comunicazione tra sistemi differenti. Le tecnologie web ampiamente diffuse e standardizzate vengono in aiuto fornendo protocolli condivisi e librerie ampiamente testate, minimizzando così rallentamenti ed errori di sviluppo. Grazie alle tecnologie cloud si eliminano i tipici problemi di congestione delle architetture client-server, aggirando l'ostacolo del single-point-of-failure e distribuendo la computazione e lo storage delle informazioni su sistemi scalabili a livello continentale.

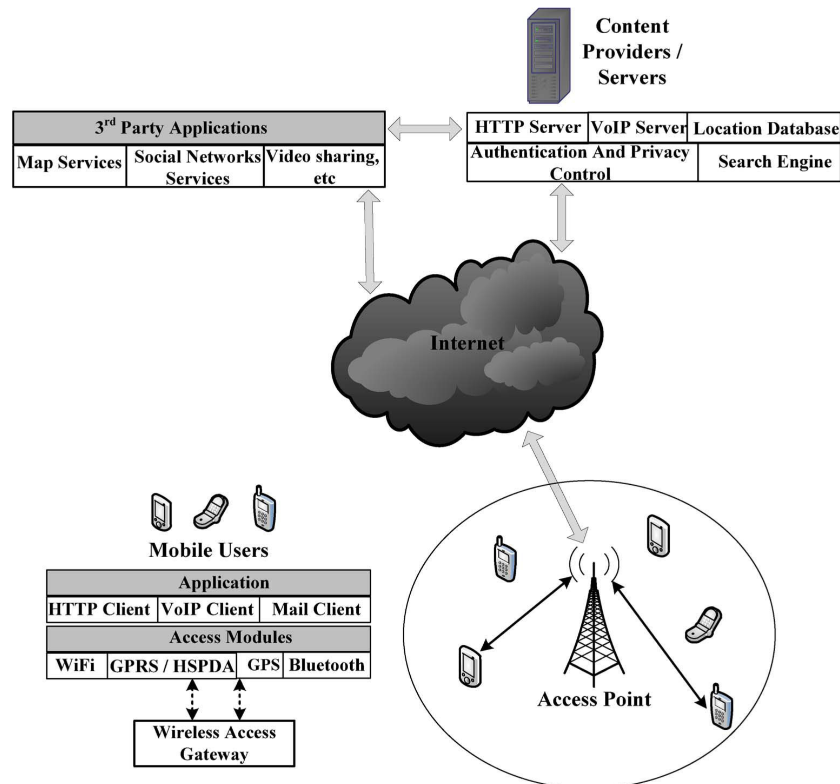

Figura 2.2: Architettura centralizzata
fonte: A General Architecture of Mobile Social Network Services

2.3.2 Architettura distribuita

Questo tipo di architettura non prevede la presenza di un server centrale, ma si basa sull'utilizzo di tecnologie di rete presenti sui dispositivi mobili e sulla interazione diretta fra essi. Le tecnologie tipicamente sfruttate sono Bluetooth e Wi-Fi: esse consentono diverse tipologie di comunicazione peculiari. Queste si possono basare, ad esempio, sulle oramai diffuse tecniche di "opportunistic networking"[5] o su altre modalità simili, specifiche delle eventuali applicazioni.

La difficoltà di sviluppare strumenti basati su questo tipo di architettura deriva da tre fattori principali. Innanzitutto la complessità intrinseca nell'uso

[5] F N09.

di tencologie "standard" su piattaforme diverse, che se in teoria dovrebbero interagire senza problemi, all'atto pratico dimostrano spesso problematiche di non semplice soluzione. Il secondo problema è quello delle librerie che dovrebbero facilitare lo sviluppo di applicazioni di questo tipo: si tratta sicuramente di strumenti di grande utilità, ma appaiono ancora piuttosto immature su quasi tutte le piattaforme. Il terzo problema è costituito dal raggiungimento della massa critica nell'uso delle applicazioni che utilizzano questo tipo di architettura: la probabilità di trovare un dispositivo con una specifica applicazione installata che consenta una interazione P2P è ovviamente dipendente dalla diffusione della applicazione stessa. La rarefazione a livello geografico degli utenti potrebbe rendere inutilizzabile la piattaforma.

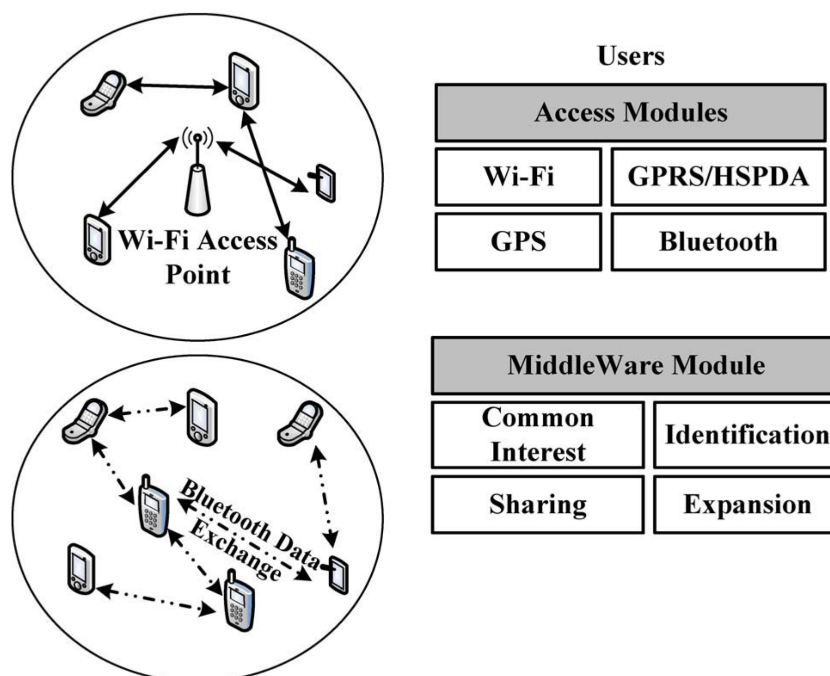

Figura 2.3: Architettura distribuita
fonte: A General Architecture of Mobile Social Network Services

2.3.3 Architettura ibrida

Questo tipo di architettura sfrutta gli aspetti positivi di entrambe le architetture definite in precedenza. Ha il vantaggio della struttura centraliz-

zata che consente la condivisione dei contenuti tramite un server, ma permette anche di usufruire dei servizi anche in quelle particolari situazioni in cui il server risulti irraggiungibile, sfruttando in questo caso i sistemi P2P dell'architettura distribuita.

Questo sistema è estremamente complesso da realizzare e ancor più complesso da gestire. Non a caso, non risultano usi commerciali di architettura ibrida. Sono però interessanti gli spunti che da questo tipo di approccio scaturiscono. Fondamentale fra i tanti è la possibilità di utilizzare il social network, seppure in modalità limitata, anche senza la connessione persistente al server, sfruttando meccanismi di caching, sincronizzazione e simili.

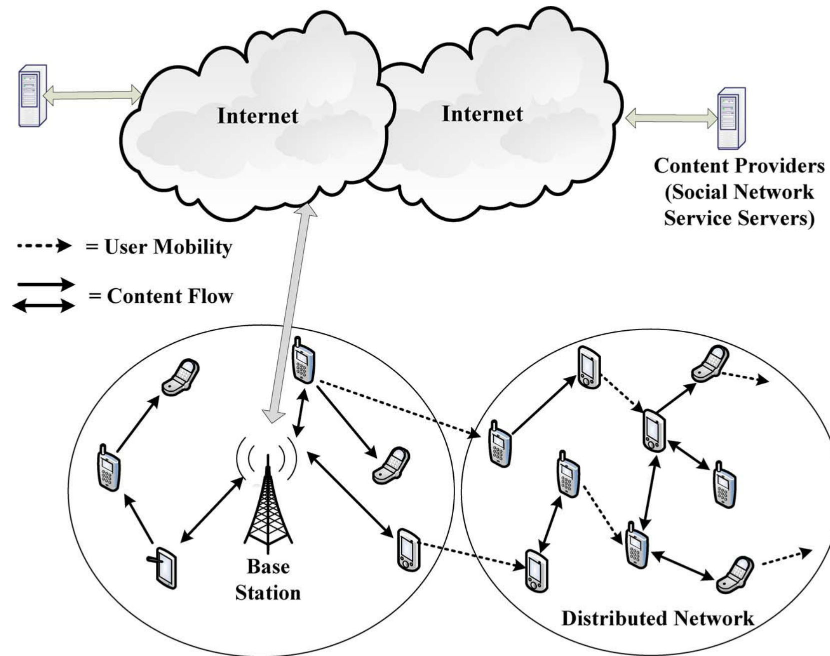

Figura 2.4: Architettura ibrida
fonte: A General Architecture of Mobile Social Network Services

2.4 Analisi dei competitors

Tra le tante applicazioni di tipo "wishlist" presenti sul Google Play Store, vengono prese in esame in questa sezione quelle che, per funzionalità e qualità implementativa, appaiono più simili e quindi più raffrontabili a Wishlist.

2.4.1 Wishlist (Beans)

Rating: 3,9

Numero di downloads: 10.000-50.000

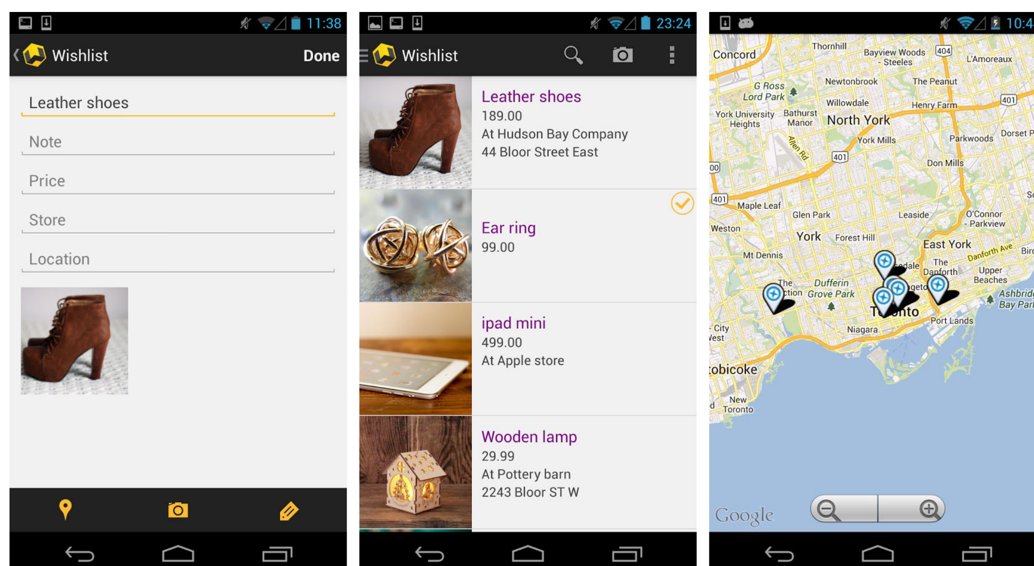

Figura 2.5: Schermate dell'applicazione Wishlist

Wishlist è un'applicazione Android che permette all'utente di inserire e gestire degli oggetti all'interno di una lista ordinata. Al momento dell'aggiunta di un nuovo elemento è possibile compilare un form con diversi campi quali Nome (obbligatorio), Descrizione, Prezzo, Nome del negozio e Posizionamento geografico. Quest'ultima informazione può essere inserita manualmente dall'utente oppure rilevata automaticamente dall'applicazione attraverso l'uso del gps. Gli oggetti ai quali è stata assegnata una posizione geografica sono visualizzabili su una mappa. Si può inoltre aggiungere una foto, scattata direttamente con la fotocamera oppure selezionata fra quelle già presenti sul

dispositivo. Ad ogni elemento, l'applicazione permette anche di associare uno o più tags creati dall'utente.

Pregi e difetti

Il punto di forza di questa applicazione è sicuramente la semplicità, è di utilizzo immediato, intuitivo e non richiede l'accesso alla connessione internet. D'altro canto, la mancanza di un meccanismo di backup la rende suscettibile alla perdita dei dati. Oltre a questo, l'applicazione presenta numerosi difetti tecnici come frequenti crashes causati dalla rotazione del dispositivo e alcuni bugs relativi al mantenimento dello stato dell'interfaccia grafica.

2.4.2 Fashiolista (Fashiolista)

Rating: 4,1
Numero di downloads: 100.000-500.000

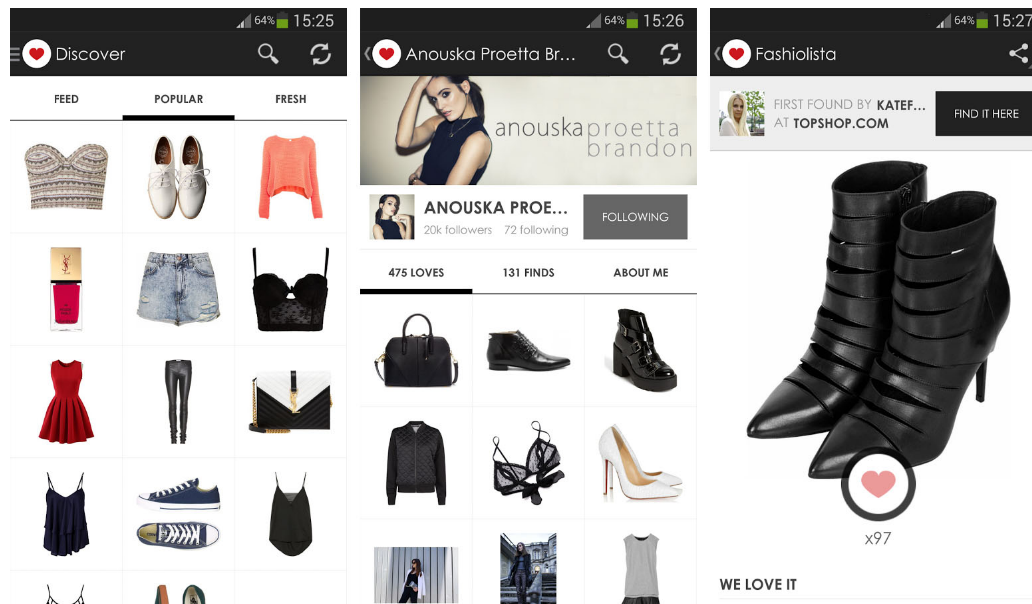

Figura 2.6: Schermate dell'applicazione Fashiolista

Questa applicazione è sicuramente quella che si avvicina maggiormente all'idea di Wishlist. Si tratta di un social network basato sulla condivisione di

capi di abbigliamento e accessori. Inizialmente l'applicazione presenta una vetrina contenente gli oggetti più "popolari" nella community, la lista degli utenti più seguiti e quella dei negozi più visitati. É possibile decidere di ricevere gli aggiornamenti su ciò che gli utenti e i negozi pubblicano attraverso un meccanismo di "following". L'utente può aggiungere a sua volta oggetti di suo interesse in due modalità: scegliendoli fra quelli già presenti sull'applicazione oppure importandoli da un'applicazione di terze parti.

Pregi e difetti

Fashiolista è un'applicazione molto ben realizzata sia dal punto di vista tecnico che da quello grafico, con un'interfaccia pulita e intuitiva. Può contare su una community molto numerosa, che partecipa in maniera attiva contribuendo significativamente alla crescita del social network. L'applicazione è dedicata quasi esclusivamente a un pubblico femminile: infatti il contesto è limitato al mondo della moda e dell'abbigliamento e i negozi presenti sono specifici per donne e ragazze. Una limitazione è rappresentata dal fatto che Fashiolista non permette l'aggiunta di oggetti in maniera "libera", non è possibile infatti aggiungere un prodotto scattando una foto e/o segnalandone la presenza in un negozio reale. Un ulteriore limite di questa applicazione è costituito dalla impossibilità di utilizzo offline: senza una connessione, nessuna funzionalità risulta disponibile.

Parte II

Wishlist

Capitolo 3

Progettazione

In questo capitolo vengono prese in esame le diverse fasi di progettazione dell'applicazione. Gli elementi riportati derivano dell'analisi di un'esigenza molto precisa che ha richiesto una serie di iterazioni di elaborazione. Idea, requisiti ed implementazione sono stati percorsi e ripercorsi più e più volte, in modo da comprendere a fondo le ripercussioni sul codice dei singoli requisiti funzionali, estetici o tecnici e, viceversa, l'impatto di specificità del linguaggio o delle librerie sulla definizone dei requisiti stessi.

3.1 L'idea

Dall'analisi dei più recenti comportamenti di fashion blogger e commentatori tecnici sui social network "vetrina" (Twitter, Instagram, Pinterest), dove le relazioni sono consapevolmente unidirezionali, emerge una tendenza sempre più evidente ad utilizzare questi social network per scopi e con modalità per i quali non offrono servizi specifici. Ad esempio Instagram è uno strumento per condividere e valorizzare la fotografia in quanto tale e non le informazioni legate all'oggetto rappresentato, relegando le meta-informazioni sul contenuto al messaggio (breve) e agli hashtag, che a causa della loro intrinseca destrutturazione risultano generare informazioni non attendibili se non con complesse analisi. Altro caso identico è quello di Pinterest.

Da queste premesse nasce l'idea di costruire uno strumento che consenta la condivisione strutturata e verticale delle informazioni legate a oggetti di interesse attraverso un social network specifico. Ovviamente, da un punto di vista strutturale, l'applicazione rimane nell'ambito dei social network "vetrina". In quanto tale, consente all'utente di seguire amici o personaggi di spicco, così come consente agli utenti più popolari di valorizzare al massimo la loro presenza on-line condividendo informazioni e organizzandole a piacimento.

3.2 Requisiti

3.2.1 Requisiti Funzionali (RF)

L'applicazione deve fornire (senza escluderne altre più specifiche) le seguenti funzionalità principali:

1. L'utente deve poter registrarsi e autenticarsi.

2. L'utente deve poter visualizzare il suo profilo e i suoi dati.

3. L'utente deve poter visualizzare il profilo e i dati di altri utenti.

4. L'utente deve poter creare una o più liste che facciano da collettori per gli oggetti inseriti successivamente, specificandone nome, descrizione, visibilità (collezione pubblica o privata) e una categoria che ne caratterizzi meglio il contenuto.

5. L'utente deve poter inserire in una lista uno o più oggetti, specificando un nome e, ove ritenuto utile, una serie di informazioni aggiuntive: descrizione, prezzo, livello di importanza, una o più fotografie, una georeferenziazione ed una categoria.

6. L'utente deve poter modificare o cancellare un proprio oggetto o una propria lista.

7. L'utente deve poter decidere se seguire o abbandonare un altro utente.

8. L'utente deve poter commentare a suo piacimento un oggetto e/o esprimere un "like" su di esso.

9. L'applicazione deve fornire all'utente una lista delle persone più popolari sul social network.

10. L'applicazione deve fornire all'utente una lista contenente tutti i suoi amici di Facebook che, tramite la suddetta piattaforma, si sono già registrati su Wishlist.

11. L'applicazione deve fornire all'utente una lista che riporti tutte le attività delle persone che sta seguendo in ordine cronologico.

12. L'applicazione deve fornire all'utente una mappa, su cui vengono visualizzati i suoi oggetti nelle vicinanze e quelli delle persone che segue.

13. L'applicazione deve informare l'utente in modo proattivo sui cambiamenti nei seguenti ambiti:

    - Un altro utente ha cominciato a seguirlo.
    - Un suo oggetto ha ricevuto un "like" da parte di un altro utente.
    - Un suo oggetto è stato commentato da un altro utente.
    - Un oggetto di un altro utente che sta seguendo è diventato popolare.
    - Nelle vicinanze sono rilevati oggetti di altri utenti che sta seguendo.

14. L'applicazione deve fornire all'utente un'esperienza d'uso consistente, sia quando il dispositivo è connesso a internet che (con limitazioni se necessario), quando il dispositivo è disconnesso.

3.2.2 Requisiti di Interfaccia utente (RUI)

Da un punto di vista grafico, l'applicazione deve rispettare le seguenti condizioni:

1. Deve seguire le linee guida fornite da Google relative a Material Design.[1]

2. I pattern di navigazione utilizzati devono essere quelli standard di Android.

3.2.3 Requisiti Tecnici (RT)

Da un punto di vista tecnico, le funzionalità devono avere i seguenti requisiti aggiuntivi:

1. L'applicazione deve essere sviluppata per Android e supportare la versione 4.0.3 (API 15) e successive.

2. La registrazione e l'autenticazione devono potersi effettuare sfruttando il protocollo OAuth2.0 offerto da Facebook, oppure direttamente sulla piattaforma senza passare da altri social network.

3. L'architettura dell'interfaccia deve basarsi su un approccio "Fragment Oriented".

4. L'applicazione deve permettere l'aggiunta agli oggetti di una o più immagini, scegliendo fra le tipiche modalità delle applicazioni di alto livello: scatto diretto di una nuova foto, selezione dalla galleria standard del telefono o dai vari servizi cloud, selezione da una griglia rapida contenente le ultime immagini scattate.

5. La georeferenziazione dei singoli oggetti deve avvenire attraverso l'integrazione con le API di Foursquare, recuperando la lista dei luoghi di interesse più vicini alla posizione geografica dell'utente.

3.3 Architettura

L'applicazione si basa su un'architettura client-server classica, alla quale vengono aggiunti elementi che supportino operazioni in condizioni di di-

[1] http://www.google.com/design/spec/material-design/introduction.html

sconnessione. Queste operazioni sono descritte nel dettaglio all'interno della sezione 4.4.

3.3.1 Server

Il server è sviluppato in Node.js utilizzando MySQL come database e fornisce un'API RESTful[2]. In aggiunta il server si occupa di avviare le operazioni push per l'invio di notifiche ai dispositivi, come spiegato nella sezione 4.3.3 (RF13 - parte server).

3.3.2 Client

Il client è sviluppato in Android e supporta la versione 4.0.3 (RT1). Il client accede alle API RESTful fornite dal server per tutte le operazioni principali: la comunicazione avviene attraverso la libreria Retrofit, attorno alla quale è stato creato un sistema di persistenza in caso di disconnesione volontaria o involontaria (RF14). Tra le tante funzionalità sono state rese disponibili offline quelle relative alla gestione delle liste e degli oggetti personali dell'utente (RF2, RF4, RF5, RF6). Inoltre l'applicazione è in costante ascolto di messaggi push da parte del server attraverso GCM, come esposto dettagliatamente nella sezione 4.3.3 (RF13 - parte client).

Per la realizzazione dell'interfaccia grafica sono state utilizzate diverse librerie: v7 appcompat library e v7 cardview library per la retrocompatibilità e costruzione del design material (RUI1); Picasso e image-chooser-library per il supporto alla visualizzazione e selezione delle immagini (RT4); AndroidStaggeredGrid e Android Sliding Up Panel per la creazione di alcuni componenti e pattern particolari (RUI2); Mapbox Android SDK per la gestione della mappa (RF12).

Le singole librerie vengono analizzate nel dettaglio all'interno della sezione 4.2.2.

[2]vedi tabella A.1 in appendice

Capitolo 4

Implementazione

4.1 Server

Il server di Wishlist è composto da un'applicazione in Node.js e un database MySQL. Questo sistema fornisce un'API RESTful[1], attraverso la quale l'applicazione Android può comunicare. Il lato server non viene esaminato ed esposto nel dettaglio in quanto l'argomento di questo elaborato è l'analisi e l'esposizione dell'applicazione Android (lato client). Aggiungere un'approfondita trattazione del server avrebbe inutilmente appesantito l'esposizione, rischiando di distogliere l'attenzione dal punto focale di questo lavoro.

4.1.1 Tecnologie utilizzate

L'applicazione server utilizza la versione di Node.js 0.10.33 ed è formata da una serie di moduli: alcuni svolgono una funzione di primaria importanza, altri sono stati aggiunti per ottimizzare la stesura del codice e/o per utilizzare delle piccole funzionalità:

- express (v4.12.x)
- waterline (v0.10.x)

[1] vedi tabella A.1 in appendice

- sails-mysql (v0.10.x)
- aws-sdk (v2.1.x)
- fb (v0.7.x)
- node-foursquare (v0.3.x)
- underscore (v1.8.x)
- async (v0.9.x)
- bluebird (v2.9.x)
- bcrypt (v0.8.x)
- crypto (v0.0.x)
- walk (v2.3.x)
- compression (v1.4.x)
- body-parser (v1.12.x)
- cors (v2.5.x)
- connect-multiparty (v1.2.x)
- mime (v1.3.x)
- gravatar (v1.1.x)
- request (v2.9.x)
- cheerio (v0.18.x)

4.2 Architettura dell'applicazione

In questa sezione viene presa in esame l'architettura di Wishlist, presentando innanzitutto le caratteristiche principali delle applicazioni Android, in modo da permettere la comprensione del contenuto delle sezioni successive. Vengono poi elencate e descritte le tecnologie utilizzate per la sua realizzazione e la sua struttura interna.

4.2.1 Architettura delle applicazioni Android

Android possiede una vasta ed esauriente documentazione ufficiale che copre tutti gli aspetti fondamentali (e non) della sua architettura. Sulla rete si possono reperire facilmente numerose pubblicazioni e guide per chiunque si approcci al sistema per la prima volta o voglia approfondirne un aspetto specifico. Inoltre, la comunità degli sviluppatori è estremamente ampia e attiva, ed esistono portali specifici per lo scambio di informazioni, come ad esempio Stack Overflow, Google Groups, il blog ufficiale di Android Developers, ecc. Appare quindi inutile (oltre che controproducente ai fini della chiarezza espositiva) una descrizione dettagliata di ogni aspetto riguardante la progettazione delle applicazioni.
Perciò in questa sezione vengono fornite soltanto le informazioni fondamentali per capire la struttura di un'applicazione Android, limitandosi a ciò che è necessario per accedere agevolmente al contenuto dei successivi paragrafi, riguardati specificamente lo sviluppo di Wishlist.

Componenti

I componenti sono gli elementi fondamentali per la costruzione di un'applicazione Android. Ci sono quattro tipi diversi di componenti, ognuno dei quali svolge un compito specifico e possiede un proprio lifecycle che definisce come il componente viene creato e distrutto.

- Un **Activity** è uno degli elementi centrali di ogni applicazione Android, perchè si occupa della gestione dell'interfaccia, permettendo all'utente va-

ri modi di interagire (scattare una foto, inviare un email, visualizzare una mappa, ecc.).
Possiamo vedere l'Activity come un vero e proprio contenitore di componenti grafici che potranno occupare tutto lo spazio sullo schermo, parte di esso oppure essere completamente nascosti.
Solitamente un'applicazione Android è composta da numerose activities, ognuna delle quali visualizza e gestisce una schermata: possono essere schermate che visualizzano semplicemente delle informazioni, oppure che interagiscono con l'utente fornendo funzionalità specifiche.
Android organizza le activities secondo una struttura a stack ("back stack") dove l'activity attiva in quel momento, occupa il primo posto. Quando una nuova Activity viene lanciata, va ad occupare la prima posizione del back stack, mettendo in pausa quella precedente. Quando l'utente termina l'utilizzo dell'activity corrente, premendo il taso "back", questa viene rimossa dal back stack (e distrutta) e si ripristina la precedente.

- Un **Service** è un componente che non fornisce un'interfaccia grafica, ma resta in esecuzione in background. In questo modo è possibile eseguire dei compiti particolarmente lunghi e/o svincolati dal lifecycle dell'applicazione (esempio computazioni complesse, operazioni di I/O, sincronizzazione dati, ecc.).

- Un **Content provider** è un meccanismo che permette alle applicazioni di condividere dati tra loro. Si tratta sostanzialmente di un'interfaccia che fornisce metodi standard (query, insert, delete e update), attraverso i quali è possibile accedere alle informazioni. Il sistema non impone l'utilizzo di nessuna specifica struttura dati, infatti può trattarsi di un singolo file o un vero e proprio database SQLite.

- Un **Broadcast Receiver** è un componente che ricevere i messaggi inviati in broadcast dal sistema. Il sistema invia continuamente un grosso numero di messaggi, ognuno dei quali rispecchia un evento specifico (esempio la batteria è quasi scarica, è stata scattata una foto, si è spento lo schermo,

ecc.) appena accaduto sul dispositivo. La maggior parte delle volte il broadcast receiver viene utilizzato come un "gateway" per altri componenti e svolge un lavoro molto limitato. Ad esempio, potrebbe avviare un service per svolgere un compito a seconda dell'evento ricevuto.

Manifest

Su ogni applicazione Android deve essere presente, nella root directory, un singolo file chiamato AndroidManifest.xml. Esso contiene le informazioni essenziali dell'applicazione che vengono richieste dal sistema per eseguirla. Tutti i componenti devono essere registrati su questo file, insieme ai permessi necesseri per funzionare.

4.2.2 Tecnologie utilizzate

È stata utilizzata una serie di librerie per velocizzare e ottimizzare il processo di sviluppo di Wishlist:

- **v7 appcompat library (v21.0.3)** è una libreria di supporto per la retrocompatibilità generale dell'applicazione, nel caso specifico di Wishlist è stata fondamentale per ottenere un perfetto "Material Design" anche su dispositivi con una versione di Android precedente alla 5.0 (API 21).

- **v7 cardview library (v21.0.3):** permette di utilizzare il nuovo widget CardView, introdotto recentemente con Android 5.0 (API 21), anche su versioni precedenti del sistema operativo.

- **Google Play services (v6.5.87):** consente l'utilizzo dei servizi di Google all'interno dell'applicazione. È stata sfruttata per le funzioni di localizzazione.

- **Retrofit (v1.7.0):** è un REST client per Android (o Java) che permette di trasformare l'API RESTful offerta dal server in vere e proprie interfacce

Java. Per mezzo di Retrofit viene creato un livello di astrazione capace di semplificare l'interazione col server, sfruttando la libreria Gson per convertire i body delle richieste HTTP in JSON e viceversa.

- **Butter Knife (v5.1.2):** semplifica molti passaggi, gestendo automaticamente delle operazione che solitamente richiederebbero molto codice aggiuntivo da parte dello sviluppatore.

- **Picasso (v2.3.4):** gestisce il download e il caching delle immagini in maniera semplice ed efficiente.

- **image-chooser-library (v1.3.56):** fornisce un supporto per la selezione delle immagini e dei video dalla memoria del dispositivo oppure attraverso i principali servizi cloud.

- **AndroidStaggeredGrid (v1.0.5):** permette l'utilizzo di una lista di tipo "staggered", che supporta un numero di colonne multiple e righe di varie dimensioni.

- **Android Sliding Up Panel (v2.0.4):** consente di aggiungere all'applicazione uno o più pannelli che scorrono dall'alto verso il basso, capaci di contenere altri elementi della UI a discrezione dello sviluppatore.

- **Mapbox Android SDK (v0.5.0):** è un servizio[2] che fornisce mappe personalizzabili sfruttando i dati di OpenStreetMap, NASA e DigitalGlobe, utilizzato all'interno dell'applicazione per la visualizzazione georeferenziata degli oggetti e dei luoghi.

- **Facebook SDK for Android (v3.23.0):** consente l'integrazione dell'applicazione con Facebook, offrendo una serie di servizi quali login, share, app events, l'accesso alla Graph API. Essa fornisce anche degli elementi grafici come pickers e dialogs.

[2]https://www.mapbox.com

4.2.3 Fragment Oriented

L'interfaccia grafica di Wishlist è basata quasi completamente su un'architettura di tipo "Fragment Oriented", cioè costituita da una sola activity (escluse le activities delle schermate iniziali e quelle di creazione/modifica delle liste e oggetti) che aggiunge e rimuove vari fragment all'interno di un unico FrameLayout.

Questa approccio porta notevoli vantaggi, sia da un punto di vista implementativo, sia per quanto riguarda la user experience.

Vantaggi implementativi

- Ha permesso di modulare ogni sezione del progetto, circoscrivendo il compito specifico di ciascun fragment e quindi consentendo il riuso della stessa struttura in diverse sezioni dell'applicazione.
- Ha portato alla razionalizzazione della logica globale dell'applicazione raggruppandola in un'unica activity permettendo pertanto di realizzare:
  - il controllo delle transizioni fra i fragments e del loro backstack
  - la gestione dell'intera sessione (app e/o Facebook) in un unico punto
  - il monitoraggio dello stato della connessione, con possibilità di inviare un feedback all'utente in caso di mancanza della stessa
  - la gestione delle notifiche
  - la chiamata di activities di altre applicazioni e la ricezione/gestione dei risultati

Vantaggi per l'utente

Fluidità nel cambio delle schermate che si traduce in una percezione di continuità e maggiore velocità durante l'uso dell'applicazione.

Disponibilità di navigazione attraverso il menù laterale (NavigationDrawer) da qualunque schermata dell'applicazione, permettendo all'utente di ri-

salire direttamente ad una delle sezioni principali/root[3] senza ricorrere all'uso del "back".

4.3 Notifiche push

Una notifica push è un breve messaggio inviato a un dispositivo mobile (smartphone o tablet) che abbia installato applicazioni specifiche. Il messaggio appare sullo schermo indipendentemente dallo stato di accensione o sleeping del dispositivo e dallo stato di attivazione o meno dell'applicazione. Il contenuto della notifica push consiste in un avviso sintetico, recapitato in tempo reale, di un evento accaduto sull'applicazione. Poichè la notifica fornisce un'anteprima sulla natura dell'evento (un nuovo commento, una richiesta di amicizia, un messaggio di chat, ecc.), l'utente può scegliere se visualizzarne immediatamente il contenuto o posticiparne la disamina.

4.3.1 Google Cloud Messaging (GCM)

Google Cloud Messaging (GCM) è un servizio gratuito predisposto da Google che, fungendo da tramite, permette agli sviluppatori di inviare dati dal proprio server alle proprie applicazioni Android e viceversa. Questo messaggio può essere sfruttato come "lightweight message" che notifichi semplicemente la presenza di nuovi dati da scaricare sul server, oppure come contenitore diretto di dati fino a una dimensione di 4Kb. GCM gestisce autonomamente le code dei messaggi e la loro distribuzione ai relativi dispositivi.

[3] "Mio profilo", "Le mie wishlist", "Esplora", "Novità" e "Mappa"

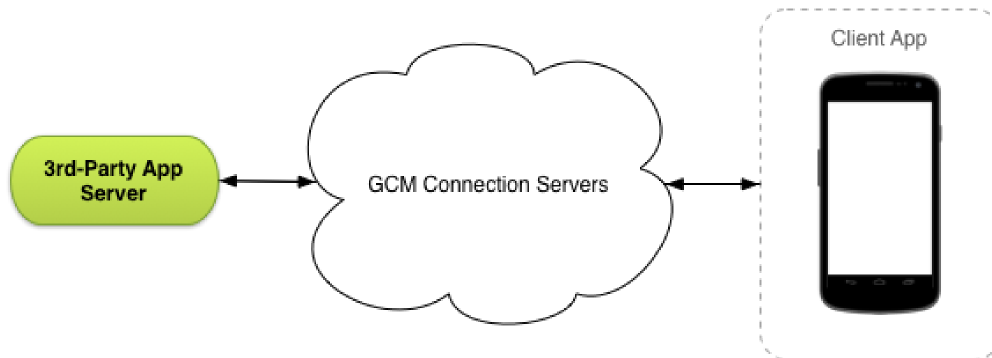

Figura 4.1: Architettura di GCM
fonte: `https://developer.android.com/google/gcm/gcm.html`

Caratteristiche

- Permette di inviare messaggi provenienti da server di terze parti ad applicazioni Android.

- Permette di ricevere messaggi in "upstream" provenienti dai dispositivi degli utenti.

- Un'applicazione Android, può ricevere messaggi anche quando non è in esecuzione. All'arrivo del messaggio il sistema attiverà l'applicazione attraverso un Intent (in broadcast) che verrà ricevuto da uno specifico BroadcastReceiver.

- La funzione di GCM è unicamente quella di consegnare il messaggio, mentre la sua gestione è competenza esclusiva dell'applicazione.

- È necessario un dispositivo Android 2.2+ con l'applicazione del Google Play Store installata, o eventualmente un emulatore Android 2.2+ con Google APIs.

- Per le versioni Android inferiori a 3.0 è necessario inoltre un Google account sul dispositivo.

4.3.2 Amazon Simple Notification Service (SNS)

Amazon Simple Notification Service (Amazon SNS) è un servizio appartenente all'infrastruttura di Amazon Web Services (AWS) che coordina e gestisce l'invio e la consegna di messaggi distribuendoli ai vari endpoints/clients registrati. Ci sono due tipi di clients:

- I publishers comunicano in maniera asincrona versi i subscribers inviando un messaggio attraverso un punto d'accesso logico chiamato topic.
- I subscribers (web server, indirizzi email o code SQS) ricevono i messaggi o le notifiche attraverso uno dei protocolli supportati (SQS, HTTP/S, email, SMS) una volta che questi vengono associati ad un topic.

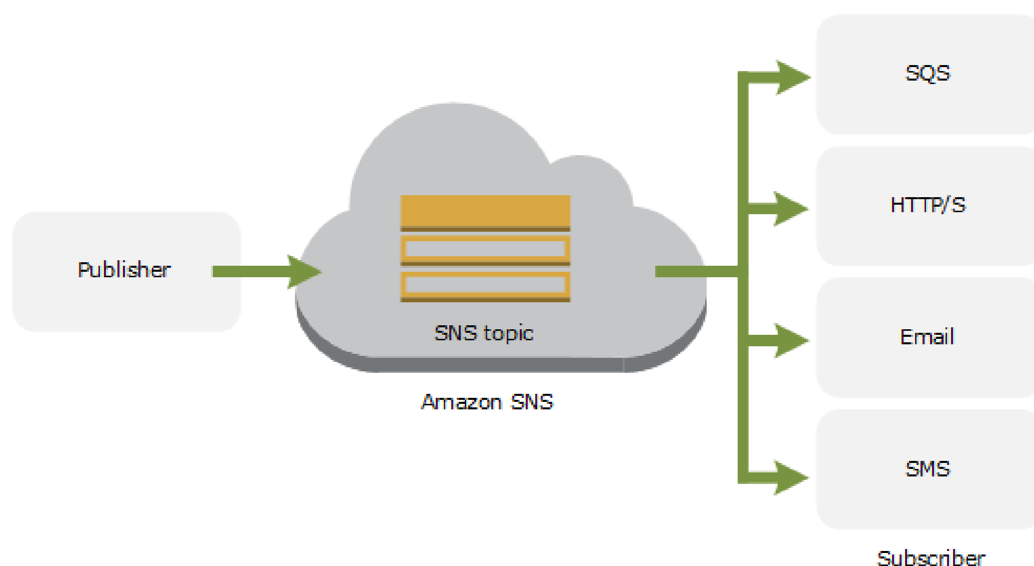

Figura 4.2: Architettura di Amazon SNS
fonte: `http://docs.aws.amazon.com/sns/latest/dg/SNSMobilePush.html`

Come si evince dalla figura 4.3 Amazon SNS è in grado di inviare messaggi a diversi servizi di push notifications.

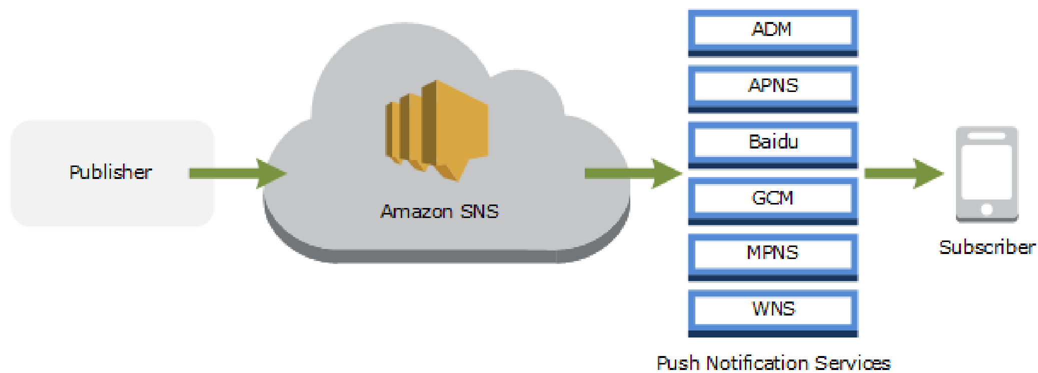

Figura 4.3: Architettura di Amazon SNS
fonte: `http://docs.aws.amazon.com/sns/latest/dg/SNSMobilePush.html`

La vocazione di Wishlist a diventare un vero e proprio social network distribuito su più piattaforme mobile (è in fase di sviluppo la versione per iOS) e la necessità di impostare fin dall'inizio il server nel modo più elastico ed efficiente possibile, hanno portato in maniera spontanea e irrinunciabile alla scelta di utilizzare Amazon SNS, che fornisce allo sviluppatore un unico punto di accesso con cui interagire. In questo modo lo sviluppatore stesso viene sollevato dall'impiego delle diverse API e SDK delle molteplici piattaforme.

4.3.3 Implementazione e flussi

In questa sezione viene descritto il processo di gestione delle notifiche push che coinvolge le diverse entità (applicazione Android Wishlist, server, Amazon SNS, GCM) rappresentate nel diagramma sottostante.

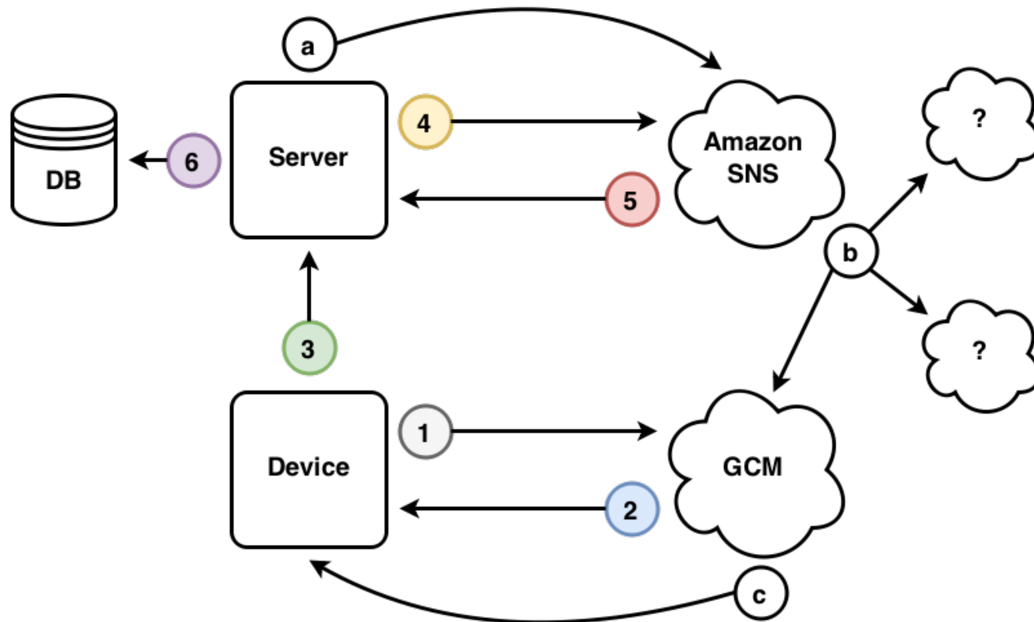

Figura 4.4: Flussi delle notifiche push

Il diagramma rappresentato nella figura 4.4 viene di seguito illustrato punto per punto, introducendo per ogni fase il relativo codice implementato.

Fase di inizializzazione

1. Come prima cosa viene verificato che sia presente una APK dei Google Play services compatibile sul dispositivo (codice 4.1 riga 1). In caso affermativo viene richiamato il metodo getGcmRegistrationId (codice 4.2). Se il metodo restituisce una stringa vuota (codice 4.1 riga 4), allora l'applicazione effettua la registrazione ai server di GCM (codice 4.1 riga 5).

```
if (PlayServicesUtils.checkPlayServices(this)) {          1
  gcm = GoogleCloudMessaging.getInstance(this);           2
  gcmRegistrationId = getGcmRegistrationId(context);      3
  if (gcmRegistrationId.isEmpty()) {                      4
    registerToGcmInBackground();                          5
```

```
  }                                                          6
} else {                                                     7
  LOGI(TAG, "No valid Google Play Services APK found.");     8
}                                                            9
```

Codice 4.1: Controllo dell'APK dei Google Play services

Questo metodo 4.2 verifica se è già presente un token di GCM all'interno delle shared preferences (riga 2 e 3). In caso affermativo, viene confrontata la versione corrente dell'applicazione con quella salvata precedentemente (riga 9). Se la l'applicazione è stata aggiornata, è necessario rieffettuare la registrazione, poichè non è garantito che il token sia compatibile con la nuova versione.

```
private String getGcmRegistrationId(Context context) {              1
  String registrationId = PreferenceUtils.getGcmRegId(context);     2
  if (registrationId.isEmpty()) {                                   3
    LOGI(TAG, "Registration not found.");                           4
    return "";                                                      5
  }                                                                 6
  int registeredVersion = PreferenceUtils                           7
    .getGcmAppVersion(context);                                     8
  int currentVersion = AppInfoUtils.getAppVersion(context);         9
  if (registeredVersion != currentVersion) {                        10
    LOGI(TAG, "App version changed.");                              11
    return "";                                                      12
  }                                                                 13
  return registrationId;                                            14
}                                                                   15
```

Codice 4.2: Metodo per il controllo del token di GCM

Poichè il metodo register di GCM è bloccante (codice 4.3 riga 10), la registrazione deve avvenire su un thread in background. E' stato per-

tanto utilizzato un AsyncTask istanziato direttamente dentro il metodo registerToGcmInBackground (codice 4.3).

2. GCM restituisce un nuovo token che viene salvato all'interno delle shared preferences, unitamente alla versione corrente dell'applicazione (codice 4.3 riga 12 e 13).

3. Il token di GCM viene inviato al server (codice 4.3 riga 11) tramite una specifica chiamata API. All'interno del body HTTP viene aggiunto il token e una stringa platform che identifica la piattaforma.

```
private void registerToGcmInBackground() {
  ...
  @Override
  protected String doInBackground(Void... params) {
    try {
      if (gcm == null) {
        gcm = GoogleCloudMessaging.getInstance(context);
      }
      gcmRegistrationId = gcm
        .register(Config.GOOGLE_API_PROJECT_NUMBER);
      sendGcmRegistrationIdToBackend();
      PreferenceUtils.setGcmRegID(context, gcmRegistrationId);
      PreferenceUtils.setGcmAppVersion(context,
        AppInfoUtils.getAppVersion(context));
    }
    ...
```

Codice 4.3: Metodo per effettuare la registrazione ai server di GCM

4. Il server invia una richiesta ad Amazon SNS composta dal PlatformApplicationArn (un stringa formata da ID personale SNS, tipo di piattaforma a cui appartiene il token, nome dell'applicazione) ed il token appena ricevuto (codice 4.4 riga da 4 a 7). L'azione CreatePlatformEndpoint è idempoten-

te, quindi, se il richiedente possiede già un endpoint con lo stesso token, SNS lo restituisce senza crearne uno nuovo.

5. Amazon SNS ritorna al server (codice 4.4 riga 10) un EndpointArn (il token con cui il server potrà inviare messaggi all'applicazione, tramite SNS).

6. Il server salva l'EndpointArn sul database assegnandolo all'utente che gli aveva inviato il token di GCM (codice 4.4 da riga 16 a 21).

```
async.series({                                                    1
  register: function(callback) {                                  2
    var sns = new AWS.SNS();                                      3
    sns.createPlatformEndpoint({                                  4
      PlatformApplicationArn: 'arn:aws:sns:...:app/'              5
        + req.body.platform + '/Wishlist',                        6
      Token: req.body.token,                                      7
    }, function(err, data) {                                      8
      if (err) return callback(err);                              9
      notificationEndpoint.endpoint_arn = data.EndpointArn;       10
      callback(null, true);                                       11
    });                                                           12
  },                                                              13
  notification_endpoint: function(callback) {                     14
    NotificationEndpoint                                          15
      .create(notificationEndpoint)                               16
      .exec(function(err, notificationEndpoint) {                 17
        if(err) return callback(err);                             18
        callback(null, notificationEndpoint);                     19
      });                                                         20
  }                                                               21
},                                                                22
```

Codice 4.4: Creazione dell'endpoint su Amazon SNS

Così conclusa la fase di inizializzazione il server protrà recapitare messaggi direttamente all'applicazione.

Fase di utilizzo

Per illustrare la fase di utilizzo verranno descritti, a titolo di esempio, i flussi relativi a un evento, applicabili ad uno qualunque dei cinque tipi di notifica precedentemente descritti (sezione 3.2.1), tutti riconoscibili dall'applicazione.

a) Nel momento in cui viene creato un nuovo commento, il server esegue una funzione specifica per inviare il messaggio a SNS. Il metodo prende in input i seguenti parametri:

   - actionKey: l'identificatore del tipo di notifica
   - data: i dati aggiuntivi che devono essere inviati insieme alla notifica
   - userIDs: un'array contenente gli ID degli utenti a cui la notifica dev'essere inviata
   - callback: una funzione che viene eseguita nel momento in cui il processo di invio è terminato o qualora si verifichi un errore.

   Il metodo riportato nel codice 4.6 carica gli identificatori dei dispositivi (endpoints) associati agli utenti (da riga 4 a 6), serializza l'oggetto dei dati aggiuntivi (riga 12) e invia la notifica a tutti gli endpoints caricati (da riga 16 a 22).

```
oThis.publish('NOTIFY_ITEM_LIKED', { item_id: item.id,
    item_like_id: itemLike.id }, [item.user]);
```

Codice 4.5: Chiamata della funzione che pubblica una notifica push

```
publish: function(actionKey, data, userIDs, callback) {
   var endpointArns = [];
   async.series({
     endpoints: function(callback) {
        ... //carica gli endpoints associati agli utenti
     },
     publish: function() {
        if (endpointArns.length === 0) {
          return callback(null, true);
        }
        var params = {
          Message: JSON.stringify(_.extend(data, { action:
              actionKey })),
          TargetArn: null,
        };
        var sns = new AWS.SNS();
        _.each(endpointArns, function(endpointArn, index) {
          params.TargetArn = endpointArn;
          sns.publish(params, function(err, data) {
            if (err) return callback(err);
         if (index+1 == endpointArns.length) callback(null,
             true);
          });
        });
      }
    }, function(err, data) {
      if (err) return callback(err);
      callback(null, true);
    });
}
```

Codice 4.6: Implementazione della funzione che pubblica le notifiche push in Node.js

b) Amazon SNS consegna il messaggio al servizio di messaggistica push corrispondente, in questo caso GCM.

c) GCM invia il messaggio al dispositivo, l'applicazione lo riceve grazie a un WakefulBroadcastReceiver in ascolto, il quale lancia un IntentService che ne elabora il contenuto (codice 4.7). Viene recuperata l'actionKey che identifica il tipo di notifica (riga 8) e istanziata la classe relativa a quel comando (riga 10 e 11). Se esiste una corrispondenza, viene invocato il metodo execute (vedi codice 4.8 riga 3), in caso contrario un messaggio di log segnala l'errore.

```
protected void onHandleIntent(Intent intent) {
  ... //controlli vari
  String extraDefault = extras.getString("default");
  try {
    JSONObject json = new JSONObject(extraDefault);
    String action = json.optString("action");
    if (!"".equals(action)) {
      GCMCommand command = MESSAGE_RECEIVERS
        .get(action.toLowerCase());
      if (command != null) {
        command.execute(this, action, json);
      } else {
        LOGE(TAG, "Unknown command received: " + action);
      }
    } else {
      LOGE(TAG, "Message received without command action");
    }
  } catch (JSONException e) {
    e.printStackTrace();
  }
}
```

Codice 4.7: Gestione del messaggio di GCM sull'applicazione

Ogni classe che viene istanziata nel codice 4.7 estende NotifyCommand (codice 4.8). All'interno del metodo execute viene creato l'intento che avvia l'activity principale dell'applicazione e il builder della notifica (riga 4 e 5). Questi due oggetti vengono configurati con i parametri comuni a tutti i tipi di notifiche(da riga 6 a 10) e passati al metodo build (riga 11). Quest'ultimo è implementato da tutte le sottoclassi che così possono aggiungere, in base alla loro tipologia, le informazioni necessarie alla corretta visualizzazione della notifica, scaricandole dal server. Una volta terminate le operazioni asincrone (chiamata alle API e/o creazione di un bitmap per l'immagine), viene invocato il metodo sendNotification che imposta gli ultimi dati (riga 14) e invia la notifica (riga 17).

```
public abstract class NotifyCommand extends GCMCommand {                1
  @Override                                                             2
  public void execute(Context context, String type, JSONObject          3
      json) {
    Intent intent = new Intent(context, HostActivity.class);            4
    Builder builder = new Builder(context);                             5
    intent.putExtra(EXTRA_ACTION, getID());                             6
    builder.setWhen(System.currentTimeMillis())                         7
      .setContentTitle(...)                                             8
      .setSmallIcon(R.drawable.ic_stat_notification)                    9
      .setAutoCancel(true);                                             10
    build(context, builder, intent, json);                              11
    }                                                                   12
  protected void sendNotification(Context context, Builder              13
      builder, Intent intent) {
    builder.setContentIntent(PendingIntent                              14
      .getActivity(context, getID(), intent,                            15
        PendingIntent.FLAG_CANCEL_CURRENT));                            16
    ((NotificationManager) context                                      17
      .getSystemService(Context.NOTIFICATION_SERVICE))                  18
        .notify(getID(), builder.build());                              19
```

```
    }                                                          20
  protected abstract void build(Context context,               21
    Builder builder, Intent intent, JSONObject json);          22
  protected abstract int getID();                              23
}                                                              24
```

Codice 4.8: Classe NotifyCommand e relativi metodi

Una volta ricevuto l'intento, l'activity carica il fragment corrispondente alla tipologia di notifica, passandogli i dati contenuti.

```
private void checkIncomingIntent(Intent intent, boolean        1
    withAnimation) {
  switch (intent.getIntExtra(EXTRA_ACTION)) {                  2
    case NOTIFY_ITEM_LIKED:                                    3
      addFragment(new ItemViewFragmentFactory()                4
        .setItem((Item) intent.getParcelableExtra(EXTRA_ITEM)) 5
          .build(), withAnimation);                            6
    break;                                                     7
    ...                                                        8
  }                                                            9
}                                                              10
```

Codice 4.9: Caricamento del fragment corrispondente alla notifica

4.3.4 Servizio di tracking in background

In questa sezione viene approfondito il funzionamento della notifica che avvisa l'utente se sono presenti oggetti limitrofi alla sua posizione. Essa è stata implementata utilizzando un servizio in background che controlla costantemente la posizione del dispositivo e la comunica al server in caso di cambiamenti significativi.

Il Service viene avviato grazie a uno specifico BroadcastReceiver registrato sul Manifest per ricevere una serie di eventi:

- com.xxx.wishlist.receiver.usertracker.START
- android.intent.action.BOOT_COMPLETED
- android.intent.action.BATTERY_LOW
- android.intent.action.BATTERY_OKAY

Il primo è utilizzato direttamente dall'applicazione per lanciare il servizio a comando, gli altri 3 sono eventi generati dal sistema operativo e vengono utilizzati rispettivamente per riavviare il servizio dopo un reboot del dispositivo, avvisare il servizio che la batteria è scarica, informarlo che lo stato della batteria è di nuovo sopra la soglia minima.

Il Service viene avviato attraverso il comando onStartCommand, configurato in modalità START_STICKY. Questo gli permette di non essere vincolato a un componente dell'applicazione (come nel caso del bind) e continuare la sua esecuzione in background per un tempo indefinito.

Come si può vedere dal codice 4.10 la funzione di aggiornamento della posizione permette di ricevere ogni 30sec (riga 2) nuove posizioni che possono provenire da altre applicazione/servizi. In ogni caso ogni 60sec viene richiesto un nuovo aggiornamento (riga 3). Non ho previsto l'utilizzo del GPS, poichè trattandosi di un servizio sempre attivo, il consumo della batteria sarebbe risultato inaccettabile. Ho scelto perciò di configurarlo nella modalità "balanced power", che utilizza le reti WiFi e i ripetitori per cellulari, portando la precisione a circa 100m. Quando la carica della batteria è troppo bassa viene comunque mantenuta la ricezione di aggiornamenti da altre fonti, mentre viene sospesa la richiesta di nuove posizioni ogni 60sec: una volta ricaricata la batteria questa funzione viene ripresa automaticamente. Lo spostamento minimo perchè una nuova posizione venga presa in considerazione è di 250m.

```
locationRequest = LocationRequest.create()
  .setFastestInterval(30000)
  .setInterval(60000)
  .setPriority(LocationRequest.PRIORITY_BALANCED_POWER_ACCURACY)
  .setSmallestDisplacement(250);
```

Codice 4.10: Configurazione degli aggiornamenti di localizzazione

Prima di comunicare la nuova posizione al server viene effettuato un ultimo controllo, che valuta se quest'ultima sia preferibile a quella accettata precedentemente. Per questa funzionalità mi sono servito del metodo isBetterLocation presente sulla guida ufficiale di Android all'indirizzo: `http://developer.android.com/guide/topics/location/strategies.html`.

4.4 Utilizzo offline e sincronizzazione

Permettere l'utilizzo offline delle funzionalità principali dell'applicazione si è rivelata una sfida piuttosto complessa. Le problematiche da affrontare sono legate alla difficoltà di gestire i due eventi chiave che si possono riscontrare nell'uso comune: il momento di rottura della connessione e, conseguentemente, la gestione dell'utilizzo offline e il momento della riconnessione, con l'eventuale criticità della ri-sincronizzazione fra le parti che potrebbero aver vissuto interazioni completamente diverse.

La gestione dell'utilizzo offline può essere ridotta, in sintesi, a tre principali componenti che interagiscono tra loro: il Persister, che svolge il compito di adapter fra l'applicazione e la gestione dei dati; il SyncHelper, che si occupa di sincronizzare tutti gli elementi presenti sul database locale che risultino desincronizzati rispetto ai dati originari delle API; il database, che insieme al suo gestore DAO fornisce l'accesso in lettura e scrittura sui dati locali. Per ottimizzare l'efficienza di questo sistema, e quindi la user experience, è stato aggiunto un broadcast receiver (NetworkChangeReceiver) che

avvia il processo di sincronizzazione attraverso il SyncHelper al ritorno della connettività.

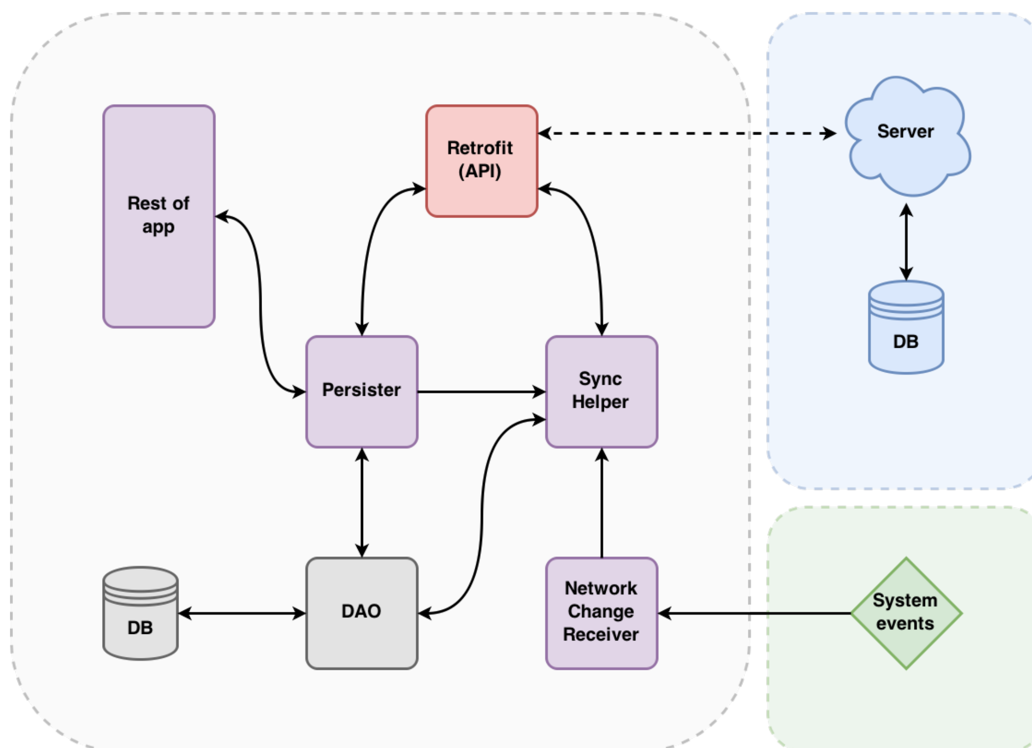

Figura 4.5: Architettura della persistenza e sincronizzazione

4.4.1 Persister

La classe Persister, come già evidenziato, funge da adapter fra l'applicazione e il sistema interno di persistenza dei dati. Nello specifico fornisce dei metodi CRUD per la gestione degli elementi persistibili. I metodi forniti sono:

- getAllWishlists
- createWishlist
- deleteWishlist
- updateWishlist

- getAllItems
- createItem
- deleteItem
- updateItem

Data la struttura molto simile di questi elementi, verranno di seguito riportati ed analizzati getAllWishlists, dal funzionamento analogo a getAllItems, e createWishlist, dalla struttura simile (ma comportamento diverso) ai restanti metodi di gestione del singolo dato.

Come possiamo vedere dal codice 4.11, per prima cosa viene controllato lo stato della connessione (riga 2): in caso positivo, viene lanciata la sincronizzazione attraverso il SyncHelper (riga 3), altrimenti vengono restituiti i dati dal database locale (riga 21). Nel caso si verifichi il primo dei due scenari, una volta completata la sincronizzazione dei dati, il Persister può procedere con la vera chiamata GET al server (riga 6): in questo modo vi è la sicurezza che non ci siano differenze tra le due versioni. Se la chiamata ha successo, all'interno della callback success vengono effettuate due operazioni: refresh dal database locale attraverso un'AsyncTask (riga 10), che sostituisce i "vecchi" dati con quelli "nuovi", e restituzione delle Wishlist al chiamante.

```
public void getAllWishlists(final ApiCallback<List<Wishlist>>       1
    callback) {
  if (NetworkUtils.isNetworkAvailable(context)) {                   2
    new SyncHelper(context).performSync(new SyncHelperListener() {  3
      @Override                                                     4
      public void onSyncCompleted() {                               5
        new WishlistList( ApplicationData.getInstance().getUser(),  6
            new ApiCallback<List<Wishlist>>() {                     7
          @Override                                                 8
          public void success(List<Wishlist> wishlists) {           9
            new RefreshDB().execute(wishlists.toArray(...));        10
            callback.success(wishlists);                            11
```

```
          }                                                    12
          @Override                                            13
          public void failure(ApiError apiError) {             14
            callback.failure(apiError);                        15
          }                                                    16
        });                                                    17
      }                                                        18
    });                                                        19
  } else {                                                     20
    callback.success(wds.getAll());                            21
  }                                                            22
}                                                              23
```

Codice 4.11: Metodo getAllWishlists del Persister

Come evidenziato in precedenza, i metodi di creazione, modifica e rimozione delle liste e degli oggetti, possiedono una struttura pressoché identica. Se andiamo ad analizzare il codice 4.12, possiamo notare una logica simile a quella del metodo get sopra riportato, ma senza la sincronizzazione dei dati attraverso il SyncHelper. Anche in questo caso viene controllato lo stato della connessione (riga 2): se presente, viene effettuata immediatamente la creazione della lista sul server (riga 3), altrimenti viene creata sul database locale (riga 15).

```
public void createWishlist(final Wishlist wishlist, final          1
    ApiCallback<Wishlist> callback) {
    if (NetworkUtils.isNetworkAvailable(context)) {                2
        new WishlistCreate(wishlist, new ApiCallback<Wishlist>() {  3
            @Override                                              4
            public void success(Wishlist w) {                      5
                wds.create(w);                                     6
                callback.success(w);                               7
            }                                                      8
            @Override                                              9
```

```
            public void failure(ApiError apiError) {
                callback.failure(apiError);
            }
        });
    } else {
        int id = wds.create(wishlist);
        callback.success(wds.get(id));
    }
}
```

Codice 4.12: Metodo createWishlist del Persister

4.4.2 SyncHelper

La classe SyncHelper si occupa di effettuare la sincronizzazione completa dei dati desincronizzati. La procedura in sè consiste nell'esecuzione di una serie di metodi, contenenti operazioni asincrone, controllati attraverso l'uso di un semaforo.

Il metodo performSync, visualizzabile nel codice 4.13, è l'unico pubblico di tutta la classe (escluso il costruttore): esso prende in ingresso un'interfaccia su cui è dichiarato il metodo onSyncCompleted (codice 4.11 riga 3) e dà avvio al processo di sincronizzazione, invocando ogni metodo singolarmente.

```
public void performSync(final SyncHelperListener listener) {
    this.listener = listener;
    syncDeletedWishlists();
    syncUpdatedWishlists();
    syncCreatedWishlists();
    syncDeletedItems();
    syncUpdatedItems();
    syncCreatedItems();
}
```

Codice 4.13: Metodo performSync del SyncHelper

Di seguito è riportato il metodo syncDeletedWishlists (codice 4.14), scelto fra tutti quelli presenti come modello per spiegare la struttura e il funzionamento di tutti gli altri. Per prima cosa viene recuperata, dal database locale, la lista delle Wishlist eliminate dall'utente durante il periodo di disconnessione (riga 2). Se non sono presenti liste da sincronizzare, allora viene invocato il metodo finish (codice 4.15) e il syncDeletedWishlists ritorna immediatamente. La seconda parte è composta da un ciclo che elimina tutte le liste cancellate dal server: il tutto viene regolato da un secondo semaforo interno al metodo (riga 3) che blocca l'invocazione del finish, finchè non sono terminate tutte le chiamate all'API (riga 14 e 20).

```
private void syncDeletedWishlists() {                                    1
    final List<Wishlist> deletedWishlists = wds.getDeleted();            2
    final AtomicInteger wtodo = new                                      3
        AtomicInteger(deletedWishlists.size());
    if (deletedWishlists.size() == 0) {                                  4
        finish();                                                        5
        return;                                                          6
    }                                                                    7
    for (int i = 0; i < deletedWishlists.size(); i++) {                  8
        final Wishlist w = deletedWishlists.get(i);                      9
        new WishlistDelete(w, new ApiCallback<Wishlist>() {              10
            @Override                                                    11
            public void success(Wishlist wishlist) {                     12
                wds.delete(w.id);                                        13
                if (wtodo.decrementAndGet() == 0) {                      14
                    finish();                                            15
                }                                                        16
            }                                                            17
            @Override                                                    18
            public void failure(ApiError apiError) {                     19
                if (wtodo.decrementAndGet() == 0) {                      20
                    finish();                                            21
```

```
            }                                                           22
        }                                                               23
    });                                                                 24
  }                                                                     25
}                                                                       26
```

Codice 4.14: Metodo syncDeletedWishlists del SyncHelper

Questo metodo (codice 4.15) controlla semplicemente lo stato del semaforo, se uguale a 0, allora viene lanciata la callback per avvertire il chiamate che la sincronizzazione è stata completata.

```
private void finish() {                                                  1
    if (semaphore.decrementAndGet() == 0) {                              2
        listener.onSyncCompleted();                                      3
    }                                                                    4
}                                                                        5
```

Codice 4.15: Metodo finish del SyncHelper

4.4.3 Database

La gestione dei dati locali deve avere la stessa espressività delle API fornite da Retrofit (che rispecchia i modelli delle API RESTful fornite dal server). Per questo motivo si è scelto di utilizzare un pattern DAO per dare al database SQLite fornito da Android un'interfaccia che porti le operazioni SQL standard allo stesso livello di astrazione delle API.

Le classi che gestiscono la persistenza dei dati sull'applicazione sono quattro:

- **DatabaseHelper** è una classe che estende SQLiteOpenHelper[4] e interagisce direttamente con il database SQLite. Nello specifico, si occupa della creazione, dell'aggiornamento (cambio di versione) e dell'acceso ai dati in lettura e scrittura.

[4] http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html

```
private static final String CREATE_TABLE_WISHLIST =
   "CREATE TABLE " + TABLE_WISHLIST + "("
   + COLUMN_ID + " INTEGER PRIMARY KEY, "
   + COLUMN_EDITED + " BOOLEAN DEFAULT 0, "
   + COLUMN_DELETED + " BOOLEAN DEFAULT 0, "
   + COLUMN_JSON + " TEXT NOT NULL" + ");";

private static final String CREATE_TABLE_ITEM =
   "CREATE TABLE " + TABLE_ITEM + "("
   + COLUMN_ID + " INTEGER PRIMARY KEY, "
   + COLUMN_WISHLIST_ID + " INTEGER REFERENCES "
      + TABLE_WISHLIST + " (" + COLUMN_ID + ") "
      + "ON DELETE CASCADE ON UPDATE NO ACTION, "
   + COLUMN_EDITED + " BOOLEAN DEFAULT 0, "
   + COLUMN_DELETED + " BOOLEAN DEFAULT 0, "
   + COLUMN_JSON + " TEXT NOT NULL" + ");";
```

Codice 4.16: Query per la creazione del database SQLite

- **DatabaseGate** è un singleton che fornisce le operazioni CRUD (insert, query, update e delete) che permettono ai DAO di comunicare col database. Esso contiene un'istanza di DatabaseHelper e viene utilizzato, per evitare problemi di concorrenza quando ci sono più threads che competono per il lock.[5]

- **WishlistDataSource** fornisce al Persister e al SyncHelper tutti i metodi necessari per manipolare le Wishlist salvate sul database SQLite sottoforma di JSON. La differenziazione delle Wishlist create offline, rispetto a quelle già presenti sul server, avviene direttamente dentro il metodo 4.17. Come è possibile vedere dalla riga 3, viene controllato l'ID della Wishlist in questione: se è uguale a 0, allora viene identificata come prodotta offline,

[5] Gal14.

quindi il metodo 4.19 le assegna un ID negativo, e utilizzandolo come primary key, la salva sul database. Le Wishlist con ID diverso da 0 vengono salvate direttamente. In questo modo è possibile recuperare con facilità le liste da sincronizzare, attraverso una semplice query (codice 4.18).

```
@Override                                                              1
public int create(Wishlist wishlist) {                                 2
  if (wishlist.id == 0) {                                              3
      wishlist.id = getNewNegativeId(TABLE_WISHLIST);                  4
  }                                                                    5
  ContentValues values = new ContentValues();                          6
  values.put(COLUMN_ID, wishlist.id);                                  7
  values.put(COLUMN_JSON, new Gson().toJson(wishlist));                8
  if (gate.insert(TABLE_WISHLIST, values) == -1) {                     9
      LOGE(TAG, "...");                                                10
  }                                                                    11
  return wishlist.id;                                                  12
}                                                                      13
```

Codice 4.17: Metodo create di WishlistDataSource

```
@Override                                                              1
public List<Wishlist> getUnsync() {                                    2
  List<Wishlist> wishlists = new ArrayList<>();                        3
  String selectQuery = "SELECT " + COLUMN_JSON                         4
    + " FROM " + TABLE_WISHLIST                                        5
    + " WHERE " + COLUMN_ID + " < 0";                                  6
  Cursor c = gate.query(selectQuery);                                  7
  if (c.moveToFirst()) {                                               8
    do {                                                               9
      String json = c.getString(c.getColumnIndex(COLUMN_JSON));        10
      Wishlist wishlist = new Gson()                                   11
        .fromJson(json, Wishlist.class);                               12
      wishlists.add(wishlist);                                         13
```

```
        } while (c.moveToNext());                                  14
    }                                                              15
    c.close();                                                     16
    gate.close();                                                  17
    return wishlists;                                              18
}                                                                  19
```

Codice 4.18: Metodo getUnsync di WishlistDataSource

```
protected int getNewNegativeId(final String table) {               1
    String selectQuery = "SELECT " + COLUMN_ID                     2
            + " FROM " + table                                     3
            + " ORDER BY " + COLUMN_ID                             4
            + " ASC LIMIT 1";                                      5
    Cursor c = gate.query(selectQuery);                            6
    if (c.moveToFirst()) {                                         7
        int id = c.getInt(c.getColumnIndex(COLUMN_ID));            8
        if (id >= 0) {                                             9
            return -1;                                             10
        }                                                          11
        return id - 1;                                             12
    }                                                              13
    return -1;                                                     14
}                                                                  15
```

Codice 4.19: Metodo getNewNegativeId di DataSource

- **ItemDataSource** fornisce in maniera analoga lo stesso tipo di servizio del precedente, ma relativamente agli Item.

Capitolo 5

Validazione

5.1 L'importanza delle notifiche push

Esaminando diverse ricerche riguardanti le notifiche push, si possono ricavare dati interessanti sulla loro funzione e la loro utilità.

Mediamente il 52% degli utenti abilita le notifiche push, con una percentuale più alta per Android (59%) in cui sono già attive di default, rispetto ad iOS (46%) in cui ne viene richiesta l'accettazione al primo utilizzo.
Le notifiche possono essere di due tipi:

- "broadcast" che hanno un contenuto generico e vengono inviate indiscriminatamente a tutti gli utenti.

- "segmented" che hanno un contenuto mirato e vengono inviate a specifici segmenti di utenti in base alle loro caratteristiche, comportamento, preferenze.

Gli sviluppatori che hanno scelto di utilizzare notifiche segmented hanno un aumento significativo dell'"open rate" (% di apertura della notifica e conseguentemente dell'applicazione) rispetto a chi utilizza le notifiche ti tipo broadcast: queste ultime infatti vengono aperte dal 3% degli utenti, mentre le segmented vengono aperte dal 7%. Per quanto riguarda il "conversion rate" (% di utenti che una volta aperta l'applicazione, completa l'azione

relativa alla notifica) la diversa efficacia tra i due tipi di notifiche è ancora più evidente: le broadcast ottengono un conversion rate del 15%, mentre le segmented raggiungono il 54%. Per maggiore chiarezza esprimo in termini numerici il significato di questi dati:

- su 100 utenti che ricevono una notifica broadcast solo 0,45 utenti completeranno l'azione
- su 100 utenti che ricevono una notifica segmented l'azione sarà completata da 8,1 di essi[1]

L'utilizzo delle notifiche push appare pertanto appropriato nel favorire il coinvolgimento degli utenti durante i periodi di inattività. Gli utenti con le notifiche abilitate lanciano l'applicazione mediamente 88% di volte in più rispetto a quelli con le notifiche disabilitate. Questo dato riguarda la media di tutte le tipologie di applicazioni, con un massimo per e-commerce del 278% e un minimo per health and fitness del 34% ; le applicazioni di tipo social network si attestano sul 60%.

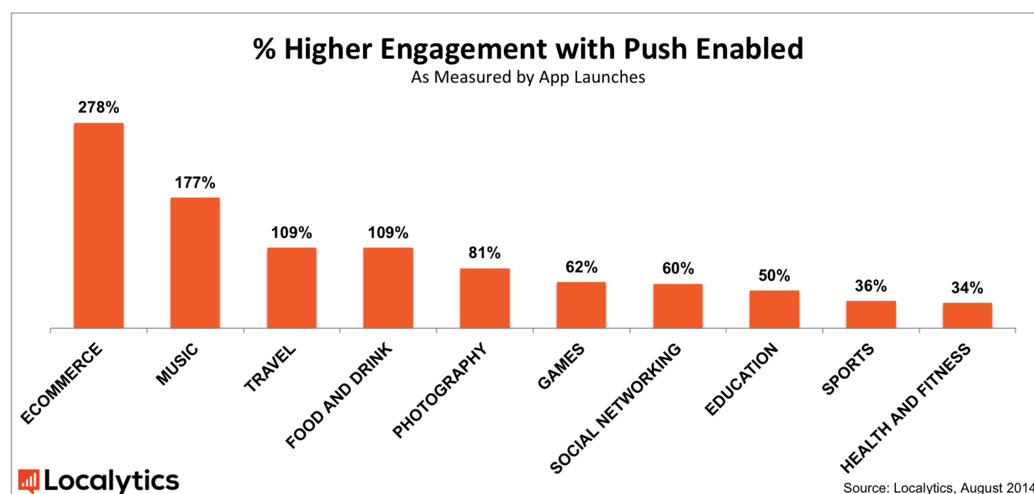

Figura 5.1: Incremento del numero di lanci dell'applicazione con le notifiche push abilitate

fonte: Localytics, Agosto 2014

[1] Loc14a.

Uno degli obiettivi principali dello sviluppatore è quello di minimizzare la percentuale di abbandono dell'applicazione. Anche a questo riguardo l'utilizzo delle notifiche push si rivela efficace. Nel grafico 5.2 si vede come l'11% degli utenti con notifiche abilitate abbandona l'applicazione dopo il primo utilizzo, contro il 21% dei non abilitati. Viceversa nella colonna delle applicazioni utilizzate 11 o più volte si nota come gli utenti con le notifiche abilitate vi siano rappresentati con un 53%, contro il 38% dei non abilitati. È evidente quindi come gli utenti abilitati abbiano una minore frequenza di abbandono e tendono a spendere più tempo sull'applicazione rispetto agli altri.

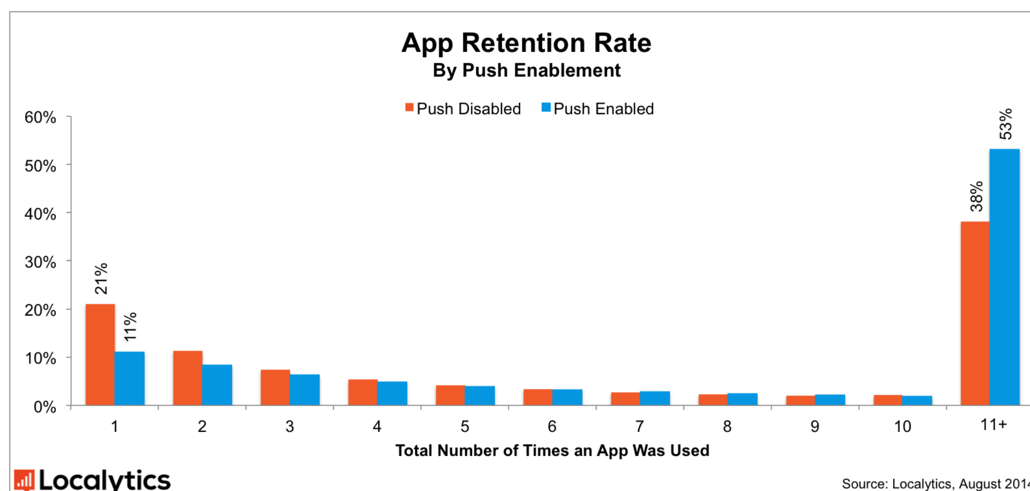

Figura 5.2: Numero di volte in cui l'applicazione viene utilizzata
fonte: Localytics, Agosto 2014

Un'altro parametro per verificare l'efficacia delle notifiche push è il perdurare nel tempo dell'utilizzo dell'applicazione. Come si vede dal grafico 5.3 il 62% degli utenti con le notifiche abilitate ritorna ad utilizzare l'applicazione nel mese successivo alla prima sessione, contro il 32% degli utenti non abilitati. Questo gap andrà ad aumentare nel tempo tanto che dopo 4 mesi oltre 1/3 degli utenti abilitati è ancora attivo, contro il 14% dei non abilitati.[2]

[2]Loc14b.

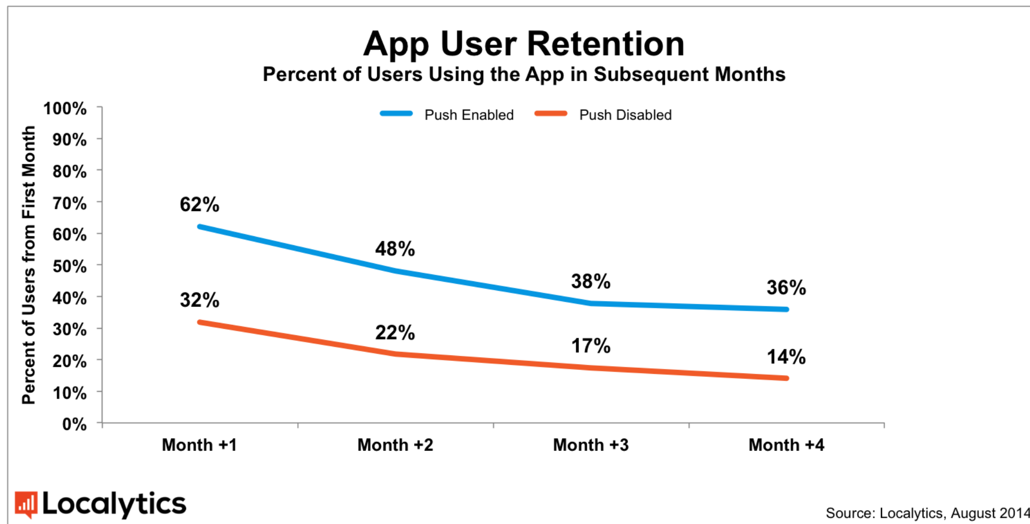

Figura 5.3: Tasso di utilizzo dell'applicazione nei mesi successivi
fonte: Localytics, Agosto 2014

5.2 Analisi delle prestazioni

In questa sezione vengono analizzate le prestazioni di alcune funzionalità di Wishlist. Mi sono servito di un'applicazione chiamata PowerTutor[3] per monitorare il consumo della batteria tenendo conto dell'utilizzo di tre componenti: display LCD, CPU, 3G. Il dispositivo su cui sono stati effettuati i test è un LG Nexus 5 con la luminosità del display impostata al 50%. Ho scelto di monitorare le operazioni utilizzando la rete 3G invece del Wifi, per rispecchiare la più frequente modalità di utilizzo dell'applicazione.

La prima analisi mostra l'energia spesa per la ricezione ed elaborazione di una notifica push; nello specifico ho considerato quella utilizzata come esempio nella sezione 4.3.3: nuovo commento. Il consumo della batteria è suddiviso tra CPU e 3G, rispettivamente di 328,0 mJ e 2,4 J. Confrontando i valori con le operazioni eseguite sul codice, è facilmente intuibile come questi consumi siano strettamente legati alla creazione del bitmap e alla chiamata GET per scaricare i dati completi.

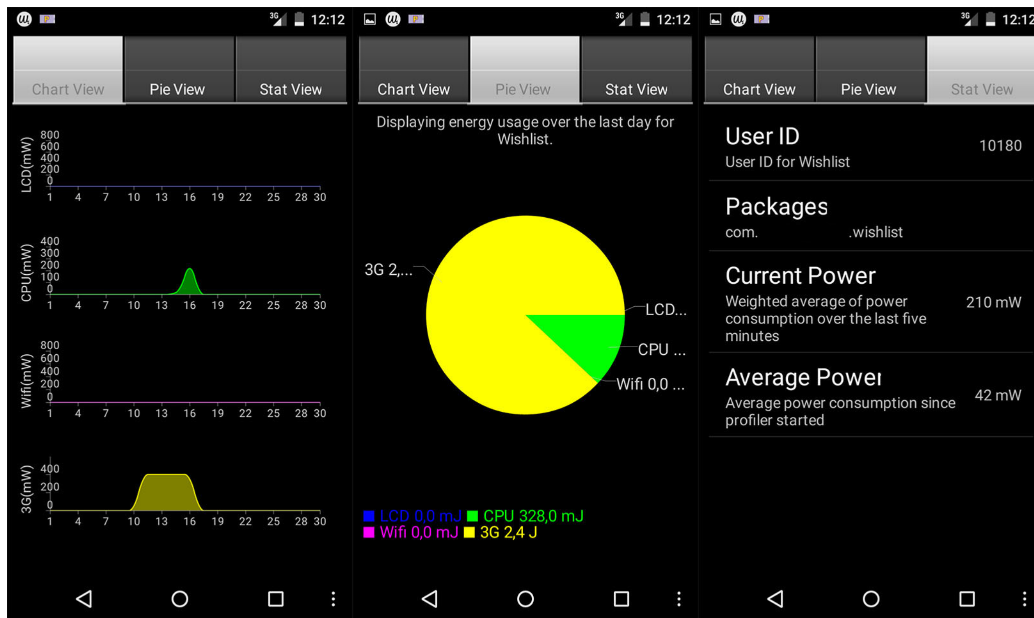

Figura 5.4: Consumo energetico per la ricezione di una notifica push
fonte: PowerTutor

[3]https://play.google.com/store/apps/details?id=edu.umich.PowerTutor

La seconda analisi si concentra sul consumo del servizio di tracking in background descritto nella sezione 4.3.4 di questo elaborato: ogniqualvolta una nuova posizione viene trovata e inviata al server, il consumo della CPU è di 13,0 mJ, quello del 3G invece, 2,4 J. Possiamo notare come il costo di ogni singolo aggiornamento sia molto ridotto, e limitato quasi esclusivamente all'operazione di invio dei dati.

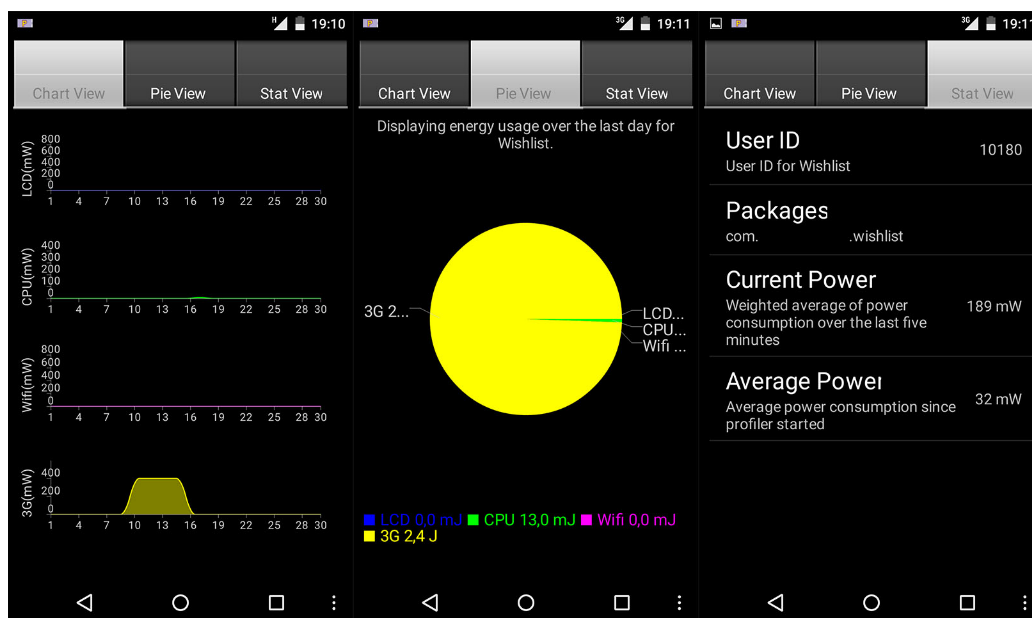

Figura 5.5: Consumo energetico del servizio di tracking
fonte: PowerTutor

La terza analisi, prende in esame l'aggiunta di un nuovo oggetto attraverso l'activity specifica che ne gestisce la creazione. Ho ritenuto importante monitorare questa azione, poichè rappresenta in qualche modo il fulcro delle funzionalità riservate all'utente. Per ottenere una valutazione precisa e realistica ho effettivamente eseguito l'operazione di inserimento di un nuovo oggetto, compilando il form in maniera completa e aggiungendo una foto, un luogo e una categoria. In questo caso il consumo totale è di 58,4 J che si divide in 41,4 J del display, 9,8 J del 3G, 5,0 J della CPU e 2,2 J del GPS (quest'ultimo viene monitorato in una schermata a parte, riservata al sistema). La maggior parte del consumo dipende dall'utilizzo del display, infatti in questa

analisi viene considerata una sezione dell'applicazione provvista di interfaccia grafica. Possiamo anche notare un alto consumo da parte della CPU e del 3G, sicuramente causato dall'elaborazione e dall'upload dell'immagine.

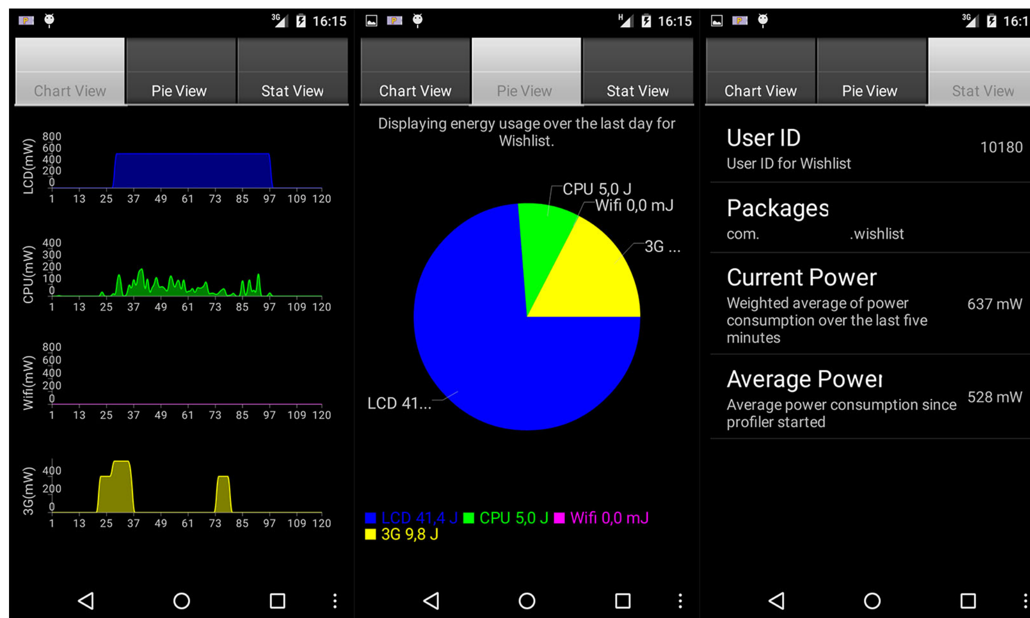

Figura 5.6: Consumo energetico per l'aggiunta di un nuovo oggetto fonte: PowerTutor

Il consumo del GPS è limitato, dato che viene attivato solo per pochi secondi, giusto il tempo di recuperare una posizione precisa e comunicarla al server (utilizzata per la lista dei luoghi).

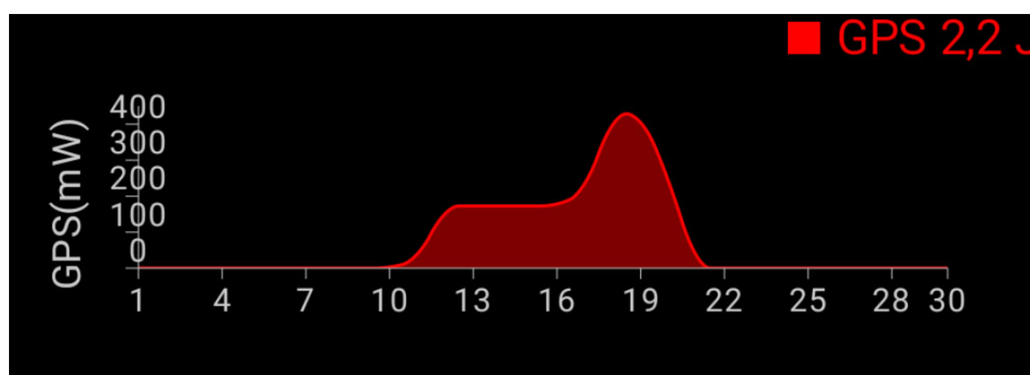

Figura 5.7: Consumo energetico del GPS per l'aggiunta di un nuovo oggetto fonte: PowerTutor

5.3 Schermate

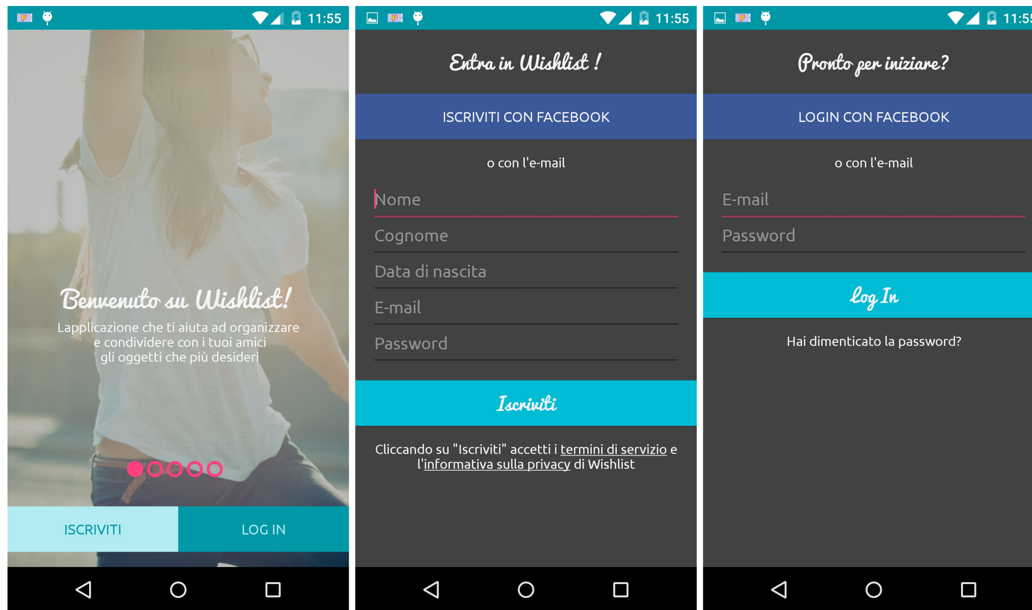

Figura 5.8: Schermate iniziali, da sinistra verso destra: welcome, registrazione, login.

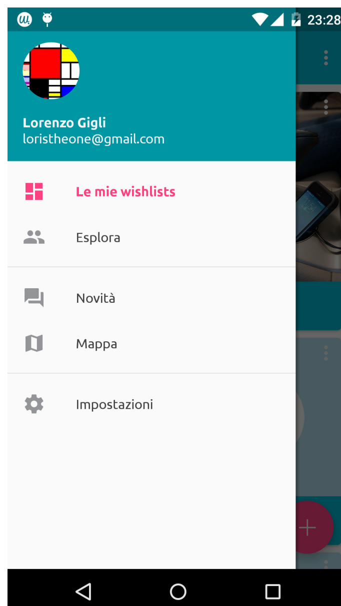

Figura 5.9: Schermata del menu laterale

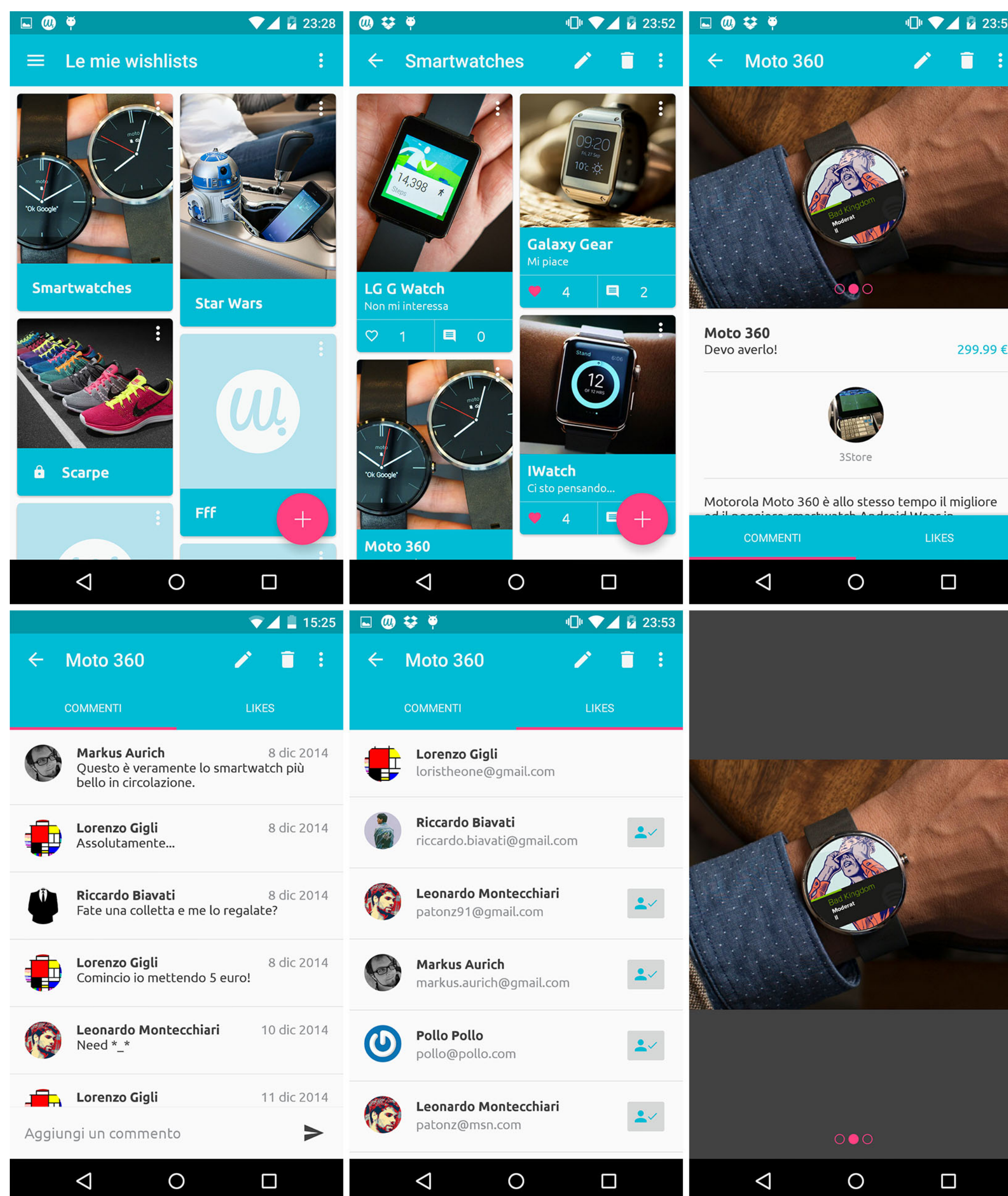

Figura 5.10: Schermate principali, da sinistra verso destra e dall'alto al basso: lista delle wishlists, lista degli oggetti di una wishlist, dettaglio dell'oggetto, commenti di un oggetto, likes di un oggetto, galleria.

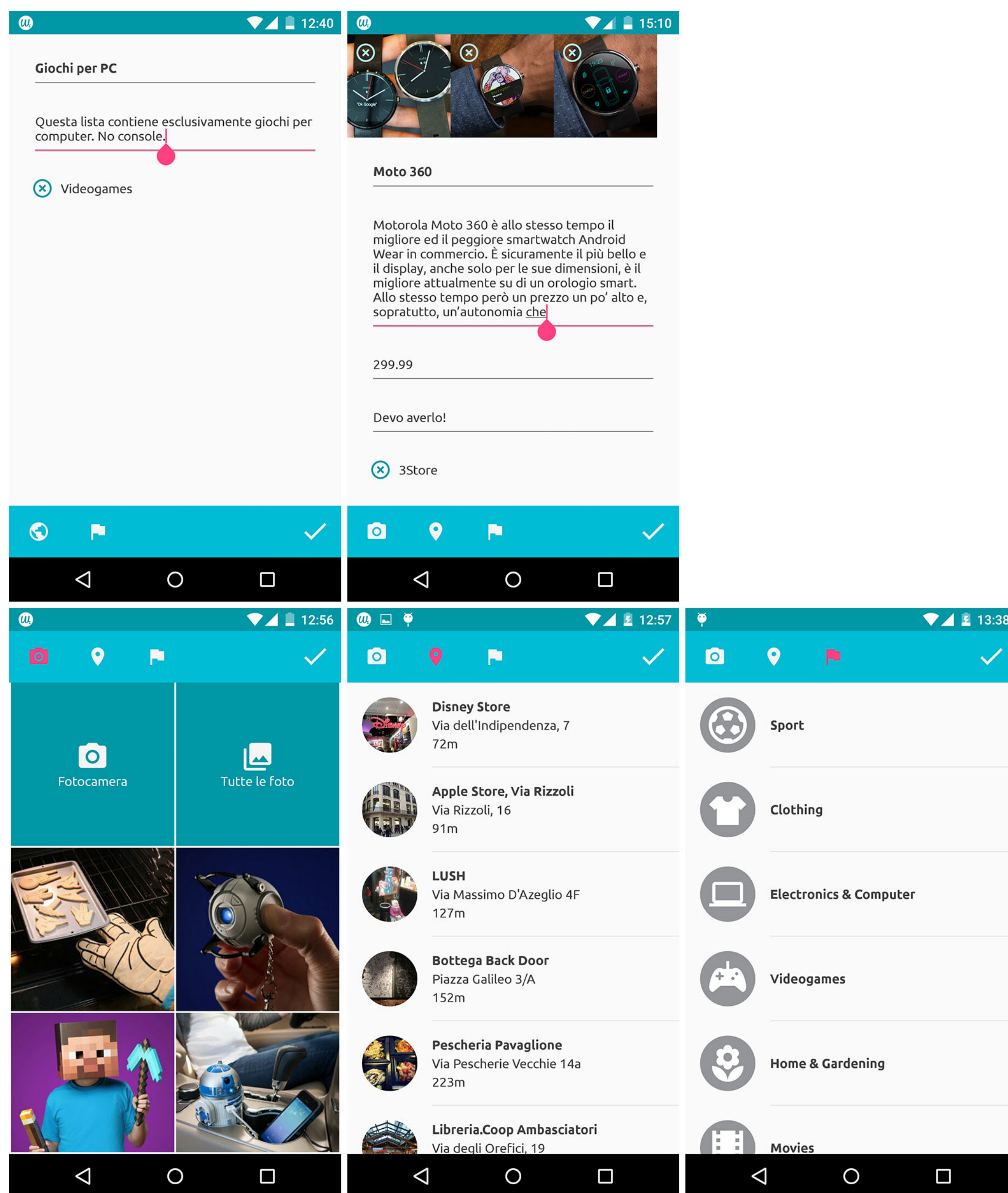

Figura 5.11: Schermate di creazione, da sinistra verso destra e dall'alto al basso: creazione di una wishlist, creazione di un oggetto, selezione delle immagini, selezione del luogo, selezione della categoria.

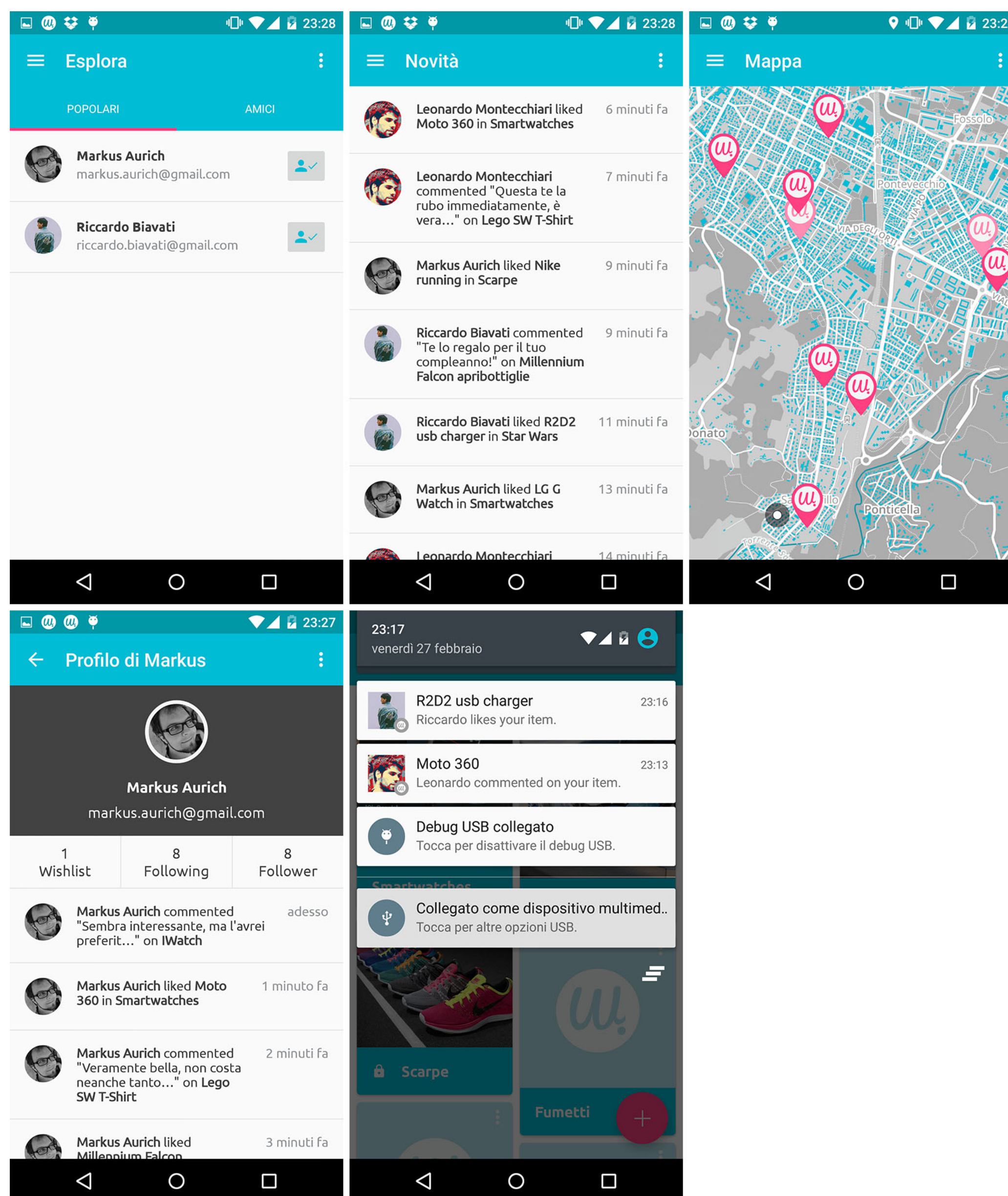

Figura 5.12: Schermate social, da sinistra verso destra e dall'alto al basso: esplora, attività, mappa degli oggetti, profilo utente, alcune notifiche.

Conclusioni

In questo elaborato viene presentata la realizzazione di una piattaforma di mobile social network per la creazione, gestione e condivisione di un insieme di "wishlist". Essa è basata su un'architettura di tipo client-server: il server è composto da un'applicazione scritta in Node.js e relativo database MySQL, mentre il client consiste in una applicazione Android.

Per prima cosa mi sono dedicato a una necessaria fase preliminare di studio degli aspetti scientifici ed economici riguardanti lo sviluppo di un'applicazione destinata alla commercializzazione. In seguito a questa analisi ho scelto Android come sistema operativo, vista la sua enorme diffusione, le prospettive per i prossimi anni e la mia buona conoscenza del sistema.

Il lavoro svolto si è rivelato molto lungo e impegnativo, infatti si trattava di sviluppare un'applicazione ambiziosa, strutturalmente complessa, composta da numerose schermate e diverse funzionalità, tanto da richiedere 13613 righe di codice per l'applicazione Android (.java e .xml) e 4154 per il server (.js). Ho dedicato un'attenzione particolare alla progettazione dell'interfaccia grafica, trattandosi di un'applicazione destinata a un'utenza numerosa ed elementare, abituata ad applicazioni graficamente accattivanti, di semplice e intuitivo utilizzo e sostanzialmente prive di difetti. Un altro aspetto particolarmente impegnativo è stata la gestione delle notifiche push geolocalizzate, che mi ha imposto di affrontare e risolvere problematiche di particolare attualità. Era inoltre indispensabile offrire all'utenza la possibilità di utilizzo offline delle principali funzionalità: questa caratteristica è imprescindibile in un'applicazione di questo tipo, ma la sua realizzazione si è rivelata di par-

ticolare complessità, poichè richiede la sincronizzazione dei dati tra client e server.

Riguardo gli sviluppi futuri il mio primo obiettivo è quello di portare l'applicazione sugli stores nel più breve tempo possibile. Per arrivare a questo sono necessarie ancora alcune fasi: realizzazione dell'applicazione per iOS, introduzione delle Google Analytics (per monitorarne l'andamento e il comportamento degli utenti), fase di "beta" (per rilevazione bugs e ottimizzazione), valutazione delle implicazioni legali, compresa la stesura dei "Termini di Servizio" e della "Informativa sulla privacy".

L'esperienza è stata sicuramente molto profiqua e positiva. Il lavoro svolto infatti mi ha permesso di accrescere in maniera significativa le mie competenze professionali, in particolar modo riguardo la conoscenza e l'utilizzo del sistema Android e di altre tecnologie e servizi all'avanguardia. Inoltre il fatto che l'applicazione Wishlist sia destinata alla pubblicazione mi ha fornito una forte motivazione e senso di responsabilità nel curare il progetto in ogni dettaglio, con grande completezza e precisione: gli utenti sono esigenti, senza rendersi conto del lavoro lungo e complesso che sta dietro un software apparentemente semplice.

Appendice A

Server API

Resource Name	HTTP Verbs	HTTP Methods
Activity	list Activity	GET /activity
Category	list Category	GET /category
	view Category	GET /category/{id}
	create Category	POST /category
	update Category	PUT /category/{id}
	delete Category	DELETE /category/{id}
FoursquareVenue	list FoursquareVenue	GET /foursquare/venue
Friendship	list Friendship	GET /friendship
	view Friendship	GET /friendship/{id}
	create Friendship	POST /friendship
	update Friendship	PUT /friendship/{id}
	delete Friendship	DELETE /friendship/{id}
Item	list Item	GET /item
	view Item	GET /item/{id}
	create Item	POST /item
	update Item	PUT /item/{id}
	delete Item	DELETE /item/{id}

ItemComment	list ItemComment	GET /item_comment
	view ItemComment	GET /item_comment/{id}
	create ItemComment	POST /item_comment
	update ItemComment	PUT /item_comment/{id}
	delete ItemComment	DELETE /item_comment/{id}
ItemImage	list ItemImage	GET /item_image
	view ItemImage	GET /item_image/{id}
	create ItemImage	POST /item_image
	update ItemImage	PUT /item_image/{id}
	delete ItemImage	DELETE /item_image/{id}
ItemLike	list ItemLike	GET /item_like
	view ItemLike	GET /item_like/{id}
	create ItemLike	POST /item_like
	update ItemLike	PUT /item_like/{id}
	delete ItemLike	DELETE /item_like/{id}
Endpoint	list Endpoint	GET /endpoint
	view Endpoint	GET /endpoint/{endpoint}
	create Endpoint	POST /endpoint
	delete Endpoint	DELETE /endpoint/{endpoint}
SocialProfile	list SocialProfile	GET /social_profile
	view SocialProfile	GET /social_profile/{id}
	create SocialProfile	POST /social_profile
	update SocialProfile	PUT /social_profile/{id}
	delete SocialProfile	DELETE /social_profile/{id}
User	list User	GET /user
	sview User	GET /user/me
	view User	GET /user/{id}
	create User	POST /user
	update User	PUT /user/{id}
	delete User	DELETE /user/{id}

UserLocation	create UserLocation	POST /user_location
UserToken	create UserToken	POST /user_token
Wishlist	list Wishlist	GET /wishlist
	view Wishlist	GET /wishlist/{id}
	create Wishlist	POST /wishlist
	update Wishlist	PUT /wishlist/{id}
	delete Wishlist	DELETE /wishlist/{id}

Tabella A.1: Server API

Bibliografia

Libri

[S W94] K. Faust S. Wasserman. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994. ISBN: 9781107631052.

Articoli

[N D06] B. Mulloth N. D. Ziv. “An Exploration on Mobile Social Networking: Dodgeball as a Case in Point”. In: *International Conference on Mobile Business* (2006).

[F N09] A. Seneviratne F. Nazir J. Ma. “Time Critical Content Delivery Using Predictable Patterns in Mobile Social Networks”. In: *Computational Science and Engineering* 4 (2009).

[Sap10] A. Sapuppo. “Spiderweb: A social mobile network”. In: *Wireless Conference (EW)* (2010).

[NH11] Ping Wang Nipendra Kayastha Dusit Niyato e Ekram Hossain. “Applications, Architectures, and Protocol Design Issues for Mobile Social Networks: A Survey”. In: *Proceedings of the IEEE* 99.12 (2011).

Online

[Gar13] Gartner. *Gartner Says Mobile App Stores Will See Annual Downloads Reach 102 Billion in 2013*. [ultima vista 18.02.2015]. 2013. URL: http://www.gartner.com/newsroom/id/2592315.

[com14] comScore. *The U.S. Mobile App Report*. [ultima vista 18.02.2015]. 2014. URL: http://www.comscore.com/ita/Insights/Presentations-and-Whitepapers/2014/The-US-Mobile-App-Report.

[Flu14a] Flurry. *App Install Addiction Shows No Signs of Stopping*. [ultima vista 18.02.2015]. 2014. URL: http://www.flurry.com/blog/flurry-insights/app-install-addiction-shows-no-signs-stopping#.VOSy1vmG_vx.

[Flu14b] Flurry. *Mobile to Television: We Interrupt this broadcast (Again)*. [ultima vista 18.02.2015]. 2014. URL: http://www.flurry.com/blog/flurry-insights/mobile-television-we-interrupt-broadcast-again#.VOSyWfmG_vx.

[Gal14] Kevin Galligan. *What are the best practices for SQLite on Android?* [ultima vista 04.03.2015]. 2014. URL: http://stackoverflow.com/questions/2493331/what-are-the-best-practices-for-sqlite-on-android/3689883#3689883.

[Gar14] Gartner. *Gartner Says Sales of Smartphones Grew 20 Percent in Third Quarter of 2014*. [ultima vista 20.02.2015]. 2014. URL: http://www.gartner.com/newsroom/id/2944819.

[Loc14a] Localytics. *52% OF USERS ENABLE PUSH MESSAGING ON THEIR MOBILE DEVICES*. [ultima vista 14.02.2015]. 2014. URL: http://info.localytics.com/blog/52-percent-of-users-enable-push-messaging.

[Loc14b] Localytics. *PUSH MESSAGING DRIVES 88% MORE APP LAUNCHES*. [ultima vista 15.02.2015]. 2014. URL: http://info.localytics.

com/blog/push-messaging-drives-88-more-app-launches-for-users-who-opt-in.

[Ope14] OpenSignal. *Android Fragmentation Visualized*. [ultima vista 21.02.2015]. 2014. URL: http://opensignal.com/reports/2014/android-fragmentation.

[Wik15] Wikipedia. *Sistema operativo per dispositivi mobili*. [ultima vista 19.02.2015]. 2015. URL: http://it.wikipedia.org/wiki/Sistema_operativo_per_dispositivi_mobili.

Ringraziamenti

Ringrazio
il mio relatore Prof. Marco Di Felice che mi ha dato fiducia
il mio maestro Marco Montanari
Markus, Leo e Jacopo: i miei compagni di corso che hanno condiviso con me ansie e soddisfazioni
Rich, Anna, Giappo, Mett, JJ, Babba e Mello che mi hanno sopportato e sostenuto
tutti i miei cugini che sono sempre presenti nella mia vita