

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 5, 2017

1 Introduzione

Il connettore *Publish-Subscribe* presenta caratteristiche interessanti per sistemi caratterizzati da una elevata dinamicità, grazie all'elevato livello di disaccoppiamento che è 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 è caratterizzato dal cosiddetto *topic* che rappresenta il suo argomento d'appartenenza al quale un *consumatore* può sottoscrivere se interessato a tale classe di informazioni; il *produttore* a sua volta può pubblicare eventi associati ad un determinato *topic*. Nello specifico è possibile caratterizzare un determinato evento (dopo averlo associato ad un *topic*) in base al suo contenuto utilizzando la politica del tipo *content-based*. Tale modalità 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 entità 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 entità 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 è 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 modalità *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 entità:

- *Event-Service*, il quale si preoccupa di gestire gli eventi generati e di recapitare notifiche
- *Filter-Service*, il quale realizza la modalità 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 possibilità al client di potersi disattivare o spegnere immantinente 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, così 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. Tutto ciò viene attuato per promuovere un utilizzo offline dell'applicazione in caso di disconnessione, scenario possibile nell'ambito dei sistemi mobili.

Di seguito illustriamo l'architettura attraverso uno schema in Figure 1.

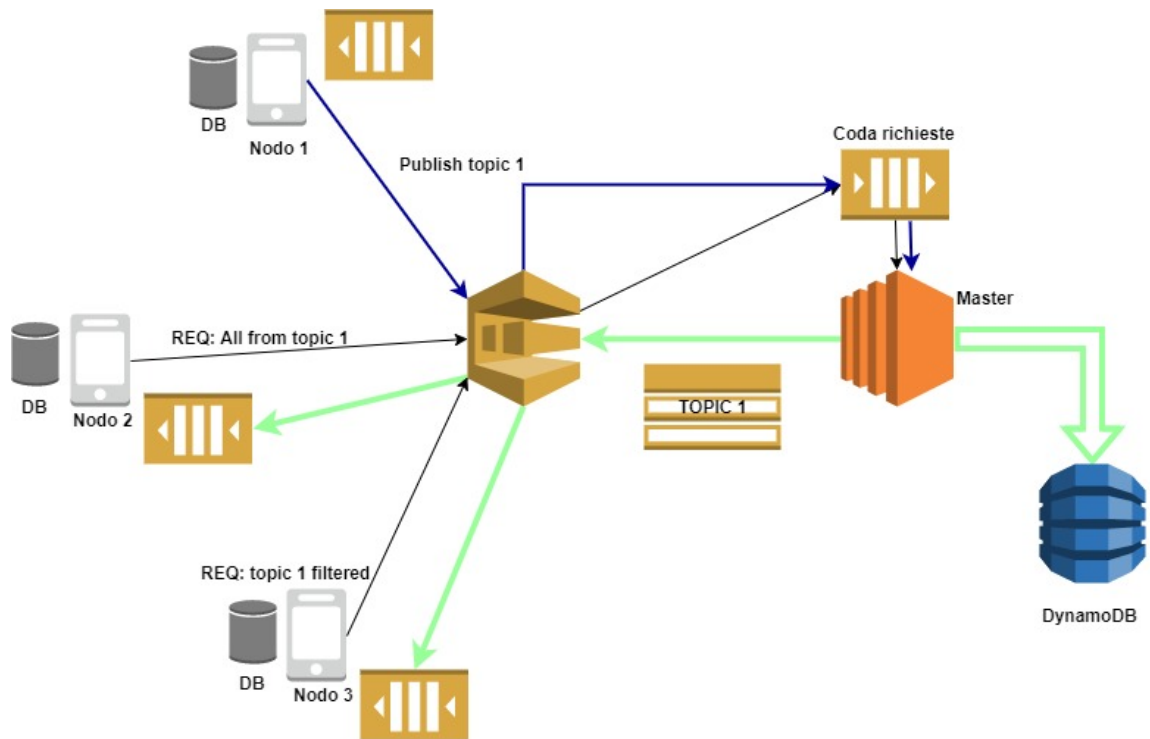


Figure 1: Architettura

3.1 Servizi Cloud

Come *Cloud Provider* è 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:** è 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 è possibile impostare un algoritmo di *scheduling* dei messaggi: nel nostro caso è stato utilizzato l'algoritmo *FIFO*
- **Amazon DynamoDB:** è un servizio di *datastorage* distribuito basato su un *database NoSQL* di tipo *key-value*. *DynamoDB* implementa un sis-

tema di tipo *AP*, disponibilità e tolleranza alle partizioni dei dati, secondo il *CAP Theorem*. L'alto grado di disponibilità 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 più letture che scritture.

3.2 Ambiente Client

Il nostro sistema è pensato per un utilizzo da parte di *client leggeri* con sistema operativo *Android*. *Android* è 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*, è 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*. È stato scelto come sistema operativo *Linux Ubuntu Server v.16.04 LTS*. La scelta di tale sistema operativo è da ricondurre alla semplicità di installazione di componenti aggiuntivi e alla sua versatilità.

3.4 Interazione dei componenti

Per promuovere la comunicazione tra i client ed il server, disaccoppiandoli, è 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 sottoscrivere in modalità *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, versatilità 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 modalità *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 deleteMessageBatch*, 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* può 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 disponibile un *endpoint* per rispondere a richieste *http* di tipo *get*, il quale invoca la funzione *scan_all_topics* di *dynamo_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.

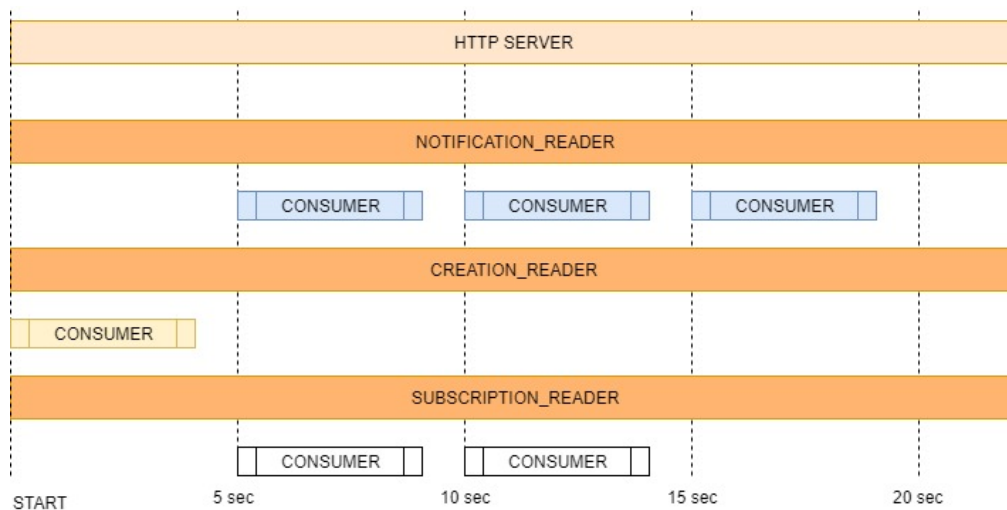


Figure 2: Processi

4.2 Client

5 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

6 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, $M_{\text{Mg}} = 16.00 \times \frac{1.31}{0.87} = 24.1 = 24 \text{ g mol}^{-1}$ (to two significant figures).

7 Results and Conclusions

The atomic weight of magnesium is concluded to be 24 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.

8 Discussion of Experimental Uncertainty

The accepted value (periodic table) is 24.3 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.

9 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

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