UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

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

Elaborato di laurea in **Programmazione II**

Analisi della piattaforma RabbitMQ per lo sviluppo di applicazioni message-oriented

Anno Accademico 2014-2015

relatore
Ch.mo prof. Marcello Cinque

candidato
Antonio Riccio
matr. N46001235

Indice

Introduzione

I middleware orientati ai messaggi sono una parte fondamentale nello sviluppo di sistemi distribuiti e di applicazioni enterprise su vasta scala. Esiste un grande interesse in ambito industriale nel loro utilizzo per la realizzazione di sistemi caratterizzati da una notevole scalabilità e criticità. Il sistema di controllo del traffico aereo in Europa, l'infrastruttura per il monitoraggio e controllo della rete elettrica del nord America sono alcuni degli esempi di applicazione di queste piattaforme.
In questo elaborato verrà analizzata una delle soluzioni di middleware orientato ai messaggi più diffuse: RabbitMQ. La trattazione verterà su un approfondimento del middleware seguita da un'analisi di alcuni aspetti implementativi legati al principale protocollo di comunicazione adottato, ovvero AMQP. E' presente inoltre una breve guida allo sviluppo di una semplice applicazione che utilizza le API di RabbitMQ. L'analisi si conclude con un confronto con un'altra soluzione middleware molto diffusa, quale ActiveMQ.

Capitolo 1

Message Oriented Middleware

Un *Message Oriented Middleware* è un middleware orientato alla comunicazione basato sul concetto di *messaggio* che viene scambiato tra le diverse entità di un sistema distribuito mediante l'astrazione di coda di messaggi. Nel modello a scambio di messaggi la comunicazione è di tipo peer-to-peer, segue il paradigma produttore/consumatore ed è asincrona. La gestione della ricezione e dell'inoltro dei messaggi è affidata ad entità chiamate *broker* che garantiscono maggior disaccoppiamento tra senders e receivers oltre che ad una migliore tolleranza ai guasti. Il tipo di comunicazione instaurata tra le varie entità prende il nome di *comunicazione indiretta* [2]. Essa può essere implementata in diversi modi, i più importanti sono:

- Publish/Subscribe;
- Code di messaggi;

Il modello publish/subscribe si basa sul concetto di *topic*. Vi sono dei publishers che pubblicano messaggi riguardanti un certo topic e dei subscribers che ricevono i messaggi solo se interessati a quel determinato topic. In questo modello un messaggio pubblicato da uno o più publisher verrà inoltrato a tutti i subscriber sottoscritti.
Nel modello a code i messaggi vengono scambiati attraverso una *coda* gestita con politica FIFO. Un messaggio prodotto da un produttore verrà consumato da un consumatore collegato alla coda. In questo modello è possibile avere più produttori ma un singolo consumatore. Quindi anche nel caso in cui ci siano più consumatori collegati alla coda, il messaggio verrà inoltrato al primo consumatore sottoscritto.

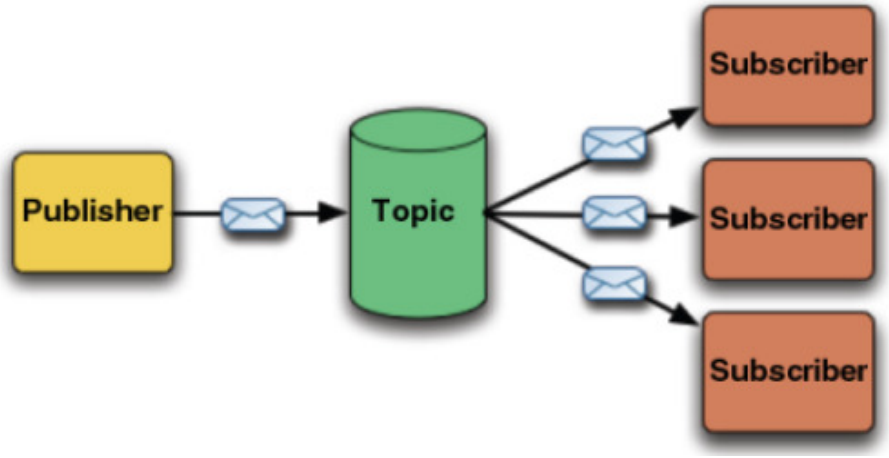

Figura 1.1: Schema publish/subscribe

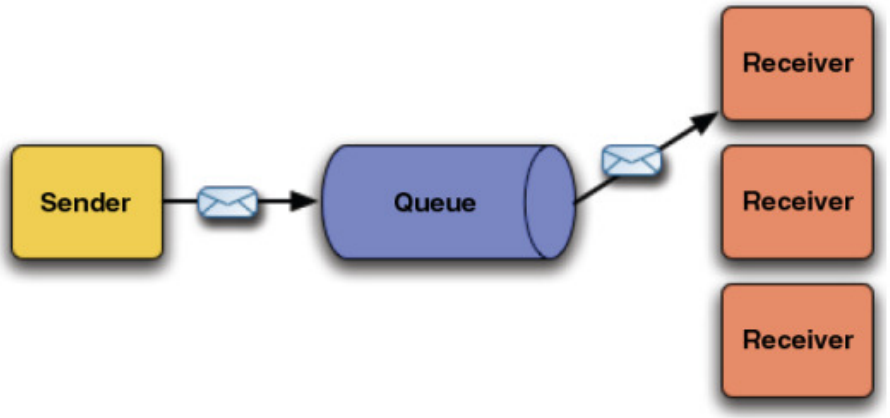

Figura 1.2: Schema a coda di messaggi

1.1 Java Message Service

JMS è uno standard che specifica un insieme di interfacce, utilizzate dalle applicazioni Java per usufruire dei servizi offerti da un broker MOM in maniera portabile. Attraverso questo standard un'applicazione può essere eseguita su diverse piattaforme di produttori differenti con una modifica minima del codice sorgente. Siccome JMS specifica solo un insieme di interfacce e non una implementazione, risulta evidente che è necessaria una connessione con una particolare implementazione per poter eseguire l'applicazione. Lo standard prevede che la connessione con il broker venga instaurata dal client mediante l'utilizzo di particolari oggetti detti *Adrministred Objects*: questi sono oggetti pre-configurati creati dal broker al suo avvio che contengono tutte le informazioni necessarie per la comunicazione con il broker. Sebbene siano dipendenti dalla piattaforma utilizzata, le loro interfacce sono parte dello standard JMS. Gli administred objects presenti in JMS sono due: *ConnectionFactory* e *Destination*. Il ConnectionFactory è l'oggetto che consente di ottenere un oggetto *Connection*, che astrae una connessione TCP con il broker, mentre Destination rappresenta l'entità alla quale i messaggi sono inviati o ricevuti. Le code ed i topic sono oggetti di tipo Destination. Ogni entità che vuole inviare o ricevere messaggi deve inoltre creare un oggetto *Session*. Un oggetto Session rappresenta un canale virtuale single-thread, che fa parte di una Connection, dal quale ricavare un entità *Message Producer* per l'invio di messaggi oppure un *Message Consumer* per il consumo. Gli oggetti Session, Message Producer e Message Consumer sono progettati per contesti single-threaded poichè progettati per supportare comunicazioni di tipo transazionale, difficili da implementare in contesti multi-threaded.

Capitolo 2

RabbitMQ

In questo capitolo verrà affrontata la trattazione del middleware RabbitMQ. In particolare la trattazione inizierà con una breve descrizione delle caratteristiche, seguita da alcuni particolari riguardanti il protocollo che utilizza per la comunicazione su rete. La trattazione comprende inoltre lo sviluppo di un semplice applicativo esemplificativo che illustra l'uso delle API.

2.1 Caratteristiche

RabbitMQ è un message-oriented-middleware open-source che implementa il protocollo *Advanced Message Queuing Protocol* alla versione 0-9-1 ed è scritto nel linguaggio Erlang, un linguaggio specializzato per lo sviluppo di sistemi distribuiti, soft real-time, non-stop e altamente scalabili.
RabbitMQ permette il clustering, ovvero il raggruppamento logico di diversi server fisici in un unico broker logico per garantire maggiore scalabilità e affidabilità. Il clustering non supporta bene le gerarchie di reti [6], quindi non può essere usato su WAN. Per sfruttare il clustering a livello WAN RabbitMQ supporta il meccanismo di routing federativo.
Altra caratteristica di affidabilità è quella che permette la replicazione delle code sui nodi di un cluster assicurando che i messaggi non siano mai persi anche dopo un fallimento di un nodo.
Il sistema può essere arricchito attraverso l'uso di numerosi plugin. Uno di questi è un'interfaccia grafica HTTP per la gestione ed il monitoraggio che si affianca alla gestione da riga di comando. Le librerie client per interfacciarsi a questo broker sono disponibili per diversi linguaggi di programmazione, tra cui C, C++, Java, Erlang e così via. Esiste una versione commerciale sviluppata da VMware che implementa lo standard JMS.

2.2 Sviluppare un'applicazione per RabbitMQ

In questo paragrafo verranno illustrate le fasi principali per la scrittura di un'applicazione "Hello World" che fa uso del broker RabbitMQ nel linguaggio di programmazione Java. In particolare, verranno sviluppati due applicativi: un produttore che invia un singolo messaggio su una coda ed un consumatore che rimane in attesa di ricevere messaggi e ne stampa a video il contenuto.

2.2.1 Sender

Le librerie da importare sono le seguenti:

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
```

A questo punto è possibile effettuare la connessione al broker:

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

Le fasi attraverso cui si compone la connessione al broker prevede innanzitutto la creazione di un oggetto ConnectionFactory, un administred object che serve a facilitare la creazione di un oggetto Connection con il broker. Un oggetto di tipo Connection rappresenta un canale di comunicazione TCP con il broker. In questo caso la connessione è effettuata verso un broker locale. Nel caso di un broker remoto basta specificarne il nome o l'indirizzo IP.
Dalla Connection si passa alla creazione di un oggetto di tipo Channel. Un Channel rappresenta un canale virtuale, costruito sulla Connection, che mette in comunicazione il sender ed il broker. Il suo utilizzo è necessario per non sovraccaricare eccessivamente il sistema operativo di troppe connessioni TCP, che possono risultare onerose.
Per poter inviare bisogna dichiarare una coda sulla quale pubblicare il messaggio:

```
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
String message = "Hello World!";
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
System.out.println(" [x] Sent '" + message + "'");
```

Si noti che il messaggio è inviato come arrray di byte, quindi è possibile codificare qualsiasi tipo di informazione al suo interno.
Infine, viene effettuata la chiusura del canale e della connessione:

```
channel.close();
connection.close();
```

2.2.2 Receiver

Le librerie da importare sono le stesse del sender con l'aggiunta delle seguenti:

```
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
```

Si effettuano poi le medesime operazioni del sender: creazione di un oggetto ConnectionFactory, apertura di una Connection, apertura di un Channel associato alla Connection e dichiarazione di una coda dalla quale consumare i messaggi in arrivo.
Si noti che, in questo caso, il nome della coda deve coincidere con quello usato dal sender, ciò è dovuto al protocollo AMQP che fa uso del concetto di exchange, che verrà approfondito nella sezione successiva.
Per la predisposizione alla ricezione di messaggi sulla coda:

```
QueueingConsumer consumer = new QueueingConsumer(channel);
channel.basicConsume(QUEUE_NAME, true, consumer);
```

Il receiver può ricevere messaggi sia in maniera asincrona mediante l'utilizzo di un listener, che in maniera sincrona. In questo caso viene utilizzata la modalità sincrona per cui:

```
QueueingConsumer.Delivery delivery = consumer.nextDelivery();
String message = new String(delivery.getBody(), "UTF-8");
System.out.println(" [x] Received '" + message + "'");
```

Nel caso si voglia far in modo che il receiver rimanga costantemente in attesa di messaggi si possono racchiudere le istruzioni precedenti in un ciclo.

2.3 AMQP

L'*Advanced Message Queuing Protocol* è un protocollo di comunicazione standard per middleware orientati ai messaggi. AMQP è un *wire-level protocol*, un protocollo che definisce in maniera univoca il formato dei dati scambiati per la comunicazione su rete. In tal modo, sistemi MOM AMQP differenti possono comunicare tra loro a patto di usare la stessa versione del protocollo. In JMS, invece, si specifica solamente un'API che consente alle applicazioni di comunicare con un MOM, pertanto applicazioni Java che fanno uso di JMS ma sono eseguite su piattaforme MOM diverse possono non riuscire a

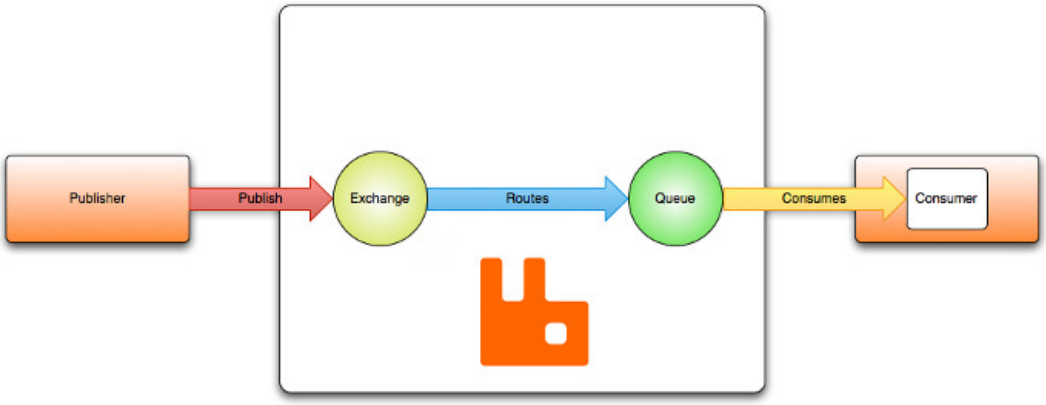

Figura 2.1: Funzionamento di AMQP

comunicare.
Il modello di funzionamento del protocollo AMQP prevede l'invio dei messaggi a entità note come *exchange* le quali inoltrano, a loro volta, i messaggi alle code dei consumer in base a determinate regole dette *binding*, rappresentate da *binding key*. La Figura 2.1 illustra il funzionamento di questo modello.
A causa dell'inaffidabilità delle reti, il protocollo prevede un meccanismo di acknoledgement mediante il quale un consumer notifica il broker dell'avvenuta ricezione di un messaggio. La notifica può essere effettuata sia automaticamente dal broker che manualmente attraverso determinate funzioni. Si noti che l'acknoledgement non serve ad avvisare il sender dell'avvenuta ricezione del messaggio: esso viene utilizzato per avvisare il broker dell'avvenuta ricezione consentendo a quest'ultimo di rimuovere dalla coda, in maniera sicura, il messaggio. Code, exchange e bindings sono definite *entità AMQP*.

2.3.1 Exchange

Gli exchange sono le entità AMQP che ricevono i messaggi e li instradano verso una o più code. L'algoritmo di routing usato dipende dal tipo di exchange e dai bindings. Lo standard specifica quattro tipi di exchange:

- Direct exchange;
- Fanout exchange;
- Topic exchange;
- Headers exchange.

Direct exchange

Un direct exchange utilizza una routing key per l'instradamento dei messaggi. Il suo funzionamento è semplice: una coda è collegata ad un exchange con una certa binding key, quando un messaggio giunge all'exchange esso viene inoltrato alla coda soltanto se la routing key del messaggio corrisponde a alla binding key. Il suo funzionamento è rappresentato nella Figura 2.2.
L'utilizzo di un direct exchange, nel sender, è preceduto dalla sua creazione seguito dalla specifica nell'atto di pubblicazione di un messaggio:

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");
...
channel.basicPublish(EXCHANGE_NAME, routing_key, ...);
```

Opzionalmente, è possibile creare una coda sulla quale memorizzare i messaggi inviati poiché nel caso in cui non ci siano code collegate all'exchange il messaggio verrà scartato automaticamente dal broker.
Per quanto riguarda il receiver, basta creare una coda con un qualsiasi nome, è possibile ottenere un nome in maniera automatica come nell'esempio seguente, e collegarla con l'exchange opportunamente dichiarato, il cui nome e tipo devono coincidere con quelli dichiarati nel sender:

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");
String queueName = channel.queueDeclare().getQueue();
...
channel.queueBind(queueName, EXCHANGE_NAME, routing_key);
```

Default exchange

Il default exchange è un direct exchange senza nome, stringa vuota, creato di default dal broker ad ogni avvio. Ogni coda che viene creata è automaticamente collegata a questo exchange con una routing key uguale al nome della coda stessa.
L'applicazione illustrata nel paragrafo precedente fa uso di un default exchange.

Fanout exchange

Un fanout exchange inoltra messaggi a tutte lo code ad esso associate indipendentemente dalla routing key, che viene ignorata. Se n code sono collegate ad un fanout exchange, quando un nuovo messaggio viene pubblicato su quell'exchange, una copia del messaggio è inviata a tutte le n code. Il suo funzionamento è rappresentato nella Figura 2.3.
Le istruzioni utilizzate per utilizzare un fanout exchange sono simili a quelle

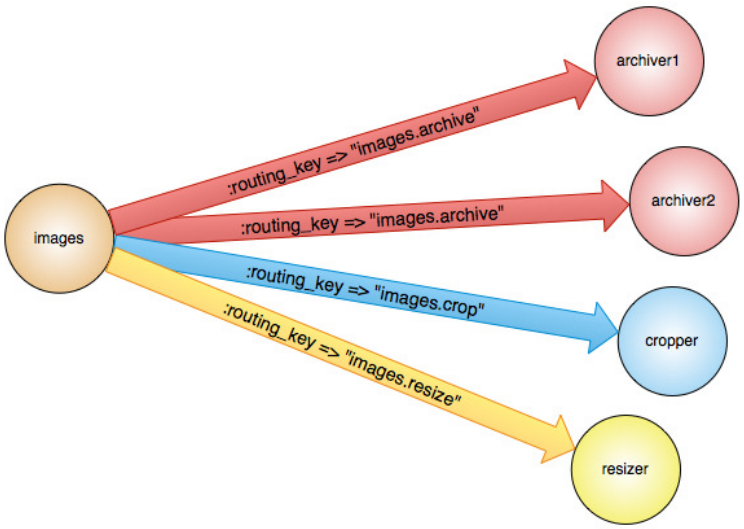

Figura 2.2: Direct exchange

viste per il direct exchange, sia per il sender che per il receiver, tranne che per la creazione:

```
channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
```

Tutte le parti che richiedono l'inserimento di una routing key possono essere sostituite da stringhe vuote dal momento che esse vengono ignorate.

Topic exchange

Il funzionamento di un topic exchange è simile a quello di un direct, un messaggio inviato con una particolare routing key verrà inoltrato a tutte quelle code la cui binding key corrisponda con la routing key. La differenza con il direct exchange risiede nell'utilizzo di routing key e di binding key più complesse, formate da più parole separate da punti. Esiste inoltre la possibilità di utilizzare caratteri speciali quali asterisco * e cancelletto # che hanno una particolare semantica: il primo permette qualsiasi corrispondenza in quella parte della routing key mentre il secondo può essere usato per sostituire zero o più parole e non considera l'asterisco come carattere di separazione. La Figura 2.4 ne illustra il funzionamento. La creazione di questo exchange è effettuata mediante l'istruzione:

```
channel.exchangeDeclare(EXCHANGE_NAME, "topic");
```

Il codice restante è identico a quanto visto in precedenza, a parte per la forma delle routing key.

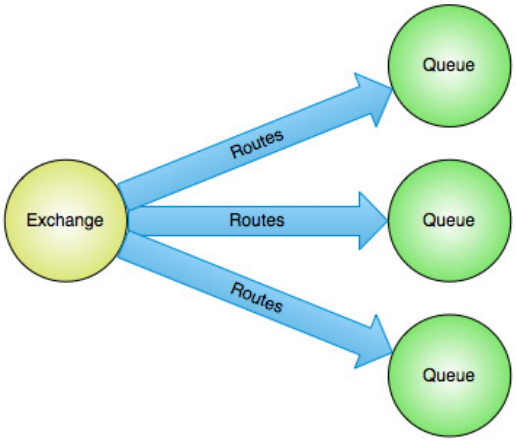

Figura 2.3: Fanout exchange

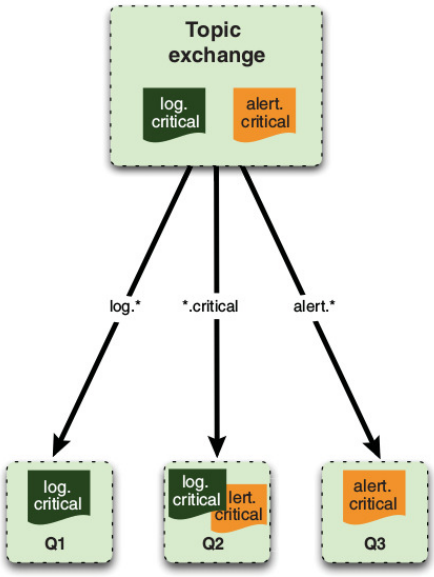

Figura 2.4: Topic exchange

Headers exchange

Gli headers exchange sono progettati per instradare messaggi in base ad informazioni presenti negli header dei messaggi piuttosto che nelle routing key, le quali sono appunto ignorate. Un messaggio è inoltrato ad una certa coda soltanto se la sua regola di binding coincide con uno o più valori presenti nell'header del messaggio. Il suo comportamento può essere assimilato a quello di un direct exchange con routing key che non sono necessariamente stringhe ma, ad esempio, interi o hash.
Per approfondimenti sull'implementazione si veda [5].

2.3.2 Bindings

I bindings sono regole utilizzate dagli exchange per instradare i messaggi alle code. Per fare in modo che un exchange E possa instradare messaggi alla coda Q, Q deve essere collegata a E attraverso una regola di binding. I bindings possono essere caratterizzati da routing key nel caso di determinati tipi di exchange. Una routing key è una sorta di filtro che ha lo scopo di selezionare solo certi messaggi pubblicati da un exchange per essere inoltrati alla coda associata.

Capitolo 3

Confronto con ActiveMQ

In questo capitolo verrà effettuato un confronto tra due midlleware molto diffusi: RabbitMQ e ActiveMQ. Il confronto verrà effettuato sia dal punto di vista delle caratteristiche che dal punto di vista prestazionale.

3.1 Caratteristiche

Entrambi i broker sono progettati per essere altamente scalabili, robusti e affidabili. ActiveMQ presenta alcuni problemi di scalabilità a causa della sua architettura [4] ma mostra i suoi limiti solo per carichi molto elevati. Sebbene queste defezioni il broker può contare comunque su una certa stabilità acquisita nei diversi anni di servizio che lo portano ad essere ancora una soluzione largamente adottata.
ActiveMQ è scritto in Java mentre RabbitMQ in Erlang. Entrambi supportano lo standard JMS, sebbene RabbitMQ attraverso delle API che sfruttano il protocollo AMQP. Per quanto riguarda il supporto ad AMQP si può osservare che, allo stato attuale, ActiveMQ risulta essere compatibile con la versione 1.0 mentre RabbitMQ con la versione 0-9-1, sebbene il supporto per la versione 1.0 sia in fase di sviluppo [4]. Entrambi i broker supportano funzionalità di clustering e di persistenza dei messaggi, e possono essere arricchiti con numerosi plugin, tra i quali delle interfacce grafiche in HTML per la gestione ed il monitoraggio.
Queste sono soltanto alcune delle molteplici caratteristiche che i broker supportano. Per ulteriori dettagli consultare le relative pagine sui siti ufficiali.

3.2 Prestazioni

In questa sezione si affronterà il confronto tra i due broker da un punto di vista prestazionale mediante dei test nei quali verrà misurato il tempo medio di risposta di un'applicazione receiver in diverse condizioni di funzionamento. Il paradigma utilizzato per i test è quello delle code di messaggi. Ciascun valore temporale è preso come media dei tempi ottenuti da 5 ripetizioni dello stesso test nelle stesse condizioni di funzionamento. Ad ogni serie di test è accompagnato il calcolo di una deviazione standard. Gli applicativi sono scritti in Java e verranno eseguiti su macchina Linux, distribuzione Fedora 21, modello HP Compaq nx7300 dotata di CPU Intel Celeron M con 1,5 GB di memoria RAM. La macchina virtuale è openJDK in versione 0.8.1. La versione usata di RabbitMQ è la 3.1.5, reperita dai repository ufficiali di Fedora, mentre per ActiveMQ viene utilizzata la versione 5.12.0. A parte la creazione delle code di test, tutti i broker eseguono con la loro configurazione di default.

3.2.1 Numero di sender

Il test consiste nella misurazione del tempo di risposta al variare del numero di sender. I sender sono gestiti mediante un un pool di thread.

RabbitMQ

Ogni sender invia 100 messaggi da 119 byte di payload. Come si può osservare dalla Figura 3.1, il tempo medio di risposta aumenta linearmente fino a 600 sender, il tempo di risposta a 600 sender è circa 6 volte il tempo ottenuto con 100 sender. A partire da 800 sender l'andamento del tempo di risposta non è più lineare.

ActiveMQ

In questo caso l'unica variazione rispetto al test effettuato con RabbitMQ è data dal numero di messaggi per sender, sceso a 10 unità. Come si può osservare dalla Figura 3.2, il numero di messaggi per sender è stato ridotto a causa degli alti tempi di risposta del broker. Si può comunque notare come i tempi aumentano con una certa linearità anche dopo i 600 sender.

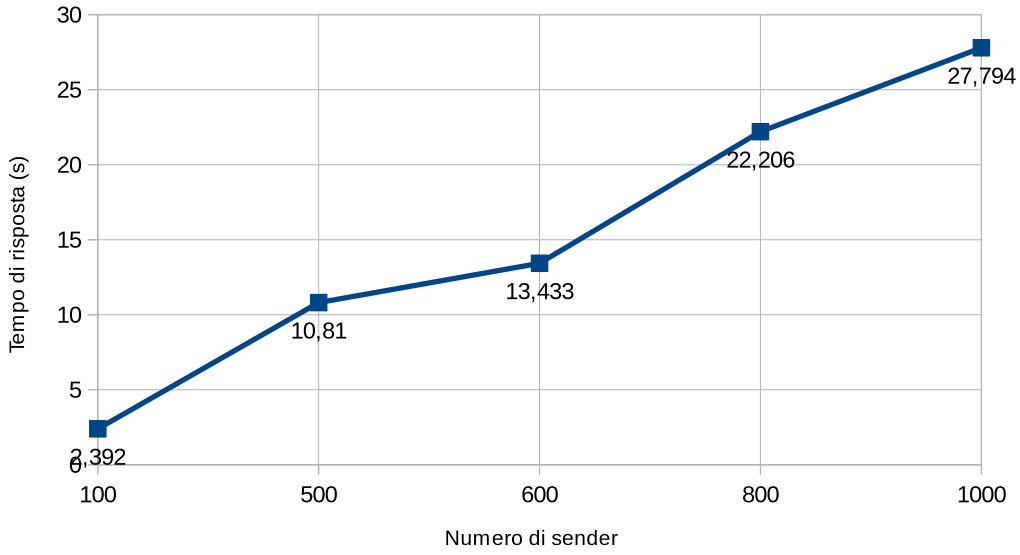

Figura 3.1: Variazione numero di sender per RabbitMQ

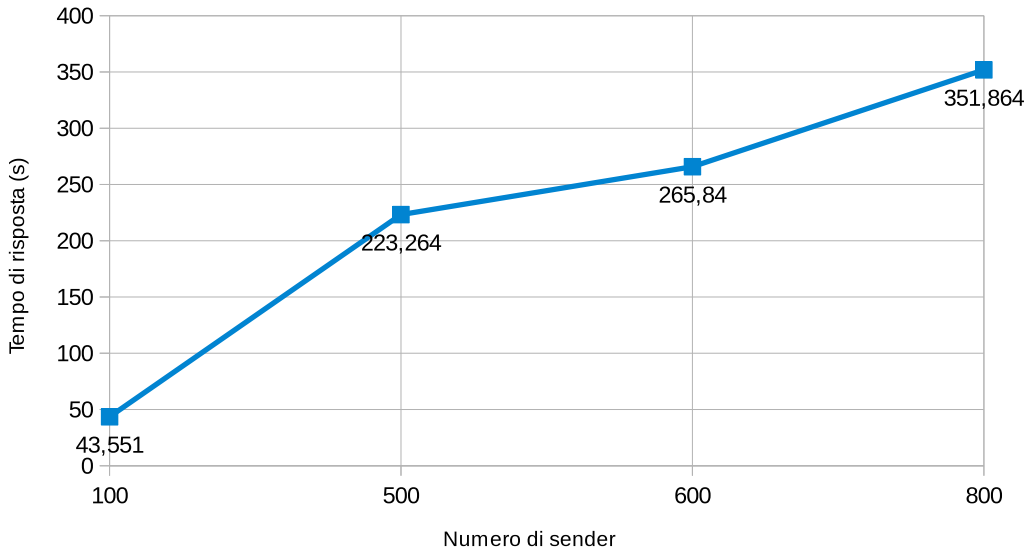

Figura 3.2: Variazione numero di sender per ActiveMQ

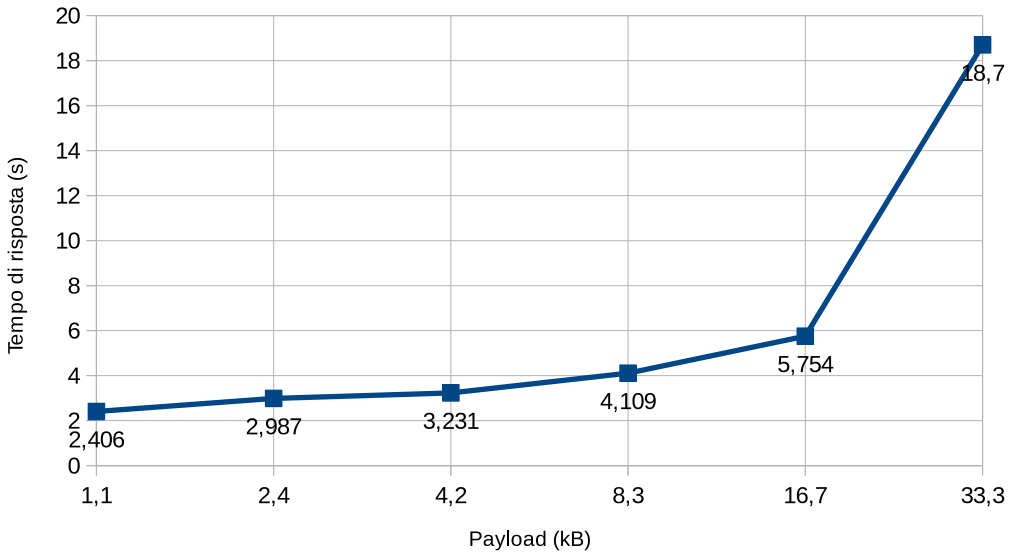

Figura 3.3: Variazione del payload per RabbitMQ

3.2.2 Payload del messaggio

Il test consiste nella misurazione del tempo di risposta al variare della dimensione del payload dei messaggi. Il numero di sender e di messaggi per sender viene mantenuto costante.

RabbitMQ

Il payload dei messaggi viene aumentato ad ogni test fino ad un massimo di 33,3 kB. Il numero di sender rimane fissato a 100 unità così come il numero di messaggi per sender, fissato a 100 messaggi. Dalla Figura 3.3 si può notare un incremento poco significativo dei tempi di risposta fino ad un payload di 16,7 kB. Oltre questo valore si può notare un aumento netto dei tempi di risposta: ciò è dovuto alla quantità di memoria necessaria per memorizzare i messaggi, oltre 300 MB, che porta il broker ad utilizzare il supporto di memorizzazione di massa, le cui operazioni di I/O sono più lente di quelle in memoria centrale e quindi aumentano di molto i tempi di risposta.

ActiveMQ

Anche in questo caso l'unica variazione rispetto al test con RabbitMQ è data dal numero di messaggi per sender, sceso a 10 unità, a causa degli elevati

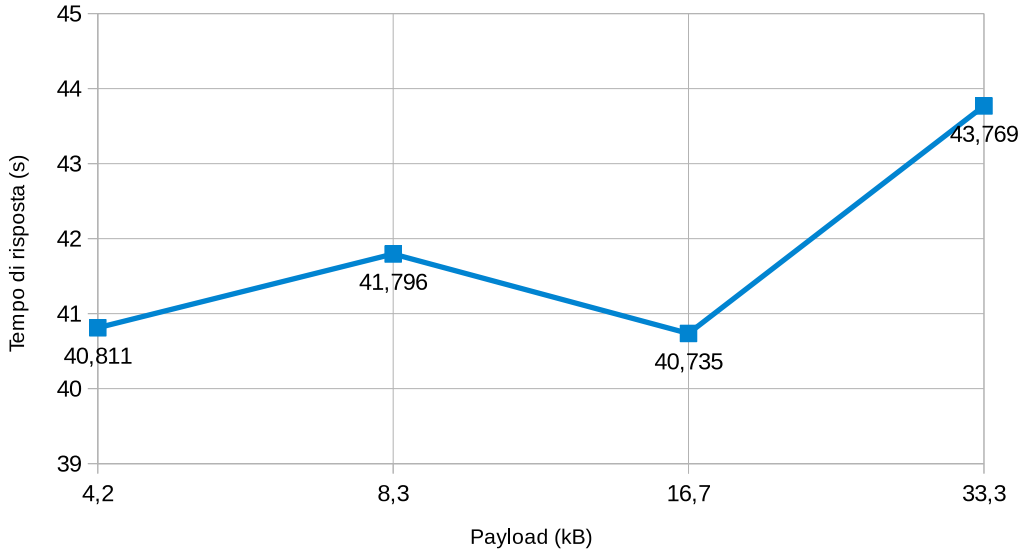

Figura 3.4: Variazione del payload per ActiveMQ

tempi di risposta. Come si può osservare dalla Figura 3.4, anche riducendo il numero di messaggi per sender, i tempi di risposta sono più alti di quelli con RabbitMQ sebbene aumentino con una certa linearità.

3.2.3 Risultati

Dai diversi test emerge una maggiore capacità di RabbitMQ di gestire carichi pesanti mantenendo comunque buoni tempi di risposta.
Altra cosa interessante da notare è la capacità di entrambi di lavorare bene con messaggi di grandi dimensioni. In particolare ActiveMQ sembra garantire tempi pressoché costanti, sebbene con un numero totale di messaggi minore rispetto a RabbitMQ.
Anche se i risultati potrebbero far pensare a RabbitMQ come soluzione da adottare a causa dei maggiori tempi di risposta di ActiveMQ, in realtà la scelta non è poi così scontata poiché ci sono tanti altri fattori, come tecnologie usate, documentazione, supporto che necessitano di essere prese in considerazione ma che non è stato possibile affrontare in questo elaborato.

Conclusioni

Come sempre accade nell'ambito dell'IT non esiste un'unica soluzione da adottare in qualsiasi contesto. Il miglior approccio è dettato dai requisiti richiesti e dal particolare problema. ActiveMQ è una soluzione presente da molto tempo sul mercato, per cui ha sviluppato una certa maturità che lo porta ad essere la prima scelta in termini di affidabilità. Se i requisiti sono orientati in termini di throughput, si può pensare all'adozione di soluzioni più moderne come RabbitMQ e altri. E' da sottolineare la presenza di tante altre soluzioni molto valide che per motivi di tempo non è stato possibile affrontare in questo elaborato ma che ricoprono una certa importanza nell'universo MOM.

Bibliografia

[1] Jason J. W. Williams Alvaro Videla. *RabbitMQ in Action, distributed messaging for everyone.* 2012.

[2] S. Russo C. Savy D. Cotroneo A. Sergio. *Introduzione a CORBA.* McGraw-Hill, 2002.

[3] Margaret Rouse. Erlang programming language. www.whatis.techtarget.com/definition/Erlang-programming-language, September 2007.

[4] http://www.predic8.com/activemq-hornetq-rabbitmq-apollo-qpid-comparison.htm.

[5] stackoverflow.com/questions/19240290/how-do-i-implement-headers-exchange-in-rabbitmq-using-java.

[6] Clustering and network partitions. www.rabbitmq.com/partitions.html.