ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Corso di laurea in Ingegneria Informatica

Distribuzione Efficiente di Mobile Sensing Data tramite Modello Push e Integrazione con PubSubHubBub

Candidato:
Andrea Candini
0000217543

Relatore:
Ing. Paolo Bellavista
Correlatore:
Ing. Luca Foschini

Sessione Terza
Anno accademico
2012 – 2013

Indice

Introduzione

Lo sviluppo del mondo mobile è andato sempre incrementandosi nel corso dell'ultimo decennio. I telefoni cellulari sono passati dall'essere un bene di lusso riservato a pochi, a dispositivi portatili presenti nelle tasche di tutti noi.
Analizzando l'uso che se ne fa oggi e paragonandolo a quello dei primi anni duemila (quindi senza nemmeno andare troppo indietro nel tempo) si nota come essi abbiano assunto un ruolo sempre più centrale nella nostra quotidianità: non sono più soltanto un dispositivo telefonico per chiamare liberamente anche fuori di casa o inviare un sms ad amico quando non risponde, sono diventati dei veri e propri piccoli calcolatori, con tutto ciò che questo comporta.
Con le nostre mani ci ritroviamo a telefonare con tutta naturalezza senza più premere nemmeno un tasto fisico, appuntiamo note per non dimenticarci dell'impegno che abbiamo domani, aggiungiamo una ricorrenza al calendario (che contemporaneamente si sincronizza con il nostro account di posta), carichiamo video e musica e tanto altro ancora.
Questa sempre maggiore potenza di calcolo ha reso i dati di profilazione dell'utente finale sempre più interessanti agli occhi delle aziende, che nel corso del tempo hanno imparato a sfruttare i nostri dati per proporci esperienze sempre più personalizzate.
Quando acquistiamo un libro dallo store del nostro smartphone, stiamo dando un'indicazione su quale genere e autore ci piacciono. È quindi probabile che al nostro ritorno troveremo in prima pagina altre sue opere o lavori simili, così da essere invogliati ad effettuare un nuovo acquisto. Altre applicazioni tengono invece traccia delle nostra posizione per proporci le inserzioni di negozi nella nostra zona. Allo stesso modo, il geo tagging può essere sfruttato per invogliare gli acquirenti a recarsi presso un venditore specifico, il quale offre uno sconto particolare quando si effettua il *check-in* presso il suo esercizio.
Accanto ad applicazioni di questo tipo, utilizzate nell'ambito del social networking, ne troviamo altre più specifiche, di uso più immediato e che sfruttano l'hardware presente nel dispositivo. Chi acquistava uno smartphone fino a 4 o 5

anni fa, raramente vi trovava all'interno un chip Global Positioning System (GPS) e, quando accadeva, il costo finale del telefono era tutto fuorché abbordabile. Ad oggi tale tecnologia è invece accessibile sin dai modelli di fascia bassa. A noi non interessa la differenza di prezzo, quanto piuttosto vedere come l'evoluzione di questi dispositivi abbia portato ad un miglioramento anche di applicazioni già esistenti. Con il vecchio modello ci saremmo trovati infatti a conoscere la nostra posizione senza avere un'indicazione chiara della direzione verso cui siamo rivolti. È bastata l'introduzione di un giroscopio per ottenere una precisione superiore, particolarmente utile quando si sta camminando per una città sconosciuta e non si sa bene quale direzione prendere per raggiungere la propria destinazione: ruotando lo smartphone, anche la nostra posizione sulla mappa cambia e noi possiamo farci un'idea chiara di quale strada percorrere.
Di particolare importanza sono poi tutti i sensori che nel tempo sono stati inseriti nei nostri dispositivi. Non parliamo solo di nuovi chip hardware, ma anche di sensori software, che monitorano il verificarsi di certi eventi o situazioni per avvisare l'utente o modificare il comportamento di un'applicazione. Su di uno smartphone possiamo trovare ad esempio un sensore di prossimità: quando avviciniamo l'orecchio al display per rispondere ad una chiamata, lui rileva la nostra vicinanza e lo disattiva per risparmiare batteria ed evitare involontarie pressioni di tasti. Allo stesso modo sappiamo che certi notebook montano al loro interno un accelerometro: tale sensore rileva gli spostamenti del computer e, nel caso venga percepita un'accelerazione troppo forte, disattiva la scrittura sull'hard disk in quanto potrebbe trattarsi di una caduta ed è quindi necessario cercare di limitare al massimo potenziali danni.
Come vedremo nel Capitolo 1, i dati ricevuti dai sensori assumono tanta più importanza quanto più le informazioni sono collegate tra di loro: ecco perché è così importante utilizzarli e ampliare il nostro punto di vista non limitandoci al solo mondo mobile.

Con questa tesi vogliamo studiare proprio come poter aggregare dati e informazioni per poi distribuirli a tutti i possibili interessati. Per la raccolta dei valori ci focalizzeremo sul mondo mobile, sfruttando le opportunità che ci

vengono offerte dai moderni smartphone, mentre per la loro raccolta e diffusione ci serviremo di due protocolli giovani, rispettivamente Cosm e PubSubHubBub (PuSH).
Le motivazioni dietro alla scelta di queste due soluzioni sono fortemente legate all'evoluzione del mondo informatico, che nel corso degli anni si è spostato sempre più sul web e verso un approccio di tipo Cloud, quindi con dati memorizzati su server appositi e sempre accessibili ovunque ci troviamo, spesso anche indipendentemente dalla piattaforma che stiamo utilizzando. Di Cosm cercheremo di capire cosa può offrire ad oggi, che possibilità si hanno di integrarlo in un'applicazione distribuita e se presenta potenziali limiti; di PubSubHubBub proveremo invece a realizzare un'implementazione su cui poi effettuare diversi stress test, analizzando i risultati e valutando quanto sia affidabile come protocollo a seconda della quantità di dati inviati.

Al fine di trattare in modo chiaro ciascun argomento si è scelto di suddividere la tesi in diversi capitoli, ciascuno dei quali si focalizzerà su un argomento preciso.
Il Capitolo 1 presenterà dei tipici scenari applicativi e mira ad offrire una visione generale dell'ambiente in cui è possibile operare, andando oltre il solo concetto di sensing di dati dal mondo mobile. Sarà offerta una breve panoramica su possibili alternative che avremmo potuto usare al posto dei protocolli e delle architetture scelte, cercando di far capire il perché sono state scartate.
Cosm verrà trattato in dettaglio nel Capitolo 2: grazie alla ricca documentazione offerta dagli autori, saremo in grado di spiegare gli elementi costitutivi di questo protocollo e come è possibile interagire con essi. Ciò che abbiamo visto nel capitolo precedente sotto forma di esempi verrà qui ampliato, spiegando le relazioni che intercorrono tra ambienti, sensori e misurazioni.
A PuSH è dedicato l'intero Capitolo 3: sarà analizzata l'architettura e il suo funzionamento, dando anche spazio ad un confronto con protocolli precedenti che hanno provato a raggiungere gli stessi obiettivi che gli autori si pongono di migliorare. Di particolare interesse sarà capire la differenza tra protocolli Light Ping e Fat Ping e il perché PuSH abbia optato per quest'ultima soluzione. Per la loro spiegazione si rimanda ai Capitoli 3.2 e 3.3.

Il Capitolo 4 è dedicato alla parte applicativa: verrà descritta in dettaglio l'applicazione realizzata, citando sia le operazioni svolte lato server che lato smartphone. Avendo scelto di lavorare su sistema operativo Android, la prima parte del capitolo sarà dedicata a mostrare le principali differenze che intercorrono tra la programmazione in questo ambiente e quella in Java tradizionale, oltre a motivare questa scelta. Tutta la parte conclusiva si concentrerà poi sulla metodologia adoperata per i test, sottolineando risultati e problemi riscontrati nell'esecuzione.

Capitolo 1

SCENARI APPLICATIVI

Nel capitolo introduttivo verranno introdotti alcuni scenari tipici che si possono presentare quando si interagisce con un telefono cellulare. Salvo rarissimi casi, l'evoluzione del mercato ha fatto sì che anche sui modelli di fascia più bassa siano presenti tutta una serie di funzioni che trasformano un semplice cellulare in uno smartphone.
È tuttavia necessario ampliare il nostro raggio di azione per iniziare a comprendere meglio il perché si sia deciso di studiare un possibile metodo per aggregare i dati e distribuirli.

1.1 Oltre il mondo dei cellulari

Il concetto di mobile non deve trarci in inganno facendoci erroneamente pensare che le informazioni di interesse provengano soltanto dai dispositivi che portiamo con noi ogni giorno. Basta guardarsi attorno per accorgersi di quante fonti di dati ci circondino, dal sensore della temperatura di casa nostra a quelli per la pressione delle gomme nella nostra auto.
Tutti questi dati, che presi singolarmente possono risultare di scarso interesse, vengono spesso aggregati per fornire un quadro più preciso dell'ambiente in cui ci troviamo, del comportamento dell'auto o dei nostri gusti.

1.1.1 Qualche esempio specifico

Al fine di comprendere al meglio alcuni dei casi in cui un dato a prima vista di scarsa importanza può dar vita a tutta una serie di informazioni utili, è bene fare alcuni esempi specifici.
Gli allarmi installati nelle case sono un ottimo punto di partenza. A seconda della

tipologia, essi sono utilizzati per rilevare l'apertura di porte e finestre o eventuali movimenti all'interno di un ambiente, facendo così scattare la sirena con la speranza di mettere in fuga il ladro.

Se questa modalità d'uso è probabilmente più che sufficiente in un palazzo con molti appartamenti, dove ci sono molte persone che possono essere svegliate dal suono della sirena e che possono avvisare le forze dell'ordine, lo è sicuramente meno se si vuole proteggere un'abitazione isolata in campagna.

Per porre rimedio a questa lacuna si può collegare l'allarme alla rete telefonica, così che, quando scatta, avvisi in automatico il proprietario e le forze dell'ordine della possibile effrazione in atto.

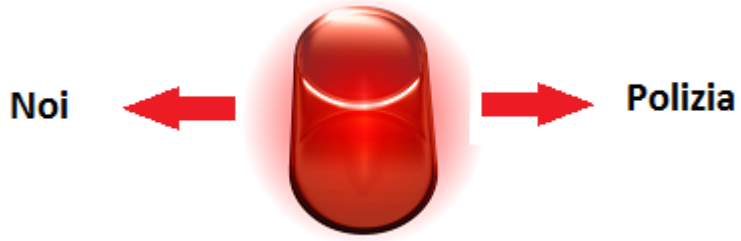

Fig. 1: Un uso limitato delle risorse

Quei sensori che di primo acchito sembravano avere un raggio d'azione limitato iniziano così ad espandere la loro interazione con il mondo circostante: la Polizia potrebbe memorizzare tutte le segnalazioni ricevute dagli allarmi a lei collegati per poi stilare una statistica sulla loro efficacia e su quali zone siano maggiormente prese di mira dalla criminalità, noi potremmo sfruttare la cosa per avvisare parenti e amici chiedendo loro di andare a controllare se siamo distanti.

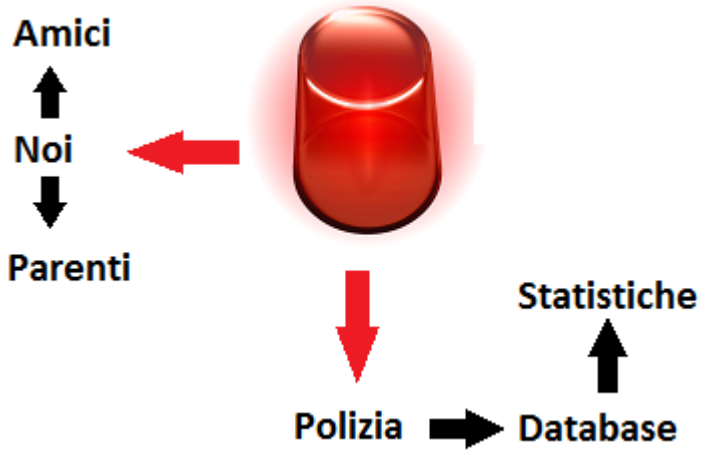

Fig. 2: L'allarme ha un'interazione estesa

Pensiamo ora ad un navigatore satellitare. Durante la nostra marcia, il nostro GPS invia costantemente dati ad uno o più satelliti al fine di ricevere le informazioni su come raggiungere la destinazione desiderata nel minor tempo possibile.

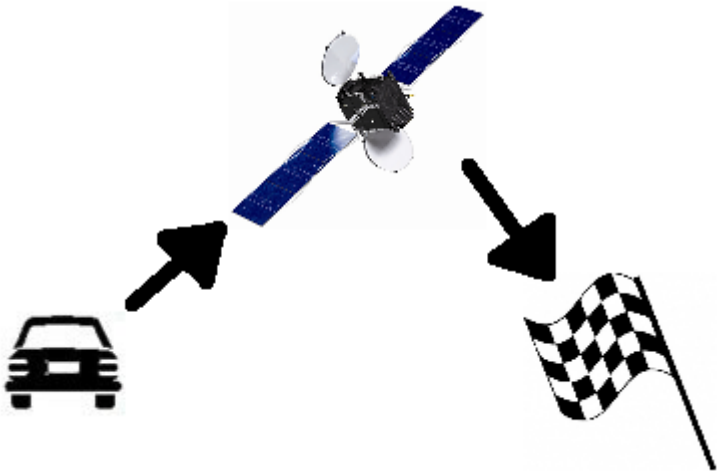

Fig. 3: Il navigatore ci guida a destinazione

La nostra posizione è però di particolare utilità anche per gli altri viaggiatori. Unendo la posizione di più veicoli si può capire se una strada sia particolarmente congestionata e avere una previsione più corretta dei tempi di percorrenza. Il navigatore può quindi elaborare un percorso alternativo e proporcelo per cercare di minimizzare la durata del nostro viaggio o magari segnalarci quanti chilometri di coda ci aspettano.

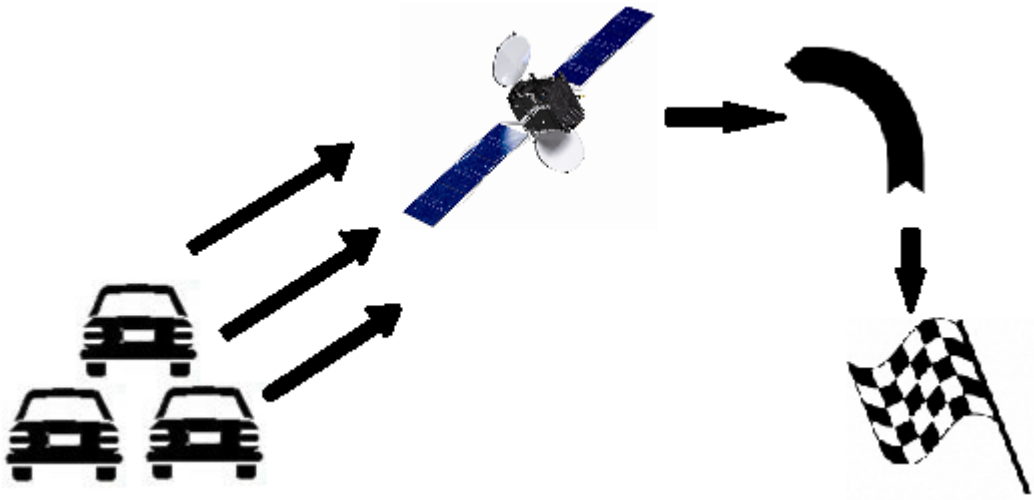

Fig. 4: Il navigatore ci suggerisce una strada alternativa

Un ulteriore caso in cui più dati aggregati forniscono un servizio di maggiore utilità rispetto ai dati singoli è quello delle previsioni del meteo.

Quando decidiamo di andare in vacanza, può capitare di ritrovarci in zone in cui il

mare sia più o meno calmo a seconda della direzione in cui soffia il vento. Allo stesso modo potremmo decidere di non voler uscire di casa quando la Bora è raggiunge velocità troppo elevate da rendere difficile camminare agevolmente.

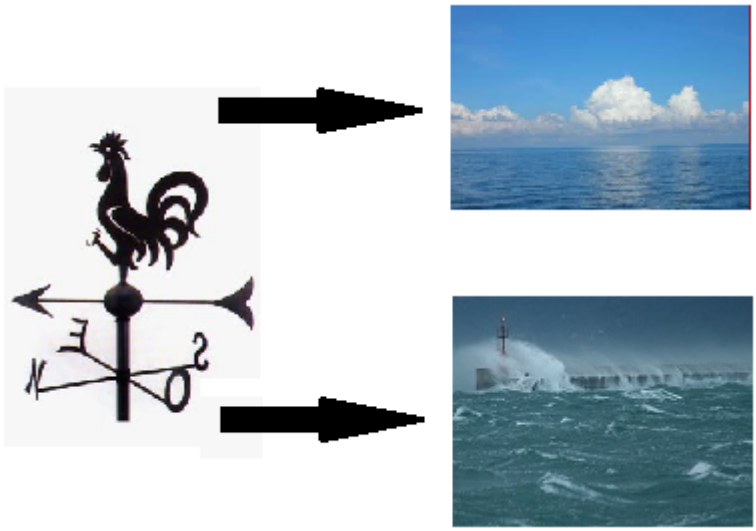

Fig. 5: Meglio restare in casa o uscire?

Prendendo invece un termometro, conoscere la temperatura è sicuramente d'aiuto per sapere cosa indossare, se è necessario coprirsi bene o se ci aspetta una giornata particolarmente calda. Ma siamo sicuri che questo sia sufficiente? Ricevere soltanto un valore non ci dice nulla delle reali condizioni meteo: potremmo avere 25 gradi senza una nuvola in cielo o, al contrario, 25 gradi con la pioggia e un altissimo tasso di umidità.

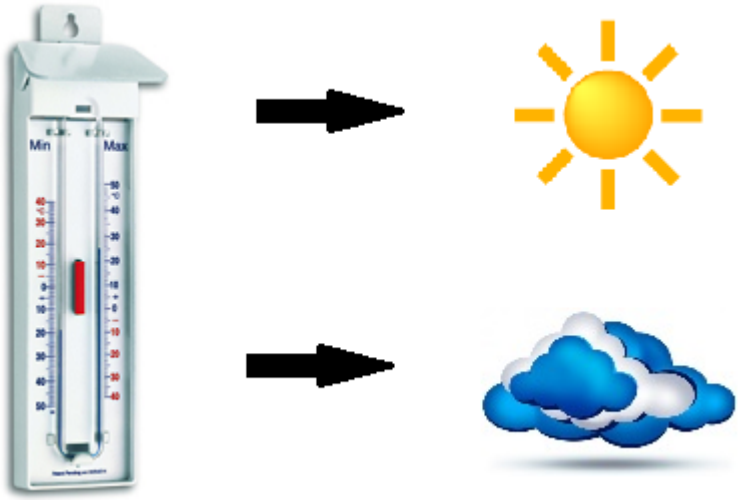

Fig. 6: Basta conoscere i gradi per sapere che tempo fa?

Unendo tutte queste informazioni possiamo invece avere una conoscenza completa del meteo e decidere al meglio come pianificare la nostra giornata.

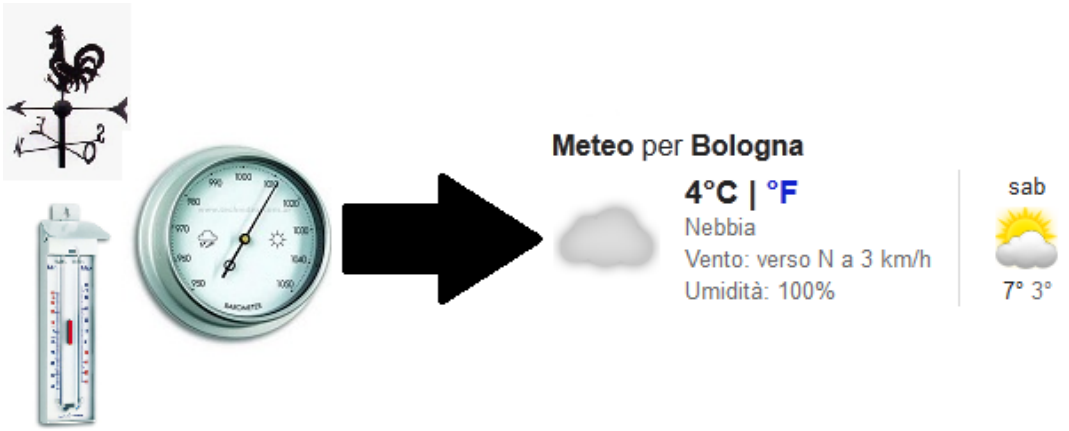

Fig. 7: Un quadro completo della situazione meteorologica

1.1.2 Proteggere i nostri dati

I casi citati nel paragrafo precedenti possono essere tutti considerati virtuosi: si parte da uno o più dati semplici e li si unisce per fornire un servizio più completo e personalizzato. Questi scenari sono legati all'utilizzo di molteplici sensori e dispositivi che tutti conosciamo e che consideriamo parti integranti delle nostre vite, per cui non ci meraviglia sapere che essi vengano aggregati per giungere ad una rilevazione più completa.

La raccolta delle informazioni presenta tuttavia un'altra faccia della medaglia: si rischia di esporre dati sensibili non sapendo bene come verranno trattati. Se la temperatura di casa nostra non interessa a nessuno al di fuori del nostro nucleo familiare, le nostre preferenze in fatto di abbigliamento, musica o film sono informazioni importantissime per tutti quei siti che sponsorizzano la propria attività online. Nel Settembre del 2011 venne reso pubblico il caso di un noto social network che sembrava continuasse a tracciare la nostra navigazione anche dopo avere effettuato il logout, così da proporre poi le inserzioni degli sponsor più accattivanti per l'utente in base ai suoi gusti (con la speranza che vi cliccasse sopra e magari acquistasse il relativo prodotto).

Il tema della privacy sta diventando un elemento sempre più centrale, tanto che tutti i browser più recenti offrono un sistema per evitare di essere tracciati durante le sessioni in internet.

La sicurezza e le misure per tutelarsi esulano dall'argomento di questa tesi e sono

state riportate per completezza: è importante conoscere anche questo aspetto per iniziare ad avere una visione più globale di ciò che ci circonda, di come le nostre informazioni vengano trattate e di cosa si parli quando si fa riferimento ad esse.

1.2 La raccolta dei dati

Quando si vuole realizzare un progetto basato sullo scambio di dati è inevitabile chiedersi quale metodo scegliere e dove archiviarli. Fino a qualche anno fa si è assistito ad un vero e proprio abbattimento del costo dei dispositivi di archiviazione di massa come gli hard disk: chiunque poteva acquistare un disco da un Terabyte o più a prezzi irrisori e ciò poteva erroneamente portare a pensare che qualsiasi scelta adottata per la propria applicazione potesse andare bene, dato che lo spazio non mancava. La sempre maggiore interazione con la rete e l'integrazione con il mondo mobile, tuttavia, ha ricordato come fosse invece ancora necessario focalizzarsi su formati che fornissero quante più informazioni possibili con il minimo peso (in termini di byte), così da essere facilmente trasportabili anche con connessioni non particolarmente veloci.

1.2.1 XML e JSON

Uno dei formati più utilizzati quando si tratta di memorizzare informazioni è sicuramente l'Extensible Markup Language (XML). In un solo documento di testo troviamo organizzati tutti i dati in modo comprensibile anche da noi senza bisogno di programmi appositi per la loro lettura: ciascun elemento è delimitato da tag che vanno chiusi rigorosamente nell'ordine in cui sono stati aperti. Ciò ne facilita la comprensione, evitando così possibili situazioni ambigue come in HyperText Markup Language (HTML).
Molti dei linguaggi di programmazione più diffusi, inoltre, hanno apposite librerie per analizzarli e permettere così di recuperare tutte le informazioni del caso, tipicamente effettuando una scansione del file fino a quando non si trova il tag a cui si è interessati oppure caricando tutto il documento in memoria, analizzandolo poi lì.

Successivo a XML e nato con l'intento di offrire un formato di dati più efficiente c'è poi JavaScript Object Notation (JSON). Il concetto dietro ad esso è lo stesso di XML, ma mentre quest'ultimo pone maggiore enfasi sulla definizione di marcatori per descrivere al meglio ciascun elemento, JSON è nato proprio con l'intento di favorire il trasferimento di dati. I relativi parser si sono adattati, permettendo quindi di creare direttamente i rispettivi oggetti in base ai valori letti dal file.

1.2.2 I database

Fondamentali per memorizzare grandi quantità di dati e garantirne la loro persistenza, i database sono un sistema assai comune di memorizzare informazioni. Abbiamo a che farci quotidianamente, anche se magari non ce ne rendiamo conto. Quando ci registriamo ad un sito o andiamo in un ufficio e forniamo le nostre generalità, la nostra utenza viene inserita o recuperata da dei database a cui tipicamente si accede previa autenticazione.
Il pregio maggiore di questo approccio è di poter ottenere grandi quantità di dati con una semplice richiesta: se volessimo realizzare un sito web che elenca una serie di ricette di cucina dovremmo preoccuparci di implementare soltanto le chiamate per ricevere tutte le ricette in questione, così che il sito possa poi popolarsi in maniera autonoma. Questo porta ad un grande risparmio di tempo e dà la possibilità di concentrarsi sulla realizzazione di un template unico per mostrare le informazioni come più preferiamo.
Quando parliamo di database, tuttavia, dobbiamo tenere ben presente che spesso essi possano portare a richieste hardware piuttosto importanti, sia in termini di spazio disponibile per l'archiviazione che (soprattutto) di rapidità nel recupero delle informazioni. In ambito mobile, tempi lenti potrebbero sommarsi alle connessioni non sempre affidabili, causando quindi un numero di dati persi più elevato di quanto la natura stessa della connessione implichi.
L'integrazione con i database più comuni è spesso garantita da apposite Application Programming Interface (API), ma è comunque un elemento da valutare in quanto non è detto a priori che il nostro ambiente di sviluppo le supporti.

1.2.3 L'approccio di Cosm

Un protocollo nuovo che si occupa di tener traccia delle informazioni di sensing, aggiornate di continuo, è Cosm. Nato pochi anni or sono, questo protocollo fornisce l'ambiente ideale su cui memorizzare i valori di sensori racchiudendoli all'interno di un ambiente, come avviene nel mondo reale.
È la scelta che è stata effettuata per questa tesi, così da capirne meglio punti di forza e debolezze.
Il costo d'ingresso nullo è un buon biglietto da visita (è infatti sufficiente registrarsi sul sito ufficiale), in quanto ci libera infatti da eventuali costi di un nostro server, permettendoci di sfruttare risorse già esistenti. Questo ha sicuramente il vincolo maggiore nel numero di interazioni possibili al minuto, come vedremo in seguito. Tutto ciò, unito all'interazione esclusivamente via messaggi HTTP, rende assai facile la sua implementazione, garantendo anche un certo grado di personalizzazione sui dati che si vogliono ricevere dal server.
Ad una sua trattazione dettagliata è stato riservato il Capitolo 2.

1.3 Un accenno alla distribuzione dei dati

Sino ad ora ci siamo occupati di quella che può essere considerata la prima fase in un progetto dedicato alla raccolta dati e alla loro diffusione. Abbiamo citato le possibili fonti e l'importanza di aggregare più informazioni contemporaneamente, quindi abbiamo spostato la nostra attenzione su dove memorizzarle.
Il passaggio mancante è quello legato alla distribuzione delle rilevazioni che abbiamo raccolto. Come si può raggiungere il numero più ampio possibile di interessati, fornendo al contempo sia puntualità nell'erogazione del servizio che un uso efficiente delle risorse?
In passato sono già stati adottati protocolli realizzati ad hoc per questo scopo, ma il loro uso sul campo ha rivelato la presenza di limiti che mal si adattano ad un loro uso intensivo. È quindi bene valutare a dovere quale possa essere la scelta più corretta in base allo scopo che si vuole raggiungere: protocolli efficienti per un determinato tipo di comunicazione potrebbero comunque non rivelarsi adatti al

nostro scopo.

1.3.1 SIP

Il Session Initiation Protocol (SIP) è un protocollo particolarmente usato per le applicazioni di telefonia su Internet Protocol (IP) o Voice over IP (VoIP). Proprio per questo, in principio è stato pensato per utilizzare lo User Datagram Protocol (UDP) e solo più recentemente gli è stato aggiunto il supporto a Tranfer Connection Protocol (TPC).
Esso permette di trasferire sia audio, che video e testo, prestandosi dunque a più usi a seconda della necessità. Viene tipicamente integrato in altri protocolli per completarli e permettere una comunicazione completa.
Analizzando i passaggi necessari all'instaurazione di una connessione ritroviamo in maniera molto intuitiva quelli che sono i classici gesti di una telefonata: il primo passo consiste nell'identificare gli interessati alla comunicazione, cosa che avviene tramite un three-way handhaking simile a quanto accade in TCP, quindi si deve valutare se essi siano disponibili e, in tale caso, impostare gli opportuni parametri che permettano la comunicazione prima e la terminazione della sessione poi.
L'immagine sottostante ci aiuta a capire meglio come avviene uno scambio di dati:

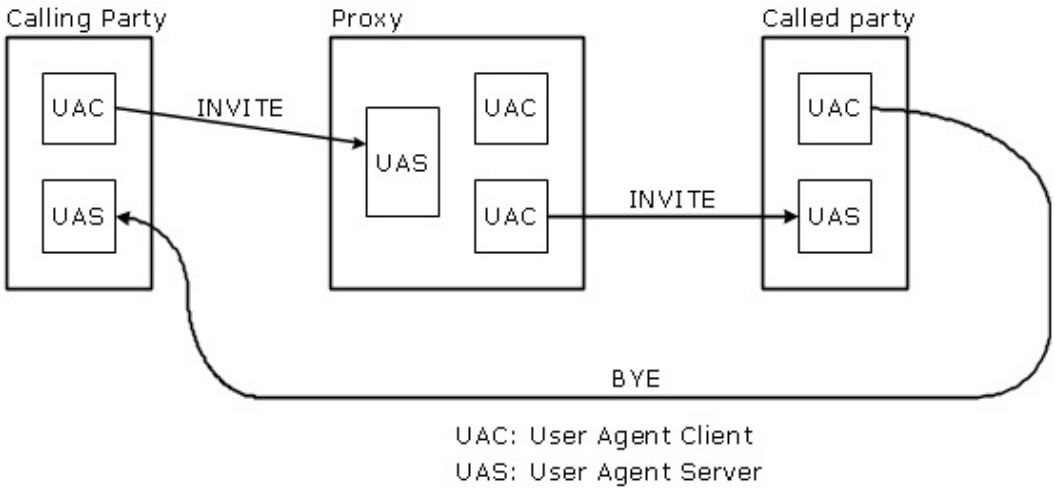

Fig. 8: Il funzionamento di SIP

Quello mostrato in figura è un caso tipico. I due estremi devono essersi

precedentemente registrati presso un proxy che smista poi le chiamate a seconda di chi è il chiamante e chi il ricevente. Quest'ultimo chiude la conversazione con un messaggio di BYE.
Per quanto questa modalità di scambio dati funzioni, male si sposa con la nostra necessità di distribuzione: come vedremo in seguito, le entità in gioco sono simili a quanto accade con PubSubHubBub (che introdurremo a breve nel Capitolo 1.3.3), ma vi è una differenza importante. In SIP il chiamante specifica in maniera precisa chi sia il ricevente. Nel nostro caso invece non abbiamo questa informazione a priori: riprendendo quanto scritto nel Capitolo 1.1, se misuriamo la temperatura della nostra casa possiamo immaginare chi siano gli interessati, ma se forniamo invece un servizio di previsioni meteo decisamente no. Il nostro compito è quello di fornire il servizio: dobbiamo trovare qualcun altro che si occupi poi di smistarlo ai diretti interessati.

1.3.2 XMPP

XMPP sta per Extensible Messaging and Presence Protocol ed è un protocollo nato nel 1999 per lo streaming XML. Usato prevalentemente nei programmi di messaggistica istantanea (si pensi a Pidgin o a Google Talk), si è evoluto nel corso degli anni ponendo sempre più attenzione alla sicurezza e alla privacy nella trasmissione dei messaggi.
I client XMPP comunicano per mezzo di un gateway remoto, cosa che permette a qualsiasi programma che implementi tale protocollo di interagire sulla stessa rete.
Ulteriori dettagli su XMPP verranno dati nel Capitolo 3.3.1: qui anticipiamo che questo protocollo non è stato scelto per la tesi per via delle troppe modifiche che i dati subiscono prima di giungere a destinazione. Vista la necessità di sviluppare un'applicazione che gestisca anche grandi quantità di dati nel modo più efficiente possibile, l'attenzione è caduta su PuSH in quanto più recente (e come tale da testare) e dotato di opportuni meccanismi per prevenire attacchi e gestire i possibili interessati.

1.3.3 La scelta di PubSubHubBub

PubSubHubBub è un protocollo semplice, aperto, publish/subscribe di tipo server-to-server strettamente legato al web che può essere visto come un'estensione ad Atom e Really Simple Syndacation (RSS).
Le parti in gioco (ovvero i server) possono ottenere notifiche quasi istantanee quando un argomento (*feed URL*) a cui sono interessati è aggiornato.
Lo scopo che PubSubHubBub si prefigge è migliorare l'esperienza di chi utilizza Internet eliminando il traffico indesiderato che si genera con i modelli a polling.
Data la scarsità di materiale su questo protocollo, si è ritenuto interessante cercare di capire quanto possa rappresentare la scelta giusta per la diffusione di nuovi contenuti.
Il suo funzionamento sarà descritto in dettaglio nel Capitolo 3, per ora ci limitiamo a dare qualche informazione generale.
PuSH si basa su tre parti in gioco: un publisher, un hub e uno o più subscriber. Rispettando un preciso protocollo, il publisher si preoccupa di avvisare l'hub quando ha nuovi contenuti, questo li gira poi a tutti gli interessati che si erano precedentemente sottoscritti.
Il fatto che l'hub tenga traccia delle richieste di sottoscrizione ricevute ed effettui controlli sui dati in arrivo è un valore aggiunto che si è ritenuto essere interessante rispetto ad altri protocolli, così da evitare possibili attacchi e usi non desiderati dell'interazione.

1.4 Obiettivi della tesi

Con la scelta di Cosm e PuSH per memorizzare e distribuire i dati ci siamo posti come obiettivo l'approfondimento della conoscenza di questi due protocolli per capire quali possibilità offrano e come possano essere sfruttati al meglio.
Per Cosm cercheremo di capire come integrarlo con il mondo mobile e in che modo possa riprodurre un ambiente reale, registrando i valori misurati da sensori. Valuteremo il livello di personalizzazione offerto al programmatore e all'utente finale per recuperare le informazioni caricate sul server e il numero di interazioni

possibili per non saturarlo.

PuSH richiederà invece un'analisi più dettagliata per testare se le premesse degli sviluppatori vengano rispettate alla prova pratica. Oltre a verificare il sistema di sottoscrizione e rimozione della stessa (al fine di evitare che i nuovi valori pubblicati siano inviati ad un destinatario inesistente, sprecando quindi risorse) saranno necessari una serie di stress test per capire che throughput si può raggiungere nella pubblicazione dei dati. Sfruttandolo in contemporanea a Cosm vogliamo arrivare a capire se i due protocolli possano interagire per diffondere con successo un elevato numero di misurazioni.

Nel nostro caso specifico i valori arriveranno da uno smartphone, per cui sarà necessario pensare a come suddividere le varie parti del progetto così da sfruttare al meglio le risorse disponibili, evitando che uno degli elementi in gioco faccia da possibile collo di bottiglia.

Capitolo 2

COSM E L'INTERNET DELLE COSE

Nato nel 2008 con il nome di Pachube, Cosm si presenta come una piattaforma, un insieme di API e una comunità dove dispositivi, informazioni, sviluppatori e applicazioni si uniscono per dare vita alle loro idee e a prodotti tra loro connessi. Il concetto fondamentale è che *aperto* sia meglio di *chiuso* e che la *condivisione* sia preferibile al *tenere tutto per sé*.

Cosm si pone come obiettivo la costruzione dell'*Internet of Things (IoT)*, ovvero l'Internet delle Cose. È quindi bene chiarire sin dal principio a cosa si fa riferimento con questo termine e per farlo utilizzeremo alcuni esempi.

L'Internet of Things può essere visto come un'ulteriore evoluzione di Internet: in principio il World Wide Web (WWW) era un mezzo a cui accedere per condividere documenti ipertestuali; nei primi anni duemila si è diffuso sempre più il cosiddetto Web 2.0, in cui gli utenti sono passati dal ruolo di consumatori passivi a quello di utilizzatori attivi, non limitandosi più a fruire di un servizio, ma proponendone uno loro stessi o andando ad arricchire e completare qualcosa di già esistente. Cosm vuole portare un nuovo interlocutore nella rete: le cose.

Immaginate di andare in vacanza per qualche tempo. In caso voleste sapere se il frigorifero sia ancora in funzione, così da non mandare a male tutto ciò che è presente al suo interno, l'unica soluzione sarebbe contattare un parente o un amico che vada a casa vostra e verifichi che non sia mai saltata la corrente. E se tale elettrodomestico fosse invece collegato alla rete e inviasse ad intervalli regolari la temperatura misurata da un sensore? Ecco che noi potremmo essere sempre informati sullo stato del frigorifero senza bisogno di scomodare nessuno: basterebbe una piccola applicazione sul nostro cellulare per tenere sempre sott'occhio tutto ciò che ci interessa.

Uno schema di base per rappresentare questa interazione è il seguente:

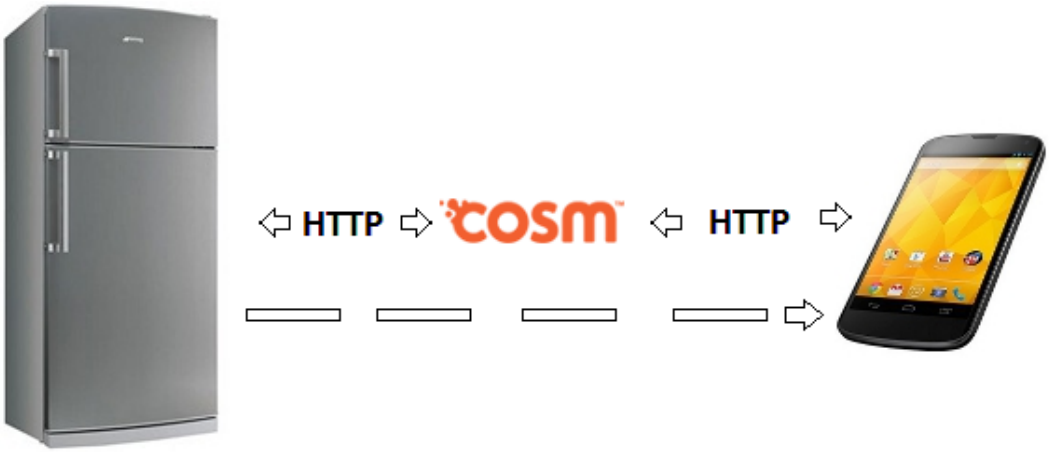

Fig. 9: Interazione tra sorgente e destinatario

Se ad un primo sguardo Cosm potrebbe sembrare un inutile appesantimento dell'interazione tra i due estremi della comunicazione (noi vorremmo infatti che il frigorifero inviasse direttamente i dati al cellulare, come da freccia tratteggiata), ad un'analisi più attenta si osserva invece come il tutto avvenga tramite l'invio di messaggi HTTP, cosa che non ci vincola ad un preciso ambiente di sviluppo. A disposizione del programmatore ci sono già librerie per i più diffusi linguaggi, ma nulla ci vieta di crearne altre ex-novo nel momento in cui si rendesse disponibile una nuova piattaforma: dovremo solo assicurarci che abbia accesso alla rete per inviare richieste HTTP.

Le frecce nello schema mostrano un'interazione bidirezionale: come vedremo in seguito analizzando più in dettaglio le API, prima di poter inviare i propri dati, il dispositivo (in questo caso il frigorifero) deve accertarsi che Cosm abbia un'opportuna area adibita alla ricezione delle informazioni; il dispositivo ricevente, allo stesso modo, deve prima sincerarsi che tale area ci sia, quindi può procedere alla ricezione dei dati.

È interessante notare come le informazioni prodotte da un singolo sensore possano essere comunicate a più di un ricevente: sta a noi sfruttare questa possibilità per raggiungere il maggior numero possibile di utenze interessate, preoccupandoci di scalare opportunamente la nostra applicazione onde evitare di saturare il numero di richieste a nostra disposizione verso il server Cosm. Questi limiti saranno trattati più in dettaglio nella sottosezione 2.1.10.

Quello appena presentato è un esempio banale, ma serve già a dare un'idea di massima delle potenzialità che Cosm mette a disposizione. Tramite una centralina potremmo tenere sempre monitorato l'inquinamento di una determinata parte della nostra città, piuttosto che la temperatura di un componente critico in una macchina, facendo scattare un allarme nel caso in cui si superasse una determinata soglia.
Gli autori definiscono Cosm come una piattaforma stabile, sicura, che connette dispositivi e prodotti con le applicazioni per fornire un controllo in tempo reale e lo storage dei dati. Offre inoltre la possibilità di realizzare prodotti collegati ad Internet senza bisogno di avere una infrastruttura alle spalle.

Tutto questo può essere effettuato grazie alle API fornite, le quali permettono allo sviluppatore di creare prodotti in grado di connettersi alla rete offrendo dati in tempo reale, tracciabilità della storia passata e tool di gestione utente per i vari dispositivi e ambienti in giro per il mondo.
Giunte alla versione 2, è importante prendere confidenza con alcuni termini ricorrenti nelle API Cosm. Queste parole chiave descrivono al meglio il contesto in cui si lavora, dando un'idea di massima su cosa sia possibile fare e su come sia possibile realizzarlo.

2.1 Informazioni generali sulle API

Nel sistema Cosm si utilizza una gerarchia di tipi di dato per rappresentare i dati e le informazioni:

Feed → Datastream → Datapoint

2.1.1 Environment (o "Feed")

Un "environment" (ambiente) è una collezione di misurazioni effettuate in un contesto specifico, spesso con una particolare geolocalizzazione, definita dal creatore dell'environment e misurata da sensori e dispositivi. Il termine può

riferirsi sia ad entità fisiche come una stanza, un dispositivo mobile, un edificio o una foresta, sia ad entità virtuali quali un modello stile Second Life, il monitoraggio della banda di un server ecc. Un “feed” Cosm è la rappresentazione del dato di un environment e dei suoi datastream. I suoi metadati possono specificare in modo opzionale se è fisico o virtuale, fisso o mobile, interno o esterno, ecc.

2.1.2 Datastream

Un datastream rappresenta un sensore individuale o un dispositivo/script di misurazione in un environment. Ciascun datastream deve avere un ID alfanumerico univoco all'interno dell'ambiente. Può anche specificare le “unità di misura” (ad esempio, watt) così come “tag” definiti dall'utente (ad esempio, “temperatura_frigo”).

2.1.3 Datapoint

Un datapoint rappresenta un singolo valore di un datastream in uno specifico punto nel tempo. È semplicemente una coppia chiave/valore di un timestamp e del valore in quel momento.

Esempio: installiamo in una stanza (“environment”) sensori (“datastream”) per la temperatura, umidità e CO2. Creiamo un feed Cosm chiamato “La mia stanza”, con tre datastream i cui ID potrebbero essere: “temperatura”, “umidita” e “CO2”, a loro volta identificati dai tag “termico,senza-contatto”, “capacitivo, SHT21” e “MG811” rispettivamente. Avrebbero inoltre le unità di misura “Celsisus”, “%RH” e “ppm”. Singoli datapoint in un certo momento potrebbero essere “32,2”, “34” e “3820”, rispettivamente.

2.1.4 Versioni

Al momento ci sono due versioni attive delle API. Dove possibile si dovrebbero

usare sempre quelle più recenti, in quanto le funzionalità più avanzate difficilmente saranno rese disponibili per le versioni precedenti.

Dare un numero di versione alle API permette di bloccare un'applicazione ad una certa versione delle stesse. Ciò significa che gli unici cambiamenti che gli autori implementeranno saranno:

- aggiungere nuove feature lasciando inalterate quelle già presenti
- fix che non comprometteranno la compatibilità all'indietro
- incrementare i numeri del formato della versione come richiestro per soddisfare anche i cambiamenti minori

I formati JSON e XML hanno entrambi numeri della versione inclusi nella loro struttura. Questo ci permette o di specificare un formato particolare al fine di garantire la consistenza dei dati, oppure di verificare in che formato un documento è rappresentato.

2.1.5 HTTP-based & RESTful

Le API di Cosm sono al momento interamente basate su richieste HTTP e soddisfano i principi di design Representational State Transfer (REST).
I tre principali tipi di dato sono rappresentati in maniera gerarchica nell' Uniform Resource Locator (URL), allo stesso modo della relazione che c'è fra loro (Environment → Datastream → Datapoint). Gli accessi RESTful utilizzano il linguaggio HTTP per determinare quale azione effettuare su di un particolare oggetto:

- GET: ottiene lo stato attuale dell'oggetto
- PUT: setta lo stato attuale dell'oggetto
- POST: crea un nuovo oggetto
- DELETE: cancella l'oggetto

2.1.6 Codici di stato HTTP

Le API Cosm provano a restituire l'appropriato codice HTTP di stato per ciascuna richiesta.
Quelli più comuni includono:

- 200 Ok: richiesta processata con successo.
- 401 Not Authorized: o devi fornire le credenziali per l'accesso, o quelle fornite non sono valide.
- 403 Forbidden: Cosm capisce la tua richiesta, ma si rifiuta di svolgerla. Un messaggio di errore dovrebbe spiegare perché.
- 404 Not Found: o stai chiedendo un Uniform Resource Identifier (URI) non valido, o la risorsa non esiste.
- 422 Unprocessable Entity: Cosm non è stato capace di creare un feed perchè l' Extended Environments Markup Language (EEML)/JSON non era completo/valido.
- 500 Internal Server Error: qualcosa è andato storto... si suggerisce di scrivere sul forum per avviare le opportune indagini.
- 503 No server error: di solito accade quando ci sono troppe richieste in arrivo su Cosm – se questa è la risposta che si ottiene da una richiesta formulata con le API, allora il messaggio di errore sarà ritornato in formato XML con la risposta.

2.1.7 Richieste PUT e DELETE alternative

Non tutti i client HTTP sono in grado di effettuare richieste PUT o DELETE (ad esempio i browser web). Per superare questa limitazione è possibile inviare una richiesta POST con un parametro "_method":

- *http://api.cosm.com/v2/feeds/504/datastreams/0.xml?_method=put*
  simulerà una PUT

- *http://api.cosm.com/v2/feeds/504/datastreams/0.xml?_method=delete* simulerà una DELETE

2.1.8 Sicurezza – SSL/HTTPS

Gli standard industriali HyperText Transfer Protocol over Secure Socket Layer (HTTPS)/Secure Sockets Layer (SSL) per l'accesso sicuro sono disponibili per tutti i metodi di autenticazione, semplicemente rimpiazzando HTTP con HTTPS nella richiesta.

2.1.9 Account e disponibilità delle feature

Per usare le API di Cosm è sufficiente creare un account Cosm. Per registrarsi basta cliccare sul pulsante "Get Started" alla pagina https://cosm.com. La registrazione garantisce il pieno accesso a tutte le feature e la possibilità di sfruttare le API con un ben definito *rate limit*.

2.1.10 Rate Limit

L'utilizzo delle API ha un limite di utilizzo che è al momento settato a 100 richieste al minuto. Mentre il rate limit delle API è espresso come "richieste per minuto", il calcolo è in realtà una media effettuata ogni 3 minuti.

Se la propria applicazione eccede tale limite, non potrà più fare con successo alcun richiesta, indipendentemente che stia cercando di accedere ai dati di un feed o di inviarli, e riceverà un errore (503 nelle API v1, 403 nelle v2) sino a quando il rate non sarà diminuito.
Una volta che ciò sarà avvenuto, il servizio tornerà a funzionare normalmente quindi non bisogna far altro che diminuire la frequenza con cui il dispositivo o lo script effettua le richieste.

Questo significa che si hanno due possibilità: o si effettuano richieste ad intervalli

regolari, oppure le si effettuano in rapida sequenza una dopo l'altra (ma bisogna poi preoccuparsi che per i successivi due minuti non ve ne siano altre). Di conseguenza, se si effettuano costantemente più richieste del limite accettato, allora nessuna di queste richieste terminerà mai con un successo.

2.1.11 Accesso senza autenticazione

L'accesso senza autenticazione è disponibile per le immagini PNG, riassunti della storia passata e feed RSS/ATOM disponibili liberamente al pubblico. Questo permette, ad esempio, di inserire un grafico in un sito a cui accedono utenti non Cosm. Un controllo più fine sui tool di controllo dell'accesso saranno forniti in futuro, al momento si fa riferimento a quanto indicato nella versione 1 delleAPI.

2.2 Formato dei dati

Le API COSM supportano 3 formati per i dati: JSON, XML e Comma-Separated Values (CSV). Ciascuno di questi formati è particolarmente adatto a determinati scopi. Hanno inoltre la caratteristica comune di essere formati aperti e di essere ampiamente usati, soprattutto in ambito mobile. Non è infatti raro trovare applicazioni che esportano i messaggi ricevuti o i contatti della propria sim in uno di quei formati, così da poterli poi visualizzare anche su di un normale computer o sfruttarli per ulteriori elaborazioni.
Nella versione 2 delle API le rappresentazioni JSON e XML di un environment sono interscambiabili: rappresentano gli stessi dati e metadati.

Ci sono due modi per specificare il formato in uso:

- appendere l'identificatore del formato all'URL, ad esempio `/v2/feeds.json`
  passare l'header “Accept” nella richiesta HTTP, ad esempio
- `Accept: application/json`

Indipendentemente dalle API usate, se non si specifica alcun formato, allora si

sceglie JSON. Ciò significa che, quando non si specifica un formato, Cosm proverà a fare il parsing dei dati in ingresso come JSON e li invierà sempre sotto forma di JSON. Se questo non è il comportamento desiderato, si deve specificare un formato in maniera esplicita.

Se si utilizzano entrambi i metodi per indicare il formato, l'identificatore nell'URL ha la precedenza.

2.2.1 JSON

Il formato JSON è particolarmente indicato per applicazioni web based in quanto se ne può facilmente fare il parsing utilizzando Javascript nel browser. Inoltre è anche un ottimo formato per la generica serializzazione dei dati in quanto ha un overhead di calcolo molto inferiore rispetto all'XML e usa meno banda per trasmettere.

JSON callback: si utilizza solo quando è richiesta una risposta formattata in JSON. Includere questo parametro permetterà alla risposta di essere racchiusa (*wrapped*) nel metodo di callback che si è scelto (particolarmente utile con Javascript e per creare applicazioni AJAX).
Ad esempio:
http://api.cosm.com/v2/feeds/504.jsoncallback=myCallbackFunction avrà come risultato un *response body* di myCallbackFunction{...} al posto del solo {…}. I callback dovrebbe contenere solo caratteri alfanumerici e underscore; qualsiasi carattere invalido sarà scartato.

2.2.2 XML

L'API Cosm usa uno specifico formato di XML chiamato EEML. È ben predisposto all'integrazione con sistemi esistenti come quelli per la creazione di sistemi di gestione anche se contiene le stesse informazioni di JSON.

2.2.3 CSV

CSV è pensato per essere utilizzato da dispositivi integrati molto semplici, come un Arduino o altri microcontroller. Non contiene nessuno dei metadati che sono presenti nei formati XML e JSON (possono comunque essere aggiunti separatamente utilizzando le API o l'interfaccia web).

2.2.4 Compressione

Cosm supporta la compressione sulle richieste sia in ingresso che in uscita. Questo è particolarmente utile se, ad esempio, si ha un dispositivo General Packet Radio Service (GPRS) che periodicamente carica un batch di dati in quanto la compressione li comprimerà del 75% o più. Cosm usa il metodo standard HTTP di specificare la codifica per ottenere la compressione.

- Compressione in **ingresso**: per inviare dati a Cosm è necessario specificare la codifica “gzip”. Prima bisogna comprimere i dati usando gzip, quindi aggiungere l'header “Content-Encoding: gzip” alla richiesta. È comunque bene indicare l'header Content-Type affinché combaci con il dato non compresso in quanto quest'ultimo verrà poi utilizzato dopo la fase di decompressione.
- Compressione in **uscita**: per ricevere dati da Cosm compressi in formato gzip basta semplicemente aggiungere “Accept-Encoding: gzip” all'header della richiesta.

2.2.5 Time zone

Cosm supporta orari indicati dall'utente.
Ci sono due posti dove si può specificare la time zone: nel profilo utente sul sito e attraverso un parametro apposito in una richiesta API. Di default il fuso orario è UTC.
Per sovrascrivere il comportamento di default, che vede tutte le api GET ritornare

un timestamp in UTC, occorre inviare *timezone* come parametro di una richiesta. Esso può essere specificato sia come offset di un certo numero di ore rispetto all'orario UTC oppure come il nome di un posto.

Ad esempio, per vedere l'orario a Bologna:

http://api.cosm.com/v2/feeds?timezone=+1.0

oppure

http://api.cosm.com/v2/feeds?timezone=Rome

Le API forniscono i nomi di tutte le località supportate.
In caso si voglia usare una time zone non-UTC per inviare dati a Cosm, tutto ciò che si deve fare è indicare il time offset in una stringa per l'orario formattata secondo International Organization for Standardization (ISO) 8601 quando si inviano i dati.

Per indicare Bologna bisognerebbe scrivere quanto segue:

2012-10-10T10:20:30.123456+01:00

Come scritto in precedenza, all'interno del contesto di una singola richiesta alle API, la time zone sarà o a UTC di default, oppure prenderà il valore del parametro timezone. Il significato di questo in riferimento ai trigger è che se non si aggiunge il parametro timezone quando si invia un update, allora qualsiasi generatore di eventi avviato dall'update sarà inviato in broadcast come UTC, anche se il timestamp all'interno dell'aggiornamento è in un'altra time zone o se il proprio profilo ne specifica uno diverso.

Se si vuole essere sicuri che qualsiasi trigger ricevuto abbia una specifica time zone, allora bisogna modificare lo script o il dispositivo che invia gli

aggiornamenti affinché includa la time zone desiderata nell'URL della richiesta quando invia ogni update. Questo non avrà effetto sul parsing di un qualsiasi time stamp presente nell'update, tutti gli orari saranno comunque convertiti in UTC per il salvataggio sui server Cosm.

2.3 Scenari d'uso

Terminata una prima panoramica su ciò con cui ci troviamo a lavorare, è bene entrare più nel dettaglio di certi elementi. Le API forniscono infatti una spiegazione dettagliata su come creare, gestire e modificare feed, datastream e datapoint, oltre ad altri aspetti come i trigger e la gestione degli utenti.
Sin dal principio abbiamo parlato di dispositivi, non solo di applicazioni: il motivo è da ricercare nella natura stessa di Cosm, ovvero il suo utilizzo per connettere tra loro le "cose". Il sito ufficiale riporta una lista di dispositivi supportati che facilitano l'interazione con il protocollo: quelli più noti sono sicuramente le piattaforme Arduino e il recente Raspberry Pi, piccole soluzioni hardware alla portata di tutti grazie al basso costo e alla grande configurabilità.

Siamo partiti citando il caso del frigorifero: riprendiamolo ora per spiegarlo più in dettaglio, dipingendo così un tipico caso d'uso di Cosm.

2.3.1 Definizione dell'ambiente

Il primo passo consiste nel definire quello che abbiano chiamato environment o feed, l'ambiente con cui interagiamo. Data la collocazione di questo elettrodomestico, possiamo pensarlo di disporlo nell'ambiente Cucina.
Un environment rappresenta un gruppo di datastream: ciascuno di essi è un sensore che monitora un determinato evento, quale ad esempio la nostra temperatura del frigorifero.
La creazione del feed avviene inviando al seguente URL un documento XML/JSON correttamente formattato oppure includendo i dati nel body di un messaggio POST:

http://api.cosm.com/v2/feeds

Un feed contiene diversi attributi: l'unico obbligatorio è il titolo, mentre dettagli quali lo status, il sito di riferimento, posizione geografica e altri sono opzionali. La loro lista completa è visibile dalla relativa pagina delle API.
Tramite opportuni messaggi HTTP si può interagire con il feed in diversi modi, andando ad aggiornarlo o visualizzarlo a seconda delle nostre necessità.

- L'aggiornamento del feed avviene tramite un messaggio POST a

  http://api.cosm.com/v2/feeds/feed_id

  Esso aggiorna [environment ID] di environment e datastream. Se ha successo, i valori attuali del datastream sono memorizzati e qualsiasi cambiamento nei metadati dell'environment sovrascrive i valori precedenti. Cosm salva un timestamp lato server nell'attributo “updated” e imposta il feed “live” se prima non lo era. La richiesta PUT richiede che il corpo contenga il file JSON, EEML o CSV che deve essere salvato/registrato.
  Se il proprio client non è in grado di effettuare richieste PUT, queste possono allora essere simulate da una richiesta POST e un parametro “_method=put”, ad esempio facendo un POST a

  http://api.cosm.com/v2/feeds/1977?_method=put

- Per ottenere la lista dei feed occorre inviare un messaggio GET a

  http://api.cosm.com/v2/feeds

  Esso ritorna una pagina con la lista dei feed Cosm che sono visibili dagli account autenticati con una dimensione di default di 50 feed per pagina. La visualizzazione può essere personalizzata con diversi parametri, inclusi

quelli che permettono una ricerca basata sulla posizione.
Questi parametri possono essere combinati utilizzando la normale notazione HTTP GET:

http://api.cosm.com/v2/feedslat=51.52&lon=0.13&distance=100.0&q=energy

- La visualizzazione di un feed richiede invece un messaggio GET a

  http://api.cosm.com/v2/feeds/feed_id

  Tale messaggio ritorna il più recente datastream da un environment ID [number]. Se quell'ambiente è un environment "automatico" (fornisce i dati a seguito di una richiesta), allora richiedere i dati attraverso Cosm farà da trigger anche per una richiesta di aggiornamento dei dati, così che la prossima (dopo la soglia dei 5 secondi) ritorni i dati aggiornati.
  L'attributo "updated" (in EEML e JSON) indica quando questo dato è stato salvato l'ultima volta da Cosm. L'elemento "status" indica invece cose leggermente differenti a seconda che si tratti di un feed "manual" o "automatic". Nel primo caso "live" significa che l'environment remoto o il device ha aggiornato Cosm (solitamente con una PUT) con i propri dati negli ultimi 15 minuti, "frozen" significa invece che è stato aggiornato più di 15 minuti fa. Nel secondo, "live" significa che Cosm ha ricevuto con successo i dati dall'environment remoto o dal dispositivo negli ultimi 15 minuti, mentre "frozen" vuol dire che o non ha ricevuto dati negli ultimi 15 minuti, oppure ha provato a riceverli (sempre entro 15 minuti) ma non ha avuto successo.
  È anche possibile filtrare i datastream ritornati con il feed usando il parametro "datastreams" e inviando datastream ID separati da una virgola:

  http://api.cosm.com/v2/feeds/10011.json?datastreams=energy,power

- In qualsiasi momento è infine possibile cancellare un feed: la richiesta

DELETE non ha bisogno di un formato per essere usata. Una richiesta fatta a

http://api.cosm.com/v2/feeds/feed_id

cancellerà l'oggetto riferito dall'ID. L'azione è definitiva e non può essere annullata.

2.3.2 Interagire con i sensori

La temperatura del nostro frigorifero è misurata da un termometro, che diventa così il nostro sensore da aggiungere al feed. Cosm non pone limiti al numero di sensori che è possibile aggiungere al nostro ambiente.
Un datastream rappresenta un unico sensore/stream di un dispositivo all'interno di un environment. È personalizzabile con appositi attributi, di cui obbligatori sono lo Stream ID (l'identificativo dello stream) e il Current Value (il valore attuale).
Le API prevedono la possibilità di interagire con il datastream.

- Per creare un datastream occorre inviare una richiesta POST all'URL

  http://api.cosm.com/v2/feeds/feed_id/datastreams

  In questo modo si crea un nuovo datastream in un environment [feed ID]. Il corpo della richiesta dovrebbe contenere un rappresentazione in formato JSON, XML o CSV del datastream da creare.
- L'aggiornamento di un datastream avviene tramite una PUT a

  http://api.cosm.com/v2/feeds/feed_id/datastreams/1

  Questo metodo aggiorna il singolo datastream [datastream ID] o quello di un environment [environment ID].
- Per ottenere la lista dei datastream è sufficiente richiedere il feed.
- La visualizzazione di un datastream funziona come nei “feed”, solo ritorna il datastream richiesto.

- Cancellare un datastream è infine possibile inviando una richiesta DELETE a

  http://api.cosm.com/v2/feeds/feed_id/datastreams/1

  Essa richiede un formato per essere utilizzata. Una richiesta effettuata a questo URL cancellerà l'oggetto riferito dall'ID. L'azione è definitiva e non può essere annullata.

2.3.3 Le misurazioni

Ciascun sensore, se attivo, invia al server Cosm una misurazione effettuata in un preciso momento.
Essa è un datapoint, che rappresenta il valore di un datastream in un certo momento ed è semplicemente una coppia chiave valore del timestamp e del suo valore in quel momento. Come conseguenza di ciò, gli attributi con cui descrivere il datapoint sono soltanto due, entrambi obbligatori: At, che indica il preciso istante temporale della misurazione, e Value, che indica il suo valore.
Il timestamp esatto, in formato ISO 8601, è molto importante in quanto questo è il modo con cui il datapoint viene referenziato in maniera univoca. I timestamp sono forniti con 6 cifre decimali e devono essere univoci in ciascun datastream (in quanto il datapoint fa riferimento allo stato del datastream in un certo istante di tempo, di conseguenza non può essere in più di uno stato contemporaneamente).

- Per creare un datapoint occorre una richiesta POST all'URL

  http://api.cosm.com/v2/feeds/feed_id/datastreams/1/datapoints

  Ciò creerà un nuovo datapoint per questo datastream, cosa che permette di inserire i datapoint nella storia del datastream. I datapoint dovrebbero avere un timestamp univoco. Inviare nuovi datapoint con lo stesso timestamp di quelli già esistenti sovrascriverà i vecchi con i nuovi. Se un singolo update contiene più valori con lo stesso timestamp (assieme ad

altre registrazioni), allora il risultato saranno datapoint registrati per tutti i timestamp univoci, mentre per quelli duplicati si salverà solo l'ultimo processato.

Al momento si possono inviare un massimo di 500 datapoint in un singolo update. Se si prova ad inviarne di più si avrà come risultato un errore, con la conseguenza che nessun datapoint verrà salvato.

- Aggiornare un datapoint richiede una PUT a

  http://api.cosm.com/v2/feeds/feed_id/datastreams/1/datapoints/2010-07-28T07:48:22.014326Z

  modificherà il valore del datapoint per questo timestamp.
- Per ricevere tutte le informazioni disponibili in un feed o in un datastream, occorre applicare parametri di storia al feed o al datastream genitore, specificando i parametri *start* ed *end*.
- È possibile visualizzare un datapoint tramite una richiesta GET a

  http://api.cosm.com/v2/feeds/feed_id/datastreams/1/datapoints/2010-07-28T07:48:22.014326Z

  ritornerà il valore dello specifico timestamp. Questo metodo è fornito per completezza, in quanto la maggior parte delle applicazioni richiederanno diversi range di dati dal livello datastream.
- Per cancellare un datapoint è necessario inviare la richiesta DELETE a

  http://api.cosm.com/v2/feeds/*feed_id*/datastreams/1/datapoints/2010-07-28%2007:48.014326Z

  Essa non necessita di un formato specifico per essere utilizzata. Una richiesta effettuata a questo URL cancellerà l'oggetto referenziato dall'ID. L'azione è definitiva e non può essere annullata.
- Nel caso si volessero cancellare più datapoint, una richiesta DELETE a

  http://api.cosm.com/v2/feeds/feed_id/datastreams/1/datapoints/2010-07-

28%2007:48.014326Z

rimuoverà un insieme di datapoint dal datastream. Fornendo un timestamp di start e uno di end come parametri della query, è possibile rimuovere tutti i datapoint presenti tra quelle due date. Se si invia la richiesta con il solo timestamp start, tutti i datapoint successivi a tale valore saranno rimossi. Similmente, fornendo solo il timestamp end verranno cancellati tutti i datapoint ad esso precedenti. Questo endpoint supporta inoltre un parametro *duration* (ad esempio "duration=3hours") che cancellerà tutti i datapoint da un timestamp start a quello start + duration (se si fornisce il parametro start), o da un timestamp end ad uno end – duration (in caso si fornisca l'end).

2.4 Chiavi Api e modello dei dati

Nella versione più recente di Cosm è possibile creare chiavi API con permessi molto più granulari che in passato. Si può permettere un accesso read-only a tutte le risorse pubbliche, oppure la scrittura ad uno specifico feed o da uno specifico IP.

I permessi per una key si propagano in cascata, così che se una chiave non include alcuna restrizione specifica, allora potrà avere accesso a tutte le risorse a cui accede l'utente che l'ha creata, così come se si limita l'accesso di una chiave ad un feed particolare, allora essa avrà lo stesso livello di accesso di tutti i datastream contenuti in tale feed.

È possibile operare sulle chiavi con i seguenti metodi.

- Creare un'API key inviando una richiesta POST a

  http://api.cosm.com/v2/keys

  Questo metodo si usa per creare una nuova API key (conosciuta anche come Secure Sharing Key). I dati per creare la chiave possono essere inviati solo o in formato JSON o XML.
- Aggiornare un'API key non è possibile. Occorre cancellare la vecchia

chiave e crearne una nuova.

- Per fare la lista delle API key serve una GET all'URL

  http://api.cosm.com/v2/keys

  Si otterranno tutte le chiavi utente. Al momento questa lista non è organizzata in pagine, per cui ritornerà tutte le chiavi in un solo documento.
- Per visualizzare un'API key è sufficiente una richiesta GET all'URL

  http://api.cosm.com/v2/keys/<key_id>

- La cancellazione di un'API key è possibile inviando una DELETE a

  http://api.cosm.com/v2/keys/<key_id>

2.4.1 Autenticazione

Le chiavi API sono utilizzate per controllare l'accesso alle risorse attraverso le API stesse. Possono essere inviate utilizzando due metodi: come l'header di una request o come un parametro nell'URL della request (il primo è preferibile per motivi di sicurezza).

Inviare tramite un HTTP header è il metodo raccomandato: nonostante continui a non essere sicuro se inviato su di una connessione non criptata, è meno probabile che venga rintracciato come parte dell'URL.

X-ApiKey: YOUR_API_KEY_HERE

Inviandolo come un parametro della request, la forma sarà invece questa:

http://api.cosm.com/v2/feeds?key=YOUR_KEY_HERE
Di conseguenza, delle tipiche request GET potrebbero essere:

```
GET /v2/feeds/504.xml HTTP/1.1
Host: api.cosm.com
X-ApiKey: ENTER_YOUR_PACHUBE_KEY_HERE
```

oppure

```
GET /v2/feeds/504.xml?key=ENTER_YOUR_PACHUBE_KEY_HERE HTTP/1.1
Host: api.cosm.com
```

È inoltre possibile utilizzare un'autenticazione di base per identificarsi al servizio API, ma è comunque raccomandato usare le chiavi API se si stanno inserendo le proprie credenziali in un'applicazione onde evitare di mettere a rischio la propria password.

2.4.2 La gerarchia dei dati

Il modello dei dati per le chiavi API può essere rappresentato con una gerarchia di oggetti in questo modo:

Key → Permission → Resource

Una Key deve avere almeno un oggetto Permission come parte della sua struttura, mentre gli oggetti Resource sono completamente opzionali. Gli oggetti contengono i seguenti attributi:

- **Key**: la chiave è la struttura dati di livello più alto. Ha tre attributi obbligatori: api_key, label e permissions.
- **Permission**: almeno un oggetto Permission è richiesto quando si crea una chiave, ma per quelle più complesse se ne può usare più di uno. L'unico attributo obbligatorio è access_methods.
- **Resource**: Resource può essere aggiunto agli oggetti Permission in maniera opzionale e può restringere l'accesso al feed o al datastream

specificato. L'attributo feed_id è obbligatorio solo se è specificato anche un datastream.

2. 5 Trigger

I trigger (ovvero le “notifiche”) forniscono capacità “push” inviando richieste HTTP POST ad un URL di propria scelta quando le condizioni da noi imposte su di un datastream sono state soddisfatte. Possono essere creati tramite l'interfaccia web o utilizzando le API, caso in cui sono anche accessibili da api.cosm.com previa autenticazione.

L'intervanno minimo tra l'invio di una notifica e l'altro è di 5 secondi. I trigger partono quando la condizione è raggiunta e non si ripetono fino a quando essa rimane tale. Se si ha bisogno di sapere quando la condizione non è più rispettata si può creare un altro trigger con le impostazioni opposte.

Il trigger effettua una richiesta HTTP POST all'URL fornito alla sua creazione e include un parametro “body” in JSON simile a questo:


```
{
  "environment": {
    "description": "",
    "feed": "http:\/\/api.cosm.com\/v2\/feeds\/343",
    "id": 343,
    "location": {
      "lat": 55.74479,
      "lng": -3.18157,
      "name": "location description"
    },
    "title": "test feed yes"
  },
  "id": 1,
```

```
  "threshold_value": 9.0,
  "timestamp": "2009-09-07T12:16:02.001403Z",
  "triggering_datastream": {
    "id": "0",
    "url":
"http:\/\/api.cosm.com\/v2\/feeds\/343\/datastreams\/0"
,
    "at": "2009-09-07T12:16:02.000063Z",
    "value": {
      "current_value": "9.07624035140473",
      "max_value": 9.99650150341,
      "min_value": 0.00471012639984
    }
  },
  "type": "gte",
  "url": "http:\/\/api.cosm.com\/v2\/triggers\/1"
}
```


Possibili valori per il trigger “type” sono:

```
gt   greater than
gte  greater than or equal to
lt   less than
lte  less than or equal to
eq   equal to
change    any change
frozen    no updates for 15 minutes
live      updated again after being frozen
```

- Creare un trigger è possibile con una richiesta POST a

  http://api.cosm.com/v2/triggers/

I dati possono essere codificati in XML o JSON.

- Per aggiornare un trigger si invia una richiesta POST a

http://api.cosm.com/v2/triggers/0

Occorre che il trigger esista già. Qualsiasi tentativo di cambiare environment_id o stream_id è ignorato.

- Fare la lista dei trigger richiede una GET a

  http://api.cosm.com/v2/triggers

  Fornisce una lista di tutti i trigger per l'account autenticato.
- Si può visualizzare un trigger con una GET a

  http://api.cosm.com/v2/triggers/0

  Usare questo metodo per ottenere la rappresentazione XML o JSON di un singolo trigger.
- La cancellazione di un trigger richiede invece una DELETE a

  http://api.cosm.com/v2/triggers/0

  Cancella l'oggetto trigger indicato. L'azione è definitiva e non può essere annullata.

2.6 Utenti

Le API per gli utenti sono progettate per permettere ad applicazioni terze di essere costruite basandosi sulle API Cosm. Permettono allo sviluppatore di creare, modificare, gestire e cancellare utenti. Gli utenti creati "appartengono" allo sviluppatore che li ha generati nel senso che creare un account lo rende fondamentalmente un admin per il nuovo utente. Gli attributi disponibili tramite le API sono indicate sotto.

Per poter creare un utente tramite le API occorre che il proprio account abbia tale funzionalità attivata, ma è possibile visualizzare e aggiornare le proprie preferenze in qualsiasi momento. Se si desidera avere questa possibilità è sufficiente

contattare gli autori al fine di farsela abilitare.

- Per creare un utente occorre una richiesta POST a

  http://api.cosm.com/v2/users

  Questo metodo creerà un nuovo utente con i permessi di ruolo specificati. Occorre che il ruolo sia indicato. Sono accettati solo i formati XML e JSON.
- Aggiornare un utente si può fare con una PUT a

  http://api.cosm.com/v2/users/mylogin

- Fare la lista degli utenti richiede una GET a

  http://api.cosm.com/v2/users

  Come risultato avremo la lista di tutti gli utenti che l'account autenticato ha creato.
- Per visualizzare i dettagli di un utente specifico inviare una GET a

  http://api.cosm.com/v2/users/mylogin

  La quantità di dati ricevuti dipenderà dal fatto se si è o meno l'utente in questione, se si è il suo proprietario o dalle impostazioni sulla privacy che l'utente ha definito.
  Nei primi due casi si otterranno tutte le informazioni disponibili, altrimenti la quantità di dati varierà da un minimo del solo username dell'utente ad eventuali altri dati che egli ha dichiarato visibili sul proprio profilo.
- La cancellazione di un utente è un'azione definitiva e non può essere annullata. Si effettua con una DELETE a

  http://api.cosm.com/v2/users/mylogin

2.7 Ottenere la storia dei feed

Una delle nuove caratteristiche introdotta nella versione 2 delle API è la possibilità di fare query su un certo range di dati storici usando date formattate secondo ISO 8601. Possono essere fatte sia a livello datastream che environment, ma sono limitate entro una certa finestra per ciascuna richiesta alle API. Se per esempio si volessero ottenere tutti i datapoint della settimana precedente bisognerebbe fare sette richieste individuali alle API, ciascuna con una finestra di 24 ore.

Le query storiche vengono notificate inserendo i parametri rilevanti nell'URL della richiesta, sia per i feed che per i datastream:

- start: indica il punto di partenza di una query sotto forma di datastamp. Il valore di default è quello vuoto;
- end: indica il punto finale sotto forma di timestamp dei dati ritornati. Il valore di default è settato al timestamp attuale;
- duration: indica la durata della query;
- page: definisce in quale pagina stiamo guardando per cercare valori che combacino. Se non settato ha valore 1;
- per_page: indica quanti valori sono ritornati per pagina. Se non settato vale 100, il massimo è 1000;
- time: ritorna il feed con i valori che aveva ad un determinato timestamp;
- find_previous: ritorna i dati precedenti all'intervallo di date richiesto;
- interval_type: se impostato su “discrete” i dati verranno forniti nel formato dell'intervallo specificato. Se non settato, darà indietro datapoint raw;
- interval: specifica quale intevallo di dati è richiesto ed è definito in secondi tra i datapoint. Se un valore passato non combacia con questi, sarà arrotondato a quello successivo.

Per leggere lo storico dei feed si devono applicare i parametri in una richiesta GET ad un certo feed.

```
http://api.cosm.com/v2/feeds/504.json?start=2010-08-
02T14:01:46Z&end=2010-08-02T17:01:46Z&interval=0
```

Lo stesso vale per i datastream.

```
http://api.cosm.com/v2/feeds/504/datastreams/0.csv?
start=2010-08-02T14:01:46Z&end=2010-08-
02T17:01:46Z&interval=0
```

Capitolo 3

PUBSUBHUBBUB

Introdotto nel Capitolo 1.3.3, ora ci preoccuperemo di analizzare più in dettaglio questo protocollo. Introdurremo i modelli disponibili quando si parla di interazione in rete e si cercherà di capire le motivazioni che hanno portato alla realizzazione di PuSH. Si darà inoltre una descrizione dettagliata di come avviene il dialogo tra le parti in gioco, così da comprendere meglio il funzionamento di ciascuno degli interessati.

3.1 Confronto tra modelli

I modelli principali con cui si lavora quando si parla di Internet e di diffusione di dati sono due: Pub/Sub e Push/Pull.
Entrambi sono modelli di tipo cliente/servitore. Non è tuttavia possibile stabilire a priori quale dei due sia da prediligere: come spesso accade, solo dopo un'analisi del contesto in cui ci troviamo ad operare è possibile fare una scelta consapevole.

3.1.1 Il modello Publish/Subscribe

Alla base di questo modello c'è l'interesse di un utente verso determinate informazioni. Il cliente deve effettuare una registrazione all'argomento di suo interesse presso un apposito gestore, il quale tiene traccia di tutte le richieste.

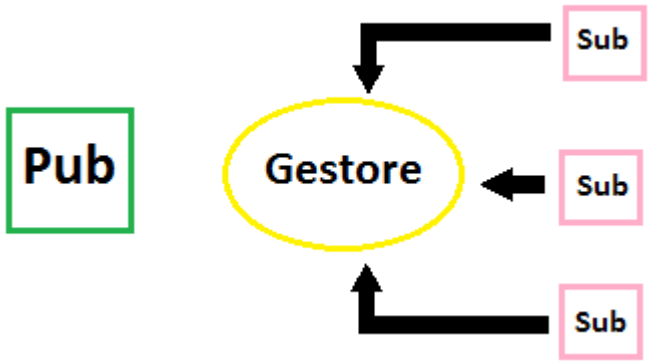

Fig. 10: Gli interessati si sottoscrivono

Il gestore riceve poi i nuovi dati da un publisher e provvede poi ad inviare le informazioni appena ricevute a tutti i clienti interessati.

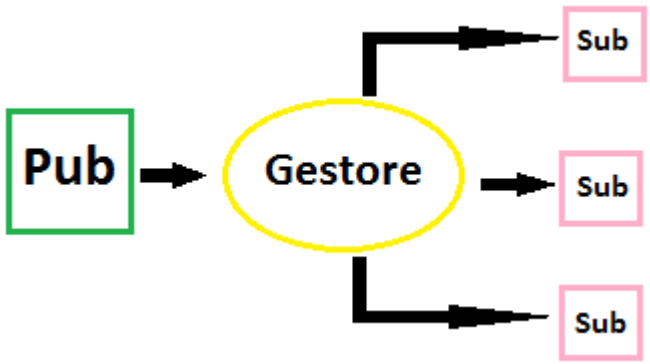

Fig. 11: La distribuzione dei dati

3.1.2 Il modello Push/Pull

In questo modello abbiamo sempre una delle due parti in gioco (il cliente e il servitore) che esegue un'azione esplicita per ottenere o inviare i dati.
Nel modello pull è il cliente che ha sempre l'iniziativa: è lui che contatta il server per chiedergli il servizio di cui ha bisogno. Sarà sua premura occuparsi di gestire la risposta e le eventuali situazioni in cui questa non dovesse arrivare.

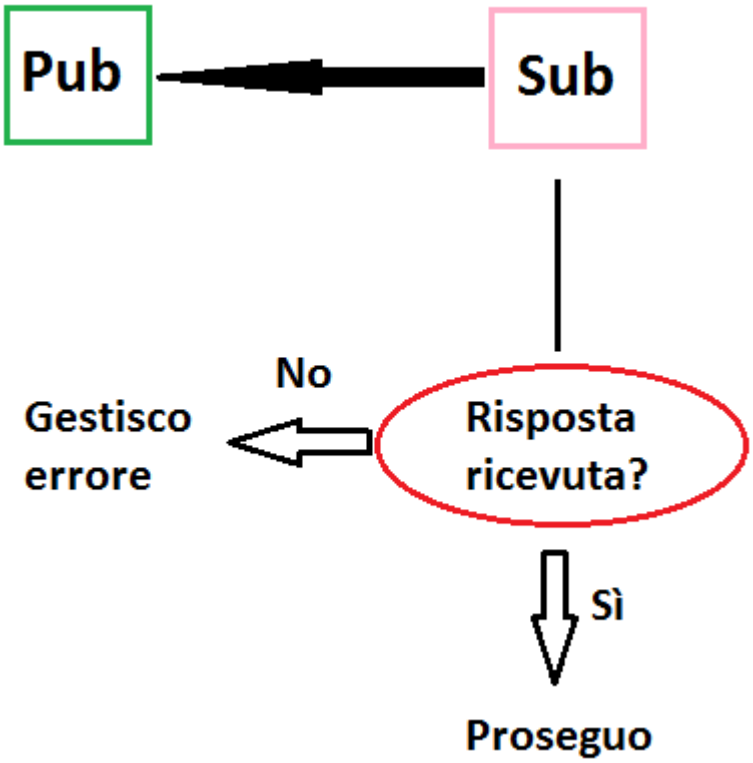

Fig. 12: Il subscriber chiede il dato e gestisce il risultato

In quello push, invece, il server diventa a sua volta cliente di ciascun cliente: quest'ultimo provvede infatti a richiedere il servizio, quindi può scegliere se aspettare la risposta o fare altro nel frattempo. Il server si deve quindi preoccupare di consegnare il risultato al cliente, facendo gli opportuni controlli che questo sia arrivato.

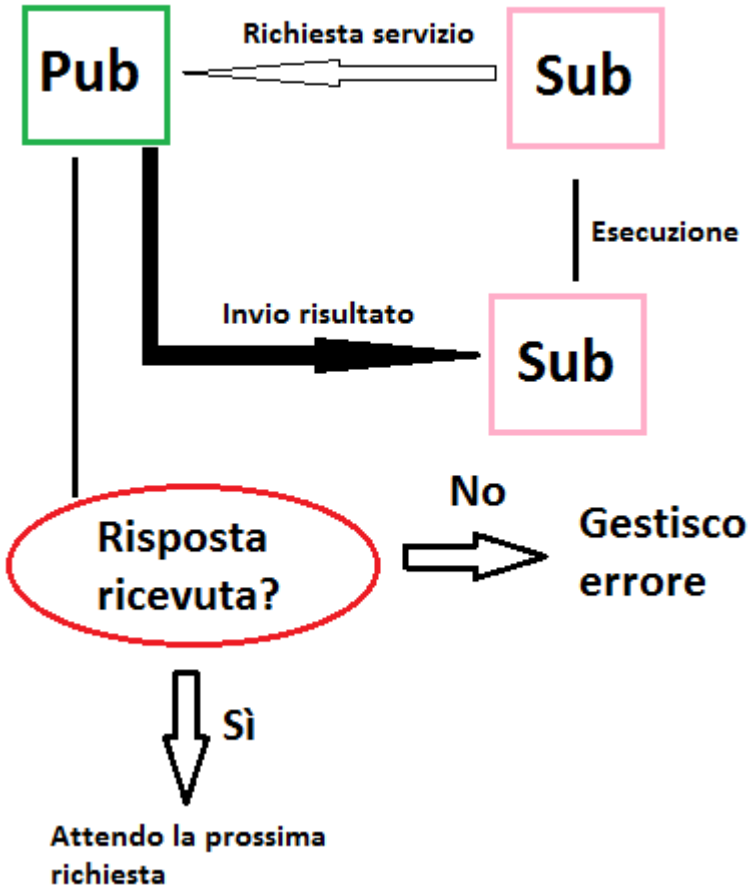

Fig. 13: Qui è il Publisher ad avere il maggior carico

Entrambi questi casi sbilanciano il carico computazionale su una delle due parti: cliente per il pull, servitore per il push.

3.1.3 Differenze tra i modelli

La differenza principale tra i due modelli è sicuramente quella di come clienti e servitori interagiscano tra loro. Se nel caso Pub/Sub si ricorre ad un intermediario, che riceve le richieste di entrambe le parti, in quello Push/Pull sono esse a richiedere o fornire direttamente un servizio.

In caso sia noto che le forze in gioco abbiano una forte disparità in termini di prestazioni, con il secondo modello si può scegliere quale delle due parti caricare maggiormente così da non sovraccaricare quella più debole. Nel primo, invece, la scelta effettuata è concettualmente differente: clienti e servitori non hanno compiti

aggiuntivi; i primi si preoccupano solo di registrarsi all'argomento di interesse e di ricevere i nuovi risultati, i secondi di fornirli. Spetterà poi al gestore coordinare l'invio e la ricezione dei dati.
PubSubHubBub ha scelto il modello Pub/Sub nel tentativo di creare un nuovo standard per ridurre il più possibile l'utilizzo del polling in Internet: il modello Pull prevede che siano i diretti interessati a richiedere le nuove informazioni. Questo fa sì che possano essere effettuate interazioni inutili: un cliente potrebbe voler ricevere dati ogni 15 minuti, ma non è detto che il nuovo invio corrisponda effettivamente ad una nuova informazione, se il dato non è stato modificato. Ritrasmettere qualcosa di già noto occupa inutilmente risorse, cosa da evitare ove possibile.
Addentrandoci più in dettaglio nel modello, tuttavia, osserviamo come all'interno di un comportamento tipicamente Pub/Sub abbiamo singole interazioni di tipo Push/Pull.
Vediamo ora il tutto con maggior precisione.

3.1.4 PubSubHubBub e il modello Pub/Sub

In PubSubHubBub i clienti e il servitore interagiscono con un gestore chiamato Hub.
Il protocollo può essere brevemente descritto come segue:

- un feed URL (ovvero un argomento, un topic) dichiara il/i server hub nel suo file XML Atom o RSS utilizzando il tag <*link rel=”hub” ...*>. Gli hub possono essere avviati dal publisher del feed oppure essere gestiti dalla comunità, liberi di essere usati da chiunque.
- Un *subscriber* (un server che è interessato ad un topic) fa inizialmente il fetch dell'URL Atom come normalmente previsto. Se il file Atom/RSS dichiara i suoi hub, il sottoscrittore può allora evitare il ripetuto polling dell'URL, registrandosi invece all'hub del feed e sottoscrivendosi ai suoi aggiornamenti.
- Il sottoscrittore si sottoscrive al Topic URL dal Topic URL dichiarato

nell'HUB.

- Quando il *publisher* in seguito aggiorna il Topic URL, il software legato al publisher sveglia l'hub dicendogli che c'è un aggiornamento.
- L'hub fa il fetch in maniera efficiente del feed pubblicato e invia tramite un multicast di livello applicativo il contenuto nuovo o modificato a tutti i sottoscrittori che si sono registrati.

Non c'è nessuna compagnia che controlli o gestisca PuSH. Chiunque può avviare un hub, in alternativa si possono utilizzare hub aperti per pubblicare e sottoscriversi.
PubSubHubBub è un protocollo giovane, nato negli ultimi anni e ancora lontano dalla sua specifica definitiva: sorge quindi spontaneo chiedersi perché optare per una soluzione che può variare in futuro e o magari correre il rischio di non avere la diffusione che ci si aspetterebbe.
Per cominciare è utile fare un'opportuna suddivisione tra i protocolli che lavorano con un trasferimento dati di tipo *fat pinging* (PuSH, XMPP pubsub) e quelli che invece utilizzano *light pinging* (rssCloud, XML-RPC ping, changes.xml, SUP, SLAP).

3.2 Light ping

Il primo metodo per il trasferimento dei dati è quello chiamato light ping. Come il nome lascia presagire, si tratta di un trasferimento leggero: i protocolli che implementano questo scambio di dati trasmettono esclusivamente l'URL del feed che è stato aggiornato. Spetterà poi al ricevente preoccuparsi di cercare le informazioni aggiornate.

3.2.2 XML-RPC ping

È un protocollo di Remote Procedure Calling (chiamata a procedura remota) che si basa su Internet. Un messaggio XML-RPC è una richiesta HTTP-POST. Il body è in formato XML e fa eseguire una procedura al server, il quale fornisce poi il

risultato sempre in XML.
L'obiettivo di questo protocollo è di avere una base solida con cui lavorare in diversi ambienti. Si è cercato di ottenere un formato pulito e semplice da estendere, così da permettere un suo rapido adattamento a nuovi environment o su altri sistemi operativi.
Le maggiori critiche che possono essere mosse riguardano proprio l'eccessivo overhead che si genera durante la trasmissione delle richieste: tutto ciò che si fa con un messaggio XML-RPC potrebbe essere fatto anche con XML. Si va quindi ad aggiungere ulteriore pesantezza ad un metalinguaggio già sostituito da qualcosa di ancora più efficiente (JSON).
Un eccessivo scambio di dati può portare al thundering herd, il cui risultato è simile a quello di un attacco Denial Of Service (DoS), attacco a cui XML-RPC non è immune.

3.2.3 RssCloud

Le API RssCloud sono un servizio web XML-RPC e SOAP 1.1 che consentono al client di ricevere notifiche sugli aggiornamenti di documenti RSS. Un server chiamato “cloud” (ovvero nuvola) riceve le richieste di notifica riguardo ai cambiamenti di un particolare documento RSS.
In maniera simile a PuSH, RssCloud vuole porre rimedio all'utilizzo del polling per lo scambio di dati. Riesce tuttavia solo in parte nel suo scopo: se delle notifiche se ne occupa il server, eliminando così la necessità che i client effettuino continuamente il polling dei dati, questi devono poi andare a recuperare l'informazione aggiornata. Ciò può dar vita al thundering herd, con conseguenze simili a quanto visto per XML-RPC.
La sottoscrizione viene automaticamente rimossa dopo 25 ore e ciò forza il client a chiedere il monitoraggio di un determinato URL ogni 24 ore, se ha la necessità di continuare a ricevere gli aggiornamenti.

3.2.3 SUP

SUP (Simple Update Protocol) è stato sviluppato come un semplice ping feed (un avviso che il feed è cambiato) per segnalare ai servizi web che uno o più dei feed che stanno monitorando è cambiato e devono quindi recuperare il nuovo valore. Eliminando il polling ripetuto ad intervalli frequenti, il protocollo offre già una buona ottimizzazione delle risorse se paragonato al caso in cui non sia utilizzato. La necessità di un controllo da parte dell'applicazione web rimane comunque presente: SUP aiuta ad allargare i tempi tra un controllo e l'altro, ma l'aleatorietà della rete forza in ogni caso ad accertarsi che non si siano persi degli aggiornamenti per problemi di connessione o altro.
Per alleggerire il trasferimento dati si è scelto di utilizzare il SUP-ID al posto dell'URL del feed: è un identificativo inserito all'interno del tag <link> che ha lo scopo di evitare eventuali ambiguità nel caso di URL simili, di abbreviare la lunghezza dei feed di aggiornamento e di poter essere utilizzato nel caso in cui il dato dovesse avere qualche riferimento all'interno di un database.
Il formato scelto per i dati è JSON, più leggero di XML, ma altrettanto semplice da gestire. È comunque una trasformazione in più: tipicamente lavoriamo con feed Atom o RSS e vorremmo poter inviare direttamente quelli senza perdere inutilmente tempo in trasformazioni di vario tipo.

3.2.4 SLAP

Simple Lightweight Announcement Protocol è un protocollo che si pone lo stesso obiettivo di quelli visti sino ad ora: fornire un sistema rapido per avvisare della disponibilità di nuovi contenuti senza che gli interessati debbano fare un polling ripetuto.
A differenza di quelli analizzati in precedenza, SLAP non utilizza HTTP per trasferire le informazioni, bensì UDP. L'idea alla base di questa scelta è di non impiegare troppe risorse al livello più basso dell'architettura: in HTTP si ricorre alle connessioni TCP e questo appesantisce la trasmissione. Scegliere UDP è tuttavia un'arma a doppio taglio: se da un lato ne guadagnano le prestazioni,

dall'altro diminuisce l'affidabilità dell'invio. Le specifiche di questo protocollo non sembrano dare troppo peso a questa cosa: si cita infatti la possibilità da parte del subscriber di contattare il publisher per assicurarsi di avere ricevuto tutto. Questo, assieme al fatto che non si trasmette l'intero nuovo feed, ma solo la sua presenza, vanifica il risparmio sull'overhead ottenuto con UDP.
SLAP manca inoltre di un'implementazione specifica: sulla sua realizzazione è lasciato campo libero al programmatore, che si deve anche preoccupare di garantire la sicurezza nello scambio di informazioni.

3.3 Fat ping

Quando parliamo di fat ping dobbiamo fare attenzione a non farci fuoriviare dal nome. Un aspetto importante nel trasferimento di dati è limitare il più possibile l'overhead. Il termine fat potrebbe quindi far pensare ad un qualcosa di particolarmente pesante e poco adatto allo scambio di informazioni, ma non è così: esso fa semplicemente riferimento al fatto che, quando si notifica la presenza di un aggiornamento, invece di inviare soltanto l'URL che indica dove andare a reperirlo, si manda proprio il nuovo feed prodotto.
PuSH rientra in questa categoria: le nuove informazioni vengono inviate tramite un modello *push*, cosa che elimina la necessità da parte dei subscriber di effettuare polling (anche se ad intervalli di tempo ampi, come succede per alcuni protocolli light ping).

3.3.1 XMPP pubsub

XMPP è un'applicazione di XML che permette lo scambio di dati strutturati quasi in real time.
Per la comunicazione è necessario conoscere l'indirizzo IP e la porta del destinatario, quindi aprire una connessione TCP e inviare su di essa uno stream XML, possibilmente criptando il canale tramite Transport Layer Security (TLS). In seguito occorre autenticarsi usando un meccanisco Simple Authentication and Security Layer (SASL), scambiarsi i dati e chiudere lo stream XML e la

connessione TCP.
I passaggi appena elencati fanno capire facilmente come l'implementazione di client e server per questo protocollo non siano esattamente votati alla semplicità, cosa che può giocare a svantaggio di chi non abbia buone conoscenze di rete. Lo scambio di messaggi, in particolare, è lontano dall'obiettivo che si vorrebbe raggiungere, ovvero l'invio diretto del feed aggiornato senza modifiche di sorta. Se da un lato è garantita una buona sicurezza nello scambiarsi le informazioni, dall'altro ne risente la complessità del progetto, che viene decisamente aumentata.
Ciò impedisce inoltre il deploy di un'eventuale applicazione su host a basso costo, obbligando ad esborsi economici maggiori chi volesse costruire un'infrastruttura basata su XMPP.

3.4 PubSubHubBub in dettaglio

Con l'introduzione di PubSubHubBub, gli autori hanno voluto proporre un nuovo protocollo che andasse a sopperire alle mancanze di tutte le precedenti proposte, condividendo con esse lo stesso obiettivo finale: fornire notifiche nel minor tempo possibile su feed aggiornati, eliminando così la necessità da parte dei subscriber di effettuare polling ripetuto.
Se nei protocolli citati nel capitolo precedente si aveva sempre una modifica di qualche tipo ai dati trasmessi, con PuSH questo non avviene: i dati trasmessi sono sempre nel formato originale, Atom o RSS.
Tre sono le parti in gioco: publisher, hub e subscriber. Tutta la complessità dovrebbe essere posta sull'hub, così da mantenere publisher e subscriber il più semplici possibili.

3.4.1 Schema di funzionamento

Prima di descrivere l'architettura di PubSubHubBub è bene partire dando una visione generale del suo funzionamento. Lo schema generale che lo rappresenta è il seguente:

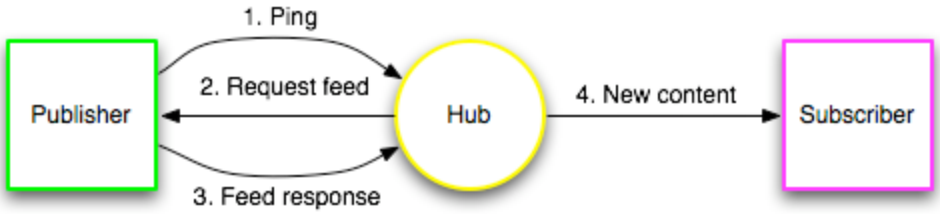

Fig. 14: Schema base di PubSubHubBub

Il publisher contatta l'hub per avvisarlo che c'è un nuovo contenuto disponibile, questi provvede poi a farselo inviare e lo gira a tutti i subscriber interessati. Ciò libera il publisher dal compito di inviare lui stesso gli aggiornamenti, cosa che potrebbe richiedere un ingente quantitativo di risorse.

Quando si è parlato di modello Push/Pull all'interno del protocollo ci si riferiva proprio a questi due passaggi.

Il publisher effettua una notifica push all'hub per avvisarlo della presenza di un nuovo valore.

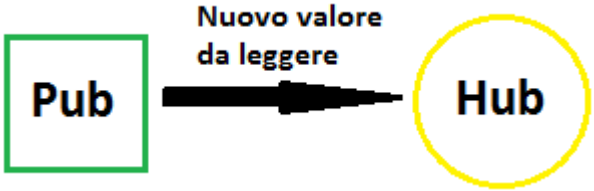

Fig. 15: Qui si ha una push del dato

L'hub andrà poi a chiedere al publisher tale nuovo contenuto, effettuando quindi un pull del dato da smistare ai subscriber.

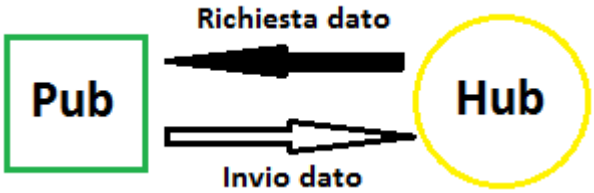

Fig. 16: L'hub "tira" a sè il nuovo valore

Questo schema può ulteriormente essere migliorato unendo publisher e hub, così da non occupare le risorse di rete richieste dai primi tre passaggi. Ci ritroveremmo di conseguenza in questa situazione.

Fig. 17: Uniamo il Publisher e l'Hub

Dal caso particolare si passa quindi a quello generale, idealmente composto da moltissime entità, ciascuna con le sue necessità.

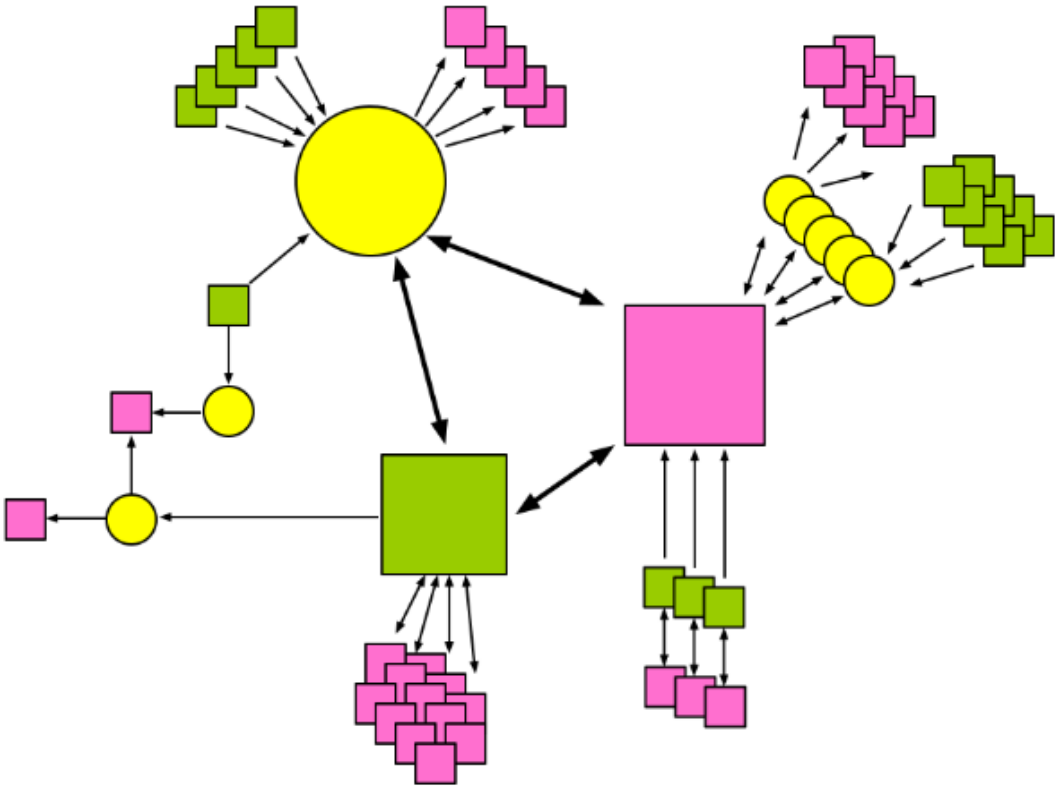

Fig. 18: Molte entità prendono parte al protocollo

La mia informazione potrebbe passare attraverso diversi hub prima di giungere a destinazione, in alternativa potrei sceglierne diversi di piccole dimensiono o uno solo in grado di gestire tutte le richieste che riceve.

3.4.2 L'architettura del protocollo

Da una visione più generale di PuSH passiamo ora ad un'analisi più attenta della sua architettura. I diagrammi precedenti rappresentano infatti il funzionamento a regime del protocollo e trascurano un aspetto importante dello stesso: la fase di inizializzazione. Il subscriber sembra infatti avere sempre un ruolo puramente passivo, ma non è così: il publisher non sa a priori se ci sarà qualcuno interessato alle informazioni che pubblica, dobbiamo essere noi nelle vesti di subscriber a mostrare interesse verso tali dati. Occorre quindi contattare l'hub per segnalargli che vogliamo ricevere gli aggiornamenti di un determinato feed e che deve inviarli ad un certo URL.

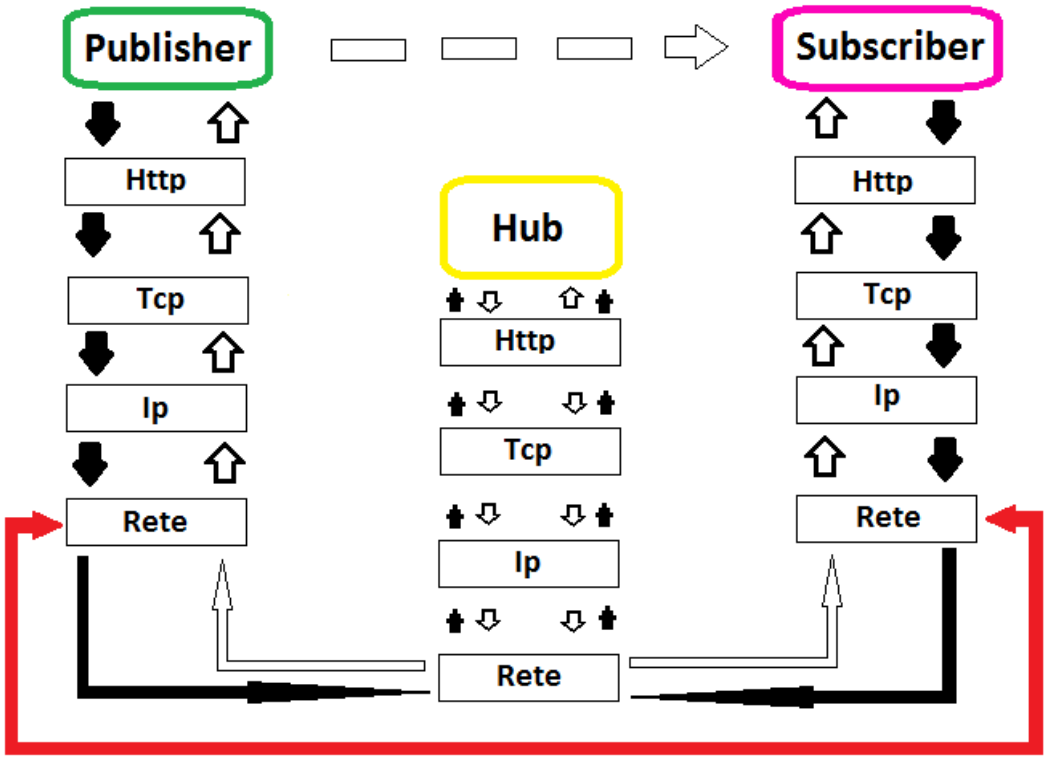

Fig. 19: L'architettura di PubSubHubBub

L'interazione tra le parti in gioco avviene tramite protocollo HTTP. Il primo passo che viene compiuto è evidenziato in rosso: il subscriber contatta il publisher per segnalare la sua intenzione di voler ricevere aggiornamenti. Tramite un apposito valore contenuto nel feed, il publisher indica al subscriber a quale hub si appoggia per la distribuzione dei contenuti, dandogli così modo di iscriversi e di non dover

più fare polling ripetuto per controllare se ci sono nuovi messaggi.
Il publisher invia una richiesta POST all'hub ogni qual volta un nuovo elemento viene aggiunto al feed che rappresenta. L'hub provvede quindi a recuperare tale nuova informazione tramite una richiesta GET. Rimane inoltre in ascolto di eventuali subscriber che vogliono ricevere gli aggiornamenti di un certo feed. Il subscriber invia una richiesta POST all'hub, il quale risponde con una GET per sincerarsi che la richiesta di sottoscrizione sia effettivamente desiderata.
La freccia tratteggiata rappresenta lo scenario ideale che vogliamo raggiungere, ovvero un publisher che invii ai subscriber i suoi aggiornamenti: nel momento in cui in cui l'utilizzatore finale si iscrive ad un feed di interesse, la sua preoccupazione è di ricevere le nuove notizie nel minor tempo possibile. La presenza di passi intermedi deve essere il più trasparente possibile.

3.4.3 La fase di discovery

Quando un subscriber vuole iscriversi ad un feed, inizia contattandolo per scoprire a quale hub dovrà poi segnalare il proprio interesse. Ciascun feed che desideri essere utilizzato con il protocollo PuSH deve pubblicare, come figlio di `//atom:feed` o di `//rss:rss/channel`, un elemento `atom:link` il cui attributo `rel` abbia il valore `hub` e il cui attributo `href` contenga l'URL a cui si trova l'hub.
I feed possono contenere più link agli hub se il publisher ne vuole notificare più di uno. Quando un subscriber li incontrerà, potrà decidere se iscriversi a solo uno o a più di essi.
Gli hub devono utilizzare un solo URL sia per l'interfaccia di pubblicazione che per quella di sottoscrizione.

3.4.4 Il subscriber e il suo funzionamento

La fase di sottoscrizione inizia con il subscriber che effettua una richiesta POST all'URL dell'hub. Tale richiesta ha un Content-Type di tipo `application/x-www-form-urlencoded` e i seguenti parametri:

- hub.callback: richiesto. È l'URL di callback del subscriber a cui le notifiche dovrebber essere inviate.
- hub.mode: richiesto. È una stringa di tipo “subscribe” o unsubscribe”, a seconda di ciò che si vuole fare.
- hub.topic: richiesto. L'URL del feed a cui il subscriber si vuole sottoscrivere.
- hub.verify: richiesto. Può essere “sync” o “async”, a seconda che si voglia una verifica sincrona o meno. In modalità sincrona, la verifica deve essere completata prima che l'hub ritorni una risposta. In modalità asincrona la verifica può essere rimandata in un secondo momento. Questo può essere utile per alleviare il carico dell'hub in caso sia sovraccaricato.
- hub.lease_seconds: opzionale. Numero di secondi per i quali il subscriber desidera tenere attiva la sottoscrizione.
- hub.secret: opzionale. Una stringa segreta fornita dal subscriber, sarà usata per calcolare un digest HMAC per la distribuzione di contenuti autorizzati.
- hub.verify_token: opzionale. Un token opaco fornito dal subscriber che sarà rimandato indietro nella richiesta di verifica per identificare quale richiesta di sottoscrizione sta venendo verificata.

Per sapere se la richiesta è andata è stata verificata e la sottoscrizione è attiva occorre osservare la risposta inviata dall'hub, che deve essere di tipo HTTP 204 “No Content”. Se la sottoscrizione deve essere ancora verificata, invece, la risposta deve avere un codice 202 “Accepted”. In caso di errore l'hub dovrà inviare un codice della famiglia 4xx o 5xx; può anche aggiungere una descrizione dell'errore sotto forma di testo nel body della request.

Non è tuttavia sufficiente che il subscriber invii all'hub una richiesta di sottoscrizione: sono necessari infatti altri due passaggi prima che essa sia confermata. Il primo messaggio è quindi paragonabile alla voglia del subscriber di iscriversi al feed. L'hub a questo punto deve inviare una richiesta GET contenente vari parametri tra cui uno denominato hub.challenge (per i dettagli precisi vedere il paragrafo dedicato all'hub). Affinchè la sottoscrizione vada a buon fine, il subscriber deve rispondere con un messaggio POST che abbia i parametri hub.topic e hub.verify_token uguali a quelli di una richiesta in sospeso e nel body

deve contenere il valore hub.challenge inviato in precedenza dall'hub.

3.4.5 Pubblicare nuovi contenuti

Seguendo il principio per cui publisher e subscriber dovrebbero essere il più leggeri possibili, chi pubblica nuovi contenuti ha solo un compito: inviare all'hub l'URL del topic che è stato aggiornato. Questo avviene con un messaggio POST con Content-Type `application/x-www-form-urlencoded` e i seguenti parametri nel body.

- hub.mode: richiesto. È la stringa “publish”.
- hub.url: richiesto: l'URL del topic che è stato aggiornato.

Se la richiesta è stata ben ricevuta dall'hub, il publisher riceverà un messaggio HTTP con codice 204 “No content”.

3.4.6 Il compito dell'hub

Il cuore del protocollo PubSubHubBub è l'hub, la parte che si occupa di recuperare i nuovi aggiornamenti e distribuirli a tutti i subscriber. Per svolgere il suo compito si affida allo scambio di ben precisi richieste HTTP.
Partendo dall'interazione con il subscriber, sappiamo che quest'ultimo invia una richiesta con determinati parametri quando vuole sottoscriversi. L'hub deve ignorare qualsiasi parametro che non capisce. Deve inoltre consentire ai subscriber di inviare più volte la richiesta di sottoscrizione: quella più recente sostituirà la precedente.
Per evitare che vengano effettuate azioni non volute da parte di malintenzionati, è sempre necessario che le azioni di subscribe e unsubscribe siano confermate da chi le ha inviate. Affinché questo avvenga, l'hub invia una richiesta HTTP GET all'URL di callback del subscriber. Essa contiene i seguenti parametri.

- hub.mode: richiesto. È una stringa “subscribe” o “unsubscribe”, come presente nella richiesta originale del subscriber.
- hub.topic: richiesto. L'URL di interesse fornito al momento della registrazione.

- hub.challenge: richiesto. È una stringa casuale generata dall'hub. Deve essere ritornata anche dal subscriber.
- hub.lease_second: opzionale. Il numero di secondi per cui rimarrà attiva la sottoscrizione. È determinato dall'hub.
- hub.verify_token: è lo stesso fornito dalla richiesta del subscriber, se specificato.

Il subscriber dovrà quindi inviare una risposta HTTP success (2xx) e confermare che i parametri hub.topic e hub.verify_token combacino con una richiesta in sospeso, oltre ad aggiungere nel body la stringa hub.challenge generata dall'hub.
Nel caso sincrono, l'hub deve considerare le risposte con un qualsiasi codice diverso o con il parametro hub.challenge che non combacia come un fallimento della conferma. In quello asincrono, invece, codici di tipo 3xx, 4xx e 5xx e il parametro hub.challenge diverso sono da considerare come un fallimento temporaneo. L'hub dovrebbe provare a verificare la sottoscrizione per un certo numero di volte all'interno di un lasso di tempo più ampio, terminato il quale la sottoscrizione deve essere considerata nulla.
Prima che una sottoscrizione termini, l'hub deve ricontrollare con il subscriber se desidera rinnovarla. Per fare questo invia una richiesta di verifica con l'hub.mode uguale a “subscribe”. Essa deve essere uguale a quella inviata all'inizio (ma con un nuovo hub.challenge). La risposta deve quindi essere interpretata come una nuova richiesta di sottoscrizione e, in caso questa non arrivi, si deve rimuovere quella precedente dopo un numero ragionevole di tentativi di riconferma falliti.
In caso di sottoscrizioni permanenti (ovvero senza il parametro hub.lease_seconds specificato nella richiesta originale), il valore di hub.lease_seconds fornito dall'hub nella richiesta di verifica al subscriber dovrebbe rappresentare il tempo dopo il quale l'hub si inizierà automaticamente il servizio di refresh della sottoscrizione. In questo modo di mantiene semplice il subscriber, ma si permette comunque all'hub di fare pulizia di quelle sottoscrizioni che hanno riscontrato problemi.
Terminata la descrizione dell'interazione tra hub e subscriber, si passa a quella tra hub e publisher che, a sua volta, segue un iter preciso.
Quando un nuovo contenuto viene aggiunto ad un feed, il publisher invia una

notifica all'hub. Questi deve accettare un messaggio POST con Content-Type `application/x-www-form-urlencoded` e i seguenti parametri.

- hub.mode: richiesto. È una stringa “publish”.
- hub.url: richiesto. Si tratta dell'URL del topic che è stato aggiornato.

Se la notifica è andata a buon fine, l'hub deve ritornare una risposta HTTP 204 No Content, in caso contrario un codice di errore (4xx e 5xx).
Per ottenere il nuovo contenuto, l'hub invia una GET all'URL del feed. Se dopo il fetch dei dati si accorge che qualcosa è cambiato, allora deve inviare la nuova informazione a tutti i subscriber. Per fare ciò utilizza un messaggio POST con la lista degli interventi nuovi e modificati verso il callback del subscriber. Questa richiesta deve avere un Content-Type di tipo `application/atom+xml` se nel body è presente un documento Atom, oppure `application/rss+xml` nel caso di un feed RSS.
Nel caso in cui i dati siano consegnati con successo, la risposta dal callback del subscriber deve essere un codice HTTP 2xx. Tutti gli altri vanno interpretati come fallimento: in questo caso si dovrebbe procedere ad effettuare un numero ragionevole di altri tentativi, prima di decretare che ci sono veri e propri problemi. La risposta che arriva dal subscriber indica soltanto che il messaggio è stato ricevuto con successo, non che è anche stato processato a dovere.

3.4.7 PuSH: vantaggi, svantaggi, stato corrente dell'implementazione

Nell'introdurre PubSubHubBub è stato detto che gli autori hanno voluto prendere quanto di buono offerto dai precedenti protocolli al fine di migliorarli, andando a risolvere le loro criticità.
L'idea di fornire un modo per avere notizie in tempo reale prestando contemporaneamente attenzione all'utilizzo delle risorse in gioco è sicuramente l'aspetto più positivo del protocollo. Bisogna però porre attenzione al contesto in cui lo si vuole utilizzare: come possiamo integrarlo con i dispositivi mobile, che spesso sfruttano connessioni dati poco stabili e quindi poco adatte ad interagire con TCP? Che implementazioni sono disponibili per permettere allo sviluppatore di lavorare concentrandosi sulla logica della propria applicazione, piuttosto che

sull'implementazione da zero del protocollo? La situazione in questo caso non è delle più rosee: benché esistano implementazioni in diversi linguaggi, la documentazione spesso è scarsa e non si trovano molte realizzazioni da studiare per capirne al meglio il funzionamento.
A seconda dell'applicazione che abbiamo in mente sta quindi a noi pensare a come poter integrare PuSH nel nostro progetto, così da poter usufruire dei suoi vantaggi anche in ambienti che a prima vista potrebbero sembrare poco adatti a tale protocollo.

3.4.8 Il perché di questa scelta

Cerchiamo di analizzare meglio il perché sia interessante utilizzare questo tipo di protocollo.
Quando si parla di trasferimento dati possono venire subito alla mente le socket. Queste ultime sono uno strumento potente, che ci permettono di lavorare di fino per ottenere esattamente il tipo di trasferimento desiderato. Ci troviamo tuttavia a lavorare con un livello di complessità decisamente superiore.
In uno dei primi prototipi realizzati per testare il corretto funzionamento delle librerie Cosm, il primo problema che si è presentato è stato quello di avvisare il potenziale subscriber che un nuovo dato era disponibile per essere prelevato e letto. Sapendo che avremmo lavorato in ambito mobile, dove la qualità della connessione è tutto fuorché garantita, la scelta per l'invio delle notifiche è ricaduta su socket UDP. Ogni volta che un nuovo valore era generato e inviato al server, un messaggio era spedito al cliente per avvisarlo che poteva procedere al suo recupero. Ciò porta a ragionare sulla sensatezza di tale operazione: se conosco il destinatario, perché non posso inviargli subito il dato? Dopo tutto si tratta di una stringa da aggiungere ad un'opportuna richiesta POST, non un file da svariati Megabyte. Nulla ce lo vieta, ma in questo modo perdiamo l'utilità di Cosm: il dato è “volatile”, ovvero si perde nel momento in cui il nostro cliente non ne ha più bisogno o ha un malfunzionamento. Bisognerebbe pertanto occuparsi anche di gestire la sua persistenza. A questo aggiungiamo un altro quesito: ponendo di avere migliaia di clienti, ha senso che ciascuno di essi tenga in memoria lo storico

degli eventi? La risposta è ovviamente negativa: in caso di problemi di linea, ciascun cliente potrebbe ritrovarsi con una serie di informazioni non sincronizzate, cioè con dati mancanti in quanto non ricevuti. Cosm ci permette invece di effettuare un'apposita query per recuperare le informazioni all'interno di un certo lasso di tempo, cosa che possiamo sfruttare se abbiamo la necessità di avere sempre dati consistenti.

Avvisare direttamente l'interessato che un nuovo dato è disponibile si è dimostrato quindi particolarmente svantaggioso. Vorremmo avere più flessibilità nella nostra comunicazione, separando meglio i compiti di ciascuna entità in gioco: non solo degli endpoint che hanno il compito di svolgere tutto il lavoro (e che potrebbero trovarsi in difficoltà a farlo, se ci rivolgiamo a dei dispositivi mobile), ma anche una architettura organizzata per la diffusione di queste informazioni.

PubSubHubBub ci permette di fare tutto questo: il publisher si occupa dell'elaborazione dei dati ricevuti, l'hub li prende e li distribuisce.

Capitolo 4

RACCOLTA DATI E LORO DISTRIBUZIONE

COSM e PuSH sono due protocolli e architetture tra loro assai differenti. Ad un primo sguardo si potrebbe dire che condividano soltanto il fatto di avere API basate su HTML: il primo si occupa di raccogliere dati per renderli disponibili al mondo esterno, il secondo si concentra maggiormente sulla distribuzione efficiente degli stessi.

Avendo preso atto del fatto che COSM abbia un rate limit ben preciso, oltre il quale ogni richiesta viene rifiutata rendendo inutile ogni ulteriore invio di informazioni, è interessante valutare come poter superare questo limite, così da garantire comunque che tutte le rilevazioni giungano ai relativi destinatari. È qui che entra in gioco PubSubHubBub, fungendo da intermediario tra i server COSM e tutti i possibili subscriber.

Fino a quando si monitora la temperatura del frigorifero di casa propria o la carica della batteria del proprio cellulare (quindi un ambiente ristretto e di scarso interesse per chiunque sia al di fuori del proprio nucleo familiare), PuSH offre come unico vantaggio la possibilità di ricevere gli aggiornamenti nel minor tempo possibile. Nel momento in cui volessimo però realizzare un'applicazione che monitori le variazioni atmosferiche per fornirci previsioni meteorologiche dei prossimi giorni, al contrario, ci apriremmo ad un numero di potenziali utenti interessati molto più elevato, nell'ordine di centinaia di migliaia o addirittura milioni di sottoscrizioni. Si capisce quindi facilmente che, in tale situazione, la distribuzione efficiente dei dati assume un ruolo importantissimo onde evitare di congestionare i server e lasciare l'utenza senza il servizio desiderato.

Per testare i due protocolli descritti in precedenza si è voluto realizzare un'applicazione per dispositivi Android: essa prende i dati da un sensore (fisico o software) e li invia al server Cosm. Tramite PuSH li distribuiamo a tutti gli interessati per renderne poi possibile una visualizzazione su di un altro dispositivo

mobile. La nostra applicazione segue quindi questo schema.

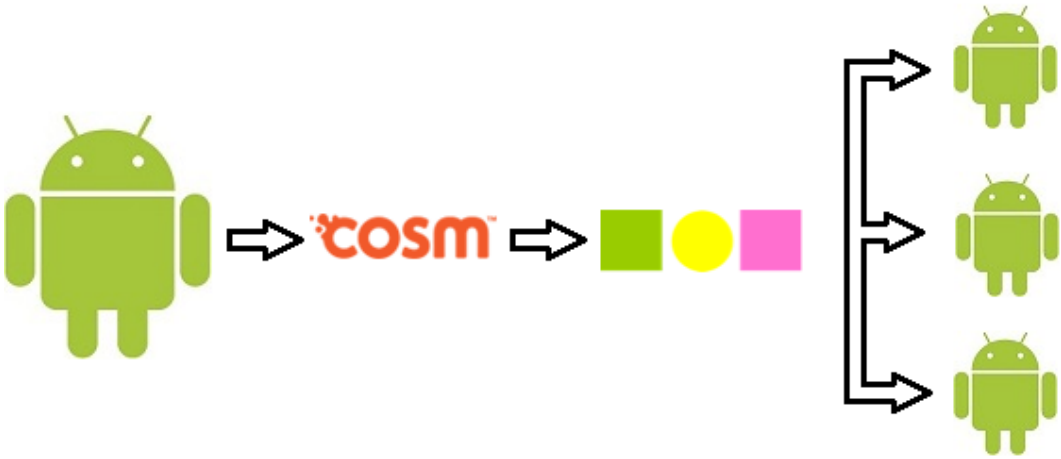

Fig. 20: L'interazione tra le varie parti del progetto

Come vedremo in seguito, il diagramma indica Android come endpoint della nostra elaborazione, ma le scelte possibili sono in realtà multiple: qualsiasi dispositivo/software in grado di manipolare un file XML è infatti in grado di ricevere gli aggiornamenti da noi pubblicati, così come per inviarli ne è sufficiente uno capace di spedire messaggi HTML.

4.1 Android, panoramica e differenze con Java

La scelta di utilizzare Android come ambiente di sviluppo non è stata casuale. Viste le sue somiglianze con Java, la realizzazione di applicazioni per tale sistema operativo è facilitata nel caso si conosca già tale linguaggio. La grande diffusione di dispositivi che lo montano, poi, è stato un ulteriore elemento che ha fatto propendere per questa scelta: nel terzo quarto 2012, infatti, tre smartphone su quattro montano il sistema di Google, raggiungendo un potenziale pubblico nel periodo citato di 136 milioni di utenze (fonte: International Data Corporation, http://bit.ly/TefQSE).

Rispetto al tradizionale modo di programmazione, che prevede una classe principale da cui avviare l'applicazione, Android adotta alcuni accorgimenti che è necessario rispettare durante lo sviluppo e che lo differenziano dal Java tradizionale.

4.1.1 Le attività

Il primo concetto importante è quello delle activity. Una attività è di fatto una singola schermata con la sua interfaccia grafica nella quale l'utente può compiere azioni. In generale, ci aspettiamo che un'applicazione tipo sia composta da più attività che interagiscono tra loro, una delle quali è quella principale che viene caricata e mostrata quando si avvia il programma

Un'activity può infatti avviarne un'altra che fa parte dello stesso programma o di un'applicazione completamente differente: quando si scatta una foto e la si vuole inviare ad un amico tramite mail, ad esempio, non è l'applicazione che scatta l'immagine che deve integrare al suo interno un client di posta (cosa che sarebbe assolutamente ridondante); essa avvisa soltanto il sistema che è stata richiesta questa operazione. Sarà poi Android ad avviare il giusto client di posta o a mostrare all'utente più scelte nel caso ne siano presenti più di uno.

All'avvio di una nuova activity, la precedente viene messa nel *back stack*, ovvero una coda gestita con politica “last in, first out” richiamata alla pressione del tasto Back sul dispositivo. Il programmatore deve ricordarsi di salvare i dati essenziali nel momento in cui un'applicazione non è più in primo piano, così da rendere possibile il loro caricamento nel momento in cui si richiama quella activity: se l'utente sta creando un nuovo contatto e ha già inserito tutti i dati salvo il numero di cellulare, per il quale deve rivedere l'ultimo sms arrivato nel quale è scritto, quando torna a compilare il profilo si aspetta di trovare ancora compilati i dati precedentemente inseriti, senza dover ricominciare da capo ogni volta soltanto perché ha aperto un messaggio.

Le activity devono essere dichiarate nel file manifest (vedere Capitolo 4.1.6 per i dettagli) dell'applicazione perché possano poi essere accessibili.

4.1.2 I servizi

Un service è un componente di un'applicazione che lavora in background, non ha un'interfaccia grafica e viene di solito utilizzato quando è necessario svolgere operazioni particolarmente lunghe che renderebbero inutilizzabile il cellulare. Si

vuole sempre far sì che l'utente abbia un'esperienza d'uso il più rapida possibile, come tale bisogna evitare che un'activity si blocchi per un tempo non definito in attesa che un determinato compito venga svolto.
I servizi servono proprio a questo. Sono tipicamente presenti in due forme:

- Started: il servizio è avviato invocando il metodo *startService()*. In generale non ritorna alcun risultato; compie soltanto un'azione in background (come il download di un file) e poi termina.
- Bound: il servizio è avviato invocando il metodo *bindService()*. Il chiamante si lega ad esso con una relazione di tipo cliente-servitore: invia la richiesta e riceve il risultato. Il servizio rimane attivo fino a quando c'è un componente che lo usa.

Come per le attività, anche i servizi devono essere esplicitati nel manifest.

4.1.3 I content provider

I content provider sono un altro componente fondamentale di un'applicazione. Grazie ad essi è possibile condividere e salvare dati, indipendentemente che si lavori sul file system, in un database o online. Se il content provider lo permette, altre applicazioni possono chiedere il dato e modificarlo.
Sono implementati come una sottoclasse di ContentProvider e devono implementare un certo set standard di API.

4.1.4 I broadcast receiver

Il broadcast receiver è un elemento importante in Android: il suo compito è infatti quello di rispondere ad un determinato tipo di evento che si è verificato. Nella nostra applicazione potremmo avere bisogno di avvisare l'utente quando la batteria inizia a scaricarsi: al generarsi dell'evento corrispondente, il receiver potrebbe ricevere tale segnalazione e mostrare un'opportuna notifica.
In generale, un broadcast receiver fa da tramite verso altri componenti, i quali si occuperanno poi di gestire in maniera più adeguata l'evento e le sue conseguenze.
Di base troviamo già varie notifiche generate da Android, ma se ne abbiamo

necessità possiamo crearne noi stessi per avvisare altre applicazioni che abbiamo eseguito una certa operazione e che quindi possono ora procedere con il resto dell'esecuzione. Per fare ciò ciascun broadcast è inviato sotto forma di oggetto Intent.

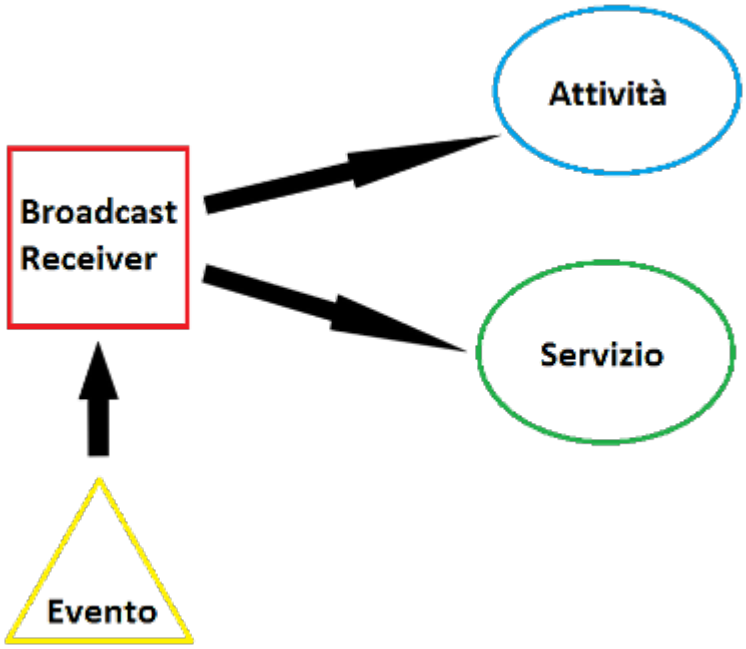

Fig. 21: Il Broadcast Receiver riceve le notifiche degli eventi e avvia i componenti necessari alla loro gestione

4.1.5 Gli intent

L'ultimo elemento che trattiamo per concludere questa breve panoramica sul funzionamento di Android sono gli Intent. Essi sono dei messaggi che vengono passati ad attività, servizi e broadcast receiver per avviarli o segnalare il verificarsi di un determinato eventi.

Come il nome lascia intendere, gli intent rappresentano l'intenzione di fare qualcosa e possono contenere una descrizione astratta dell'operazione da compiere. Quando abbiamo parlato delle activity si è fatto il caso della fotografia da inviare via e-mail: in quel caso, il nostro programma creerebbe un intent apposito da inviare in broadcast (ad esempio) nel quale segnalare l'intenzione di

voler inviare una e-mail. Tutti i client di posta installati nel nostro sistema potrebbero quindi rispondere, dando la possibilità all'utente di scegliere il preferito. In alternativa, il programma potrebbe inviare un intent specifico all'applicazione di Gmail: esso non sarebbe più un avviso che si è verificata una certa condizione, quanto piuttosto l'intenzione da parte nostra di voler avviare un programma preciso.
Più in dettaglio, distinguiamo tra due tipi di intent:

- explicit intent: dichiarano il componente da avviare in maniera esplicita, inserendo quindi un valore preciso nel campo ComponentName. Sono utilizzati tipicamente per inviare messaggi all'interno dell'applicazione, così da avviare gli opportuni servizi o activity;
- implicit intent: non viene definito in maniera esplicita il ComponentName. Li si usa per attivare componenti in altre applicazioni.

Quest'ultimo caso richiede una spiegazione più precisa di come activity, servizi e broadcast receiver possano gestire gli intent. Ciascuno di essi può infatti avere determinati intent filter (istanze della classe IntentFiler), dei veri e propri filtri che permettono ad Android di capire quali operazioni possono svolgere. Questi filtri non sono specificati in codice Java, vengono bensì inseriti nel manifest dell'applicazione come elementi <intent-filter> (con l'unica eccezione dei filtri per i broadcast receiver, che vengono registrati dinamicamente) e hanno tre campi: action, data e category. Per far sì che un intent implicito venga consegnato ad una certa applicazione, esso deve superare tutti i test con i relativi campi.

4.1.6 Il manifest

Il manifest è il punto di partenza per qualsiasi applicazione Android. Nei capitoli precedenti sono stati citati diversi elementi che si possono trovare al suo interno: attività, servizi, filtri ecc. Troviamo in sostanza tutti quegli elementi che devono essere utilizzati dal nostro programma: se un pulsante avvia un'attività di cui abbiamo scritto il codice (quindi effettivamente presente nella nostra applicazione), ma non dichiarata nel manifest, il risultato sarà un crash con relativa chiusura dell'app.

Nel manifest troviamo inoltre il nome del package Java, usato come identificatore univoco dell'applicazione. Accanto ad esso vi saranno poi i permessi richiesti dal programma per funzionare (nel nostro caso servono i permessi per l'accesso online, ad esempio) e le versioni supportate del sistema operativo. Se facciamo uso di librerie introdotte soltanto nelle release più recenti, sta a noi assicurarci che l'applicazione non possa essere installata su dispositivi troppo vecchi, dispositivi sui quali non potrebbe funzionare:

```
<uses-sdk  android:minSdkVersion="5"
      android:targetSdkVersion="15" />
```

indica quali versioni di Android sono compatibili con il programma. Utilizziamo questi valori per definire bene su quali dispositivi vogliamo lavorare, così da evitare che hardware ormai obsoleto non riesca a far girare la nostra applicazione con performance soddisfacenti o abbia problemi per via dell'utilizzo di codice troppo recente.

4.2 L'architettura logica distribuita

Per gestire i dati generati dal nostro smartphone si è scelto di realizzare un'apposita applicazione web che si occupi di elaborarli e presentarli in modo adeguato.

Essendo necessario gestire richieste POST e GET, la scelta è caduta sull'implementazione di due servlet: la prima si occupa di ricevere i dati, caricarli sul server Cosm e avvisare l'hub della presenza di un nuovo elemento da inviare ai subscriber; la seconda svolge invece le veci dell'indirizzo di callback e raduna tutti i valori ricevuti dall'hub per mostrarli in maniera conveniente, oltre a gestire le richieste di sottoscrizione (o sua rimozione) ad un feed.

Le due servlet interagiscono con altrettante Java Server Page (JSP) per fornire in output un documento XML che rispetti i requisiti di PubSubHubBub per la prima e un feed RSS di facile comprensione per la seconda.

Tutta l'applicazione web è fatta girare sui server di AppFog, un servizio Cloud che fornisce l'infrastruttura necessaria a far girare codice scritto nei più comuni linguaggi di programmazione. Ciò permette di effettuare il caricamento dei file

Web Archive (WAR) generati da Eclipse e avere subito la nostra applicazione funzionante in rete.

Uno schema generale dell'architettura utilizzata è il seguente.

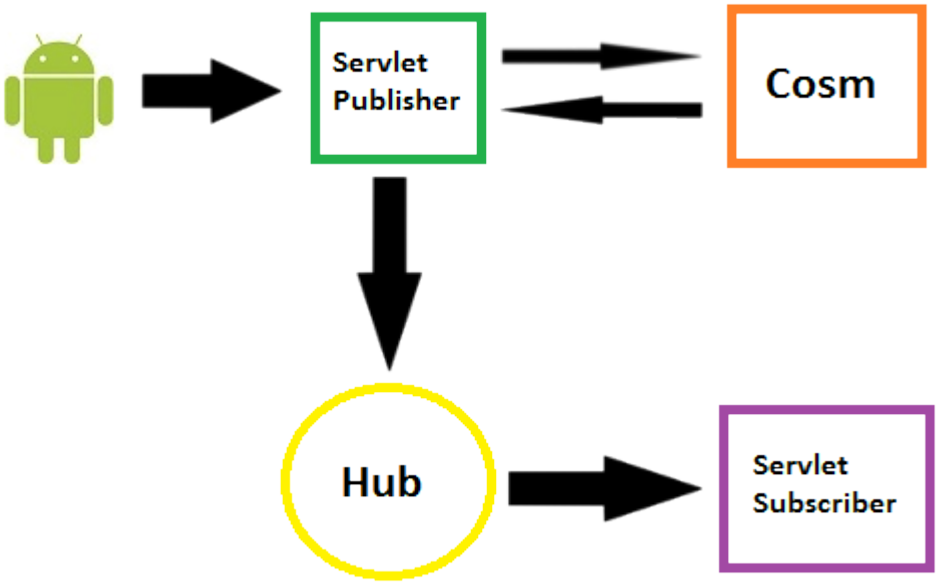

Fig. 22: Le interazioni in gioco

La servlet indicata come Publisher si occupa di ricevere la richiesta POST dal dispositivo Android e la invia al server Cosm, assicurandosi poi che quest'ultimo l'abbia ricevuta correttamente. In seguito, si occupa di pubblicare il nuovo valore. Qui entra in gioco l'Hub che, avvisato dal publisher, dovrà andare a chiedergli il nuovo dato da mandare alla servlet Subscriber, la quale lo elaborerà per mostrarlo sotto forma di feed RSS.

4.3 Interazione con Cosm

La parte di interazione con Cosm è stata sviluppata interamente lato server. Tramite una servlet (come appena visto nel Capitolo 4.2) riceviamo i valori inviati dallo smartphone e li elaboriamo.
Per la realizzazione di questa fase sono state utilizzate le librerie Java JcosmAPI, basate a loro volta su Jpachube. Quest'ultima implementazione, suggerita sul sito

ufficiale Cosm, non è ad oggi funzionante in quanto fa riferimento alla prima versione del protocollo. Di qui la necessità di utilizzarne una versione più recente, adattata alla versione 2 di Cosm.

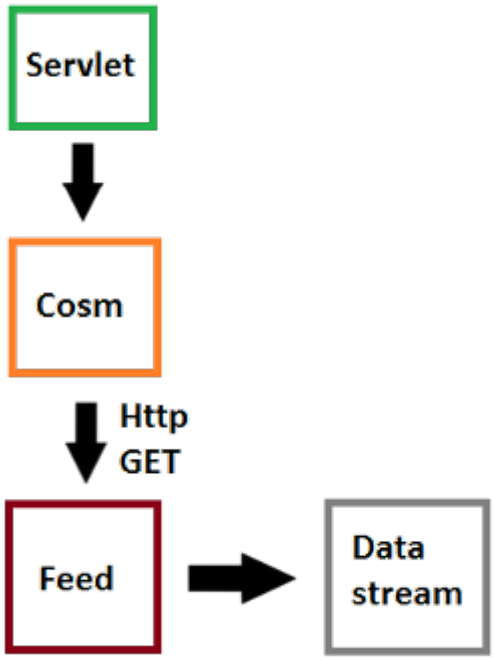

Fig. 23: I passaggi necessari per caricare il nuovo dato sul server Cosm

Il primo passo per far sì che i nostri dati possano essere inviati al server è collegare la nostra applicazione a Cosm. O, per meglio dire, collegarla al nostro account tramite l'apposita chiave API. Le librerie ci forniscono a tal proposito l'oggetto Cosm, il nostro punto d'ingresso per l'interazione con il protocollo. A lui passiamo come stringa la nostra key: il costruttore provvede quindi ad istanziare un nuovo oggetto di tipo CosmClient.

```
private CosmClient client;

public Cosm(String apikey) {
     client = new CosmClient(apikey);
}
```

Creare l'istanza di CosmClient fa sì che si inizializzi un HttpClient con il quale

invieremo poi le varie richieste HTTP e si definiscano i limiti entro i quali la nostra applicazione può agire: essi dipendono da come abbiamo impostato i permessi della API Key.

```
private HttpClient client;
private AuthMethod authMethod;
private String API_KEY;

public CosmClient(String APIKEYS) {
      this.client = new DefaultHttpClient();
      this.authMethod = AuthMethod.apikey;
      this.API_KEY = APIKEYS;
}
```

Giunti a questo punto occorre recuperare il feed e i datastream che caratterizzano il nostro progetto Cosm. Il feed è definito da un identificativo, un numero univoco che ci fornisce direttamente Cosm quando lo creiamo e che ci limitiamo a riportare per averne accesso. È necessario passare attraverso l'istanza di Cosm, tuttavia: su tale oggetto invochiamo il metodo getFeed() a cui passiamo come argomento l'identificativo. L'oggetto restituito sarà il feed a cui siamo interessati. Proseguendo nell'elaborazione si passa poi all'ottenimento dei datastream dal feed. Come spiegato in precedenza, un datastream è un sensore, di conseguenza dobbiamo aspettarci che, in un caso complesso, ve ne sia più di uno (a differenza del nostro, dove monitoriamo un solo valore). È compito del programmatore assicurarsi di leggere o scrivere i dati nel datastream corretto. A tal proposito, per controllare se si è scelto quello giusto si può ricorrere al metodo getId(), il quale ci restituisce una stringa con il suo nome (temperatura, umidità, carica della batteria ecc.).

Rimane quindi l'ultimo step di questa fase, ovvero inviare ai server Cosm il valore ricevuto tramite messaggio POST. Aggiornato il valore corrente del datastream, occorre poi passare dall'oggetto Cosm per eseguire l'operazione. Non è infatti sufficiente limitarsi ad eseguire il metodo setCurrentValue() sul datastream,

occorre anche invocare updateDatastream sull'oggetto Cosm. Il codice che svolge questi compiti è il seguente.

```
inviaValoreACosm(Cosm cosm, int feedId, String valore)
{
      Feed f = null;
      Datastream[] a = null;

      try {
            f = cosm.getFeed(feedId);
            a = f.getDatastreams();
            a[0].setCurrentValue(valore);
            cosm.updateDatastream(feedId, a[0].getId(),
a[0]);
      } catch (CosmException cosmEx)
{ cosmEx.printStackTrace(); }
```

La nostra interazione con Cosm potrebbe fermarsi qui: salvo problemi di linea, il dato è stato caricato nel nostro datastream e il nostro web server può rimanere in attesa di un altro messaggio POST.
Viste le nostre esigenze, tuttavia, serve ancora un'altra serie di passaggi volti a creare le basi per l'elaborazione PuSH. Vogliamo dunque prendere il dato appena elaborato diventi in qualche modo disponibile sul web server per poi essere inviato a tutti i subscriber.
Per realizzare questa operazione ci serviamo nuovamente dell'oggetto Cosm, sul quale cerchiamo i datastream presenti (uno solo nel nostro caso) da cui estrarre l'ultimo datapoint con il relativo valore e timestamp. Avere un'indicazione temporale di quando è stato scritto il datapoint è importante: nel momento in cui lo riceviamo possiamo così analizzare se esso sia lo stesso che abbiamo già recuperato (ad esempio perché il nuovo valore non è stato scritto correttamente sul server per problemi di rete, nonostante sia arrivata una nuova richiesta POST) o se sia effettivamente un dato fresco. Ce ne serviremo per creare il file XML che verrà

opportunamente elaborato quando la nostra servlet riceverà una richiesta GET.
Il codice per questa operazione è il seguente.

```
private ServerFeed ritornaValoreInviatoACosm(Cosm cosm,
int feedId) {

      Feed fRec = null;
      Datastream[] dRec = null;
      try {
      fRec = cosm.getFeed(feedId);
            dRec = fRec.getDatastreams();
      } catch (CosmException cosmExRec) {
            cosmExRec.printStackTrace();
      }

      return new ServerFeed(dRec[0].getCurrentValue(),
dRec[0].getAt());
      }
```

Notiamo il tipo di ritorno, ServerFeed: è una semplice classe realizzata per convenienza nella quale salvare le informazioni legate al datapoint ed è costituita da due costruttori e da quattro proprietà per settare e restituire il valore e il timestamp del datapoint di interesse.

4.4 La distribuzione dei dati tramite PubSubHubBub

Tra gli aspetti coperti da questa tesi, quella di maggior rilievo è legata al protocollo PubSubHubBub. La motivazione è da ricercare nel suo ruolo di "collante" tra le varie parti del progetto.
I dati possono essere rilevati dai dispositivi più disparati, allo stesso modo potremmo poi salvarli su di un database invece che su Cosm. Per il loro trasferimento, tuttavia, PuSH fornisce un protocollo quasi automatizzato che può rivelarsi particolarmente utile: una volta impostati correttamente gli estremi, grazie all'hub pubblico fornito da Google gli aggiornamenti in real time ci vengono offerti praticamente a costo zero.
Quando abbiamo parlato di PuSH abbiamo anche citato i tanti suoi concorrenti,

partiti tutti con l'idea di trasferire nella maniera ottimale i dati, ma che hanno mostrato il fianco ad alcuni problemi nel corso del tempo. Questo protocollo è invece ancora giovane, tanto che trovare esempi o dati legati alle sue performance è spesso difficile, se non impossibile.
Ecco perché ci siamo voluti concentrare su di esso, effettuando opportuni test per capire fino a che punto possiamo spingerci prima di mettere in difficoltà il sistema di notifiche.

4.4.1 L'implementazione di PuSH

L'implementazione del protocollo è stata effettuata seguendo le specifiche del protocollo nella versione 0.3. Il sito ufficiale del progetto fornisce i link a diverse librerie, incluse quelle Java che sono state usate come base per questo progetto. Si è reso necessario apportare alcune modifiche per adattarle al meglio all'applicazione e rispondere al meglio alle specifiche, come illustrato in seguito.

Il primo passo compiuto è stato realizzare un publisher funzionante. Il procedimento è immediato e richiede soltanto di istanziare un nuovo oggetto Publisher su cui invocare il metodo execute(), passando come argomenti l'URL dell'hub e quello del feed di interesse.

```
Publisher publisher = new Publisher();
publisher.execute(
      "http://pubsubhubbub.appspot.com/publish",
      "http://batteryserver.eu01.aws.af.cm/feedcosm/"
      );
```

Questo metodo segnala all'hub che è disponibile un nuovo contenuto e dove può andare a reperirlo tramite una GET. Al ricevimento della richiesta, la servlet prova ad aprire il file XML generato alla ricezione del messaggio di POST inviato dallo smartphone. Se tale file è presente estrae le entry di interesse con i rispettivi valori, link e descrizione, in caso contrario ne crea una di default nella quale si segnala che non è ancora stato ricevuto alcun valore.

Questo passaggio può sembrare contraddittorio: concettualmente, infatti, ci troviamo davanti a una situazione nella quale non abbiamo alcun dato, ma la nostra entry di fatto lo è. Agli occhi dell'utilizzatore finale questo non è un problema: si ritrova infatti l'equivalente di un avviso il quale gli fa notare che non è ancora stato inviato alcun valore. Dal punto di vista implementativo, invece, sappiamo che avere un oggetto inizializzato a null è ben diverso da averne uno inizializzato (ad esempio) con il valore 0: nel primo caso l'oggetto proprio non c'è, nel secondo, al contrario, non solo c'è, ma sappiamo pure con esattezza il suo valore. Il perché di questa scelta è da ricercare nel funzionamento di PuSH. Affinché la sottoscrizione vada a buon fine occorre che il feed contenga informazioni precise (come spiegato nel Capitolo 3.1.4). Senza di esse, la sottoscrizione non può essere completata in quanto l'hub non riconosce il feed di interesse.

Si passa quindi al subscriber, per il quale è stato necessario implementare una piccola modifica rispetto al codice di riferimento. Lavorando già su di un server web non abbiamo bisogno di crearcene uno in locale; la gestione delle richieste di sottoscrizione/disiscrizione viene pertanto girata al Subscriber, come vedremo a breve.
Tali richieste vengono inviate al nostro indirizzo di callback tramite messaggio POST. Nella pagina c'è una servlet con un metodo apposta per questo compito. Dato che l'indirizzo di callback riceve tramite POST anche i valori aggiornati da mostrare all'utente, occorre separare in maniera opportuna i due eventi. Il seguente codice si occupa di portare avanti le richieste di sottoscrizione/disiscrizione.

```
if (req.getParameter("modo") != null) {
            gestisciSottoscrizioni(req, subscriber);
      }
```

Se nel messaggio troviamo il campo “modo”, sappiamo allora che è in corso una richiesta da inoltrare all'hub.
Oltre al tipo di richiesta, dobbiamo anche sapere a quale feed ci vogliamo

sottoscrivere e dove vogliamo che gli aggiornamenti siano inviati. Queste informazioni le estrapoliamo dal messaggio POST e le giriamo all'hub tramite la richiesta di sottoscrizione effettuata dal subscriber. Il codice del nostro metodo è il seguente:

```
private void gestisciSottoscrizioni(HttpServletRequest
req, Subscriber sub) {
     String modalita, feed = null, callback = null;
     String hub = "http://pubsubhubbub.appspot.com/";

     modalita = req.getParameter("modo");
     // Sottoscrizione
     if (modalita.equals("subscribe")) {

     if (req.getParameter("feedinteresse") != null) {
          feed = req.getParameter("feedinteresse");
               }
     if (req.getParameter("urlcallback") != null) {
          callback = req.getParameter("urlcallback");
               }

     try {
        sub.subscribe(hub, feed, callback, null, null);
     } catch (Exception e) {
               e.printStackTrace();
               }
          }
     // ...
     }
```

Operiamo in maniera del tutto simile per la rimozione della sottoscrizione, con l'unica differenza che ci accerteremo che “modalita” sia uguale a “unsubscribe” e invocheremo il metodo *unsubscribe()* del subscriber.

Quando indichiamo la volontà di effettuare una sottoscrizione, l'hub si deve assicurare che tale richiesta sia voluta, onde evitare che qualche malintenzionato sfrutti tale possibilità per forzare l'invio di aggiornamenti non richiesti andando così a caricare server e connessione di lavoro non desiderato. Per fare ciò contatta l'indirizzo di callback tramite una GET e associa al parametro hub.challenge una stringa con l'identificativo della richiesta di sottoscrizione. Il compito del subscriber è di rispondere all'hub con uno status 200 (per segnalare che è avvenuto tutto correttamente) e la stringa associata a hub.challenge, così da confermare che quella sottoscrizione è effettivamente desiderata.
Lavorando con messaggi GET, ci dobbiamo preoccupare che la nostra servlet presente all'indirizzo di callback gestisca in maniera opportuna i diversi casi che si possono presentare.
I casi con cui ci troviamo a lavorare e che dobbiamo opportunamente gestire sono tre:

- richiesta di conferma da parte dell'hub;
- prima lettura dei valori da parte dell'utente;
- lettura dei valori dopo aver ricevuto aggiornamenti dall'hub.

Il primo caso è gestito controllando la presenza o meno del parametro hub.challenge, cosa che indica se dobbiamo fornire una risposta all'hub.

```
if (req.getParameter("hub.challenge") != null) {
                gestisciRisposte(req, resp, subscriber);
            }
```

Il metodo gestisciRisposte() verifica se l'azione richiesta è presente nel subscriber e, in caso positivo, fornisce l'opportuna risposta all'hub. Particolarmente importante è la rimozione della stringa action dalla relativa ArrayList: senza questo passaggio, la nostra richiesta di sottoscrizione rimarrebbe sempre disponibile, anche dopo esserci disiscritti. Ciò lascerebbe la porta aperta a chiunque volesse far nuovamente sottoscrivere il nostro subscriber per fargli ricevere aggiornamenti non voluti: l'hub troverebbe infatti sempre la stringa che

indica il desiderio dell'utente di ricevere dati, provocando quegli effetti dannosi che vogliamo invece prevenire.

```
private void gestisciRisposte(HttpServletRequest req,
     HttpServletResponse resp, Subscriber subscriber) {

String hubMode = req.getParameter("hub.mode");
String hubTopic = req.getParameter("hub.topic");
String hubChallenge =req.getParameter("hub.challenge");

String action = hubMode+ ":" + hubTopic + ":" + "null";

if (subscriber.getApprovedAction(action) &&
     hubMode.equals("subscribe") ||
     subscriber.getApprovedAction(action) &&
     hubMode.equals("unsubscribe")) {

          try {
               subscriber.removeAction(action);
               resp.setStatus(200);
               resp.setContentType("text/plain");
               resp.getWriter().write(hubChallenge);
          } catch (IOException e) {
               e.printStackTrace();
               }
          }
     else if (!subscriber.getApprovedAction(action)) {
               resp.setStatus(404);
          }
     }
```

Per i due casi rimanenti, invece, non ci discostiamo da quanto fatto per la prima servlet, quella deputata alla ricezione dei valori inviati dallo smartphone e al loro

caricamento sul server Cosm.
In mancanza di un file con le entry più recenti creiamo noi un valore da inviare alla JSP, così che essa possa poi elaborarlo per mostrarlo sotto forma di feed RSS, senza che l'applicazione che dovrà andare a leggerlo si preoccupi di gestire il casco specifico in cui la pagina di callback non fornisce un documento opportunamente formattato.
Se la sottoscrizione è andata a buon e l'hub ci invia con successo i vari dati aggiornati, la servlet li carica dal file XML generato dal metodo doPost() e ne effettua il forward alla già citata JSP.

4.5 Implementazione in Android

Entriamo ora più nel dettaglio di come è stata realizzata l'applicazione per la raccolta dei dati.
Lavorando nel mondo mobile, si è cercato di limitare al massimo la computazione richiesta al nostro dispositivo: immaginando di utilizzarla su di uno smartphone, vogliamo evitare un inutile spreco di batteria. Ecco perché tutto ciò che riguarda la computazione dei dati è stato demandato al server, come vedremo in seguito. Ciò ha permesso inoltre di liberarsi da eventuali incompatibilità con le librerie utilizzate sul web server, concentrandosi solamente sull'invio delle informazioni.

4.5.1 Una breve descrizione dell'architettura

Prima di cominciare la descrizione vera e propria dell'applicazione Android è bene mostrare quali sono gli elementi che la caratterizzano e in che relazione sono l'uno con l'altro.
La schermata introduttiva ci mette subito davanti a due scelte, che portano ad altrettanti possibili usi dell'applicazione. L'attività principale avvia quelle desiderate facendo uso di intent espliciti: siamo all'interno del nostro programma e sappiamo esattamente quale attività vogliamo istanziare, pertanto è un'operazione che ci è concessa.

Fig. 24: I due possibili utilizzi

Analizzando la parte di invio dati osserviamo come a sua volta offra due scelte, volte rispettivamente ad avviare e fermare il monitoraggio dei dati. Con il seguente grafico si vogliono mostrare tutte le entità che entrano in gioco al momento della pressione del relativo pulsante. Per una descrizione dettagliata rimandiamo al Capitolo 4.5.3.

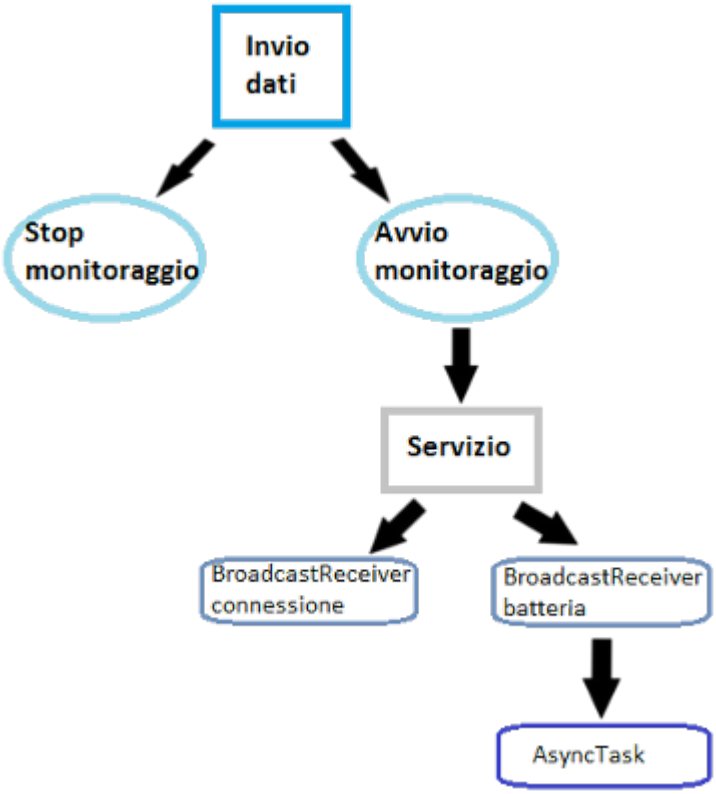

Fig. 25: Gli elementi costituitivi dell'attività di monitoraggio dati

Come mostrato in figura, il monitoraggio non viene svolto per intero dall'attività, ma viene deputato ad un servizio che fa uso di opportuni BroadcastReceiver per

comunicare con l'utente ed effettuare il sensing di dati ed eventi.

L'altra opzione a nostra disposizione, invece, sfrutta un BroadcastReceiver per controllare se la connessione è disponibile e in tal caso scarica i dati più recenti. L'utente ha modo di personalizzare sotto quali connessioni aggiornarli tramite una apposita attività.

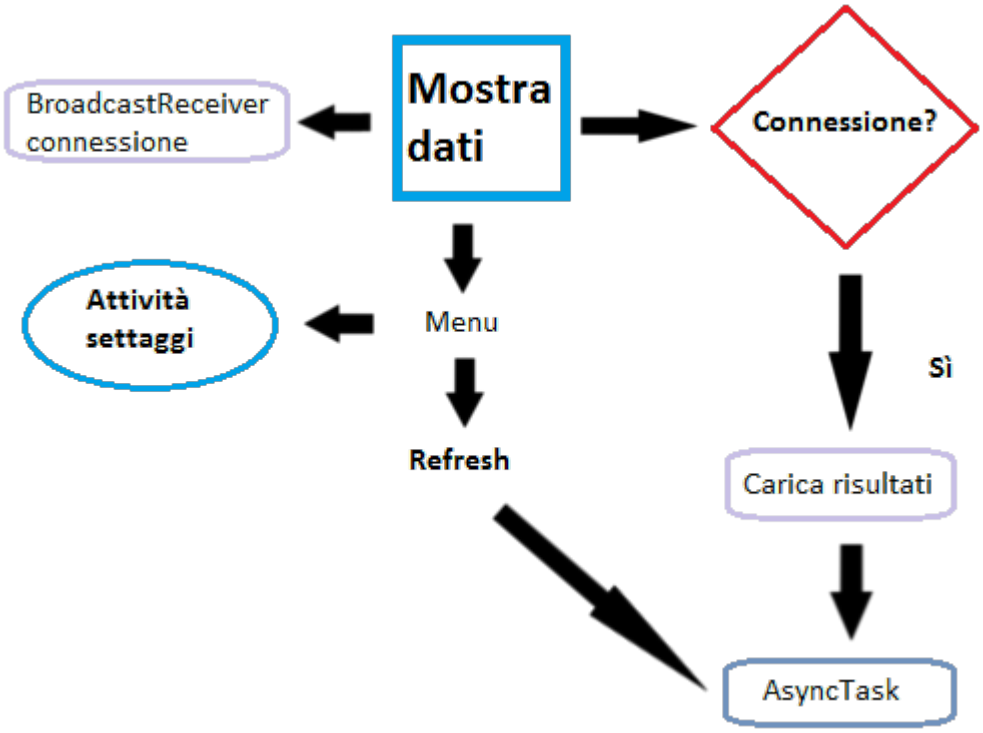

Fig. 26: L'attività interagisce con BroadcastReceiver e impostazioni dell'utente

Per una trattazione completa di questa parte si rimanda al Capitolo 4.5.5.

4.5.2 La schermata introduttiva

Per avviare l'applicazione è sufficiente cliccare sulla rispettiva icona presente nel launcher delle applicazioni. Affinché essa sia visibile è necessaria una ben precisa dichiarazione nel file AndroidManifest.xml:

```
<activity
   android:name=".MainActivity"
   android:label="@string/title_activity_main" >
<intent-filter>
```

```
        <action android:name="android.intent.action.MAIN" />
        <category
        android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
```

I tag activity permettono alla nostra attività di essere visibile. Sono necessari, come da specifica. Incontriamo poi due attributi:

- android:name definisce il nome della classe;
- android:label fa riferimento al file strings.xml e indica quello che sarà il nome dell'applicazione visualizzato dall'utente. Questo file è particolarmente utile nel caso si vogliano realizzare più versioni in lingue diverse: sarà infatti sufficiente fornire il link al giusto file strings.xml per avere il programma completamente tradotto, senza quindi bisogno di riscrivere ogni parametro o campo a mano.

La parte che più ci interessa, però, è quella inserita tra i tag intent-filter: qui incontriamo i tag activity e category, utilizzati per definire in dettaglio il funzionamento di questa attività. Il primo ha come attributo il seguente valore:

android.intent.action.MAIN

Esso vuole indicare che questa classe è quella di partenza, che viene avviata senza input particolari e non presenta alcun valore di ritorno.

Nel secondo troviamo invece

android.intent.category.LAUNCHER

È proprio grazie a questo parametro che la nostra attività compare nel launcher delle applicazioni: lo troviamo soltanto qui in quanto è questo il nostro punto di ingresso nel programma, di conseguenza non avrebbe senso assegnare tale attributo ad altre classi.

Dato che lavoriamo con la connessione, sarà fondamentale aggiungere anche i permessi d'uso che consentono all'applicazione di accedere alla rete. Per fare questo bisogna aggiungere due tag, uno che garantisca l'accesso ad Internet, l'altro che monitori l'attuale stato della connessione (attiva o meno). Rispettivamente, i tag sono i seguenti:

```
<uses-permission
android:name="android.permission.INTERNET" />
<uses-permission
   android:name="android.permission.ACCESS_NETWORK_STATE" />
```

La schermata che ci troveremo davanti all'avvio è la seguente.

Fig. 27: La schermata di avvio

La parte di nostro interessa è il pulsante Avvia monitoraggio. Ad esso è collegato il metodo `startMonitoring(View view)` alla cui pressione corrisponde l'avvio di una nuova activity in cui l'utente può decidere di iniziare o fermare la rilevazione della carica della batteria.

Per avviare la nuova attività occorre sfruttare gli intent: si crea quindi una nuova istanza passando come parametri chi sta creando l'intent (ovvero la classe attuale) e quale activity si vuole lanciare. Il codice è il seguente:

```
Intent intent = new Intent(this, BatteryMonitor.class);
startActivity(intent);
```

L'activity corrente verrà quindi messa nel back stack, pronta ad essere richiamata nel caso in cui l'utente prema il tasto back sul suo smartphone.

4.5.3 Avviare e fermare il monitoraggio, descrizione generale

Superata la schermata introduttiva, arriviamo dunque al vero cuore dell'applicazione. Questa activity, avviata dall'apposito intent, permette all'utente di avviare e fermare il monitoraggio della carica della batteria. Si presenta così:

Fig. 28: La seconda activity

L'aspetto scarno è stato scelto apposta per mettere in maggiore risalto ciò che ci interessa veramente: il valore monitorato e il buon esito del suo invio al nostro web server. Cliccando sul pulsante Inizio, in caso di corretto funzionamento

otterremo un risultato simile a questo:

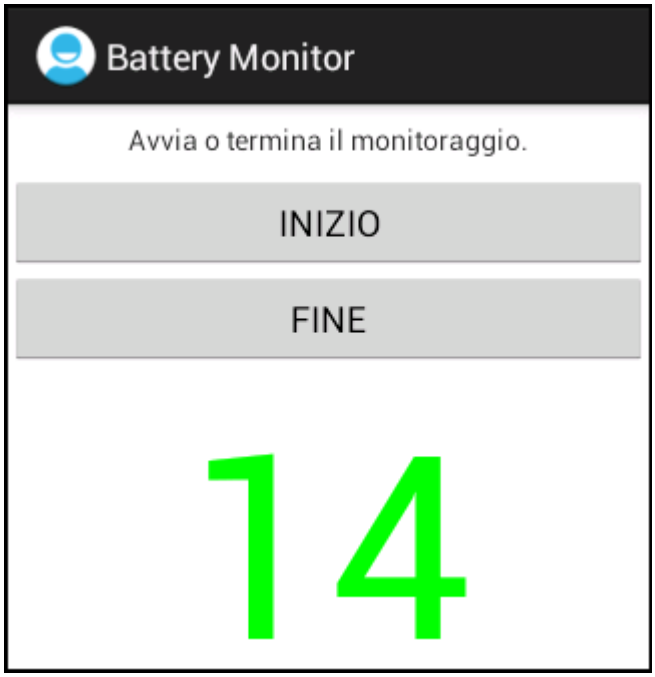

Fig. 29: Il livello di carica della batteria

Per fornire un opportuno feedback all'utente, così da tenerlo sempre informato sul buon esito o meno dell'operazione, l'applicazione fa comparire alcuni messaggi a scomparsa in sequenza. Il primo segnala la presenza della connessione dati.

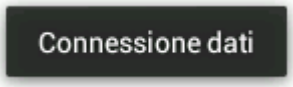

Fig. 30: Dati attivi

Il secondo invece la sua mancanza.

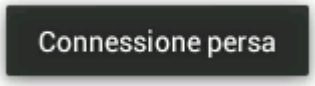

Fig. 31: Dati mancanti

Il passo successivo riguarda l'invio dei dati al server, che a sua volta può avere esito positivo

Fig. 32: Esito positivo

o meno.

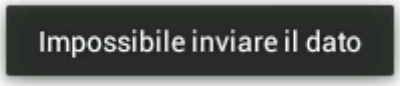

Fig. 33: Esito negativo

Terminata la descrizione rapida del comportamento di questa activity, è ora necessario spiegare più in dettaglio come ciascuna operazione avvenga. A fronte di un comportamento tutto sommato semplice, sotto la superficie troviamo alcuni accorgimenti precisi che hanno portato a questo risultato.

4.5.4 Avviare e fermare il monitoraggio, in dettaglio

All'avvio dell'activity il metodo onCreate() si preoccupa di inizializzare l'intent per il servizio adibito allo svolgimento dell'operazione di nostro interesse. Essendo quest'ultima un qualcosa con cui l'utente non deve interagire, non c'è la necessità di avere un'interfaccia grafica. Ecco pertanto giustificata la scelta di utilizzare un service, che svolge il suo compito in background. La sua inizializzazione comporta la registrazione di due BroadcastReceiver ai relativi IntentFilter: così facendo, infatti, il nostro service può monitorare sia lo stato della connessione che i cambiamenti nel livello di carica della batteria.

Il via alle operazioni arriva quando l'utente effettua un click sul pulsante INIZIO. Per prima cosa chiamiamo il metodo registerReceiver(), così da legare il nostro BroadcastReceiver all'apposito IntentFilter: quando il service invierà in broadcast il suo intent, il BroadcastReceiver riuscirà ad intercettarlo e si preoccuperà di aggiornare l'interfaccia grafica con il valore di carica attuale della batteria.
In seguito, tramite il metodo startService, diamo il via vero e proprio al monitoraggio.
È interessante notare come, avendo registrato l'apposito receiver nel service, l'attività non riceva aggiornamenti ad intervalli prestabiliti (cosa che porterebbe ad un polling ripetuto e potenzialmente dannoso per la durata della batteria), ma soltanto quando il livello di carica varia.

L'utilizzatore ha poi la facoltà di bloccare il monitoraggio tramite il tasto FINE. Esso rimuove la registrazione del BroadcastReceiver e indica al service che deve fermarsi.

Sia l'inizio che la fine del monitoraggio sono due eventi a rischio malfunzionamento, se non gestiti correttamente. Per questo motivo, prima di effettuare le rispettive azioni si controlla se il servizio è già stato avviato o meno. Senza questo controllo, se l'utente preme il tasto fine senza che il service sia stato effettivamente avviato, si ha infatti un crash dell'applicazione, la quale tenta di rimuovere la registrazione da un broadcast che ancora non ha ricevuto registrazioni.

Analizzando meglio il receiver, dobbiamo ora concentrarci sulla fase di invio dei dati al server Cosm. La nostra applicazione non ha alcuna conoscenza della destinazione delle informazioni generate: siamo totalmente indipendenti dalle API Cosm e lavoriamo come se dovessimo inviare una semplice richiesta POST ad un web server, cosa che di fatto si verifica.
È importante ragionare sul tipo di operazione che dobbiamo svolgere, analizzando bene quali problemi potrebbe generare. Trattandosi di una interazione con la rete, non sappiamo a priori quanto tempo ci vorrà per portarla a termine e se avrà un esito positivo: potremmo avere a che fare con una rete wi-fi ormai satura, oppure trovarci in una zona con scarsa copertura da parte del nostro operatore. Tutto questo rischia di tramutarsi in una lunga attesa per l'utente, che avrebbe così tra le mani un'applicazione che non risponde ai suoi comandi. Nonostante nel nostro caso specifico non si abbia a che fare con operazioni particolarmente ravvicinate ed intense, si è scelto comunque di adottare una soluzione che ci metta al sicuro da questi problemi: in caso di futuri aggiornamenti, non ci ritroveremo quindi con la necessità di riscrivere questa parte di codice per adattarla alle nuove esigenze.
Nella documentazione di Android, Google suggerisce una linea guida per raggiungere questo risultato e fornisce la classe AsyncTask per questo scopo. Essa genera un nuovo task, separato dall'interfaccia grafica. L'implementazione delle operazioni da svolgere va scritta nel metodo doInBackground(). È qui che

abbiamo indicato la pagina web verso cui inviare la richiesta Post costruita in questo metodo. Per conoscere l'esito dell'invio e rendere partecipe anche l'utente, si è scelto di utilizzare i messaggi Toast visualizzati nel capitolo precedente. Un messaggio Toast è un piccolo avviso che compare sotto forma di pop-up e sparisce automaticamente dopo poco: non prevede alcun tipo di interazione con l'utente e ha il solo scopo di informarlo riguardo l'esito dell'operazione.
Per decidere quale messaggio visualizzare ci serviamo del valore di ritorno di doInBackground(). Esso viene passato al metodo onPostExecute(), dandoci poi modo di analizzarlo e controllare che tutto sia andato a buon fine.

4.5.5 Mostrare i risultati

La schermata di avvio ci permette di svolgere due operazioni: la prima di monitoraggio, appena spiegata, la seconda di recupero delle informazioni per mostrarle all'utente.
Questa seconda opzione potrebbe essere anche separata in un'applicazione a parte: se qui forniamo in output i nuovi valori di un feed selezionato (ovvero quello relativo alla carica della nostra batteria), nel caso la si volesse ampliare per renderla più completa si potrebbe dare all'utente la possibilità di sottoscriversi a più feed o di inserirne uno di suo interesse.

La realizzazione è piuttosto semplice e sfrutta il codice di un'applicazione fornita da Google come esempio per il parsing di file XML. La pressione del tasto Risultati avvia una nuova attività che mostra a video una WebView nella quale verranno caricate le entry ricevute dal server di callback. L'utente ha la possibilità di scegliere se scaricare gli aggiornamenti solo quando connesso tramite Wi-Fi o anche tramite la connessione dati. Può inoltre aggiornare la pagina a piacere per caricare i valori più recenti ricevuti.

Fig. 34: Risultati delle misurazioni

Tramite la pressione del tasto Menu facciamo apparire le due scelte citate in precedenza: Refresh e Settings.

Fig. 35: Aggiornamento e impostazioni

La seconda ci offre la possibilità di scegliere quando scaricare gli aggiornamenti e se mostrare o meno la descrizione con le informazioni estese.

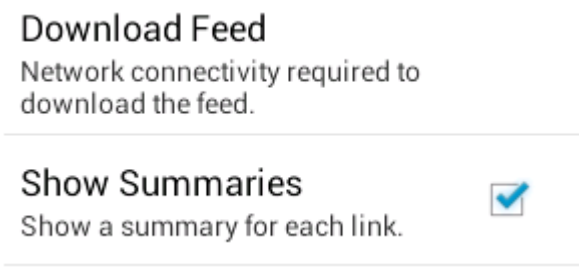

Fig. 36: Il menu Settings

L'opzione Download Feed dà infine l'occasione di scegliere quando scaricare i dati: questa opzione è particolarmente utile per evitare di scaricare dati utilizzando la connessione cellulare, rischiando così di pagare costi non voluti.

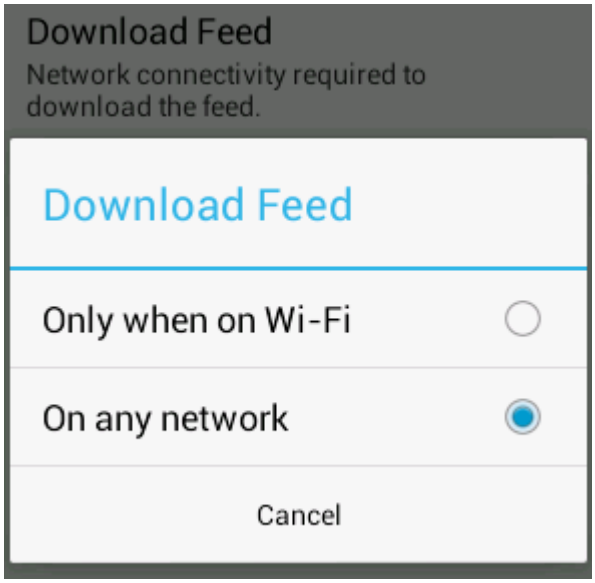

Fig. 37: La scelta della connessione

4.6 Metodologia dei test

Per testare l'applicazione realizzata sono stati adottati diversi metodi che hanno portato a risultati interessanti. Si è scelto di effettuare test manuali per valutare che il tutto funzionasse correttamente, mentre la scelta è ricaduta su test maggiormente automatizzati quando ci siamo concentrati sul protocollo PuSH.

4.6.1 I test dell'applicazione

I test dell'applicazione sono stati eseguiti sfruttando Telnet, un apposito protocollo che consente di lavorare in remoto. Nel nostro caso è stato utile per collegarci alla macchina virtuale e impostare di volta in volta il valore di carica della batteria, senza dover aspettare che il dispositivo reale variasse il suo livello di carica per inviarci un nuovo valore.
Il primo comando impartito è stato `telnet localhost port`. Il buon esito della connessione ci è confermato dalla seguente schermata.

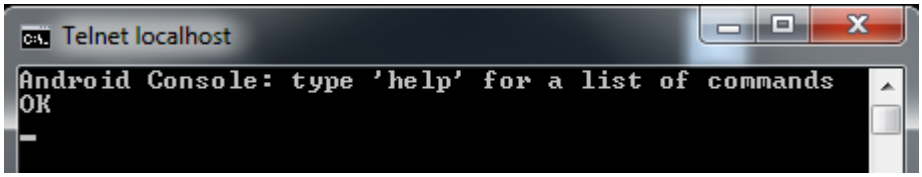

Fig. 38: Connessione all'host effettuata

Da qui è poi sufficiente digitare `power capacity val` (con val che indica la percentuale di carica, da 0 a 100) per variare a piacimento la carica della batteria. La nostra applicazione rileva così il cambiamento e inizia il processo di upload dei dati sul server Cosm, mentre contemporaneamente aggiorna la schermata visibile dall'utente e lo avvisa del corretto esito o meno dell'operazione.
Se l'indirizzo di callback è stato abilitato a ricevere gli aggiornamenti, i nostri valori appariranno in tale pagina, pronti per essere poi elaborati e mostrati all'utente.

Sin da questi test si è palesato un problema che risulterà poi molto più evidente nelle prove automatizzate. Digitando la stringa citata poco sopra molto rapidamente (facendo quindi cambiare i valori di carica in tempi rapidi), non sempre essi venivano mostrati correttamente nella pagina di callback.

4.6.2 I test del protocollo

Al fine di effettuare test del protocollo PubSubHubBub il più veritieri possibile, si è reso necessario apportare alcune modifiche alla nostra applicazione web. Ci si è

quindi preoccupati di realizzarne una versione priva di qualsiasi interazione con Cosm, così da eliminare eventuali ritardi dovuti al maggior carico computazionale richiesto al server e (soprattutto) alla rete.
Per inviare tutte le richieste POST al posto dell'applicazione Android è stato scritto un piccolo programma che genera un numero a piacere di thread, ciascuno dei quali invia un determinato numero di messaggi.
Il semplice codice che caratterizza la classe contente il main ha come punto centrale questo:

```
int val = 0;
PostSenderThread pst;

     for(int i = 0; i < 10; i++) {
          pst = new PostSenderThread(i, val);
          pst.start();
     }
```

L'intero `val` è un valore specificato da noi al momento dell'avvio del programma. Abbiamo infatti reso possibile tre scelte: decidere noi il valore di attesa tra l'invio di un messaggio e l'altro, avere un'attesa random (da 0 a 5 secondi) oppure affidarci ad una distribuzione di Poisson. Quest'ultima alternativa è stata in realtà leggermente personalizzata: qualsiasi valore utilizzato come media avrebbe portato ad attese piccolissime, trasformando di fatto il test in una serie di invii back to back (come se le richieste fossero inviate una dopo l'altra senza sosta).
Di conseguenza il codice è stato così modificato:

```
import org.apache.commons.math3.
                    distribution.PoissonDistribution;

private int val = 0;
private PoissonDistribution pd;
```

```
val = (int) (125 - (pd.probability(i) * 1000));
```

Il risultato è avere attese che spaziano da 124 millesimi di secondo a 0, simulando situazioni con carico differenziato.

Citata in precedenza, la classe PostSenderThread estende Thread e si occupa di effettuare l'invio dei messaggi POST, ciascuno dei quali contiene l'identificativo del thread e il valore a cui siamo arrivati all'interno del ciclo.

```
public void run() {
     DefaultHttpClient httpClient = null;
     String address =
"http://batteryserver.eu01.aws.af.cm/stresstestfeed/";

     int i;
     for (i = 0; i < 20; i++) {

          try {

     List<NameValuePair> nvps = new
          ArrayList<NameValuePair>();

     httpClient = new DefaultHttpClient();
     HttpPost httppost = new HttpPost(address);
     nvps.add(new BasicNameValuePair("testValue",
          ""+i));
     nvps.add(new BasicNameValuePair("thread", ""+id));

     // Effettuo la codifica dell'ArrayList
     UrlEncodedFormEntity urlEncodedFormEntity = new
               UrlEncodedFormEntity(nvps);

     httppost.setEntity(urlEncodedFormEntity);
```

```
        HttpResponse httpResponse =
             httpClient.execute(httppost);
        res = httpResponse.getStatusLine().toString();

        if (res.equals("HTTP/1.1 200 OK")) {
                        ok++;
             }

        if (poisson) {
        val = (int) (125 - (pd.probability(i) * 1000));
             Thread.sleep(val);
        } else if (random) {
             Thread.sleep(rd.nextInt(5000));
        } else {
             Thread.sleep(val);
        }

             } catch (UnsupportedEncodingException e) {
                        e.printStackTrace();
             } catch (ClientProtocolException cpe) {
                        cpe.printStackTrace();
             } catch (IOException ioe) {
                        ioe.printStackTrace();
             } catch (InterruptedException e) {
                        e.printStackTrace();
             }

             } // for
}
```

Il metodo run() prepara la richiesta POST aggiungendo ad una lista le coppie nome valore contenenti le informazioni di nostro interesse che verranno

opportunamente codificate. La richiesta sarà poi eseguita dall'oggetto httpClient.
Un elemento a cui prestare particolare attenzione è la stringa `res`: quello che a prima vista sembra solo un controllo sull'effettivo invio del dato, si rivela in realtà un aspetto vincolante di questi test. Prima di procedere con l'invio successivo, infatti, dobbiamo aspettare che arrivi dall'hub la conferma che il messaggio sia stato recapitato a dovere. Di conseguenza, i valori da noi impostati (a mano, random o Poisson) generano un'attesa che parte solo dopo la ricezione del codice di stato, causando attese che vanno man mano aumentando maggiore è il numero di invii al secondo.

I test sono stati effettuati considerando 10 thread che inviano 20 richieste l'uno, partendo da un attesa di 10 secondi tra un invio e l'altro sino ad arrivare a 250 millisecondi. Per ciascun valore il test è stato ripetuto 5 volte.
Il pc scelto per le prove era dotato di una banda in upload di oltre 10 Megabit, così da non rischiare di saturarla come potrebbe accadere con una comune ADSL da 384 kb/s.

4.6.3 I risultati dei test

Il primo test effettuato prevede un'attesa di 10 secondi tra un invio e l'altro. In una situazione ideale ci aspettiamo che tutti i thread partano e terminino nello stesso istante. I valori misurati mostrano però un comportamento differente: essi sono stati ottenuti sommando le attese riscontrate per ciascun thread dividendole poi per 5 (ovvero il numero di test effettuati). Come vedremo a breve nel grafico riassuntivo, il tempo di interarrival tra un thread e l'altro è variabile (ovvero il tempo che passa da quando il thread precedente ha concluso gli invii a quando anche il successivo ha fatto lo stesso). Inoltre il tempo che ciascun thread impiega per terminare le sue pubblicazioni va via via aumentando, portando ad una differenza di quasi 30 secondi tra il primo e l'ultimo.
Nel seguente grafico abbiamo quindi deciso di mostrare i valori medi di interarrival tra ciascun thread e il valore medio di valori ricevuti dal callback per ciascun thread. Tramite le barre di errore evidenziamo invece la varianza, che

spazia dallo 0,468% al 1,382%.

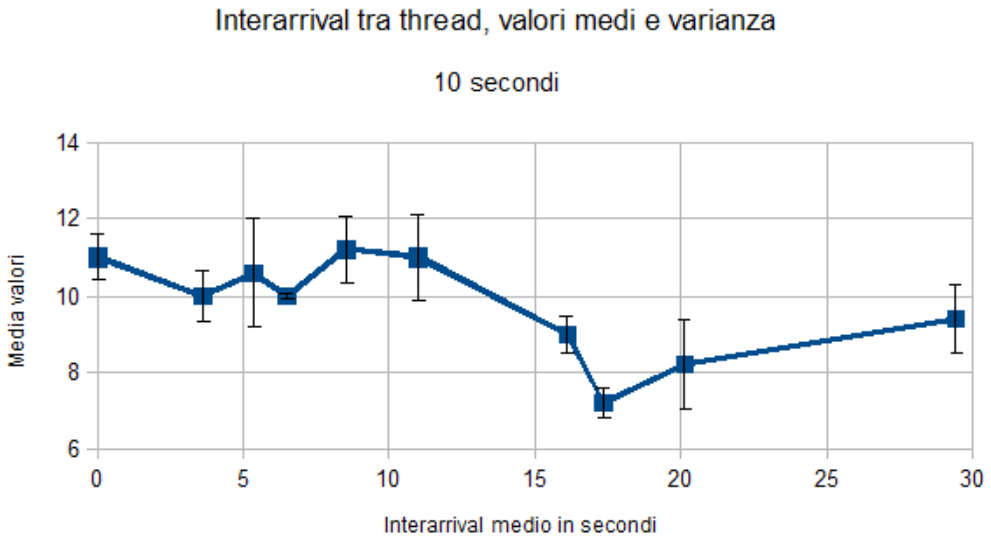

Fig. 39: I risultati nel caso di attesa 10 secondi

Di 200 dati inviati (tutti correttamente ricevuti dal server), il callback ne ha mostrati un numero che spazia da 90 a 102. La percentuale media di valori ricevuti si attesta pertanto al 48,8%.

Per la prima prova si è scelto dunque di utilizzare un'attesa elevata per vedere come si comportava l'hub in presenza di un carico non eccessivo. La seconda è stata invece effettuata con timing molto più stretti, pari a 250 millesimi di secondo, così da raggiungere idealmente a 80 richieste al secondo.
Laddove ci aspetteremmo un'attesa tra gli invii vicina a quella scelta per il test, i risultati mostrano nuovamente valori differenti.
Andando a verificare il numero di dati correttamente ricevuti dal callback riscontriamo questi risultati: l'hub ha ricevuto tutte e 200 le pubblicazioni, ma è riuscito a distribuirne un valore che spazia da 102 a 125, con una percentuale media del 56,1%.
La varianza è leggermente più alta del caso precedente e varia da un minimo di 0,246% ad un massimo di 2,411%.

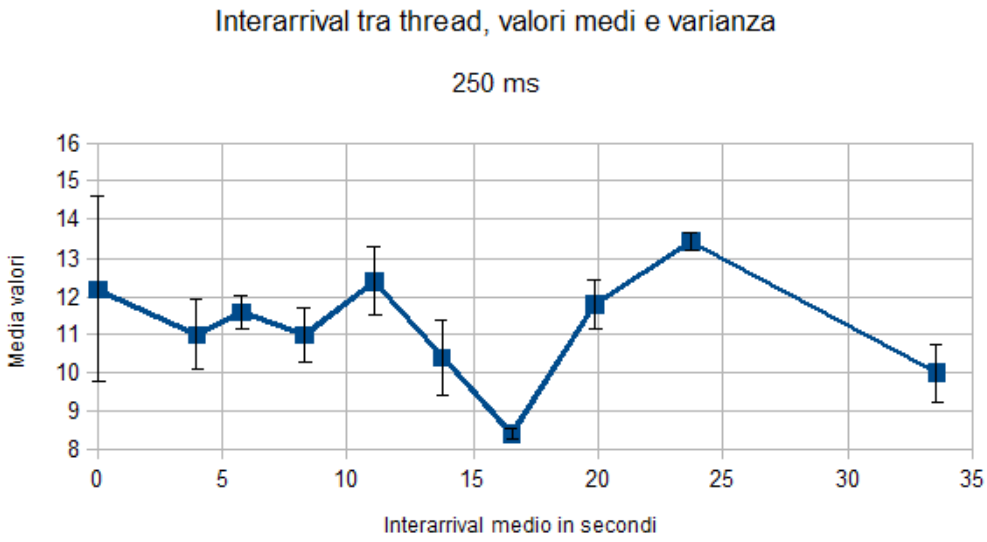

Fig. 40: I risultati nel caso con attesa di 250 ms

Rimangono i test da effettuare con Poisson. La distribuzione utilizzata ha questo andamento.

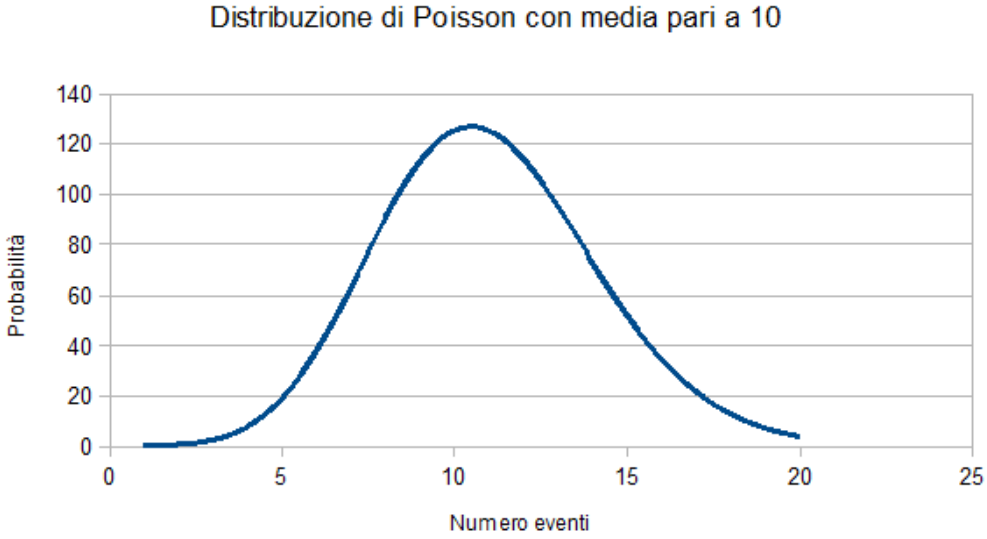

Fig. 41: Andamento della distribuzione

Come il grafico mostra, agli estremi le attese tra un invio e il successivo sono più rilassate, andando invece a ridursi fortemente (e di conseguenza a creare un maggiore stress) verso il centro.

Il valore medio di pubblicazioni ricevute con successo varia da 11,4 a 15, il che

significa che i valori ricevuti dal callback sono un minimo di 106 e un massimo di 129, con una percentuale del 60,1%.

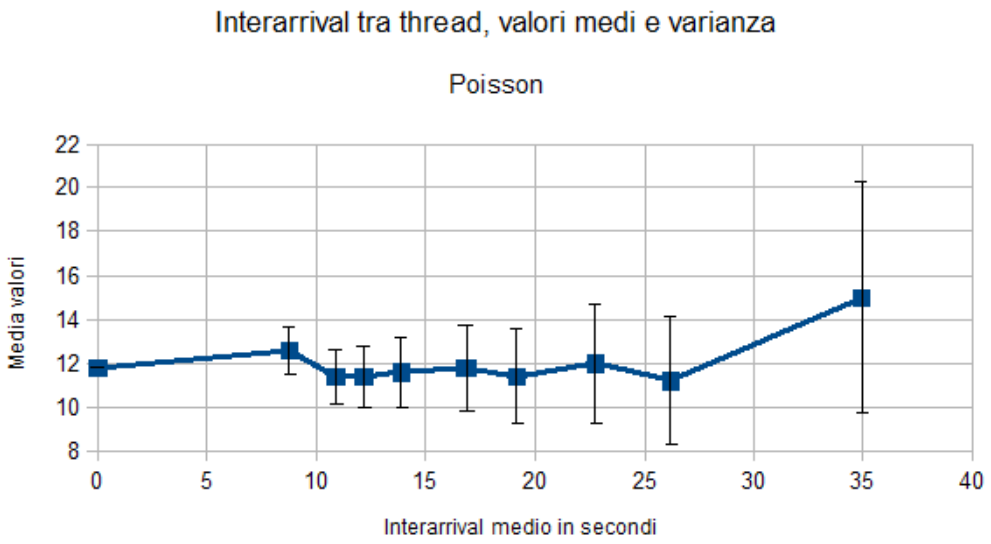

Fig. 42: I risultati dei test effettuati con Poisson

4.6.4 Un diverso approccio ai test

Visti i risultati dei precedenti esperimenti si è reso necessario cambiare la metodologia dei test per vedere se si riuscivano a raggiungere risultati più soddisfacenti.

La nostra applicazione è stata quindi modificata per utilizzare un solo thread, il quale effettua richieste POST ad una certa distanza temporale l'una dall'altra. Oltre a questo, un'altra modifica sostanziale è stata creare un secondo server che pubblicasse nuovi valori. L'hub riceve pertanto le pubblicazioni di due diversi publisher (posti su server differenti), riunendo i risultati su di un terzo server indicato come callback.

Dalle prove precedenti è interessante notare come il tempo di attesa tra un invio e l'altro cresca man mano che si susseguono le pubblicazioni. Osservando i risultati vediamo come esso sia di circa 3,5/3,7 secondi più lungo dell'attesa desiderata per il primo invio, arrivando fino a 5/5,5 secondi per l'ultimo thread. I nuovi test sono quindi stati realizzati tenendo conto di questi valori: si è infatti deciso di partire con un'attesa di 5 secondi tra gli invii, così da lasciare margine alle parti in gioco

di interagire, per poi abbassare tale valore per i test successivi.

I risultati, visibili nel seguente grafico, mostrano un netto miglioramento rispetto alle prove precedenti: ci avviciniamo alla totalità dei messaggi consegnati, raggiungendo la percentuale del 97,5%.

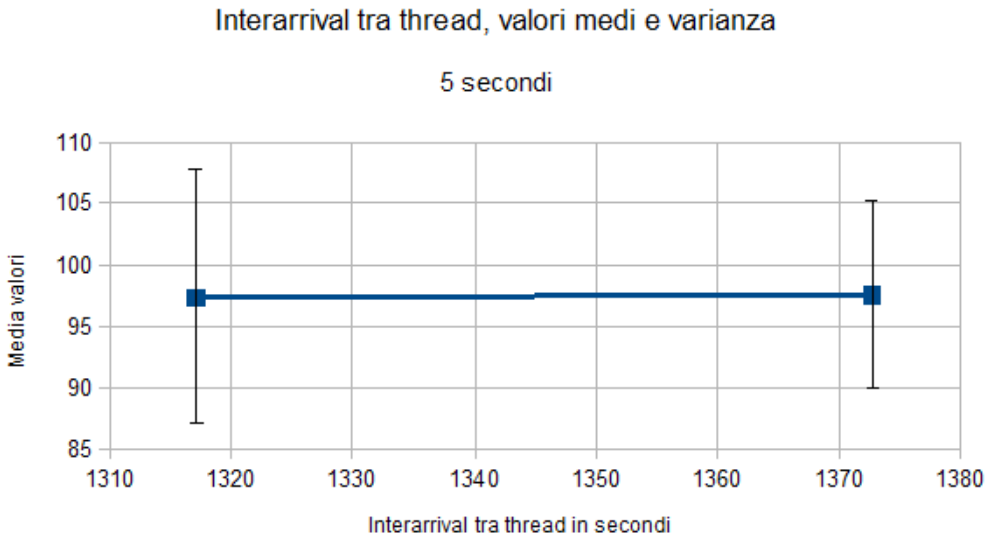

Fig. 43: I risultati con un solo thread e l'attesa di 5 secondi

Visti i dati incoraggianti, sono stati ripetuti anche i test con attesa di 250 millisecondi per invio.

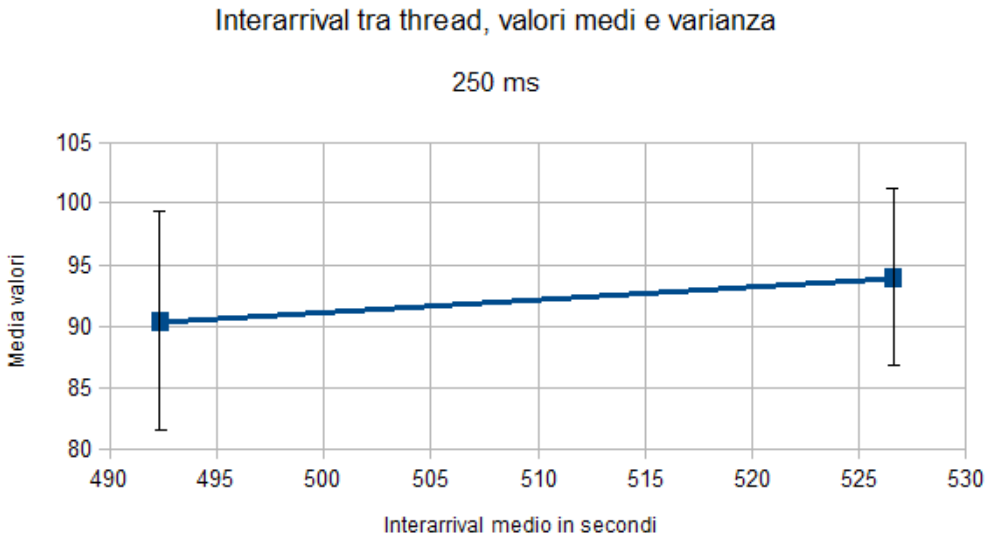

Fig. 44: I risultati con un solo thread e un'attesa di 250 ms

La percentuale media di pubblicazioni ricevute è ora del 92,2%.

4.7 Conclusioni tecniche

Le informazioni che abbiamo ricavato da questi test si sono rivelate particolarmente utili per mettere in mostra pregi e debolezze di PubSubHubBub nella nostra implementazione.
Il protocollo ha dimostrato di fare fatica a smistare grandi quantità di richieste ricevute in un arco ristretto di tempo dalla stessa sorgente. I primi test hanno portato a valori ben lontani da quelli che ci si poteva aspettare in un primo momento. Percentuali vicine al 50% e al 60% non sono affatto soddisfacenti per pensare di poter distribuire efficientemente dati prodotti da un qualsiasi sensore. L'incremento costante dei tempi di attesa tra un invio e l'altro indica come l'hub impieghi via via più tempo a dare la conferma di aver ricevuto la segnalazione di pubblicazione da parte del publisher, allontanandoci sempre più da quelle notifiche in tempo (quasi) reale che tanto ci hanno interessato al momento della scelta di questo protocollo.
Questi dati vanno tuttavia analizzati assieme al numero di pubblicazioni ricevute dall'hub: tale numero è infatti sempre pari al 100%, segno che il problema risiede nella rapidità con cui esso va a recuperare il nuovo dato dal publisher. Nel caso in cui quest'ultimo generi informazioni troppo velocemente, l'hub riceve l'avviso del nuovo valore, ma quando lo va a leggere ne trova già uno aggiornato. Una seconda lettura dello stesso valore non porta ulteriori invii verso il subscriber per via del funzionamento dell'hub stesso, che riconosce il dato come non modificato ed evita quindi di inviarlo una seconda volta (così da difendersi da possibili attacchi di tipo DDoS).

La seconda tipologia di test si è invece focalizzata su di uno scenario più vicino a quello per cui PuSH è stato pensato e i risultati ne hanno giovato. Le varie implementazioni del protocollo disponibili online sono spesso integrate in servizi di blog, i quali tipicamente non producono grandi quantità di dati in tempi ristretti, anzi. La scelta di due server differenti ha simulato il caso concreto in cui un subscriber mostri interesse per i contenuti prodotti da due fonti diverse che vogliono essere aggregate in un'unica destinazione.

Il primo test ha portato ai risultati sperati: ci si è avvicinati al 100% di messaggi consegnati, cosa che dà un'indicazione chiara su come muoversi nel caso di architettura basata su publisher, hub e subscriber su server differenti. Per una corretta distribuzione dei messaggi sembra quindi essere necessario far passare un certo lasso di tempo tra un invio e l'altro. Nel nostro caso 5 secondi si sono rivelati essere già un buon valore, ma potrebbe essere necessario variarlo a seconda delle risorse disponibili.

Ciò su cui dobbiamo ragionare, tuttavia, è il rate limit di Cosm (vedi Capitolo 2.1.10). 100 richieste al minuto equivalgono ad un invio ogni 600 millisecondi. Visti i tempi con cui l'hub completa la sua interazione con il publisher riusciamo ampiamente a non saturare di richieste il server Cosm, ma dobbiamo adattare di conseguenza il tempo tra una rilevazione e l'altra. Un ritardo di 250 millisecondi si tramuta comunque in un'attesa media pari a 5 secondi (in caso di connessione dotata di sufficiente banda), per cui è inutile effettuare sensing più frequenti. Considerato che già con un tale ritardo si ha una percentuale di rilevazioni ricevute dal subscriber pari al 92/93%, diminuire ulteriormente i tempi non conviene onde evitare la perdita di altri dati.

Conclusioni

Nel Capitolo 1.4 ci siamo posti come obiettivi quelli di imparare a conoscere i protocolli Cosm e PubSubHubBub per valutare se potessero interagire tra di loro. Del primo abbiamo voluto studiare quali possibilità offra per replicare al meglio un ambiente reale, con i suoi sensori e le sue misurazioni. Era importante capire come interagire con tale protocollo per valutare che opzioni vengono fornite per l'invio e la ricezione dei dati. Del secondo ci interessava invece verificare la capacità di trasferire dati in tempo (quasi) reale e che numero di invii si può raggiungere prima che l'hub non riesca a distribuire correttamente tutte le pubblicazioni.

Il loro studio ha portato a scoprire informazioni interessanti sul loro funzionamento.

Cosm è infatti un ottimo strumento per racchiudere in uno stesso ambiente più informazioni differenti potenzialmente legate tra di loro. Gli utilizzi delle stesse possono essere i più variegati: con la nostra applicazione abbiamo monitorato la variazione della carica della batteria del nostro cellulare, ma su di un dispositivo con più sensori potremmo decidere di tener traccia anche della temperatura della batteria, piuttosto che della nostra posizione sfruttando i dati della rete o del sensore GPS.

Le API Cosm sono spiegate in dettaglio sul sito ufficiale del progetto e possono essere usate facilmente da chiunque abbia un minimo di conoscenza di programmazione di rete. Sfruttando dei semplici messaggi HTTP (opportunamente formattati) è infatti possibile interagire con il server per caricare i nostri dati, visualizzarli o gestire il nostro ambiente. Questo compito è ulteriormente facilitato dalle numerose librerie già presenti nei più comuni linguaggi di programmazione.

C'è poi da considerare un altro vantaggio di Cosm: è totalmente gratuito. Ciò significa che non dobbiamo preoccuparci per eventuali costi aggiuntivi dei server né di trovare una piattaforma web che su cui installarli. Possiamo concentrarci esclusivamente sull'interazione con essi e su come reperire i dati quando ne

abbiamo bisogno.

Passando ora a PubSubHubBub, invece i risultati a cui siamo giunti sono decisamente più interessanti.
Presentato come un tentativo di dar vita ad un web in tempo reale, ovvero un mondo in cui si elimina il bisogno del polling a tutto vantaggio dell'efficienza nelle comunicazioni, i test ci hanno messo davanti a diversi scenari.
Nel primo scenario di testing (spiegato nel Capitolo 4.6.2 e che consiste nell'avviare 10 thread diversi sullo stesso server, ciascuno dei quali pubblica 20 dati differenti) i risultati non sono stati esaltanti: l'hub riesce sempre a ricevere tutte le richieste che noi gli inviamo, ma fallisce nell'andare a recuperare con sufficiente rapidità il nuovo valore. Ciò fa sì che spesso venga perso il valore per cui si era inviato il messaggio POST e si recuperi poi uno di quelli successivi, in quanto il nostro feed è già stato aggiornato più volte. È pertanto sconsigliabile utilizzare PuSH in presenza di server che inviano diverse decine di richieste al secondo.
Diverso è invece il secondo scenario (descritto nel Capitolo 4.6.4, con un solo thread per server che invia 100 messaggi). Qui l'hub riceve le pubblicazioni da due server differenti, i quali pubblicano il nuovo contenuto soltanto dopo aver ricevuto una risposta dall'hub (indipendentemente che essa indichi che la richiesta di venire a prelevare il dato sia stata positiva o meno).
Questo ci avvicina ad una situazione più plausibile nel mondo reale. PuSH è stato infatti pensato per aggiornare gli interessati dei nuovi contenuti pubblicati da più fonti, tipicamente situate su server diversi. Come clienti noi possiamo sottoscriverci a queste notifiche, così da sapere immediatamente quando è stata pubblicato una nuova notizia.
Immaginando il caso più classico, ovvero quello di un sito web che pubblica una news o un blog che inserisce un nuovo intervento, difficilmente troviamo pubblicazioni ripetute diverse volte al secondo. Al contrario, è decisamente più plausibile che esse vengano fatte a distanza di minuti l'una dall'altra. Questo permette all'hub di non sovraccaricarsi e di evitare che possibili limitazioni di banda impediscano un corretto scambio dei nuovi valori.

Come per Cosm, anche PuSH è un protocollo che non prevede spese da parte dell'utilizzatore, essendo implementabile anche su piattaforme gratuite.

Tutti i nostri test sono stati effettuati nel caso peggiore, ovvero quello in cui publisher e hub siano su server differenti. Questo aumenta i ritardi tra un invio e l'altro ed è fonte di possibili perdite di informazioni nel caso in cui la connessione non sia efficiente o l'hub sia oberato di richieste.
Per future implementazioni abbiamo diversi scenari da testare. L'elemento su cui concentrarsi è sicuramente PubSubHubBub. Unire sulla stessa macchina publisher e hub eliminerebbe un'interazione con la rete, rendendo quindi più efficiente la distribuzione dei nuovi contenuti: in questo scenario è interessante eseguire nuovamente i test con più thread che pubblicano contenuti contemporaneamente. Nel nostro caso abbiamo visto come l'hub faccia fatica a recuperare tutti i nuovi valori, ritrovandosi spesso con un dato più recente di quello che era venuto a prendere. Lavorando sulla stessa macchina questo problema dovrebbe invece mitigarsi se non sparire del tutto: ciò porterebbe ad una migliore diffusione di tutti i valori monitorati.
Il Capitolo 3.4.1 contiene una figura (la numero 18) che offre altri spunti su come ampliare l'architettura. PuSH non è pensato per la sola interazione tra un publisher e un subscriber, ma mira a distribuire i contenuti a quanti più interessati possibili. Come spesso accade, un subscriber può trasformarsi a sua volta in un publisher e girare le informazioni ad altri sottoscrittori. È quindi interessante effettuare test sull'architettura quando si hanno molte entità in gioco, possibilmente dislocate su tutto il globo (cosa che porta ad un potenziale aumento dei ritardi nella trasmissione dei dati). Lo scopo è ovviamente quello di realizzare un'implementazione scalabile, che si adatti al meglio alle esigenze dei subscriber offrendo al contempo un servizio sempre efficiente senza sovraccaricare una sola delle parti in gioco.
Un ultimo scenario che merita di essere approfondito comporta lo spostamento del subscriber dalla rete ad un dispositivo fisico. Nella nostra implementazione abbiamo visto come lo smartphone Android invii i dati e poi proceda alla loro lettura ogni qual volta si rinfresca l'apposita pagina. I dati vengono tuttavia

ricevuti dal server che contiene l'applicazione distribuita, così da elaborarli e mostrarli sotto forma di feed RSS. Le attuali librerie Java di PuSH presentano molto codice deprecato, mal digerito da Android. Tenendo buona la loro impostazione, che di fatto prevede già un'implementazione di publisher e subscriber (quest'ultimo da completare con la gestione che più si preferisce dei file XML che l'hub invia), si può provare ad aggiornarle per far sì che sia possibile effettivamente ricevere le nuove notifiche direttamente sul cellulare, senza dover ricorrere ad intermediari o attendere la pressione dell'apposito tasto. Realizzare un endopoint della comunicazione su di un dispositivo mobile comporta necessariamente tutta una serie di accorgimenti legati alla rete: vista la frequenza con cui si potrebbe perdere la connessione, ogni volta che ci si collega per abilitare la ricezione degli aggiornamenti è molto probabile avere un IP differente da quello della volta precedente. Ciò comporta una nuova sottoscrizione, dato che quella effettuata in passato invia dati ad un destinatario diverso. Onde evitare che l'hub sprechi banda inviando dati a destinatari inesistenti dobbiamo pertanto preoccuparci di gestire a dovere la ricezione delle pubblicazioni ad ogni nuovo collegamento, ricordandoci di effettuare anche una rimozione della nostra precedente sottoscrizione così da alleggerire il compito dell'hub, che non deve aspettare il raggiungimento del timeout prima di verificare se quel destinatario è ancora presente o meno.

Bibliografia

Android Trainer, *Updating UI from background service in android*, http://androidtrainningcenter.blogspot.it/2012/08/updating-ui-from-background-service-in.html

Antonio Corradi, Luca Foschini, *Reti di Calcolatori T*, http://lia.deis.unibo.it/Courses/RetiT/RetiT_11-12/

Commons Math 1.2 API, *Class PoissonDistributionImpl*, http://commons.apache.org/math/api-1.2/org/apache/commons/math/distribution/PoissonDistributionImpl.html

Cosm, *Where the Internet of Things is being built*, https://cosm.com/

Cosm, *API Documentation*, https://cosm.com/docs/

David Turner, Jinseok Chae, *A Simple News Feed Application*, http://cse.csusb.edu/turner/java_web_programming/news_feed_app/

Google, *API Guides*, http://developer.android.com/guide/components/index.html

Google, *Reference*, http://developer.android.com/reference/packages.html

Google, *Training*, http://developer.android.com/training/index.html

Google Groups, [Java] Status of subscription: 409, https://groups.google.com/forum/#!topic/pubsubhubbub/KhGCzIztIzQ

Paolo Bellavista, Ilaria Bartolini, *Tecnologie Web T*, http://www-db.deis.unibo.it/courses/TW/

PubSubHubBub, *A simple, open, web-hook-based pubsub protocol & open source reference implementation*, http://code.google.com/p/pubsubhubbub/

RSS Advisory Board, *RSS 2.0 Specification*, http://www.rssboard.org/rss-specification

The Apache Software Foundation, *HTTP Components*, http://hc.apache.org/

The ECLIPSE Foundation, *Eclipse*, http://www.eclipse.org/

The ROME Project, *ROME : RSS and Atom Utilities for Java*, http://rometools.org/

websmithing.com, *How to Update the UI in an Android Activity Using Data from a Background Service*, http://www.websmithing.com/2011/02/01/how-to-update-the-ui-in-an-android-activity-using-data-from-a-background-service/