Connettore Publish-Subscribe Content-Based adeguato ad ambiente Mobile per mezzo di politiche adattative

Stefano Agostini Matricola: 0234240
Paolo Salomé Matricola: 0233502
Alessandro Valenti Matricola: 0228709

August 11, 2017

1 Introduzione

Il connettore *Publish-Subscribe* presenta caratteristiche interessanti per sistemi caratterizzati da una elevata dinamicitá, grazie all'elevato livello di disaccoppiamento che é in grado di garantire tra componenti. I componenti coinvolti nell'interazione, per mezzo del connettore, sono i *produttori* e i *consumatori* di determinati eventi generabili. Un evento é caratterizzato dal cosidetto *topic* che rappresenta il suo argomento d'appartenenza al quale un *consumatore* puó sottoscriversi se interessato a tale classe di informazioni; il *produttore* a sua volta puó pubblicare eventi associati ad un determinato *topic*. Nello specifico é possibile caratterizzare un determinato evento (dopo averlo associato ad un *topic*) in base al suo contenuto utilizzando la politica del tipo *content-based*. Tale modalitá di sottoscrizione viene realizzata specificando un filtro e di conseguenza favorisce la riduzione del carico di eventi recapitati ai *consumatori*. Il connettore realizzato si presta all'utilizzo in ambiente mobile in quanto assicura un alto grado di disaccoppiamento, continuando a garantire uno scambio affidabile di messaggi tra le entitá coinvolte. Tuttavia ci siamo posti come obiettivo quello di aggiungere a tale sistema politiche adattative che tengano conto delle limitate risorse energetiche e computazionali a disposizione dei nodi mobili, continuando a garantire uno scambio affidabile di messaggi tra le entitá attraverso il connettore. Intuendo un relazione funzionale tra consumo energetico e traffico generato dal nodo mobile, abbiamo compiuto uno studio statistico per avallare la nostra ipotesi che si é rivelato soddisfacente: pertanto abbiamo potuto realizzare un modello matematico finale. La nostra politica adattativa ci permette di controllare il tasso di consumo energetico controllando direttamente la frequenza di interazione del nodo mobile con la rete, gestendo, tramite un monitoraggio continuo dello stato della batteria, le risorse limitate del dispositivo.

2 Obiettivi

L'obiettivo é quello di realizzare un connettore *Publish-Subscribe* in modalitá *content-based* (basato su *string pattern matching*) ed adattarlo ad ambiente mobile. Nel dettaglio il nostro sistema deve essere costituito da nodi mobili che possono assumere il ruolo di *consumatore o produttore* e dal connettore che deve comporsi delle seguenti entitá:

- *Event-Service*, il quale si preoccupa di gestire gli eventi generati e di recapitare notifiche
- *Filter-Service*, il quale realizza la modalitá di sottoscrizione *content-based*

Si vuole inoltre adattare tale sistema ad ambiente mobile mediante una politica che abbia come scopo quello di monitorare e controllare:

- *Il consumo energetico*
- *Il traffico generato* nell'interazione tra nodo mobile e connettore

3 Architettura generale

Il sistema si basa fondamentalmente su un'architettura centralizzata. Infatti il connettore *Publish-Subscribe* si compone di un server che svolge sia la funzione di *Event-Service* che la funzione di *Filter-Service*. I client non interagiscono direttamente con il connettore ma scambiano messaggi mediante delle code che implementano il paradigma di interazione *message queueing*. Tale paradigma offre dei vantaggi in termini di disaccoppiamento spaziale e temporale, garantendo la possibilitá al client di potersi disattivare o spegnere immediatamente una volta inoltrato un messaggio.
Il server gestisce, tramite una *database non relazionale* distribuito, la persistenza delle informazioni indispensabili per il corretto funzionamento del sistema dal punto di vista dei client, ovvero

- *Topic*
- *Sottoscrizioni e Filtri*

I nodi mobili, cosí come il server, gestiscono la persistenza in locale di alcune informazioni attraverso un *database relazionale*. Tali informazioni sono i *topic* al momento disponibili nel sistema, le *notifiche* in ingresso, le *sottoscrizioni* effettuate con i relativi *topic e filtri*. Il client attraverso procedure interne effettua, periodicamente, un'operazione di *pull* sulla coda per poter aggiornare i dati relativi alle notifiche ad esso destinate. Analogamente richiede al server la lista dei *topic* disponibili per aggiornare le sue informazioni locali.
Di seguito illustriamo l'architettura attraverso uno schema in Figure 1.

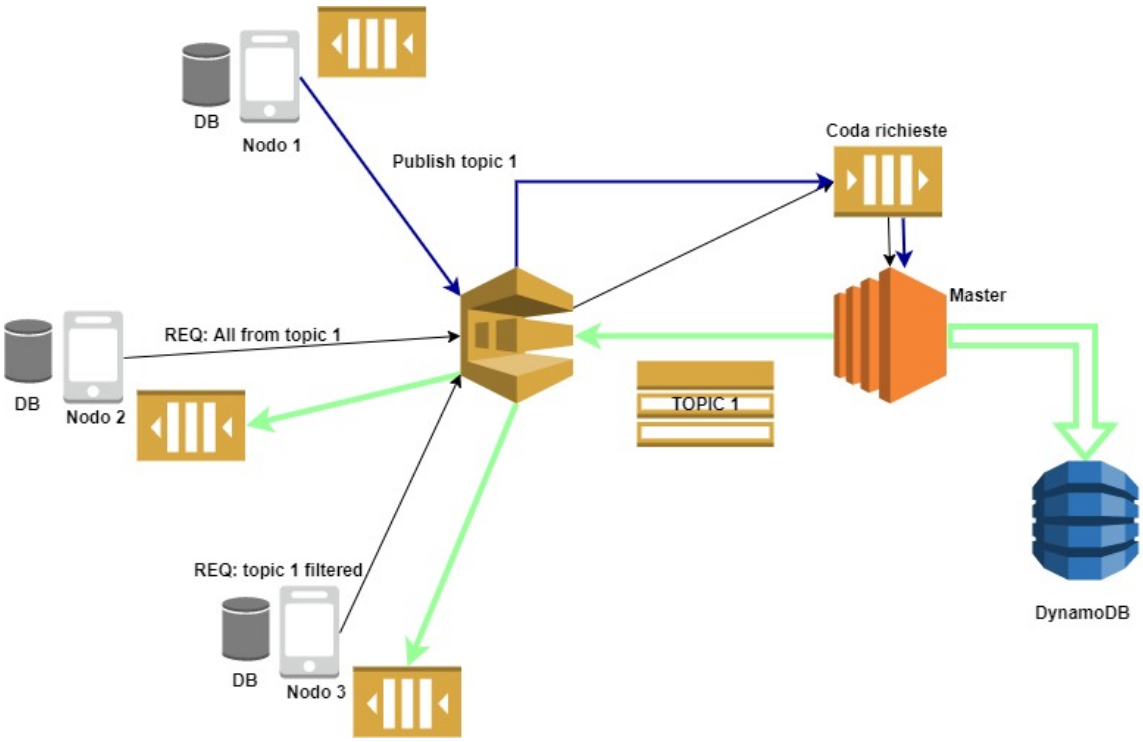

Figure 1: Architettura

3.1 Servizi Cloud

Come *Cloud Provider* é stato scelto *Amazon AWS*. I servizi utilizzati per l'implementazione del connettore sono:

- **Amazon EC2**: servizio *IaaS* sfruttato per la realizzazione del nodo master che identifica il server dell'architettura. Svolge il ruolo di *Event-Service* e *Filter-Service* nonché di gestore del *database NoSQL*. Per la comunicazione con i nodi client interagisce con le code del servizio *Amazon SQS*
- **Amazon SQS**: é un servizio di accodamento di messaggi distribuito che disaccoppia completamente i componenti dell'architettura. In tal modo semplifica e riduce i costi del coordinamento dei componenti stessi. Offre un *throughput* elevato e *tunable*, un ordinamento semplificato e distribuzione di tipo *at-least-once*. In generale é possibile impostare un algoritmo di *scheduling* dei messaggi: nel nostro caso é stato utilizzato l'algoritmo *FIFO*
- **Amazon DynamoDB**: é un servizio di *datastorage* distribuito basato su un *database NoSQL* di tipo *key-value*. *DynamoDB* implementa un sis-

tema di tipo *AP*, disponibilitá e tolleranza alle partizioni dei dati, secondo il *CAP Theorem*. L'alto grado di disponibilitá ed un accesso di tipo *anywhere* alle informazioni sono due caratteristiche adatte ad un utilizzo in ambiente mobile. Essendo inoltre un database *read-mostly*, ovvero pensato prevalentemente per operazioni di lettura, si adatta perfettamente al nostro sistema, il quale richiede piú letture che scritture.

3.2 Ambiente Client

Il nostro sistema é pensato per un utilizzo da parte di *client leggeri* con sistema operativo *Android*. *Android* é un sistema operativo per dispositivi mobili sviluppato da *Google Inc.* e basato sul kernel *Linux*. Le applicazioni che vengono eseguite sulle piattaforme *Android* sono scritte prevalentemente in linguaggio *Java*, come nel nostro caso. L'applicazione *client* interagisce con un *database relazionale* interno di tipo *SQLite*.

3.3 Ambiente Server

Sfruttando il servizio *IaaS di Amazon EC2*, é stato possibile scegliere il sistema operativo del server e le caratteristiche della macchina *host*. In particolare abbiamo scelto un'istanza *t2.micro* dotata di una *virtual cpu*, *1 GB di ram* e *8 GB di storage*. É stato scelto come sistema operativo *Linux Ubuntu Server v.16.04 LTS*. La scelta di tale sistema operativo é da ricondurre alla semplicitá di installazione di componenti aggiuntivi e alla sua versatilitá.

3.4 Interazione dei componenti

Per promuovere la comunicazione tra i client ed il server, disaccoppiandoli, é stato scelto di utilizzare diverse code *SQS*, partizionandone i compiti. In particolare vengono generate tre code fisse per richiedere l'esecuzione delle operazioni fondamentali di un sistema *Publish-Subscribe* al server:

- **Creation Queue**: i client scrivono su questa coda per richiedere al server la creazione di un determinato *topic*
- **Notification Queue**: i client scrivono su questa coda per pubblicare una notifica su un determinato *topic*
- **Subscription Queue**: i client scrivono su questa coda per sottoscriversi in modalitá *content-based* ad un determinato *topic* oppure annullare una sottoscrizione precedentemente effettuata

Ogni messaggio proveniente dai nodi mobili raggiunge il server per mezzo delle tre code fisse sopra descritte (come si nota in Figure 1). Il server elabora i messaggi in ingresso ed aggiorna le informazioni riguardanti i *topic* e le sottoscrizioni nell'istanza di *DynamoDB*.
Per la ricezione di messaggi ogni client ha a disposizione una coda dedicata, la

quale viene creata al momento dell'ingresso del nodo nel sistema ed é utilizzata per la ricezione dei messaggi di notifica.
Infine il database *SQLite*, presente in ogni client, é utilizzato per tenere traccia dei *topic* presenti al momento nell'istanza di *DynamoDB* e per memorizzare le notifiche ricevute.

4 Implementazione

4.1 Server

Abbiamo scelto di utilizzare lato server la piattaforma *event-driven Node.js* per motivi di immediatezza, versatilitá ed efficienza. Tale piattaforma supporta il linguaggio di programmazione *Javascript*, linguaggio di scripting orientato agli oggetti e agli eventi. L'efficienza di *Node.js* é legata al suo modello di *networking*, ossia *I/O event-driven*. La piattaforma richiede al sistema operativo di ricevere notifiche al verificarsi di determinati eventi e rimane in modalitá *sleep* fino alla ricezione della notifica stessa, in corrispondenza della quale avvia l'esecuzione di una funzione di *callback* di risposta. Tale modello di *networking* si rivela efficiente nella gestione di un elevato traffico di rete, come potrebbe accadere nel caso del nostro sistema.
Di seguito ci apprestiamo a descrivere puntualmente l'implementazione dei componenti server.

4.1.1 Formato *JSON* dei messaggi scambiati

I messaggi diretti da client a server possono essere di tre tipi (un tipo per coda):

- **Crezione-rimozione di un topic**:

```
{
        "type": string, // "CREATE" or "DELETE"
        "userID": string, // macaddress
        "topicName": string
}
```

- **Sottoscrizione-Cancellazione**:

```
{
        "type": string, // "SUB" or "UNSUB" or "UNSUBALL"
        "userID": string, // macaddress
        "topicName": string,
        "filter": string
}
```

- **Pubblicazione di notifiche**:

```
{
        "userID": string, // macaddress
        "topicName": string,
        "messageBody": string
}
```

I messaggi di notifica elaborati dal server e destinati alle code dedicate dei client hanno il seguente formato:

```
{
        "topicName": string,
        "messageBody": string
}
```

4.1.2 AWS SDK

Amazon AWS mette a disposizione delle librerie ai programmatori per consentirgli di interagire e accedere ai servizi *Cloud* da codice. In particolare *AWS* supporta anche *Javascript* con le proprie *SDK* (*Software Development Kit*). Sono state utilizzate le *API AWS SDK* relative ai servizi *SQS e DynamoDB* ed estese per mezzo di due librerie: *sqs_api.js e dynamo_api.js*.

sqs_api.js

- *create_queue*: crea una coda identificata dall'attributo *QueueNamePrefix* (parametro della funzione) se non esiste. Questa funzione viene utilizzata sia per la creazione delle tre code fisse che di quelle dedicate ai singoli nodi client.

- *write_queue*: scrive un messaggio nella coda specificata (messaggio e nome della coda entrambi parametri della funzione). Per risalire alla coda, ricava il suo *URL* mediante l'*API SDK getQueueUrl* utilizzando il suo nome. Se il recupero va a buon fine, viene inviato il messaggio sulla coda invocando *l'API SDK sendMessage*.

- *read_queue*: legge i messaggi presenti nella coda di cui é specificato il nome (parametro della funzione). Ricava lo *URL* della coda mediante l'*API SDK getQueueUrl* alla quale passa il nome come argomento. Se la *getQueueUrl* va a buon fine, vengono recuperati i messaggi presenti nella coda selezionata grazie all'*API SDK receiveMessage*. Nella *callback* della *receiveMessage* viene generato un nuovo processo che resta in attesa della comunicazione dei messaggi da parte del processo padre. Quest'ultimo invia i messaggi letti dalla coda al processo figlio sfruttando la funzione *send* dell'*IPC di Node.js*. In ricezione, il processo figlio (definito come *consumer*, i cui dettagli sono nella sezione 4.1.4) elabora i messaggi e termina. Tutti i messaggi letti dalla coda vengono eliminati dalla stessa attraverso la funzione *delete_messages* che ci apprestiamo a descrivere.

- *delete_messages*: prende come parametri lo *URL* della coda e la lista delle *entries* lette, oggetti che seguono un formato specificato dagli standard *AWS SDK*. Per l'eliminazione viene invocata l'*API SDK deleteMessage-Batch*, la quale permette la cancellazione di *batch* interi di messaggi.

dynamo_api.js

- *creation_table*: prende come parametro il nome della tabella da creare se non esiste (supporta la creazione delle tabelle *Topic e Subscription*). Definisce il formato della tabella per la sua creazione per mezzo dell'*API SDK createTable*. La tabella *Topic* ha come chiave il *topicName* (*string*) che ricopre il ruolo di *partition key*. Invece la tabella *Subscription* ha come chiave la coppia *(topicName, userID)* (entrambe stringhe) e la prima ha il ruolo di *partition key* mentre la seconda quello di *sort key*.

- *create_topic*: prende come parametri il *topicName* e lo *user_id* (creatore del *topic*). Utilizza l'*API SDK get* e, se il *topic* non é presente nella tabella *Topic*, crea un nuovo *item* e lo salva usando l'*API SDK put*.

- *delete_topic*: prende come parametri il *topic_name* e lo *user_id* (identificativo del client che ha richiesto al server di rimuovere il *topic*). Recupera, mediante l'*API SDK get*, il *topic* dalla tabella e controlla se l'*id* del creatore coincide con l'*id* del client richiedente. In caso affermativo rimuove l'*item* dalla tabella usando l'*API SDK delete*.

- *create_subscription*: prende come parametri *topic_name, user_id* e *filter*. Controlla l'esistenza del *topic* nella tabella *Topic*. Se é verificata scrive nella tabella *Subscription* il nuovo *item*, con chiave *(topic_name, user_id)*.

- *delete_subscription*: prende come parametri *topic_name* e *user_id*. Effettua la cancellazione dalla tabella *Subscription* dell'*item* con chiave *(topic_name, user_id)*, usando l'*API SDK delete*.

- *scan_all_topics*: utilizza l'*API SDK scan* per estrarre la lista dei *topic* al momento presenti nel sistema.

- *send_notification*: prende come parametri *topic_name* e *message_body* (rispettivamente il *topic* e il messaggio inviato dal client che vuole effettuare una *publish* su di esso). Effettua una *query* sulla tabella *Subscription*, tramite l'*API SDK query*, specificando il *topic_name*. Il risultato delle *query* é una lista di coppie *(user_id, filter)*, ognuna delle quali rappresenta un sottoscritto al *topic* ed il relativo filtro. Per ogni *item* della lista viene controllata l'occorrenza del suo filtro all'interno del *message_body*, ed in caso fosse verificata viene inoltrata la notifica alla coda dedicata del client mediante la funzione *write_queue* della libreria *sqs_api.js*.

4.1.3 *Readers*

I tre processi *Readers* si occupano di leggere periodicamente dalle tre code fisse *Creation Queue*, *Notification Queue* e *Subscription Queue*. Tali processi sono continuamente in esecuzione e sfruttano la funzione *polling* del modulo *polling.js*, la quale legge periodicamente dalla coda specificata, passata come argomento, mediante la funzione *read_queue* della libreria *sqs_api*. Nel caso in esame é stato scelto come intervallo di lettura un tempo pari a *5000 millisec*.

4.1.4 *Consumer e Updaters*

Ogni processo di tipo *Reader*, alla lettura dei messaggi dalla sua coda (tramite la funzione di lettura *read_queue* del modulo *sqs_api*), crea un processo di tipo *Consumer* invocando una *fork* con argomento il modulo *consumer.js* e successivamente gli invia i messaggi letti tramite la funzione *send* dell'*IPC* di *Node.js*. Ogni *Reader* invia al processo *Consumer* un messaggio che segue il seguente formato:

```
{
        data: data_read.Messages, // Lista dei messaggi letti dalla coda
        queue: queue_name // Nome della coda di lettura
}
```

In tal modo il processo *Consumer* puó distinguerne la coda di appartenenza per delegare al corrispondente *Updater* la gestione dei dati letti (*data_read.Messages*). Gli *Updaters* sono tre funzioni, ognuna implementata nel suo modulo:

- *notification_updater*: funzione eseguita quando *queue = notificationQueue*. Per ogni messaggio in *data_read.Messages* invoca la funzione *send_notification* della libreria *dynamo_api*.
- *subscription_updater*: funzione eseguita quando *queue = subscriptionQueue*. Per ogni messaggio, in base al tipo (*SUB o UNSUB*), invoca la funzione *create_subscription* o *delete_subscription*, entrambe appartenenti alla libreria *dynamo_api*.
- *topic_updater*: funzione eseguita quando *queue = creationQueue*. Per ogni messaggio, in base al tipo (*CREATE o DELETE*), invoca la funzione *create_topic* o *delete_topic*, entrambe appartenenti alla libreria *dynamo_api*.

Al termine della gestione dei messaggi letti il processo *Consumer* termina.

4.1.5 *Server Http e Routes*

Al fine di separare le *routes* dal server *http* é stato utilizzato il *framework Express.js*. Nel modulo *routes.js* é disponibille un *endpoint* per rispondere a richieste *http* di tipo *get*, il quale invoca la funzione *scan_all_topics di dinamo_api.js* per restituire al client richiedente la lista di tutti i *topic* presenti nel sistema. Il *Server http* é un processo che utilizza le *routes* e rimane in ascolto sulla porta *8080*.

4.1.6 *Main*

Il modulo *main.js* crea le tre code fisse del sistema invocando la funzione *create_queue* di *sqs_api* e le tabelle *Subscription e Topic* tramite la funzione *creation_table* di *dinamo_api*. Infine, per mezzo di *fork*, avvia i quattro processi stabili del sistema: i tre *Readers* e il *Server http*.
In Figure 2 troviamo lo schema dei processi attivi nel server: é possibile osservare che vi sono quattro processi sempre attivi e diversi processi temporanei che eventualmente si attivano con cadenza temporale di *5 sec*.

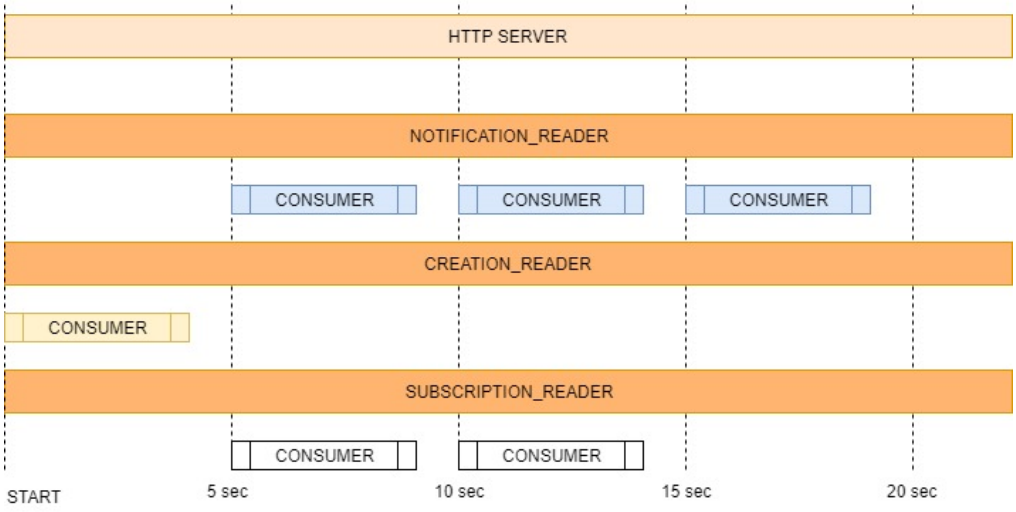

Figure 2: Processi

4.2 Client

L'applicazione lato client é stata realizzata in modo tale da potersi adattare secondo due modalitá in risposta al consumo attuale di batteria:

- **modalitá dinamica**: il client é un sistema *self-adaptive*, ovvero in base al livello attuale di batteria del dispositivo modula la frequenza di interazione con la sua coda dedicata, limitando l'utilizzo della scheda di rete. Implementa il *feedback-control-loop* di *MAPE-K*.
- **modalitá statica**: l'utente che utilizza l'applicazione puó impostare una politica di risparmio energetico a priori, con la quale si limita la luminositá dello schermo del dispositivo.

É stata dimostrata l'efficacia di queste politiche e nel seguito della trattazione verranno mostrati i test ed il modello matematico finale corrispondente.

4.2.1 *Util*

Questa classe contiene diversi metodi che si possono raggruppare in categorie:

- Metodo per accedere ai servizi di *AWS SQS*(*getProperty*).
- Metodo per rilevare il livello di batteria attuale del dispositivo (*getBatteryLevel*).
- Metodi per convertire nel corretto formato i messaggi destinati al connettore (*createDelete,subUnsub,publish*), attraverso le tre code fisse precedentemente descritte.
- Metodo per recuperare la lista dei *Topic* presenti attualmente nel sistema (*getTopicList*). Contatta l'endpoint del server (*allTopics*) ed eventualmente compatta i messaggi di risposta.

4.2.2 *DBHelper*

Questa classe estende *SQLiteOpenHelper* e fornisce i metodi *CRUD* sulle tabelle del *DB* locale. Le tabelle sono le seguenti:

- **Filtered**: é la tabella contenente le notifiche destinate all'utente ed ha il seguente schema:
  - *id*: id della notifica;
  - *date*: data di ricezione dalla coda;
  - *topic*: *Topic* di appartenenza;
  - *content*: contenuto della notifica.
- **Topic**: é la tabella contenente tutti i *Topic* aggiornati all'ultima richiesta al server ed ha il seguente schema:
  - *id*: id del *Topic*;
  - *name*: nome del *Topic*.
- **TopicFiltered**: é la tabella contenente tutti i *Topic* a cui l'utente si é sottoscritto con il filtro associato ed ha il seguente schema:
  - *id*: id del *Topic*;
  - *name*: nome del *Topic*;
  - *filter*: filtro di sottoscrizione sul topic.

4.2.3 *AWSSimpleQueueServiceUtil*

Questa classe implementa dei metodi per accedere ai servizi *AWS SQS* mediante l'utilizzo delle *API AWS SDK*. Tali metodi sono i seguenti:

- *createQueue*: prende come argomento il nome della coda e la crea utilizzando l'*API AWS SDK createQueue*. Restituisce infine l'*URL* della coda.
- *getQueueUrl*: restituisce l'*URL* della coda il cui nome é passato come argomento. Usa l'*API AWS SDK getQueueUrl*.
- *sendMessageToQueue*: invia un messaggio sulla coda (una delle tre code fisse) identificata dal suo *URL*, passati entrambi come argomento. Usa l'*API AWS SDK sendMessage*.
- *getMessagesFromQueue*: specificato l'*URL* della coda dedicata del client, legge i messaggi presenti nella stessa usando l'*API AWS SDK receiveMessage*. Infine restituisce una lista.
- *deleteMessageFromQueue*: cancella un messaggio dalla coda dedicata identificata dal suo *URL*, entrambi passati come argomento. Usa l'*API AWS SDK deleteMessage*.

4.2.4 *Fragment*

Un *Fragment* rappresenta un ambiente oppure una porzione dell'interfaccia in una Activity. Si puó pensare a un *Fragment* come sezione modulare di un'attivitá che ha un proprio ciclo di vita, riceve i propri eventi di input e che é possibile aggiungere o rimuovere mentre l'attivitá é in esecuzione. É possibile combinarne diversi in una singola *activity* per costruire una *multi-pane UI*, e questo é quello che ci ha spinti all'utilizzo degli stessi.
Nella fase di inizializzazione dei *Fragment* vengono eseguiti gli *AsyncTask*.
L'*AsyncTask* consente l'uso corretto e semplice del *thread UI*. Tale classe permette di eseguire operazioni in background e pubblicare risultati sul thread dell'interfaccia utente senza dover manipolare i *threads* e/o *handlers*. Un *AsyncTask* ha il compito di eseguire una porzione di codice su un *thread* in background, il cui risultato viene pubblicato sul *thread UI*.
Nella nostra applicazione sono presenti quattro tipologie di *Fragment*:

- **ReadFragment**: consente all'utente di leggere i messaggi presenti nel *DB* del dispositivo mobile raggruppati per *Topic*. All'avvio viene invocato un *AsyncTask* per popolare una *listView* (presente sul layout *activity_Read*) contenente tutti i *Topic* a cui l'utente si é sottoscritto. All'interno dell'*AsyncTask*, per recuperare i dati dal *DB* viene invocato il metodo *getAllTopicFilter* della classe *DBHelper*. Alla selezione di un *item* della lista viene eseguito il secondo *AsyncTask*, il quale popolerá nuovamente la *listView* con le notifiche presenti nel *DB* relative a quel *Topic*. All'interno dell'*AsyncTask*, per recuperare le notifiche dal *DB*, relative al *Topic* selezionato, viene invocato il metodo *getFilteredMessageByTopicName* della classe *DBHelper*.

- **WriteFragment**: consente all'utente di pubblicare notifiche sui *Topic* attualmente esistenti o di crearne di nuovi. All'avvio viene invocato un *AsyncTask* per popolare una *listView* (presente sul layout *activity_write*) contenente tutti i *Topic* attualmente nel sistema. All'interno dell'*AsyncTask* viene invocato il metodo *getTopicList*, fornito dalla classe *Util*, per contattare il server con il fine di ottenere la lista dei *Topic* presenti nel sistema. Ogni *item* della *listView* puó essere selezionato per inviare una notifica su quel *Topic*: questa fase richiede l'interazione con l'utente, il quale dovrá scrivere il testo della notifica per poi confermarne l'invio. La notifica viene formattata utilizzando il metodo *publish* della classe *Util*, ed inviata con l'utilizzo dell' *API AWS SDK sendMessageToQueue* specificando la coda *notificationQueue* in quest'ultima.
  La creazione di nuovi *Topic* é possibile in pochi passi:
  - L'utente seleziona un tasto presente sull'*activity_write*;
  - compare a schermo una finestra di dialogo in cui inserire il nome del nuovo *Topic*;
  - se si conferma l'operazione, viene inviata una richiesta di creazione *Topic* al server con l'utilizzo dell'*API AWS SDK sendMessageToQueue* verso la coda *creationQueue*. La creazione del *Topic* é cosí demandata al server che gestisce la richiesta come spiegato in precedenza.

- **SetFilterFragment**: consente all'utente di sottoscriversi ad i *Topic* attualmente esistenti. All'avvio viene invocato un *AsyncTask* per popolare una *gridView* (presente sul layout *activity_set_filter*) contenente per ogni *Topic* attualmente nel sistema il filtro di sottoscrizione associato. All'interno dell'*AsyncTask* si eseguono i seguenti passi:
  - Viene invocato il metodo *getTopicList*, fornito dalla classe *Util*, per contattare il server con il fine di ottenere la lista dei *Topic* presenti nel sistema;
  - vengono salvati gli eventuali nuovi *Topic* all'interno del *DB*, utilizzando il metodo *insertNotExistTableTopic* della classe *DBHelper*;
  - si recupera la lista di tutti i *Topic*, ciascuno con l'eventuale filtro associato.
  - si popola la *gridView*.

  L'utente puó selezionare ciascun *item* della griglia per cambiare il filtro corrispondente a quel *Topic* o per richiedere una nuova sottoscrizione. Se si conferma l'operazione, viene inviata una richiesta di sottoscrizione *Topic* al server con l'utilizzo dell'*API AWS SDK sendMessageToQueue* verso la coda *subscriptonQueue*.

- **SettingsFragment**: consente all'utente di impostare l'applicazione in modalitá risparmio energetico, la quale non fará altro che impostare la

luminositá dello schermo ad un valore fissato. L'utente puó attivare o meno questa modalitá grazie ad uno *Switch* presente nell'*activity_settings*.

4.2.5 *MainActivity*

La *MainActivity* contiene tutti i controlli utili per la navigazione all'interno delle sezioni dell'applicazione (ossia quelle di pubblicazione notifiche, sottoscrizione *Topic*, lettura notifiche) e passa quindi il controllo al *Fragment* corrispondente alla sezione prescelta. Inoltre é presente un *AsyncTask* con il compito di prelevare le notifiche in arrivo, leggendole dalla coda dedicata con l'utilizzo dell'*API AWS SDK getMessageFromQueue*. Quest'operazione é effettuata periodicamente con frequenza di *pull* dipendente dal livello corrente della batteria del dispositivo:

- 75% <Livello batteria <= 100% ➢ 1 pull ogni secondo
- 50% <Livello batteria <= 75% ➢ 1 pull ogni 5 secondi
- 0% <Livello batteria <= 50% ➢ 1 pull ogni 30 secondi

Questa scelta é motivata dai risultati ottenuti dallo studio sulla relazione funzionale che esiste tra frequenza di *pull* (e quindi frequenza di utilizzo della scheda di rete) e consumo della batteria. Nel prossimo capitolo viene illustrato l'esito di tale indagine.

5 Politica adattativa

5.1 Oggetto dell'analisi

La natura limitata inerente ai dispositivi mobili impone al progettista di sistemi informatici di condurre la ricerca di una politica adattativa che preservi le loro risorse computazionali e/o energetiche e/o di memoria, inevitabilmente finite e degradabili. Non possono pertanto essere trascurate, nell'atto della progettazione e realizzazione di un sistema informatico, le differenze sostanziali che intercorrono tra nodi fissi e nodi mobili in termini di *abbondanza* di risorse fisiche ed energetiche: pena il rapido degrado e la mala gestione delle stesse.
L'idea é quella di identificare una risorsa misurabile del dispositivo mobile, finita e degradabile, e di tentare di risalire ad una sua relazione funzionale con una variabile (indipendente) direttamente controllabile. Una tale relazione tra le due grandezze permetterebbe di limitare e gestire il deterioramento della risorsa del dispositivo, modificando semplicemente il valore della variabile indipendente (e controllabile). In sintesi si cerca una relazione del tipo:

$\{f(\theta),\, \theta \in K\}$

- $(f_{\mathrm{p}}(t),\, t)$ ➢ *lineare* poiché $R^2 \approx 0.9$
- $\frac{df_{\mathrm{p}}(t)}{dt}$ ➢ derivata rispetto al tempo

- $(\frac{df_{\mathrm{p}}(t)}{dt},\ p)$ ➢ *lineare* poiché $R^2 \approx 0.9$

5.2 Test e risultati

6 Experimental Data

Mass of empty crucible	7.28 g
Mass of crucible and magnesium before heating	8.59 g
Mass of crucible and magnesium oxide after heating	9.46 g
Balance used	#4
Magnesium from sample bottle	#1

7 Sample Calculation

Because of this reaction, the required ratio is the atomic weight of magnesium: 16.00 g of oxygen as experimental mass of Mg: experimental mass of oxygen or $\frac{x}{1.31} = \frac{16}{0.87}$ from which, $M_{\mathrm{Mg}} = 16.00 \times \frac{1.31}{0.87} = 24.1 = 24\,\mathrm{g\,mol^{-1}}$ (to two significant figures).

8 Results and Conclusions

The atomic weight of magnesium is concluded to be $24\,\mathrm{g\,mol^{-1}}$, as determined by the stoichiometry of its chemical combination with oxygen. This result is in agreement with the accepted value.

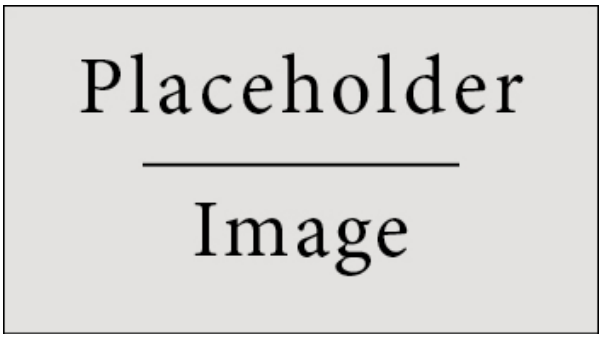

Figure 3: Figure caption.

9 Discussion of Experimental Uncertainty

The accepted value (periodic table) is $24.3\,\mathrm{g\,mol^{-1}}$ **?**. The percentage discrepancy between the accepted value and the result obtained here is 1.3%. Because only a single measurement was made, it is not possible to calculate an estimated standard deviation.
The most obvious source of experimental uncertainty is the limited precision of the balance. Other potential sources of experimental uncertainty are: the reaction might not be complete; if not enough time was allowed for total oxidation, less than complete oxidation of the magnesium might have, in part, reacted with nitrogen in the air (incorrect reaction); the magnesium oxide might have absorbed water from the air, and thus weigh "too much." Because the result obtained is close to the accepted value it is possible that some of these experimental uncertainties have fortuitously cancelled one another.

10 Answers to Definitions

a. The *atomic weight of an element* is the relative weight of one of its atoms compared to C-12 with a weight of 12.0000000. . ., hydrogen with a weight of 1.008, to oxygen with a weight of 16.00. Atomic weight is also the average weight of all the atoms of that element as they occur in nature.

b. The *units of atomic weight* are two-fold, with an identical numerical value. They are g/mole of atoms (or just g/mol) or amu/atom.

c. *Percentage discrepancy* between an accepted (literature) value and an experimental value is

$$\frac{\text{experimental result} - \text{accepted result}}{\text{accepted result}}$$