marco.canducci@studio.unibo
20/7/2018

Assignment #4
Chat distribuita e reattiva

Gruppo di lavoro

Anche questo ultimo assignment, così come il terzo, è stato svolto dal sottoscritto Canducci Marco (matr. 833069) insieme al compagno Schiavi Daniele (matr. 843692). Per quanto riguarda l'analisi del problema e tutta la parte preparatoria ed implementativa iniziale non c'è stata alcuna separazione dei compiti tra i due componenti del gruppo: si è proseguito nello sviluppo ragionando insieme su ogni dubbio, idea, scelta tecnologica e di design considerata rilevante. Solo a progetto quasi concluso si è sperimentata invece la comodità della programmazione a servizi, poiché abbiamo scelto di concludere in maniera totalmente indipendente alcuni dettagli implementativi, terminando in autonomia un servizio a testa (in particolare, il servizio relativo al database e quello atto ad implementare la sezione critica).

Premessa

Avendo già risolto in modo totalmente decentralizzato il problema nel terzo assignment, e volendo focalizzare lo sviluppo del quarto su tecnologie cloud, si è deciso di utilizzare in questo caso un algoritmo centralizzato. Così facendo viene semplificata la logica della soluzione e, allo stesso tempo, la si sposta quasi completamente su server/cloud piuttosto che sui singoli client (dando un senso maggiore alla sperimentazione su Heroku).

Si vuole inoltre sottolineare di come, al contrario della soluzione proposta per l'assignment precedente, sia ora presente un controllo al momento del login, col quale si impedisce a due omonimi di entrare in una stessa chat.

Analisi del problema

Idee e motivazioni

Essendo richiesto un *sistema reattivo*, è necessario porre particolare attenzione su una serie di requisiti non funzionali come la responsività, la resilienza e l'elasticità. Per fare ciò, si è deciso prima di tutto di organizzare il sistema in servizi (***Simple Component pattern***): moduli software indipendenti che incapsulano una determinata funzionalità e che comunicano tra loro tramite scambio di messaggi asincrono.

- Per aumentare la **resilienza** del sistema alle disconnessioni ed ai guasti si è deciso di usare un framework a messaggi come quello di RabbitMQ, dove viene garantita la persistenza in code nel caso non fosse disponibile il destinatario al momento dell'invio di un messaggio. Inoltre, utilizzando un MOM si ha la certezza che ad ogni messaggio inviato ne arriverà a destinazione "**almeno uno**", in contrapposizione a quanto accadeva invece in con il sistema ad attori del terzo assignment, dove la garanzia era quella di consegnare "al più un messaggio" per ogni invio.

- La separazione in diversi servizi per diverse funzionalità ha permesso di applicare, se necessario, una **scalabilità orizzontale diversa per funzionalità diverse**. Ad esempio, all'aumentare del numero di utenti, il servizio per effettuare il login in una chatroom avrà sicuramente meno necessità di scalare rispetto al servizio che si occupa dello smistamento di tutti i messaggi. Ma non solo, ogni servizio potrà così avere, se necessario, delle sue precise politiche di *fault tolerance*, *disponibilità*, ecc.
  Da notare come **nulla di tutto questo sarebbe stato possibile con un'applicazione monolitica**.

- Si è posta una particolare attenzione ad utilizzare **servizi completamente stateless** per ottenere forte **scalabilità orizzontale** e **resilienza**. La piattaforma Heroku è infatti già predisposta in automatico per il supporto a politiche di resistenza ai guasti; si riporta di seguito uno stralcio di testo tratto dalla sua guida ufficiale riguardo al perché i servizi dovrebbero essere, quando possibile, stateless:

  "Processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database. Apps should never assume that anything cached in memory or on disk will be available on a future request or job. With many processes running in a distributed environment, chances are high that a future request will be served by a different process with a different physical location which won't have access to the original memory or filespace. Even when running only one process, a restart (triggered by code deploy, config change, or the execution environment relocating the process to a different physical location) will usually wipe out all local (e.g., memory and filesystem) state. This process model truly shines when it comes time to scale out."

Ordinamento totale dei messaggi

Per garantire anche in questo assignment un ordinamento totale dei messaggi si è utilizzato un sistema di **ticket**: ad ogni chatroom sul database è associato un ticket che rappresenta il numero dell'ultimo messaggio inviato in quella chatroom. Il Messaging Service, prima di ridirezionare un nuovo messaggio a tutti i componenti di una chat, richiederà al DB Service di **incrementare il ticket e restituirgli il suo valore in modo atomico**, così da poterlo fornire come allegato al messaggio originale.
Sarà poi compito dei client visualizzare correttamente i messaggi, leggendo il ticket nell'header ed eliminando eventuali messaggi obsoleti (perché mandati più di una volta dal framework MOM), oppure salvando momentaneamente in un buffer i messaggi con un ticket troppo elevato.

Interfacce dei Servizi

Il sistema è composto da quattro moduli software indipendenti, di cui è stato fatto il deploy separato su piattaforma cloud Heroku. È possibile interagire con i servizi mandando opportuni messaggi nelle code messe da loro a disposizione: sarà compito del mittente includere nel messaggio anche dove si vorrà ricevere la risposta (***request-response pattern***). Durante la fase di design dell'applicazione, si è cercato di ragionare sin da subito sulle funzionalità (API) che ogni servizio avrebbe dovuto fornire verso l'esterno. Si è quindi arrivati a delineare già inizialmente il profilo di tre servizi distinti (più un quarto per la parte facoltativa), cercando di incapsulare in ognuno di essi responsabilità ben precise.

1. **Logger Service**
   Si occupa di gestire l'ingresso e l'uscita degli utenti dalle diverse chatroom.
   Funzionalità fornite:
   a. *Richiesta di login* (in una chatroom già esistente)
   b. *Effettua logout* (dalla chatroom in cui ci si trova)

2. **Messaging Service**
   Si occupa del vero e proprio *dispatching* dei messaggi inviati tra gli utenti delle diverse chatroom.
   È coordinatore e cuore del sistema: in base all'ordine di arrivo dei messaggi di ogni chatroom, li associa ad un ticket crescente e li inoltra a tutti i partecipanti di tale chatroom.
   Funzionalità fornite:
   a. *Inoltro nuovo messaggio* (a tutti i partecipanti di una chatroom)

3. **DB Service**
   La realizzazione o meno di un servizio apposito per la gestione del DataBase è stata oggetto di dibattito durante le fasi di progettazione: se da un lato è infatti vero che la sua presenza potrebbe generare dipendenze orizzontali, d'altro canto si è dato un maggior peso in questo caso alla comodità che comporta il poter incapsulare in esso tutta la logica riguardante il database. Così facendo, nel caso in futuro si volessero modificare tecnologie o strategie di storage, si potrebbe intervenire su questo unico servizio, lasciando completamente invariato tutto il resto del sistema.
   Funzionalità fornite:
   a. *Richiesta elenco chatroom*
   b. *Richiesta elenco partecipanti* (di una chatroom già esistente)
   c. *Creazione nuova chatroom*
   d. *Inserimento di un nuovo partecipante* (in una chatroom già esistente)
   e. *Rimozione di un partecipante* (dalla chatroom dove si trova)
   f. *Incremento e richiesta numero di ticket* (di una chatroom già esistente)
   g. *Inserimento partecipante in sezione critica*
   h. *Rimozione partecipante da sezione critica*

4. **CS Service**
   Si occupa di gestire l'ingresso e l'uscita degli utenti dalla sezione critica.
   Funzionalità fornite:
   a. *Richiesta di ingresso in CS* (per una chatroom già esistente)
   b. *Effettua uscita dalla CS*

Protocolli di comunicazione per ogni Use Case

- **Visualizzazione dell'elenco delle chatroom sul client**
  - L'utente avvia il client per la prima volta, oppure preme il pulsante di *refresh* sulla GUI.
  - Il client invia un messaggio al DB_Service nella sua coda di ***Richiesta elenco chatroom***.
  - Il DB_Service ***risponde*** al mittente, fornendogli l'elenco delle stanze.
  - Il client visualizza quanto appena ricevuto sulla GUI.

- **Creazione di una nuova chatroom**
  - L'utente sceglie un nome per la stanza e preme il pulsante per la creazione di una nuova chatroom.
  - Client invia un messaggio contenente il nome della stanza al DB_Service nella sua coda di ***Creazione nuova chatroom***.
  - Il DB_Service **risponde** al mittente, impostando uno *STATE_HEADER* positivo (*OK_STATE*) nel caso l'inserimento nel Database fosse andato a buon fine, oppure negativo (*ERROR_STATE*) nel caso fosse fallito (ad esempio a causa della previa esistenza di una chatroom omonima).

- In caso di risposta positiva, il client aggiunge sulla GUI il nome della nuova chatroom tra quelle disponibili, altrimenti visualizza un popup di errore.

- **Login ad una chatroom**
  - L'utente sceglie un nickname, seleziona una tra le chatroom disponibili, e preme il pulsante di login.
  - Il client invia un messaggio contenente nickname e nome della chatroom al Logger_Service nella sua coda di ***Richiesta di login***.
  - Il Logger_Service inoltra il messaggio al DB_Service, nella sua coda per l'***Inserimento di un nuovo partecipante***.
  - Il DB_Service **risponde** al Logger_Service, impostando uno *STATE_HEADER* positivo (*OK_STATE*) nel caso l'inserimento nel Database fosse andato a buon fine, oppure negativo (*ERROR_STATE*) nel caso fosse fallito (ad esempio a causa della presenza previa dello stesso username nella stanza in cui si sta cercando di entrare).
  - Il Logger_Service **risponde** al client inoltrandogli l'esito dell'inserimento da parte del DB_Service. In caso di esito positivo il client modificherà il proprio stato logico da "logged out" a "logged in" e preparerà la GUI, visualizzando il nome della stanza nella quale si è entrati e la chat vera e propria su cui verranno pubblicati i messaggi da tutti i partecipanti. In caso di esito negativo verrà visualizzato il classico popup di errore.
  - In caso di esito positivo dell'inserimento nel Database, il Logger_Service si preoccupa anche di allegare un *MESSAGE_TYPE_HEADER* di tipo *LOGIN_MESSAGE* ed inoltrare il messaggio al Messaging_Service, nella sua coda di ***Inoltro nuovo messaggio*** con lo scopo di avvisare tutti i partecipanti della chat che un nuovo utente ha effettuato l'ingresso.
  - Per dettagli su come viene inoltrato questo messaggio dal Messaging_Service a tutti i partecipanti della chat si veda di seguito lo Use Case "*Invio di un nuovo messaggio a tutti i partecipanti di una chatroom*".
  - Infine, su tutti i client della chatroom verrà visualizzato nella GUI un messaggio speciale (che si è deciso di visualizzare anche se qualcuno dovesse essere in CS) che li informa dell'ingresso del nuovo client.

- **Logout da una chatroom**
  - L'utente, quando loggato in una chatroom, preme il pulsante di logout.
  - Il client si disconnette ed invia un messaggio contenente nickname e nome della chatroom al Logger_Service nella sua coda ***Effettua logout***.
  - Il Logger_Service inoltra il messaggio al DB_Service, nella coda per la ***Rimozione di un partecipante***.
  - Il DB_Service **risponde** al Logger_Service, confermando l'avvenuta rimozione.
  - Il Logger_Service allega un *MESSAGE_TYPE_HEADER* di tipo *LOGOUT_MESSAGE* ed inoltra il messaggio al Messaging_Service, nella sua coda di ***Inoltro nuovo messaggio*** con lo scopo di avvisare tutti i partecipanti della chat che l'utente interessato ha effettuato il logout.
  - Per dettagli su come viene inoltrato questo messaggio dal Messaging_Service a tutti i partecipanti della chat si veda di seguito lo Use Case "*Invio di un nuovo messaggio a tutti i partecipanti di una chatroom*".
  - Infine, su tutti i client della chatroom verrà visualizzato nella GUI un messaggio speciale (che si è deciso di visualizzare anche se qualcuno dovesse essere in CS) che li informa dell'uscita del client.

- **Invio di un nuovo messaggio a tutti i partecipanti di una chatroom**
  - L'utente, quando loggato, digita un messaggio di testo per gli altri partecipanti e preme il pulsante *send*.
  - Il client invia un messaggio contenente il proprio nickname, il nome della chatroom a cui è loggato, ed il contenuto del testo al Messaging_Service nella sua coda di ***Inoltro nuovo messaggio***.
  - Il Messaging_Service invia un messaggio contenente il nome dell'utente e della chatroom al DB_Service, nella sua coda ***Incremento e richiesta numero di Ticket***.
  - Il DB_Service incrementa e legge il nuovo valore del ticket, fallendo nel caso ci fosse qualcuno di diverso dal mittente in sezione critica (NB: questa azione è realizzata **atomicamente**).
  - Nel caso l'operazione precedente sul DB fosse andata a buon fine, il DB_Service **risponde** al Messaging_Service allegando il valore aggiornato del Ticket al messaggio di richiesta originale.
  - Ottenuto il valore del Ticket, il Messaging_Service manda un altro messaggio al DB_Service, richiedendogli la lista dei partecipanti della chatroom sulla coda ***Richiesta elenco partecipanti***.
  - Il DB_Service legge dal database l'elenco dei partecipanti, li allega agli header del messaggio originale, e lo invia come **risposta** al Messaging_Service.
  - Arrivati a questo punto, il Messaging_Service scorre la lista dei partecipanti ed invia ad ognuno di loro il messaggio di testo, scegliendo come univoco della coda <ChatterName+chatRoomName> che identifica in modo univoco ogni cliente.
  - Il client riceve il messaggio di testo e ne controlla il Ticket confrontandolo con quello locale:
    - Ticket messaggio ≤ Ticket locale → scartare messaggio
    - Ticket messaggio = Ticket locale + 1 → visualizzare il messaggio sulla GUI
    - Ticket messaggio > Ticket locale + 1 → mettere il messaggio in un buffer locale

- **Ingresso in sezione critica**
  - Quando loggato, l'utente digita il codice per l'ingresso in Sezione Critica e preme il pulsante *send*.
  - Il client riconosce il codice inserito ed invia un messaggio contenente il nome dell'utente e della chatroom al CS_Service, nella sua coda ***Richiesta di ingresso in CS***.
  - Il CS_Service invia a due messaggi:
    - Il primo al DB_Service, nella sua coda di ***Inserimento partecipante in sezione critica***.
    - Il secondo a se stesso, nella sua coda senza consumatori per la gestione del Timeout della CS.
      (Per i dettagli, si veda il paragrafo: *Gestione del Timeout in Sezione Critica*)
  - Nel caso il DB_Service riesca con successo ad inserire l'utente in CS, allora ***risponderà*** al CS_Service.
  - All'eventuale ricezione della conferma, il CS_Service allega un *MESSAGE_TYPE_HEADER* di tipo *CS_ENTER_MESSAGE* ed inoltra il messaggio al Messaging_Service, nella sua coda di ***Inoltro nuovo messaggio*** con lo scopo di avvisare tutti i partecipanti della chat che l'utente interessato ha effettuato l'ingresso in Sezione Critica.
  - Per dettagli su come viene inoltrato questo messaggio dal Messaging_Service a tutti i partecipanti della chat si veda di seguito lo Use Case "*Invio di un nuovo messaggio a tutti i partecipanti di una chatroom*".
  - Infine, su tutti i client della chatroom verrà visualizzato nella GUI un messaggio speciale che li informa di chi è entrato in CS.

- **<u>Uscita dalla sezione critica</u>**
  - Quando loggato, l'utente digita il codice per l'uscita dalla Sezione Critica e preme il pulsante *send*.
  - Il client riconosce il codice inserito ed invia un messaggio contenente il nome dell'utente e della chatroom al CS_Service, nella sua coda ***Effettua uscita dalla CS***.
  - Il CS_Service invia un messaggio al DB_Service, nella sua coda di ***Rimozione partecipante da sezione critica***.
  - Nel caso il DB_Service riesca con successo a rimuovere l'utente dalla Sezione Critica, allora ***risponderà*** al CS_Service con un messaggio di conferma.
  - Alla ricezione di tale messaggio, il CS_Service allega un *MESSAGE_TYPE_HEADER* di tipo *CS_EXIT_MESSAGE* ed inoltra il messaggio al Messaging_Service, nella sua coda di ***Inoltro nuovo messaggio*** con lo scopo di avvisare tutti i partecipanti della chat che l'utente interessato ha effettuato l'uscita dalla Sezione Critica.
  - Per dettagli su come viene inoltrato questo messaggio dal Messaging_Service a tutti i partecipanti della chat si veda di seguito lo Use Case "*Invio di un nuovo messaggio a tutti i partecipanti di una chatroom*".
  - Infine, su tutti i client della chatroom verrà visualizzato nella GUI un messaggio speciale che li informa dell'avvenuta uscita dalla CS.

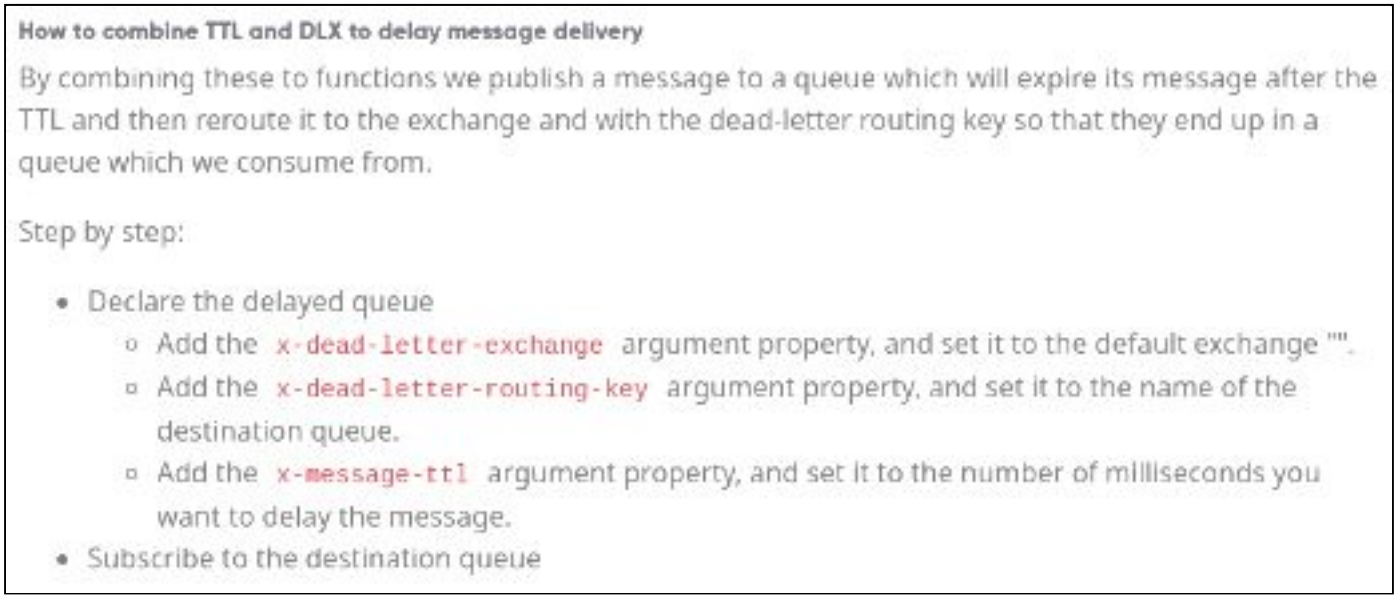
Dettagli implementativi

Tecnologie utilizzate

Sia il client che i servizi sono stati implementati in Java, e messi in comunicazione tramite il middleware messo a disposizione da **RabbitMQ**. La GUI del client è stata realizzata sfruttando la libreria di **JavaFX**, con la quale si ha avuto modo di sperimentare anche nei precedenti assignment. Per il deployment dei servizi su cloud ci si è affidati ad **Heroku**, utilizzando inoltre due plugin per lo scambio di messaggi e l'utilizzo di Database "as a Service": ***CloudAMQP*** ed ***Heroku Postgres*** rispettivamente.

Gestione del Timeout in Sezione Critica

Purtroppo, il plugin di cloudAMQP su Heroku per creare un exchange che ritardasse l'invio di messaggi era disponibile solo per i piani a pagamento. Si è quindi utilizzato una sorta di *hack*: esso era il metodo consigliato per fare lo scheduling di messaggi prima che venisse sviluppato il plugin apposito.



How to combine TTL and DLX to delay message delivery

By combining these to functions we publish a message to a queue which will expire its message after the TTL and then reroute it to the exchange and with the dead-letter routing key so that they end up in a queue which we consume from.

Step by step:

- Declare the delayed queue
  - Add the `x-dead-letter-exchange` argument property, and set it to the default exchange "".
  - Add the `x-dead-letter-routing-key` argument property, and set it to the name of the destination queue.
  - Add the `x-message-ttl` argument property, and set it to the number of milliseconds you want to delay the message.
- Subscribe to the destination queue

L'idea principale è quella di utilizzare una feature disponibile dalla versione 2.8.0 di CloudAMQP: ***dead letter exchanges***. In questo modo è possibile specificare un *exchange* sul quale pubblicare tutti i messaggi rigettati o "scaduti" su una certa coda. È possibile far "scadere" appositamente tutti i messaggi di una coda senza consumatori dopo N secondi impostando ad N il suo ***TimeToLive***. Allo scadere degli N secondi, i messaggi saranno automaticamente inoltrati al ***DeadLetterExchange*** in una nuova coda specificata dal parametro **DeadLetterRoutingKey**.

Possibili miglioramenti

Aggiornamento in tempo reale dell'elenco di chatroom

Al posto del tasto *refresh chatrooms* si potrebbe fare un servizio a parte che avvisa i client ogni volta che viene creata una nuova chatroom (evitando così di dover fare polling per verificare l'eventuale creazione di nuove chatroom).

API RESTful e Proxy Service

Il client della nostra applicazione distribuita, per come è stata realizzata, deve utilizzare obbligatoriamente determinate tecnologie (nello specifico, RabbitMQ) e conoscere esattamente quali messaggi inviare a quali servizi.

Per ovviare in un colpo solo ad entrambe queste limitazioni, e migliorare di conseguenza l'usabilità e l'interoperabilità dell'applicazione, si potrebbe aggiungere un ulteriore servizio di tipo "Proxy", dotato di un *environment* su cui poter eseguire Node.js. Tale servizio sarebbe l'unico punto di riferimento per il client, ed avrebbe il compito di tradurre richieste RESTful HTTP in messaggi asincroni da smistare opportunamente agli altri servizi tramite la preesistente infrastruttura di RabbitMQ.

A quel punto si potrebbe utilizzare un qualunque client in grado di effettuare chiamate HTTP, e non ci sarebbe nemmeno più necessità di conoscere tutti i servizi che compongono il sistema: sarebbe sufficiente conoscere unicamente il proxy, a cui inviare tutte le richieste tradotte in API RESTful.

Architettura Publish-Subscribe

Nel caso dovessero aumentare considerevolmente il numero dei servizi dell'applicazione e la complessità delle interazioni tra gli stessi, potrebbe aver senso ri-organizzare lo scambio dei messaggi, abbracciando un modello peculiare della programmazione ad eventi: *publish-subscribe*. Infatti, nell'ottimizzazione suggerita precedentemente, si era semplicemente spostata dal client al proxy la responsabilità del conoscere tutti i servizi del sistema e tutti i messaggi che ognuno di essi era in grado di ricevere e processare. Se invece non si volesse affrontare tale problema nemmeno all'interno del proxy? Come si potrebbe evitare completamente?

Si potrebbe pensare di pubblicare tutti i messaggi su un bus condiviso (come se venissero pubblicati degli eventi, che in questo caso però corrisponderebbero a dei messaggi), senza preoccuparsi del destinatario. Sarebbero poi i singoli servizi a sottoscriversi solamente agli eventi (messaggi) di loro interesse.

Ticket "circolare"

Il ticket che identifica il numero di messaggi inviati in ogni chatroom ha una limitazione: **potenziale overflow aritmetico**! Si potrebbe mettere in atto una sorta di ticket più complesso e circolare, che arrivato ad un certo valore quindi ripartisse da capo senza intoppi.