Università degli Studi di Roma Tor Vergata

Pub-Sub Message Queue with Go

Autori:

Luigi Corsi - Leonardo Petraglia

Sistemi Distribuiti e Cloud Computing
2019/2020

31 ottobre 2020

Indice

Capitolo 1

Introduzione

Lo scopo del progetto è stato quello di realizzare un sistema a code di messaggi basato su topic di tipo publish subscribe e in cui la consegna avvenisse considerando il contesto geografico dei due attori. All'interno del sistema sia il publisher sia il subscriber specificano un raggio a partire dalla loro posizione attuale in modo tale che la consegna del messaggio sia effettuata solamente quando le aree virtuali date dalle circonferenze si intersecano tra loro.
La nostra implementazione si è focalizzata nello sviluppare un'applicazione che potesse permettere agli esercenti di attività commerciali di pubblicare offerte a tempo riguardanti determinate categorie di prodotti e servizi (rappresentate dai topic) e ai sottoscrittori di tali categorie di ricevere i relativi messaggi.

Capitolo 2

Ambiente di Sviluppo

L'idea descritta è stata implementata in forma di Web Application tramite l'utilizzo dell'IDE Goland, piattaforma per lo sviluppo di applicazioni nel linguaggio di programmazione Go. La natura dell'applicazione ha previsto l'utilizzo di ulteriori linguaggi quali HTML, JavaScript e SQL.

Per quanto riguarda il lato Web dell'applicazione si è fatto uso di gin-gonic [1], un framework utile alla realizzazione di un middleware che, inserito all'interno dell'applicazione, si occupa della gestione le richieste HTTP in maniera efficiente e performante.

Capitolo 3

Architettura e Implementazione

L'architettura è di tipo client-server, pertanto, avendo a che fare con un modello publish-subscribe, c'è la necessità di avere un event broker posizionato sul server che si occupi di:

- Mantenere le sottoscrizioni degli utenti ai vari topic;
- Gestire la consegna dei messaggi in base alle sottoscrizioni e al contesto geografico.

Ogni utente registrato può agire all'interno del sistema secondo entrambi i ruoli, potendo pubblicare messaggi in qualsiasi categoria oppure riceverne in base alle proprie sottoscrizioni.

3.1 Coda di Messaggi

La coda è stata implementata in memoria tramite una struttura denominata EventBroker, costituita da:

- Una mappa chiave-valore che associa ad ogni topic i relativi messaggi;
- Una mappa chiave-valore che associa ad ogni utente le proprie sottoscrizioni;
- Un mutex per gestire l'accesso in concorrenza alla struttura.

Sono stati messi a disposizione i servizi di base di una coda publish-subscribe, quali pubblicazione e consegna di un messaggio, sottoscrizione ad un topic ed eliminazione da esso. Ciascuna di queste operazioni viene attivata tramite delle richieste AJAX HTTP di tipo POST:

- **Pubblicazione**: nella pagina dedicata alla pubblicazione dei messaggi, una volta compilati i campi che la compongono (tra cui il raggio d'interesse), il click sul bottone *Send Message* attiva una funzione JavaScript per l'ottenimento della posizione da parte del browser a partire dal quale viene creata l'area d'interesse del messaggio. In seguito, a seconda della semantica di comunicazione selezionata, viene eseguita un'ulteriore funzione JavaScript che tramite una chiamata *AJAX* invia la richiesta di inserimento del messaggio al server.

  A partire dall'oggetto JSON ricevuto viene effettuato il parsing all'interno della struttura rappresentante il messaggio composta dai seguenti attributi: Message, Title, Topic, RequestID, Radius, Lifetime, Latitude, Longitude, InsertionTime, ExpirationTime. Successivamente una *go routine* prende il lock sulla coda ed effettua l'inserimento del messaggio;

- **Gestione Sottoscrizioni**: nella pagina relativa alla gestione delle sottoscrizioni cliccando sul check button associato a ciascun topic viene attivata una funzione JavaScript che tramite una chiamata *AJAX* effettua la sottoscrizione oppure la cancellazione da esso. Anche in questo caso l'operazione è preceduta dall'acquisizione del lock sulla coda;

- **Recupero Messaggi**: nella sezione relativa ai messaggi in seguito alla selezione del raggio di interesse, attraverso uno slider viene attivata una funzione JavaScript che dapprima acquisisce la posizione dell'utente e successivamente tramite una chiamata *AJAX* invia una richiesta di recupero dei messaggi al server. Per la consegna dei messaggi, si è scelto quindi un approccio secondo il quale essi vengono inviati solamente nel momento in cui l'utente ne fa domanda e se le aree di interesse associate al messaggio e all'utente si intersecano. Così facendo si evita che l'event broker debba inviare un messaggio in broadcast al momento della sua pubblicazione diminuendone il carico. Al contrario delle operazioni precedenti, avendo a che fare con accessi in sola lettura, non si necessita dell'acquisizione del lock.

3.2 Semantiche di Comunicazione

Sono state realizzate tre diversi tipi di semantica di comunicazione per l'inserimento di un messaggio in coda, tutte attraverso l'uso di chiamate AJAX con relativi parametri di configurazione e funzioni di callback:

- **At-least-once**: secondo questo modello i messaggi vengono inviati ripetutamente al server fino a quando non si riceve un ACK entro un intervallo di tempo configurabile. L'ACK indica il corretto inserimento all'interno della coda.

  Oltre allo scadere del timeout, l'invio del messaggio viene rieffettuato alla ricezione di un NACK inviato dal server. Tale NACK sta proprio ad indicare il mancato inserimento del messaggio all'interno della coda;

- **At-most-once**: questo tipo di semantica prevede che i messaggi, se ricevuti, vengono inseriti nella coda al più una volta. Oltre al timeout descritto in precedenza è presente un ulteriore parametro configurabile che rappresenta il numero massimo di tentativi di invio a valle di possibili errori. Se si supera questo numero il messaggio non viene più inserito.

  Per far sì che il messaggio venga inserito al più una volta è stato realizzato un meccanismo che associa ad ogni richiesta un ID univoco formato dalla combinazione dell'email del publisher e del timestamp del momento di invio. Le richieste vengono inserite all'interno di una mappa chiave-valore. In questo modo il server è in grado di riconoscere le richieste duplicate così da inserire il messaggio nella coda al massimo una volta.

- **Exactly-once**: in questo caso si vuole avere accordo completo tra client e server per far sì che il messaggio venga inserito nella coda esattamente una volta. Ciò comporta che non ci sia una durata massima della loro interazione, perciò il messaggio viene ritrasmesso fintanto che entrambi non abbiano ricevuto un ACK dall'altro.

  Per realizzare questo tipo di semantica anche in questo caso si è fatto uso dello stesso meccanismo di filtraggio delle richieste duplicate descritto nella semantica precedente, rimuovendo tuttavia il numero massimo di ritrasmissioni.

L'accordo completo tra i due partecipanti prevede inoltre che l'ACK inviato dal client al server venga percepito da quest'ultimo come via libera per eliminare la richiesta dalla mappa.

Per quanto riguarda la semantica at-most-once è stato inoltre necessario implementare un meccanismo di eliminazione delle richieste, questo per evitare che la memoria sia inutilmente occupata da richieste già servite oppure che hanno superato il loro tempo di vita all'interno della mappa. Tale operazione è svolta da una go routine dedicata che viene avviata allo start-up dell'applicazione e che periodicamente effettua il compito suddetto. Il periodo di permanenza nella mappa e quello di esecuzione della go routine sono parametri configurabili.

3.3 Gestione della Sessione

La gestione della sessione di un utente all'interno dell'applicazione è stata realizzata tramite l'utilizzo di JSON Web Token (JWT) [2], ovvero un modo compatto e autonomo per trasmettere in modo sicuro le informazioni tra le parti come oggetto JSON. I JWT sono popolari perché:

- Una volta creata, la firma di un JWT non viene mai decodificata, garantendo la sicurezza del token;
- Possono essere impostati come non validi dopo un certo periodo di tempo. Ciò elimina il rischio di danni provocati da eventuali attacchi nel caso in cui il token venga dirottato.

Per accedere alla piattaforma ogni utente deve necessariamente effettuare una registrazione inserendo la propria email e una password. Una volta registratosi, ciascun utente che effettua il login riceve un token univoco [3] che viene memorizzato lato client nel local storage del browser, lato server all'interno di un database NoSQL Redis.

Un token non è più valido dopo un determinato intervallo di tempo configurabile o a seguito del logout da parte dell'utente. Il logout prevede l'eliminazione del token sia dallo storage del browser sia dal database Redis che ha quindi il compito di black list. Quando un utente richiede un'operazione che necessita di autorizzazione, tramite gin-gonic viene verificata la validità e la correttezza del token, in particolare se il token non è presente in entrambi gli storage l'operazione viene contrassegnata come unauthorized.

3.4 Persistenza

Per questioni di performance l'applicazione è stata pensata per agire principalmente in memoria, tuttavia è stato realizzato un livello di storage persistente PostregSQL per la gestione delle credenziali e la memorizzazione dei topic.
Come descritto in precedenza i messaggi e le sottoscrizioni vengono gestiti in memoria dall'event broker. Essendo però la memoria volatile in caso di crash dell'applicazione tutti i dati in essa contenuti vengono conseguentemente persi. Per ovviare a ciò è stato realizzato un meccanismo che può essere abilitato prima dell'avvio dell'applicazione e che permette il mantenimento dei messaggi e delle sottoscrizioni anche sul database. In questo modo, in caso di fallimento dell'applicazione, al nuovo riavvio i messaggi e le sottoscrizioni verranno recuperati dal database e inseriti nella relativa struttura in memoria.

Capitolo 4

Installazione - Docker Compose

In un contesto di installazione locale i servizi di storage PostgreSQL e Redis sono stati implementati tramite Docker Compose, uno strumento per la definizione e l'esecuzione di applicazioni multi contenitore. A ciascuno di essi è stato quindi riservato un container la cui definizione è specificata all'interno di un file di configurazione chiamato *docker-compose.yml*. In questo modo con l'esecuzione del comando *docker compose up* entrambi i servizi vengono eseguiti all'interno di una stessa rete alla quale l'applicazione può accedere attraverso l'utilizzo di IP statici.

Docker rende il processo di deployment particolarmente semplice e dinamico non dovendosi preoccupare della configurazione di ogni singolo servizio. Per avviare l'applicazione è quindi sufficiente accedere alla directory */script/docker* ed eseguire il comando *sh start.sh*; in successione verranno eseguiti i seguenti comandi:

- **docker-compose up -d**: inizializzazione dei container in background;
- **docker exec -it docker_db_1 psql -U postgres -c "create database sdcc"**: creazione del database con nome "sdcc" sul container dedicato a PostgreSQL;
- **go run initDB.go**: esecuzione di uno script per l'inizializzazione del database appena creato;
- **cd ../../src**;
- **go run *.go**: avvio dell'applicazione.

Infine per utilizzare l'applicazione è sufficiente accedere all'indirizzo *http://localhost:8080*.

Capitolo 5

REST API

L'applicazione è stata progettata in modo tale da essere eseguita anche da riga di comando e da ambienti esterni realizzando una REST API le cui richieste HTTP comunicano con il server attraverso la codifica di messaggi in formato JSON.

- **Registrazione**:
  curl -i -d '{"Email": "example@example.com", "Password": "examplePassword"}' -X POST http://localhost:8080/registration
- **Login**:
  curl -i -d '{"Email": "example@example.com", "Password": "examplePassword"}' -X POST http://localhost:8080/login
  Se il login va a buon fine viene restituito il token necessario per eseguire le successive operazioni.
- **Logout**:
  curl -i -b "access_token=exampleToken" -X POST http://localhost:8080/logout
- **Lista Topic**:
  curl -i -b "access_token=exampleToken" -X POST http://localhost:8080/subscriptionPage
  Viene ritornata la lista di tutti i topic in formato JSON dove l'iscrizione è contrassegnata dall'attributo booleano Flag.
- **Modifica Iscrizione**:
  curl -i -d '{"topic": "exampleTopic"}' -b "access_token=exampleToken" -X POST http://localhost:8080/editSubscription
- **Messaggi**:
  curl -i -d '{"Latitude": myLatitude, "Longitude": myLongitude, "Radius": "exampleRadius"}' -b "access_token=exampleToken" -X POST http://localhost:8080/notifications
- **Pubblica**:
  curl -i -d '{"Message": "myMessage", "Topic": "exampleTopic", "Title": "exampleTopic", "Radius": "exampleRadius", "LifeTime": "exampleLifeTime", "Latitude": myLatitude, "Longitude": myLongitude}' -b "access_token=exampleToken" -X POST http://localhost:8080/publish

Capitolo 6

Deployment

Per il deployment ed il testing dell'applicazione sul cloud ci si è avvalsi di alcuni servizi messi a disposizione da AWS Educate, quali:

- **Amazon EC2 (Amazon Elastic Computing)**, un servizio Web per l'elaborazione sicura e scalabile nel cloud;
- **Amazon RDS (Amazon Relational Database Service)**, un servizio che semplifica l'impostazione, il funzionamento e il dimensionamento di database relazionali nel cloud.

La Web Application è stata inizialmente deployata su un'istanza EC2 *t2.micro* opportunamente configurata in termini di comunicazione in ingresso e in uscita. Nel dettaglio sono state impostate le seguenti regole:

- **In entrata** - SSH, TCP, Porta: 22, Origine: Ovunque.
  Tale regola permette di connettersi da remoto secondo il protocollo SSH;
- **In entrata** - Custom TCP, Porta: 8080, Origine: Ovunque.
  Tale regola permette di eseguire i servizi esposti dall'applicazione;
- **In uscita** - Tutto il traffico.

Un'altra istanza EC2 è stata utilizzata per la realizzazione del meccanismo di blacklist implementato con Redis e descritto in precedenza. Anche in questo caso si è fatto uso di un'istanza EC2 *t2.micro* opportunamente configurata in accordo alle seguenti regole:

- **In entrata** - SSH, TCP, Porta: 22, Origine: Ovunque.
  Tale regola permette di connettersi da remoto secondo il protocollo SSH;
- **In entrata** - Custom TCP, Porta: 6379, Origine: Ovunque.
  Tale regola permette la comunicazione con il server Redis;
- **In uscita** - Tutto il traffico

RDS è stato configurato per ospitare il database PostreSQL di cui si è discusso precedentemente. Similmente ai casi precedenti la tipologia d'istanza scelta è la *db.t2.micro*.

6.1 Load Balancer - Auto Scaling

Al fine di migliorare la robustezza e la scalabilità dell'applicazione è stato realizzato un meccanismo di Auto Scaling e di Load Balancing di tipo Classic per assicurare un corretto bilanciamento del carico tra le varie istanze EC2. Dapprima si è creata una AMI dell'istanza sulla quale è stata deployata l'applicazione, essa infatti è necessaria per la creazione delle istanze ausiliarie sul quale riversare il carico. Successivamente è stato configurato un VPC (Virtual Private Cloud), una rete virtuale sulla quale avviare risorse AWS come i servizi suddetti. Dopodiché la creazione di un Gruppo di Avvio con associata l'AMI dell'applicazione è stata necessaria per la configurazione del sistema di Auto-Scaling con le seguenti caratteristiche:

- Numero di istanze minime: 1;
- Numero di istanze desiderate: 2;
- Numero di istanze massime: 3;
- Soglia per l'attivazione di un'ulteriore istanza: Utilizzazione CPU $> 50\%$

Con tale configurazione la gestione dei fallimenti delle istanze avviene in maniera dinamica e scalabile. Inoltre, sono state apportate alcune modifiche per quanto riguarda le regole di comunicazione delle istanze EC2 per accettare il flusso di dati dal load balancer. Infine, per mantenere la consistenza della sessione di un utente all'interno del sistema è stato fatto sì che all'istanza dedicata a Redis fosse assegnato un IP statico e che di conseguenza tutte le istanze dell'applicazione si riferiscano univocamente ad essa. Così facendo il fallimento di una istanza non provoca l'espulsione di un utente dal sistema.

Capitolo 7

Testing

Vengono riportati di seguito i risultati di due tipologie di test effettuati con il tool di benchmark fornito da Apache *ab*, la prima in locale[1] e in contesto docker, la seconda in versione deployata sui servizi AWS.

Il primo test ha avuto come obiettivo misurare la latenza introdotta dal meccanismo di persistenza attivabile come descritto in precedenza. Le metriche prese in considerazione sono state la durata del test e il transfer rate globale. In particolare sono stati svolti 5 run per ognuna delle seguenti configurazioni riportandone i risultati medi: 10'000 richieste di pubblicazione di un messaggio eseguite rispettivamente con un tasso di concorrenza di 100/500/1000.

Tabella 7.1: 10'000 requests - localhost:8080/publish

	Test Time (s)		**Transfer Rate (Kb/s)**	
Conc. Rate	**No DB**	**DB**	**No DB**	**DB**
100	1.4822	4.8416	4475.538	1389.062
500	1.8028	5.9772	3705.41	1165.65
1000	2.4912	7.0862	2779.478	1073.536

Come si evince dalla tabella, a parità di numero di richieste concorrenti, l'attivazione della persistenza introduce un delay non trascurabile; inoltre, all'aumentare del tasso di concorrenza, si ha comunque un andamento crescente ma più contenuto. In termini di transfer rate si ha allo stesso modo un peggioramento delle performance contestuale all'attivazione della persistenza e riscontrabile nella diminuzione del throughput globale.

Per quanto riguarda il secondo test si è voluto esaminare l'andamento delle prestazioni in funzione dell'attivazione o meno dei meccanismi di Load Balancing e Auto Scaling. Le metriche usate sono le stesse citate in precedenza e anche in questo caso sono state svolte 5 esecuzioni per ciascun tasso di concorrenza riportandone i valori medi.

[1] Configurazione usata: CPU: i7 8700, RAM: 32GB DDR4

Tabella 7.2: 1'000 requests No DB - AWS/publish

	Test Time (s)		**Transfer Rate (Kb/s)**	
Conc. Rate	**LB**	**No LB**	**LB**	**No LB**
100	3.5848	3.9066	1144.684	201.868
500	6.5548	7.7234	694.896	121.728
1000	8.7862	9.7374	413.592	93.568

Come preventivato si può notare come il miglioramento delle prestazioni introdotto dall'impiego di tali meccanismi sia proporzionale all'aumento del tasso di concorrenza.

Capitolo 8

Limitazioni Riscontrate

Una limitazione riscontrata risulta essere l'implementazione delle semantiche di comunicazione. Infatti avendo realizzato tali meccanismi con l'ausilio di funzioni JavaScript essi non vengono considerati quando si fa uso della REST API. Il risultato delle operazioni effettuate da riga comando rimane comunque facilmente riscontrabile nell'header di risposta delle richieste.

Bibliografia

[1] https://github.com/gin-gonic/gin

[2] https://github.com/dgrijalva/jwt-go

[3] https://github.com/twinj/uuid