UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Programmazione II**

Studio ed utilizzo della piattaforma a scambio messaggi RabbitMQ

Anno Accademico 2018/2019

Candidato:

Mariarosaria Barbaraci

matr. N46 003154

Ai miei genitori, ai quali devo la persona che sono oggi,
a Fabio, compagno di squadra ed amore della mia vita,
a Maria Paola, amica e sorella dall'infanzia,
agli amici, nuovi e di vecchia data, con cui ho condiviso gioie e paure,
a zia Maria, che, anche se ormai solo nei ricordi, mi è sempre vicina,
ai miei nonni, il cui affetto mi dà sempre forza.
A voi, che mi avete guidata tra le prove sul mio percorso.

Indice

STUDIO ED UTILIZZO DELLA PIATTAFORMA A SCAMBIO MESSAGGI RABBITMQ *1*

Indice **3**

Introduzione **4**

Capitolo 1: La nascita dei MOM ed il bisogno di uno standard **7**
1.1 Breve storia dei MOM 7
1.2 La soluzione offerta da JMS 9
1.3 La nascita degli standard 10
1.3.1 AMQP 10
1.3.2 MQTT 11
1.4 Obiettivi dell'elaborato 12

Capitolo 2: RabbitMQ, un Message Broker open source **13**
2.1 Modello di Messaging 13
2.1.1 Queue 14
2.1.2 Exchanges and Bindings 16
2.2 Durabilità e persistenza 20
2.3 Un broker multi-protocol 22

Capitolo 3 : Esperienze pratiche con RabbitMQ **23**
3.1 Installazione 23
3.2 Esempio di utilizzo delle work queues 23
3.2.1 Applicazione *Producer* 24
3.2.2 Applicazione *Consumer* 26
3.2.3 Conclusioni sul primo esperimento 30
3.3 Esempio di comunicazione broadcast 30
3.3.1 Applicazione *Producer* 30
3.3.2 Applicazione *Consumer* 31
3.3.3 Conclusioni sul secondo esperimento 32

Conclusioni e sviluppi futuri **33**

Bibliografia **34**

Introduzione

Oggigiorno, un sistema informatico, con molta probabilità, sarà un sistema distribuito.
Un sistema distribuito è una tipologia di sistema informatico costituito da vari componenti; questi spesso non sono localizzati sulla medesima macchina, ma sono connessi in rete e comunicano scambiandosi messaggi. L'utente di un sistema distribuito non ha percezione dell'eterogeneità che caratterizza l'architettura del sistema, anzi lo utilizza come se fosse un'unica entità.
I sistemi distribuiti sono nati e si sono affermati sempre di più come conseguenza a esigenze nate in ambito aziendale, dove sistemi software complessi non potevano essere riprogettati da zero, ma dovevano evolversi per stare al passo con i requisiti aziendali e le richieste del mercato. Pertanto, l'integrazione tra software diversi, sviluppati con linguaggi e tecniche differenti, si rende necessaria. L'eterogeneità di prodotti software e hardware determina il bisogno di tecniche di astrazione che permettano agli sviluppatori di applicazioni distribuite di potersi disinteressare della complessità del sistema.
Le infrastrutture *middleware* vengono utilizzate, di solito, per mascherare l'eterogeneità in un sistema distribuito e fungono da intermediarie tra le varie parti del sistema.
Tra i modelli di tecnologie *middleware* ricoprono particolare importanza quelli che utilizzano un modello di programmazione basato sullo scambio di messaggi, i MOM (Message Oriented Middleware). Nati inizialmente come soluzione a problematiche sorte nel settore finanziario, i prodotti *middleware* orientati ai messaggi hanno tradizionalmente

usato protocolli proprietari per le comunicazioni tra applicazioni client e broker. Ne risultava una forte dipendenza dallo specifico fornitore del servizio di messaggistica, il che rendeva difficoltosa l'interoperabilità cross-platform.

In teoria, un prodotto MOM avrebbe dovuto garantire la comunicazione indiretta tra due o più entità di un sistema distribuito fornendo delle API (Application Programming Interface), in modo da sollevare il programmatore dai dettagli di basso livello (protocolli di comunicazione su rete e codifica dei dati); utilizzando tecniche proprietarie, però, si rendeva difficile la realizzazione di tale requisito fondamentale. Per questo motivo negli anni '90 si faceva chiara l'esigenza di una metodologia standard che rendesse possibile la comunicazione tra componenti software eterogenee, una metodologia che fosse indipendente dai linguaggi di programmazione, sistemi operativi, dispositivi hardware o *message broker*.

Esistono oggi *message brokers* open source che fanno largo uso di protocolli standard per implementare la comunicazione indiretta tra applicazioni distribuite.

La nascita di broker e protocolli open source ha contribuito a diffondere sempre più l'utilizzo dei MOM, anche in progetti software di medie dimensioni, e ha portato alla nascita di community che collaborano e contribuiscono alla standardizzazione di suddette tecnologie.

L'elaborato, tramite lo studio del message broker open souce RabbitMQ, è stato redatto con lo scopo di approfondire una tecnologia largamente adoperata negli ultimi anni e dimostrare i vantaggi e l'utilità dell'utilizzo di protocolli standard in un contesto sempre più eterogeneo ed in evoluzione. In particolare, l'elaborato è suddiviso in tre capitoli:

- Il primo capitolo, dopo un breve excursus sulla nascita dei MOM, presenta le soluzioni proposte per favorire l'interoperabilità tra applicazioni distribuite eterogenee, contrastando il monopolio precedentemente attuato dai fornitori di tecnologie MOM commerciali e aziendali. In particolare, si presenta la nascita dei protocolli AMQP ed MQTT, attraverso un confronto con JMS (Java Message Service), e la loro importanza nell'evoluzione del Messaging aziendale e non.

- Il secondo capitolo si concentra sulla descrizione di RabbitMQ, un *message broker* open source molto conosciuto ed utilizzato come implementazione efficiente e scalabile di AMQP 0-9-1. Se ne descrive in maniera approfondita il modello di Messaging e alcune funzionalità avanzate, nonché il supporto ad altri protocolli standard.
- Nel terzo capitolo, invece, sono proposti alcuni esempi pratici di RabbitMQ, con lo scopo di evidenziarne le potenzialità e i possibili utilizzi. Si descrive il codice sviluppato per la sperimentazione, in modo tale da analizzare le funzionalità testate.

Capitolo 1: La nascita dei MOM ed il bisogno di uno standard

1.1 Breve storia dei MOM

La nascita dei MOM può essere collocata agli inizi degli anni ’80, quando un giovane ingegnere indiano ebbe un’idea che avrebbe rivoluzionato l’industria informatica. Vivek Ranadivè, nativo di Bombay, studiò al MIT (Massachusetts Institute of Technology), e, dopo aver conseguito una laurea in ingegneria elettrica ed un master in ingegneria meccanica, si iscrisse alla Harvard Business School. Nel 1983, a soli 26 anni, con l’MBA (Master of Business Administration) di Harvard per le mani, fondò Teknekron, l’azienda nella quale cominciò lo sviluppo della sua idea innovativa. Avendo un background in hardware design, Ranadivè concepì l’idea di un bus software che potesse far comunicare facilmente più applicazioni, proprio come fa un bus hardware, il quale realizza la comunicazione tra i diversi componenti di un computer e ne permette, così, il funzionamento.

Le prime aziende interessate all’idea del giovane ingegnere ed imprenditore indiano furono grandi compagnie del settore finanziario che videro nell’idea di Ranadivè una soluzione per migliorare la loro organizzazione del lavoro e delle risorse soprattutto per quanto riguardava il settore del trading. In quegli anni, infatti, la postazione di un operatore finanziario era piena di terminali, e ognuno di essi conteneva diverse informazioni e programmi necessari all’operatore per svolgere il suo lavoro. Il bus software avrebbe sostituito tutti quei terminali e permesso al trader di usufruire di qualsiasi programma o reperire qualsiasi informazione utilizzando semplicemente una singola postazione desktop, ed un singolo programma che gli permettesse di sottoscriversi solo alle informazioni di suo interesse. Nacque il primo sistema *publish/subscribe* ed il primo sistema di accodamento di messaggi: Teknekron’s

The Information Bus (TIB).

Non trascorse molto tempo che tale modello di trasferimento di informazioni si diffuse e trovò mille usi in altri disparati settori, da quello delle telecomunicazioni a quello della difesa. Un'applicazione *publisher* poteva pubblicare dei dati e un'applicazione *consumer* consumarli senza che fossero collegate fisicamente tra loro e senza che sapessero dell'esistenza l'una dell'altra; dopo che gli sviluppatori avessero deciso quale struttura dovesse avere il messaggio, qualsiasi applicazione consumer, dopo la sottoscrizione, avrebbe potuto ricevere una copia dei messaggi d'interesse. Il bus, che permise il disaccoppiamento tra produttori di messaggi e consumatori, prese il nome di *broker*, proprio per rispecchiare la sua funzione di intermediario.

Dato il largo utilizzo delle code di messaggi (Message Queue - MQ), grandi compagnie del settore informatico come IBM e Microsoft cominciarono a sviluppare i propri sistemi creando concorrenza sul mercato. I fornitori commerciali di software MQ, però, pur volendo favorire la cooperazione tra applicazioni eterogenee, non volevano creare degli standard che permettessero una completa interoperabilità tra broker di diversi fornitori o che permettessero addirittura la sostituzione di una piattaforma a scambio messaggi a favore di un'altra. Tale situazione può essere descritta dall'espressione "vendor lock-in", la quale ben sintetizza il monopolio dei fornitori sulla nuova tecnologia.

Negli anni '90 cominciò a nascere un generale malcontento riguardo agli alti prezzi e alla non completa interoperabilità offerta dai vari fornitori, e non solo da parte di piccole aziende che non potevano permettersi spese eccessive, ma anche da parte delle grandi aziende del settore finanziario, pioniere della tecnologia. Inevitabilmente le aziende finanziarie, essendo molto grandi, facevano uso di diversi prodotti basati su code di messaggi e quindi, quando due prodotti di questi dovevano in qualche modo interagire, si presentarono delle difficoltà. I principali problemi sorsero a causa delle diverse API (Application Programming Interface) e dei diversi protocolli di comunicazione utilizzati da ciascun broker. Perciò, si fece sempre più chiara l'esigenza di una soluzione unica che potesse andare bene in qualsiasi situazione[2].

1.2 La soluzione offerta da JMS

Nel 2001 una prima soluzione venne fornita grazie alla nascita di JMS (Java Message Service). JMS è oggi una API robusta e matura che permette di realizzare una comunicazione basata su scambio di messaggi tra applicazioni Java. Offre un livello di astrazione tale che solleva il programmatore dalle specifiche della comunicazione su rete (protocolli utilizzati, codifica dei dati, ecc…), e rende l'applicazione indipendente dal *message broker* scelto. Infatti, nella comunicazione tra due applicazioni Java, se si utilizza JMS, in base al broker scelto, verrà utilizzato un protocollo di livello applicazione differente per la comunicazione su rete, ma in maniera totalmente trasparente al programmatore; per esempio se si sceglie come broker ActiveMQ, il protocollo nativo è OpenWire, se si utilizza HornetQ, invece, verrà usato il suo protocollo nativo, ma ciò non importa, perché le funzionalità esposte dalla libreria verranno adempiute in ogni caso, senza differenze. Inoltre, con pochi cambiamenti di configurazione si potrà sostituire facilmente il *message broker* lasciando quasi inalterato il codice.

Il problema nasce se la comunicazione attraverso code di messaggi deve avvenire tra un'applicazione Java ed un'altra non-Java (es. Ruby, C#, ecc…). In questo caso infatti, non è possibile utilizzare semplicemente JMS nel programma non-Java, ma si deve trovare anche un modo per effettuare una traduzione dei protocolli e della struttura del messaggio tra le due applicazioni. Per esempio, si supponga che un programma Java debba comunicare con un programma scritto in Ruby; generalmente il protocollo maggiormente utilizzato nel caso di Ruby è STOMP (Streaming Text Oriented Messaging Protocol), quindi c'è bisogno di un *message broker* che supporti allo stesso tempo sia JMS che STOMP.

ActiveMQ, in questo caso è l'ideale, poiché è caratterizzato da un meccanismo interno che permette la traduzione dal suo protocollo nativo, OpenWire, utilizzato da JMS, al protocollo STOMP, ma anche la traduzione di un messaggio JMS in un messaggio STOMP. [4]

Sembrerebbe che il problema di interoperabilità tra un programma Java ed uno Ruby sia stato risolto. Adesso si supponga di voler cambiare *message broker*. Non è detto che si trovi un altro middleware che supporti entrambi i protocolli, e quindi si è costretti a non poter cambiare servizio di Messaging. Inoltre, ci potrebbe essere una non completa compatibilità

tra i tipi di messaggi: dati supportati, proprietà facoltative e campi dell'header del messaggio potrebbero non corrispondere.

In conclusione, seppure si è trovata una soluzione di interoperabilità, essa è molto debole in quanto legata in modo molto forte alle parti in gioco e ad uno specifico fornitore.

Se si aggiungesse nel sistema una terza applicazione, sviluppata con un terzo linguaggio, si dovrebbe nuovamente ragionare sulla compatibilità tra i protocolli e la codifica dei dati dell'applicazione e quelli supportati dal broker.

Si è capito ormai che, nel mondo dei sistemi distribuiti, in cui ogni nodo di uno stesso sistema può essere implementato attraverso l'utilizzo di qualsiasi tecnologia e può essere aggiornato, sostituito, aggiunto o rimosso in qualsiasi momento, c'è bisogno di una soluzione standard, unica, per realizzare in modo efficiente una interoperabilità cross-platform.

1.3 La nascita degli standard

JMS dimostrò che la soluzione non risiedeva nella formalizzazione di una API, poiché essa sarebbe sempre stata legata ad un particolare linguaggio di programmazione, ma si doveva agire più a fondo, magari standardizzando il protocollo di comunicazione e la struttura dei messaggi.

1.3.1 AMQP

AMQP (Advanced Message Queuing Protocol) è un *binary wire-level protocol,* cioè è un protocollo binario che opera a livello applicazione (secondo lo stack protocollare ISO/OSI)[4], e il quale definisce uno standard su come deve essere strutturato il messaggio e come deve essere trasmesso tra un'applicazione ed un'altra, attraverso la rete. Il vantaggio è, appunto, quello di garantire la comunicazione tra due applicazioni senza doversi preoccupare del sistema operativo sul quale esse sono in esecuzione o del linguaggio di programmazione utilizzato per implementarle oppure del *message broker* utilizzato.

L'obiettivo principale che AMQP realizza ed il motivo per cui è stato originariamente creato è quello di risolvere i problemi di interoperabilità tra piattaforme software eterogenee e message brokers (sia commerciali che personalizzati), in modo da poter realizzare,

adottando un unico standard, una facile comunicazione tra applicazioni tramite scambio di messaggi.

Qualitativamente, sono due le proprietà per cui oggi AMQP è molto apprezzato ed utilizzato: reliability e interoperability; reliability indica che il protocollo è affidabile, nel senso di garantire uno specificato livello di prestazioni quando è usato secondo condizioni e tempi specificati, mentre interoperability indica che il protocollo è adatto alle comunicazioni cross-platform.

Come si può dedurre dal nome, tale protocollo offre un ampio spettro di funzionalità avanzate relative allo scambio di messaggi, funzionalità che verranno approfondite più avanti attraverso l'analisi di uno specifico middleware basato su AMQP, il quale permetterà di approfondirne le peculiarità tramite alcuni esempi pratici.

1.3.2 MQTT

MQTT (Message Queue Telemetry Transport) è un protocollo di messaggistica binario che lavora su TCP, proprio come AMQP. Sviluppato originariamente da IBM, ora è un protocollo open source che deve la sua crescita e popolarità alla sua adozione nelle applicazioni mobile e nell'IoT. Questo perché la sua struttura si presenta molto più semplice rispetto a quella di AMQP. MQTT offre un modello di Messaging basato sul pattern *publish/subscribe* ed è stato studiato per le situazioni in cui è richiesto un basso consumo di risorse e dove la banda è limitata. Per queste sue proprietà tale protocollo è largamente utilizzato nella progettazione dei sistemi embedded per le comunicazioni M2M (Machine to Machine). Nonostante la semplicità che lo caratterizza, offre comunque molte proprietà avanzate che lo rendono appetibile per usi aziendali; basti pensare che Facebook lo impiega nello sviluppo dell'applicazione mobile e nel sistema di notifiche di Facebook Messanger. Molti broker che supportano MQTT come protocollo di Messaging gestiscono anche migliaia di connessioni concorrenti e usufruiscono dei tre livelli di QoS (Quality of Service) messi a disposizione dal protocollo:

- Fire and forget – inaffidabile, non si è interessati a sapere se il messaggio sia effettivamente stato recapitato correttamente.
- At least once – si vuole essere sicuri che il messaggio sia stato inviato almeno una

volta. Si potrebbero avere dei duplicati.

- Exactly once – il messaggio deve essere ricevuto esattamente una volta. Non si vogliono duplicati.

1.4 Obiettivi dell'elaborato

Conclusa la panoramica sulla problematica che si vuole affrontare, nei prossimi capitoli è approfondita, tramite lo studio del Message Broker RabbitMQ, la struttura ed il funzionamento del protocollo AMQP 0-9-1, versione nativamente supportata dalla piattaforma in esame. Dopo un intero capitolo dedicato ad una descrizione attenta del modello di Messaging utilizzato da RabbitMQ e ad alcune sue funzionalità avanzate, è presente un terzo capitolo nel quale vengono descritti esempi sperimentali volti a testare praticamente le funzionalità del broker e le potenzialità derivanti dall'adozione di un protocollo standard in un sistema software distribuito, che prevede la presenza di programmi sviluppati in linguaggi di programmazione differenti.

Capitolo 2: RabbitMQ, un Message Broker open source

RabbitMQ è un *message broker* nato quasi negli stessi anni di AMQP, e che si presentò come uno dei primi sistemi open source per lo scambio di messaggi.

2.1 Modello di Messaging

Come si è detto nel capito precedente, i *message broker* permettono di implementare uno scambio di messaggi tra applicazioni fungendo da intermediari; quindi, nella comunicazione tra applicazioni, non si fa più riferimento ai ruoli dettati dal pattern client/server, nel quale si aveva una comunicazione diretta tra due parti, ma vengono introdotte le figure di *consumers* e *producers*. Solitamente, le situazioni in cui viene utilizzato il Messaging sono quelle in cui la propria applicazione deve comunicare con uno o più fornitori di servizi e il message broker agisce da router rendendo possibile la comunicazione. In generale, quando un'applicazione si connette a RabbitMQ può mandare o ricevere messaggi e quindi di conseguenza essere un *producer* oppure un *consumer*. In RabbitMQ, ma più in generale in AMQP, il messaggio è formato da due parti: un *payload* (corpo del messaggio) ed un *header* (intestazione). Il *payload* è il dato che si vuole trasmettere e può essere qualsiasi cosa: un file di testo, un JSON o anche un file multimediale; mentre l*'header* contiene informazioni più interessanti: descrive il *payload* e contiene informazioni su come RabbitMQ determinerà a chi deve essere consegnato il messaggio. Ovviamente in AMQP non si dichiara uno specifico *receiver* o *sender* (come accade per esempio in TCP), ma viene utilizzata una speciale *label* tramite la quale il broker inoltra il messaggio agli opportuni *receivers*. La comunicazione è *fire-and-forget* e *monodirezionale*, quindi il *producer,* dopo aver affidato in messaggio con la relativa *label* al broker, ha concluso il suo compito, la responsabilità del messaggio è passata all'intermediario, e può anche cancellare il messaggio.

Il *consumer*, invece, si connette al broker e si sottoscrive ad una particolare coda; la coda può essere vista come una casella di posta, identificata da un nome, nella quale arrivano i messaggi. I messaggi, una volta nella coda, sono costituiti dal solo *payload*, infatti l'*header*, come si è già detto serve solo al broker per capire in quale coda inoltrare il messaggio. Dato che il messaggio, una volta in coda, è costituito da solo *payload*, il *consumer* non ha modo di sapere chi sia il *producer*. Se si vuole tener traccia del mittente del messaggio, allora il *producer* deve inserire tale informazione nel *payload*.

Per poter realizzare l'invio o la ricezione di un messaggio da e verso il broker, per prima cosa bisogna aprire una connessione TCP tra l'applicazione mittente e RabbitMQ, e, dopo un eventuale autenticazione, predisporre un *channel* (connessione virtuale) sul quale poi verranno effettivamente trasmessi i comandi AMQP per svolgere le operazioni di comunicazione. Il vantaggio di predisporre un canale virtuale piuttosto che trasmettere i comandi AMQP direttamente sulla connessione TCP è quello di poter sfruttare una singola connessione TCP per realizzare molteplici *channels* sui quali avviare diverse comunicazioni. Spesso ciò si verifica nel caso di un programma multithread, dove ciascun thread ha bisogno di un proprio canale in cui sia assicurata una comunicazione privata con il broker; in questo modo non si limitano le performance e il sistema operativo non ha percezione di quanti *channels* vengono aperti e chiusi e quante comunicazioni sono attive contemporaneamente. Pertanto, la flessibilità offerta da AMQP nel poter utilizzare molteplici *channels*, grazie ai quali più thread possono condividere simultaneamente la stessa connessione, permette di poter gestire richieste di messaggi in ingresso, continuando a ricevere e gestire nuove richieste senza instaurare una connessione TCP per ogni thread.

2.1.1 Queue

Come per altri servizi di Messaging, anche in AMQP e RabbitMQ si fa uso delle code (chiamate *queues*), nelle quali vengono depositati i messaggi per poter poi essere inoltrati ai *consumers* interessati. Come già si è detto in precedenza, si può vedere la coda come una casella postale dove i messaggi sostano ed aspettano di essere consumati. Per predisporre il canale della propria applicazione a ricevere messaggi si utilizza il comando AMQP `basic.consume`, il quale comporta che si ricevano automaticamente messaggi da una

determinata coda appena si sia stato consumato (o respinto) l'ultimo messaggio ricevuto. Nel caso in cui si sia interessati solo ad un singolo messaggio, si può utilizzare il comando `basic.get`, il quale comporta che il consumer si sottoscriva ad una coda, prelevi un solo messaggio e poi cancelli la propria sottoscrizione.

In RabbitMQ, quando più *consumers* sono interessati ad una stessa coda, i messaggi di quest'ultima vengono distribuiti in maniera *Round-Robin*, perciò ciascun messaggio è ricevuto da un solo *consumer* alla volta.

Il *consumer*, una volta che ha ricevuto il messaggio, deve confermarne la ricezione al broker; tale meccanismo prende il nome di *acknowledgment*. Il *consumer* può mandare esplicitamente l'*ack* tramite il comando AMQP `basic.ack`, oppure può impostare a *true* il parametro booleano `auto_ack` del comando `basic.consume` in modo tale che RabbitMQ consideri il messaggio confermato non appena il messaggio sia stato ricevuto dal consumer. L' *acknowledgment* è un modo per confermare al broker che il messaggio è stato ricevuto correttamente e che può essere rimosso dalla coda; se il *consumer* si disconnettesse o per qualche malfunzionamento andasse in crash e non avesse ancora mandato l'*ack*, il broker considererebbe il messaggio non consegnato e rimetterebbe il messaggio in coda in modo che possa essere gestito da un altro *consumer*. Inoltre, finché non riceve l'ack, il broker ritiene il *consumer* non disponibile e non gli inoltra nuovi messaggi. Pertanto, se il messaggio ricevuto da un *consumer* riguarda un task da eseguire che necessita di una certa quantità di tempo per considerarsi concluso è bene inviare l'*ack* solo quando l'elaborazione del task sia stata completata.

La scelta della strategia di *acknowledgment* appropriata dipende dal caso d'uso che si sta andando ad implementare; se si è in una situazione in cui la perdita dei messaggi non è critica allora si può usare `auto_ack`, invece, se è importante che ogni messaggio sia ricevuto e gestito correttamente, ed in caso di malfunzionamenti l'elaborazione di ogni task sia garantita, bisogna utilizzare l'*acknowledgment* esplicito tramite il comando AMQP `basic.ack` .

Come è stato accennato in precedenza, c'è anche la possibilità che il *consumer* rifiuti un determinato messaggio. Da RabbitMQ 2.0.0 in poi è possibile usare il comando

`basic.reject`, il quale permette proprio di non accettare il messaggio, e reinserirlo in coda se il parametro `requeue` del comando è settato a *true* o cancellarlo se il parametro `requeue` è settato a *false*.

Non si è ancora detto chi e come viene inizializzata una coda. La coda può essere creata sia dal *producer* che dal *consumer* utilizzando il comando AMQP `queue.declare`. La coda può essere creata con un determinato nome, oppure se non lo si specifica, è RabbitMQ ad assegnare un nome che la identifichi univocamente restituendolo come parametro di ritorno del comando `queue.declare`. Si vedrà in seguito quali sono i casi d'uso in cui conviene specificare il nome di una coda o meno.

Per il momento si vedano quali proprietà possono essere configurate al momento della creazione di una coda:

- `durable`, se impostato a *true*, implica che la coda non verrà cancellata nel caso di un riavvio del broker.
- `exclusive`, se impostato a *true*, rende la coda privata e quindi solo l'app che la dichiara ne può consumare i messaggi.
- `auto-delete`, se impostato a *true*, comporta che la coda venga cancellata non appena l'ultimo consumer cancelli la propria sottoscrizione.

Nel caso in cui venga dichiarata una coda con un nome già esistente, RabbitMQ farà riferimento alla coda già presente (senza creare duplicati) a patto che i valori dei parametri corrispondano, altrimenti verrà restituito un errore.

Se si utilizza il parametro `passive` con valore *true*, invece, `queue.declare` ritornerà un valore positivo se una coda con un determinato nome esiste, un errore altrimenti.[2]

2.1.2 Exchanges and Bindings

Il modello di Messaging completo offerto da RabbitMQ vede la presenza degli *exchanges* e dei *bindings*.

L'*exchange* è il primo componente presente nel broker e funge essenzialmente da router. Riceve in ingresso messaggi e li instrada correttamente nelle varie code basandosi su delle regole.

Finora, con informazioni sulle sole code, si sarebbe potuta implementare una comunicazione

tra *producers* e *consumers* nella quale il messaggio prodotto sarebbe stato ricevuto da un unico consumer (comunicazione *Point to Point*). Invece, se si vuole che uno stesso messaggio sia ricevuto da più consumer, si deve adottare una strategia *publish/subscribe*, e per realizzarla vengono in aiuto proprio gli *exchanges*.

Il *publisher*, l'applicazione che produce e manda il messaggio, non deve più creare la coda, ma si preoccupa solo di creare un determinato messaggio, al quale assegna una *routing key*, e un *exchange*, definendone nome e tipo.

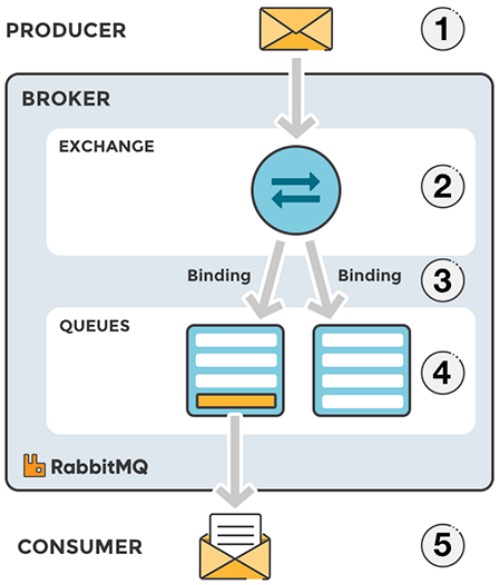

Figura 1- Stack AMQP per l'invio del messaggio[3]

Il comando AMQP per creare l'*exchange* è `exchange.declare`.

Il *subscriber*, l'applicazione che è interessata ad un determinato tipo di messaggi, ha il compito di creare una coda, definire un *exchange* e poi creare il collegamento (*binding*) tra la coda e l'*exchange*. Il collegamento viene creato tramite il comando `queue.bind` e configurando opportunamente una *binding key* .

Si è appena accennato ad una *routing key*, definita nell'atto di pubblicazione del messaggio, ad una *binding key*, che relaziona una coda ad un *exchange*, e all'esistenza di più tipologie di *exchange*. Che relazione hanno tra loro queste tre proprietà?

Ci sono quattro diversi tipi di *exchange* e ognuno di questi implementa un diverso algoritmo di routing basato sui valori di *routing key* e *binding key*:

- *direct* - quando la *routing key* corrisponde esattamente ad una *binding key*, il messaggio viene inoltrato alla coda corrispondente.
- *fanout* - ogni messaggio viene inviato a tutte le code legate all'*exchange* indipendentemente dalla *binding key*.

- *topic* - un messaggio viene inoltrato in una coda la cui *binding key* corrisponde anche parzialmente alla *routing key*.
- *header* - simile al *direct*, ma al posto della *routing key*, si utilizza l'*header* (ha prestazioni peggiori rispetto al caso *direct*).

Mentre le tipologie *direct* ed *header* sono immediate da comprendere, *fanout* e *topic* hanno bisogno di qualche parola in più.

In generale si utilizza *fanout* quando si vuole realizzare una comunicazione broadcast, quindi quando si vuole che tutte le code collegate ad uno stesso *exchange* ricevano gli stessi messaggi. In questa circostanza, nel *subscriber,* non c'è bisogno di creare una coda con un determinato nome; è il broker ad assegnare in maniera random un nome univoco alla coda, il quale servirà semplicemente a collegarla al giusto *exchange*. Nella fase di *binding* non bisogna nemmeno specificare una *binding key* poiché verrebbe comunque ignorata dall'*exchange* di tipo *fanout*.

Un caso d'uso reale potrebbe essere il seguente: si supponga di dover gestire il sistema di un social network basato sulla pubblicazione di fotografie e si supponga in particolare di dover gestire quali operazioni devono seguire il caricamento di una foto da parte di un utente. Si supponga che le operazioni da eseguire siano indipendenti l'une dalle altre, e quindi possano essere eseguite in parallelo; allora si può utilizzare un sistema di Messaging con un *exchange* di tipo *fanout*, chiamato *upload_pictures*.

Ciascuna coda ad esso connessa corrisponderà ad una operazione da eseguire dopo il caricamento di una foto. Si supponga che si voglia ridimenzionare l'immagine, in modo che sia adatta alla pubblicazione sul sito del social network, e si vogliano assegnare dei punti all'immagine; si avranno allora due code.

Questo tipo di architettura permette agli sviluppatori di aggiungere o rimuovere in qualsiasi momento delle funzionalità. Si supponga di voler aggiungere un meccanismo di notifica agli amici in seguito al caricamento di un'immagine, grazie alle funzionalità offerte da AMQP

basterà semplicemente aggiungere una coda e collegarla all'*exchange upload_pictures*.

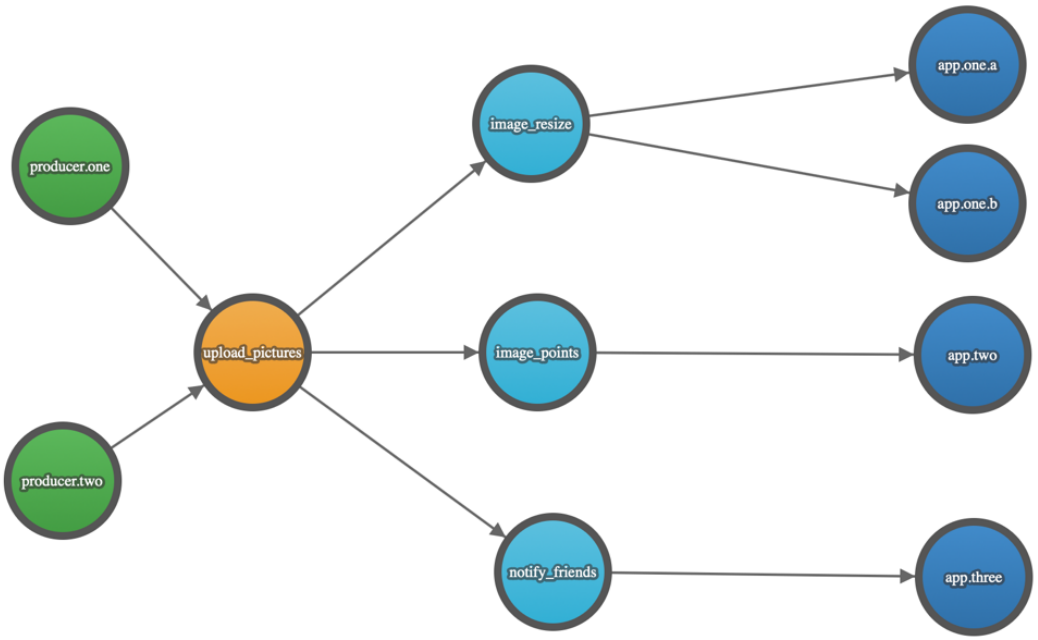

Figura 2 - topologia per l'esempio di un exchange di tipo fanout [7]

Un *exchange* di tipo *topic,* invece, permette di realizzare molteplici scenari di utilizzo e personalizzare l'architettura del broker a proprio piacimento.

Grazie alla flessibilità per la quale la *routing key* deve corrispondere solo in parte alla *binding key*, si possono implementare diverse configurazioni del sistema. La parziale corrispondenza è, però, regolata da una certa notazione, in particolare dall'utilizzo di due caratteri speciali nella definizione della *binding key*:

- * indica la presenza di esattamente una parola,
- # indica la presenza di 0 o N parole.

Per comprendere meglio l'algoritmo di routing nel caso di un *exchange* di tipo *topic*, si presenta di seguito un esempio di applicazione reale. Si supponga di avere un sistema di logging che fa riferimento ad una applicazione web e si supponga di voler raccogliere tutti i messaggi di log provenienti dai vari moduli dell'applicazione e suddividerli in base alla loro tipologia e criticità.

Si supponga una topologia del genere per il routing:

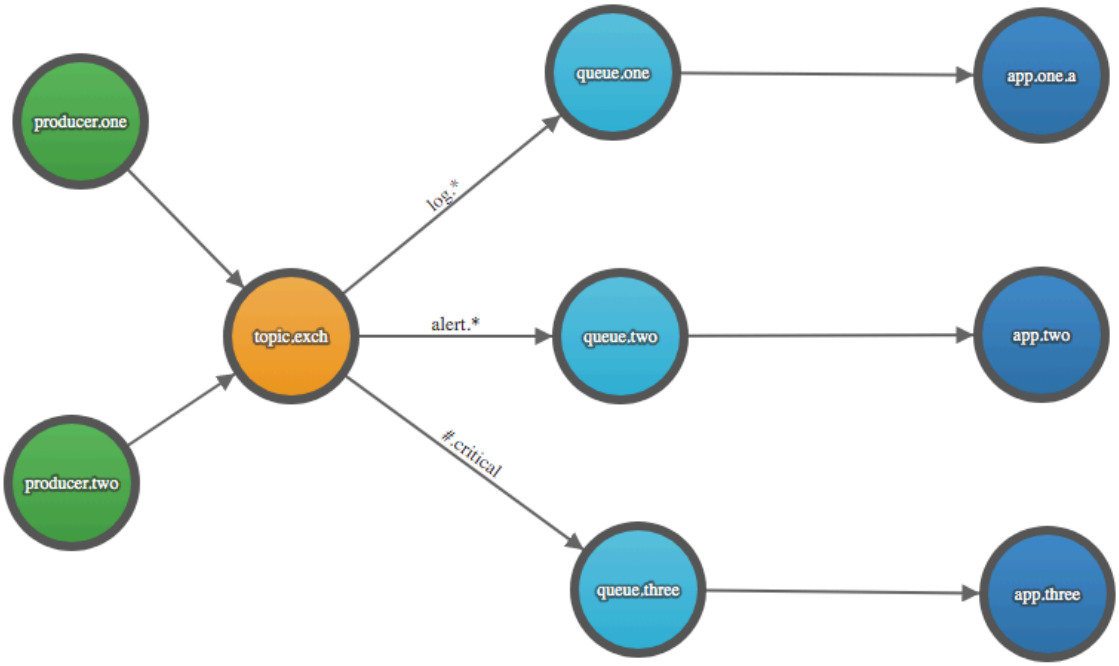

Figura 3 – topologia per l'esempio di un exchange di tipo topic[7]

Si suppongano in ingresso all'*exchange* due messaggi : il primo con una *routing key* pari a 'log.critical' e l'altro pari a 'alert.critical'. In base alla topologia disegnata e alle regole prima citate, si avrà che il primo messaggio sarà mandato alla queue.one e alla queue.three, mentre il secondo alla queue.two e alla queue.three.

Le prime due *binding keys* prevedono che dopo il punto ci sia necessariamente un'altra parola, mentre l'ultima, #.critical, prevede la possibilità che ci siano da 0 ad N parole chiave prima del punto. [1]

2.2 Durabilità e persistenza

Nella situazione in cui si presentasse un malfunzionamento del broker o questo si riavviasse, ogni informazione contenuta nelle code, le code stesse e gli exchange andrebbero persi. Si è già accennato nel capitolo riguardante le *queue* che esiste la proprietà `durable`, la quale permette di creare una coda che sopravviva al riavvio del sistema. Ma se si vuole rendere persistente il sistema, non basta semplicemente tale proprietà, c'è bisogno che anche l'*exchange* in fase di dichiarazione sia definito `durable` e che il messaggio sia etichettato *persistent* tramite il *delivery flag* impostato a 2. L'insieme di tutte e tre queste proprietà è fondamentale per garantire persistenza nel caso dello sviluppo di un sistema che deve gestire messaggi critici. RabbitMQ assicura che i messaggi *persistent* sopravvivano ad un riavvio salvandoli su disco in un *persistent log file*; questo perché, quando si pubblica un messaggio

persistent su un *exchange* `durable`, RabbitMQ non manda una risposta finché il messaggio non è stato scritto anche sul log file. Ovviamente la procedura per garantire la persistenza inficia le performance del broker: salvare i messaggi su disco è un'operazione molto più lenta che conservarli solo nella RAM e quindi decresce notevolmente il numero di messaggi al secondo che può processare il broker. In base alle necessità riguardo le performance, si dovrà scegliere la giusta strategia di persistenza.

Se il requisito di durabilità è fondamentale, perché si utilizza il broker per gestire dei messaggi di importanza critica, si possono utilizzare le *transaction*.

Le *transaction* di AMQP non vanno confuse con quelle più note utilizzate nell'ambito dei database. In AMQP, dopo aver abilitato un *channel* a gestire *transaction*, si può procedere con la pubblicazione del messaggio sensibile, seguito eventualmente da una serie di altri comandi e infine un *commit*, e, se la pubblicazione iniziale va a buon fine, verranno eseguiti gli altri comandi definiti, altrimenti no. Il problema introdotto dalle *transaction* è, però, l'instaurazione di una comunicazione sincrona, uno dei problemi che generalmente il Messaging vuole risolvere.

Per assicurare che un determinato messaggio sia stato correttamente ricevuto, esiste un'altra metodologia introdotta da RabbitMQ (estensione aggiunta alle funzionalità offerte da AMQP), che rende la comunicazione nuovamente asincrona. Tale tecnica è quella dei *publisher confirms*. Un *channel* viene configurato in modalità *confirm*, in questo modo ad ogni messaggio che transita in esso è assegnato un ID univoco; una volta che il messaggio è stato recapitato a tutti i *subscriber* il broker procede con l'invio di un messaggio di conferma al *publisher*, utilizzando l'ID precedentemente assegnato. Si noti che nel caso di code e di *exchanges* `durable` la conferma viene inviata dopo il salvataggio su disco dei messaggi *persistent*.

In conclusione, le strategie migliori da adottare dipendono dal caso d'uso che si deve implementare e da come si vuole che sia strutturato il sistema, tuttavia la prima cosa che si apprende documentandosi su RabbitMQ ed in generale sui MOM è che non sono dei database e non possono essere usati per conservare delle grandi quantità informazioni e per lungo tempo, sono semplicemente degli intermediari; perciò, se si necessita di garantire

persistenza per una grande quantità di dati è bene che si introduca un vero è proprio database, il quale potrà ricoprire poi, a secondo delle esigenze, o il ruolo di *producer* o di *consumer* all'interno dell'architettura del sistema che si sta progettando. [2]

2.3 Un broker multi-protocol

RabbitMQ è molto utilizzato oggi, non solo perché è un broker open source ed è basato su AMQP, ma anche perché è un broker che supporta molteplici protocolli standard per lo scambio di messaggi e per questo motivo si adatta a moltissimi utilizzi diversi.

In particolare, RabbitMQ supporta il protocollo MQTT e il protocollo STOMP, citati nel capitolo uno.

Inoltre, bisogna specificare che la versione nativamente utilizzata da RabbitMQ del protocollo AMQP è la versione 0-9-1, la quale prevede il modello di Messaging caratterizzato da *exchanges* e *bindings* precedentemente descritto.

Esiste da vari anni, e si sta sempre di più affermando, la versione di AMQP 1.0, molto diversa dalla precedente e che non si basa più sul suddetto modello di messaggistica. Nella nuova versione si è voluto definire un protocollo che fornisse semplicemente delle regole per la comunicazione, ma che non imponesse una determinata topologia interna per il broker; in più, tramite AMQP 1.0 si possono implementare non solo comunicazioni *client-broker*, ma anche *broker-broker* e *client-client*, quindi è possibile anche utilizzarlo per comunicazioni *peer-to-peer*.

RabbitMQ, come molti altri broker basati su AMQP, ha introdotto il supporto alla nuova e standardizzata versione del protocollo tramite l'installazione di un plug-in.

Per concludere, si ricorda che RabbitMQ è impiegato soprattutto in piccoli e medi progetti, poiché la sua eterogeneità permette di esplorare più meccanismi e strategie per poter poi scegliere quella più adatta al sistema che si sta progettando.

Capitolo 3 : Esperienze pratiche con RabbitMQ

3.1 Installazione

Per condurre gli esperimenti si è proceduto con l'istallazione di RabbitMQ. Utilizzando il sistema MacOS è bastato utilizzare Homebrew, un gestore di pacchetti molto utilizzato per il sistema operativo di Apple, e digitare il seguente comando da terminale:

```
brew install rabbitmq
```

Durante l'installazione verranno installate anche dipendenze necessarie, come il supporto per Erlang (linguaggio utilizzato per lo sviluppo del broker).

Una volta completata l'installazione gli script del server RabbitMQ e i Client tools si troveranno nella cartella `sbin` accessibile attraverso il percorso `/usr/local/opt/rabbitmq/sbin` che, per comodità, è bene aggiungere alla variabile d'ambiente PATH. Dopodiché il broker può essere avviato con il comando da terminale `rabbitmq-server`.

3.2 Esempio di utilizzo delle work queues

Per presentare un primo esempio di utilizzo di RabbitMQ si è preso come rifermento uno *use case* reale basato sull'esperienza fatta nello sviluppo del software Softonic.

Molto conosciuto ed utilizzato da circa 100 milioni di utenti al mese, Softonic è un portale che permette di poter cercare e scaricare applicativi Software di ogni genere. La testimonianza di come RabbitMQ sia inserito nell'architettura di tale piattaforma, e come sia utilizzato, è stata fornita da un articolo presente sul blog di CloudAMQP.

CloudAMQP è un servizio cloud che permette il deploy e la configurazione di clusters RabbitMQ in modo molto semplice e veloce.

Secondo l'articolo[3], RabbitMQ è utilizzato per realizzare un'architettura basata su eventi; c'è un *event bus,* il quale raccoglie tutte le operazioni da effettuare e man mano le consegna all'applicazione consumer che sa come gestirle. L'utilizzo di un *event bus* permette al web server di rispondere velocemente alle richieste che riceve, invece di eseguire delle elaborazioni lunghe e bisognose di risorse facendo rimanere gli utenti in attesa di una risposta.

Volendo basarsi su quest'esempio è stato prodotto un semplice programma che vede la comunicazione tra un *producer* ed un *consume*r tramite una coda, che funge da *event bus*. Questo è realizzato tramite una *work queue*, usata per distribuire dei task *time-consuming* tra molteplici *workers*.

L'idea principale alla base di questo tipo di code è l'evitare di eseguire dei task che hanno bisogno di un alto utilizzo di risorse immediatamente e dover poi aspettare che l'elaborazione sia conclusa per poter proseguire. Quindi, si incapsula un task in un messaggio e si manda ad una coda. Quando si mandano in esecuzione più *workers* i task verranno suddivisi tra di loro secondo uno scheduling Round-Robin.

Per dimostrare l'interoperabilità offerta da AMQP, l'esempio è stato realizzato tramite l'implementazione in due linguaggi diversi per le applicazioni producer e consumer.

L'esempio segue la seguente topologia:

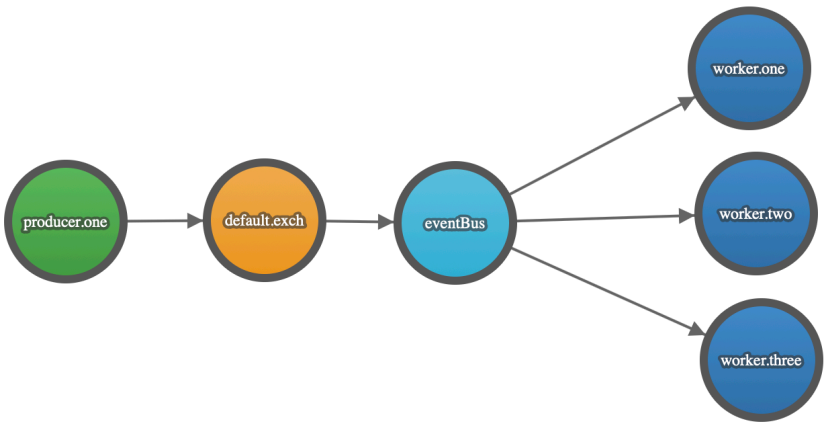

Figura 4 - topologia per il sistema implementato nell'esempio con le work queue[7]

3.2.1 Applicazione *Producer*

L'applicazione producer è stata scritta in Python. Se ne riporta il codice in figura 5.

```
1  import pika
2  import sys
3
4  QUEUE_NAME="eventBus"
5
6  connection = pika.BlockingConnection(
7      pika.ConnectionParameters(host='localhost'))
8  channel = connection.channel()
9
10 channel.queue_declare(queue=QUEUE_NAME)
11 message = ' '.join(sys.argv[1:]) or " "
12
13 channel.basic_publish(exchange='', routing_key=QUEUE_NAME, body=message)
14 print(" [x] Sent %r" % message)
15 connection.close()
```

Figura 5

Per prima cosa bisogna importare `pika`[6], libreria che realizza l'implementazione in Python del protocollo AMQP 0-9-1, dopodiché, secondo quello che è stato detto nel capitolo su RabbitMQ, le prime operazioni da effettuare sono quelle per aprire la connessione verso il broker e predisporre un *channel* sul quale mandare i comandi AMQP. Per cui si faccia riferimento alle istruzioni alle righe da 6 ad 8 che riportano le istruzioni per effettuare tali operazioni:

`BlockingConnection` crea un livello al di sopra delle funzionalità di base di `pika`, che sono di natura asincrona, fornendo dei metodi bloccanti; ciò permette al programma di non andare avanti finché non si sia instaurata la `connection`.

Dato che il broker è stato installato sulla stessa macchina dove verranno eseguiti i programmi utilizziamo come host `localhost`.

Una volta avviata la connessione si può predisporre il *channel* sul quale poi avverrà effettivamente la comunicazione (riga 8).

Successivamente, attraverso l'istanza del *channel,* si crea la coda, con un determinato nome (riga 10). Si ricordi che, all'atto della creazione della coda, si possono configurare altre proprietà quali `durable`, `passive`, `auto-delete`, `exclusive`, ma dato che il metodo `queue_declare` fornito da `pika` offre dei parametri di default per tali proprietà, e tali valori di default sono pari a false per ciascuna di queste proprietà, possiamo utilizzare la versione del metodo con il solo parametro che specifica il nome della coda se non vogliamo abilitare nessuna delle altre funzionalità.

Se abbiamo necessità di rendere la coda robusta a malfunzionamenti o riavvii del server possiamo abilitare `durable` ed useremo la versione del metodo:

`channel.queue_declare(queue=QUEUE_NAME, durable=true).`

Si deve fare poi attenzione in seguito a creare la coda nel *consumer* utilizzando, oltre allo stesso nome, gli stessi valori anche per i parametri.

Dopo aver creato correttamente anche la coda si deve procedere con le operazioni per l'invio del messaggio. In questo caso si è supposto che il messaggio venga inserito a riga di comando nel momento in cui si esegue il programma. Per procedere all'invio si utilizza il metodo `basic_publish` (linea 13), nel quale va specificato il nome dell'*exchange* che in questo caso è una stringa vuota (perchè con le *work queues* non si crea un particolare *exchange*, ma si usa quello di default), la *routing key,* che nel nostro caso coincide con il nome della coda, e il messaggio da inviare.

Alla fine delle operazioni bisogna chiudere la connessione (linea 15).

Notiamo che nel caso preso in analisi, e quindi quando si adoperano le work queue, il producer crea in maniera esplicita la coda in modo tale da indirizzare proprio su di essa i messaggi prodotti. Non si utilizza il modello di Messaging completo fornito da RabbitMQ, il quale prevede anche l'utilizzo di *bindings* ed *exchanges*.

Una volta conclusa la spiegazione su quali sono le operazioni che necessariamente deve compiere un *producer*, si presenta l'applicazione *consumer*.

3.2.2 Applicazione *Consumer*

Come già detto in precedenza, tale applicazione è stata implementata in un diverso linguaggio di programmazione: Java.

Nel caso di un programma java, per implementare il protocollo AMQP bisogna utilizzare la libreria client java per AMQP e le sue dipendenze.

Per il programma realizzato si sono quindi utilizzate le seguenti API[5]:

`amqp-client-5.5.1,`

`slf4j-api-1.7.25,`

`slf4j-simple-1.7.25 .`

Come nel *producer* anche nell'applicazione *consumer* bisogna, come prima operazione,

creare la connessione. Nel caso di Java, viene in aiuto l'uso della classe `ConnectionFactory` per semplificare l'apertura della connessione e configurare i parametri necessari.

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();
```

```
9  public class Worker extends Thread {
10
11      private int id;
12      private Connection connection;
13      private String nameQueue;
14      private Channel channel;
15
16      public Worker(int id, Connection connection, String nameQueue) throws IOException {
17          this.id = id;
18          this.connection = connection;
19          this.nameQueue = nameQueue;
20          channel = connection.createChannel();
21      }
22
23      public void run(){
24          try {
25          channel.queueDeclare(nameQueue, false, false, false, null);
26          System.out.println("[Thread "+id+"]: Waiting for a message");
27          DeliverCallback deliverCallback = (consumerTag, delivery) -> {
28              String message = new String(delivery.getBody(), "UTF-8");
29              long deliveryTag = delivery.getEnvelope().getDeliveryTag();
30              System.out.println(" [Thread "+id+"]: Received '" + message + "'");
31              try {
32                  doWork(message, deliveryTag);
33              }catch(InterruptedException | IOException e) {
34                  e.printStackTrace();
35              }
36          };
37          Boolean autoAck = false;
38          channel.basicConsume(nameQueue , autoAck, deliverCallback, consumerTag -> { });
39          }catch(Exception e) {
40              e.printStackTrace();
41          }
42      }
43
44      private void doWork(String task, long deliveryTag) throws InterruptedException, IOException {
45          if (task.compareTo("ScanFile") == 0) {
46              System.out.println("[Thread "+id+"]: comando in esecuzione ScanFile...");
47              Thread.sleep(5000);
48              channel.basicAck(deliveryTag,false);
49              System.out.println(" [Thread "+id+"]: Done ");
50          }else if(task.compareTo("DownloadFile") == 0) {
51              System.out.println("[Thread "+id+"]: comando in esecuzione DownloadFile...");
52              Thread.sleep(10000);
53              channel.basicAck(deliveryTag,false);
54              System.out.println(" [Thread "+id+"]: Done ");
55          }else {
56              channel.basicReject(deliveryTag, false);
57              System.out.println(" [Thread "+id+"]: Message rejected ");
58          }
59      }
```

Figura 6

Per simulare l'utilizzo di un consumer *multithread*, è stata aggiunta una classe `Worker` (di cui si riporta il codice in figura 6), la quale estende Thread e, utilizzando la medesima connessione e *queue,* avvia un thread con un proprio *channel*, in modo da realizzare una comunicazione privata con il broker e rimanere in ascolto di richieste da gestire.

Perciò, dopo aver istanziato la connessione, *consumer* avvia tre Workers in modo da dividere il carico delle richieste.

La classe `Worker`, oltre ad un parametro `id`, possiede altri tre attributi: la `connection`, `queueName` e il `channel`. Mentre i primi tre sono assegnati tramite i parametri nel costruttore (linee 16-21), l'ultimo viene creato tramite la connessione.

Le operazioni, che devono essere svolte dai thread, devono essere inserite nell'override della funzione `run()`.

Dato che si vogliono gestire i messaggi ricevuti dalla coda, bisogna creare la coda e predisporre il *channel* a ricevere.

La coda può essere dichiarata attraverso un metodo corrispondente a quello utilizzato nel codice Python: `queueDeclare` (linea 25).

Si ricordi che i parametri della funzione nel *consumer* devono corrispondere a quelli assegnati nella dichiarazione della coda del *producer*. Inoltre, la coda è collegata automaticamente all'*exchange* di default (non c'è bisogno di *exchange* e *binding*).

Il consumo di un messaggio avviene, invece, tramite il metodo `basicConsume`, un metodo bloccante che permette di definire il comportamento del *consumer* nel momento in cui arriva un messaggio. Per meglio esplicitare il funzionamento e i parametri richiesti da tale metodo,utilizzato nel codice alla riga 38, se ne riporta la signature:

```
String basicConsume(String queue, boolean autoAck,
DeliverCallback deliverCallback, CancelCallback
cancelCallback) throws IOException.
```

- `queue` - nome della coda
- `autoAck` - *true* se il server considera il messaggio confermato nel momento in cui viene ricevuto, *false* se si procederà con un *acknowledgment* esplicito

- `deliverCallback` - metodo chiamato nel momento in cui arriva un messaggio
- `cancelCallback` - metodo chiamato nel caso in cui il consumer sia cancellato

L' `autoack` è a *false*, in modo tale da utilizzare il metodo dell'*ack* esplicito, più appropriato nel nostro caso, e si definisce la `deliverCallback` come riportato nelle righe da 27 a 36.

`DeliverCallBack` è una FunctionalInterface, che in java rappresenta un'interfaccia con esattamente un metodo. Spesso si preferisce implementarla utilizzando la sintassi lambda, proprio come è stato fatto nell'esempio, in modo da essere più sintetici nella scrittura del codice. Nel corpo della callback viene estratto il messaggio d'interesse e il *delivery tag,* il quale servirà nel momento in cui si dovrà mandare l'*ack*.

Si è poi scritto un metodo privato `doWork()`, che contiene le elaborazioni da effettuare in base al messaggio ricevuto. Nell'esempio presentato si ipotizza che si possano ricevere due tipi di comandi: *ScanFile*, che simula una scansione antivirus da effettuare su un file caricato da un utente, e *DownloadFile*, che simula una richiesta di download pervenuta da un utente. Il programma prodotto è una versione molto semplificata di quello che potrebbe essere un caso di utilizzo reale quindi le elaborazioni vengono simulate attraverso l'utilizzo delle `sleep()`.

Il metodo implementato (righe da 44 a 59) ha come parametri d'ingresso: il `task`, in base al quale deve essere svolta una particolare operazione, e il `deliveryTag`, il quale identifica un particolare messaggio e quindi sarà utile nel momento in cui si deve comunicare al broker l'esito della ricezione. Tramite la funzione `basicAck` viene mandato l'*ack* esplicito alla fine dell'elaborazione richiesta dal comando contenuto nel messaggio ricevuto. Se ci dovesse essere qualche malfunzionamento per cui il *consumer* si dovesse disconnettere senza aver mandato l'*ack*, il messaggio verrebbe reinserito in coda.

Si noti che, oltre a gestire le situazioni in cui arrivi uno dei due comandi attesi, è stato gestito anche il caso in cui nel messaggio arrivi qualche altro comando. In quel caso viene utilizzato il metodo `basicReject`(riga 56), il quale, tramite il riferimento al `deliveryTag`, rifiuta un determinato messaggio e tramite il secondo parametro, `requeue`, definisce anche

se il messaggio deve essere cancellato oppure rimesso in coda. Nel caso in esame, dato che si sanno gestire solo due tipologie di comandi, è meglio che il messaggio sia cancellato in modo da non andare a sovraccaricare inutilmente la coda.

3.2.3 Conclusioni sul primo esperimento

Infine, si può procedere con il provare se lo scambio di messaggi avviene nel modo corretto. Come prima cosa si deve avviare il server da terminale tramite il comando `rabbitmq-server`, dopodiché, utilizzando nuove finestre del terminale ed avviando un programma *producer* ed uno *consumer*, si può osservare che, man mano che arrivano i messaggi nella coda, i thread li ricevono in maniera RoundRobin.

Per monitorare tramite un'interfaccia grafica la ricezione dei messaggi, il numero di channel attivi ed altre informazioni sull'attività del broker mentre i programmi sono in esecuzione, ci si può collegare, tramite browser, alla ManagmentUI di RabbitMQ. Bisogna collegarsi all'indirizzo `http://{node-hostname}:15672/` ed eventualmente accedere (se non si sono cambiate in precedenza) con le credenziali `guest` e `guest`.

3.3 Esempio di comunicazione broadcast

Di seguito si vuole presentare brevemente un secondo esempio[2] nel quale viene adoperato il modello di messaggistica completo messo a disposizione da RabbitMQ. In particolare, si vuole implementare l'esempio di comunicazione broadcast presentato nel capitolo due utilizzato per spiegare il funzionamento di un'exchange di tipo *fanout*. Nello scenario di un sistema software di un social network, è stato fatto riferimento al caso d'uso del caricamento di una foto da parte di un utente.

3.3.1 Applicazione *Producer*

Nell'applicazione producer, realizzata anche questa volta in Python, si deve semplicemente gestire l'invio di un messaggio del tipo *newPicture* all'*exchange* di tipo *fanout upload_picture*.

```python
import pika
import sys

EXCHANGE_NAME = "upload_pictures"

connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange=EXCHANGE_NAME, exchange_type = 'fanout')
message = "newPicture "
channel.basic_publish(exchange=EXCHANGE_NAME, routing_key='', body=message)
connection.close()
```

Figura 7

Come si può vedere, il codice (figura 7) del programma *producer* è molto simile a quello dell'esempio precedente eccetto per il fatto che, in questo caso, il producer non deve dichiarare la coda, ma solo l'*exchange* a cui deve mandare il messaggio. Utilizzando quest'architettura si realizza un maggiore disaccoppiamento tra *sender* e *receiver*, infatti il producer, una volta che il messaggio è stato preso in carico dal broker, può anche dimenticarsene.

Si noti che, nella dichiarazione dell'*exchange,* non è stata specificata una *routing key* perché verrebbe ugualmente ignorata da un *exchange* di tipo *fanout*.

3.3.2 Applicazione *Consumer*

Come nell'esempio precedente, l'applicazione *consumer* è stata scritta in Java. Nel caso d'uso in esame devono essere svolte tre operazioni in parallelo a valle del caricamento di un'immagine da parte dell'utente, allora sono stati creati tre programmi *consumer* che svolgono le operazioni di: ridimensionare l'immagine, assegnarle dei punti e notificare gli amici. Si riporta il codice di uno solo dei tre programmi, in quanto il codice è il medesimo

```java
        channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
        String queueName = "resize_image";
        channel.queueDeclare(queueName, true, false, false, null);
        channel.queueBind(queueName, EXCHANGE_NAME, "");

        System.out.println("[*] Waiting for a message");

        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            try {
                Thread.sleep(5000);
            }catch(InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(" [x] Completed resizing for'" + message + "'");
        };
```

Figura 8

dato che le operazioni sono simulate tramite delle `sleep()`. Per simulare che le operazioni sono indipendenti tra di loro e non si influenzano sono stati impostati tre tempi di elaborazione diversi per i tre task.

Il codice (Figura 8), anche in questo caso è molto simile a quello dell'esempio precedente, quindi se ne riporta la parte in cui sono presenti istruzioni diverse. In quest'applicazione consumer è stato dichiarato l'*exchange* di tipo *fanout*, in modo da ricevere tutti i messaggi ad esso pervenuti e si crea la *queue* tramite l'istruzione a riga 21. Dopodiché, tramite `queueBind`, si realizza il *binding* tra l'*exchange* e la coda (senza definire la *binding key*). Nella DeliverCallback, chiamata quando arriva un messaggio, viene simulata l'elaborazione di ridimensionamento dell'immagine che dura cinque secondi.

3.3.3 Conclusioni sul secondo esperimento

Per testare il funzionamento del sistema realizzato e della comunicazione broadcast bisogna avviare un programma producer e i tre programmi consumer. Si osserverà che, avendo tempi di elaborazioni diversi, i tre programmi consumer lavoreranno parallelamente, ma non in maniera sincrona, ricevendo tutti i messaggi pervenuti all'*exchange upload_pictures*.

La topologia del sistema è riportata nella figura 2 presente nel secondo capitolo.

Conclusioni e sviluppi futuri

I MOM sono uno strumento molto potente che permette a due programmi di comunicare anche se sono localizzati in parti opposte del mondo; ed è proprio per questo motivo che sono uno strumento indispensabile nella progettazione software odierna. Il mondo del software sarà sempre di più orientato verso topologie distribuite e i MOM propongono delle ottime strategie per realizzarle.

RabbitMQ si è dimostrato un *message broker* molto versatile, fornisce funzionalità avanzate che lo rendono perfetto per essere usato sia in ambito aziendale che da parte di sviluppatori alle prese con nuovi progetti. Inoltre, il supporto di più protocolli lo rende adatto a qualsiasi tipo di progetto. Come si è evinto anche dagli esercizi sperimentali implementati, l'utilizzo di un protocollo standard per la comunicazione rende il sistema da progettare scalabile ed interoperabile; se un giorno si decidesse di sostituire l'applicazione *producer* né il broker né tantomeno l'applicazione *consumer* ne avrebbero percezione. Sarebbe interessante testare in futuro l'utilizzo di RabbitMQ con il protocollo MQTT, oppure utilizzando AMQP e MQTT insieme, per realizzare applicazioni IoT, applicazioni che si stanno sempre più diffondendo e che probabilmente domineranno nel settore informatico negli anni avvenire.

Nel mondo dell'ingegneria del software è bene essere al passo con i tempi e conoscere le tecnologie del momento, ma ancora più importante è capire la direzione in cui si muove l'innovazione per cercare di essere un passo avanti e prepararsi, acquisire le competenze adatte per partecipare in prima persona al cambiamento.

Bibliografia

[1] RabbitMQ, https://www.rabbitmq.com , ultimo accesso19 giugno 2019

[2] Alvaro Videla, Jason J.W. Williams, RabbitMQ in Action – Distributed message for everyone, Manning Publications Co. , 2012, Capitoli 1-2-4.

[3] CloudAMQP, https://www.cloudamqp.com/blog/, ultimo accesso 19 giugno 2019

[4] Mark Richards , Understanding the differences between AMQP & JMS , No Fluff Just Stuff the Megazine, 2011.

[5] RabbitMQ Java Client 5.7.1 API, https://rabbitmq.github.io/rabbitmq-java-client/api/current/index.html, ultimo accesso19 giugno 2019

[6] Introduction to Pika, https://pika.readthedocs.io/en/stable/, ultimo accesso19 giugno 2019.

[7] RabbitMQ Visualizer, https://jmcle.github.io/rabbitmq-visualizer/, ultimo accesso19 giugno 2019.