Messaggistica (Messaging)

- Componenti distribuiti, o varie applicazioni, devono comunicare attraverso la rete
  - La rete è *inaffidabile*, ciascun elemento della rete può causare ritardi o interruzioni. La rete è *lenta* (rispetto ad una comunicazione locale)
- Le applicazioni che vogliono comunicare usano linguaggi di programmazione diversi, formati di dati diversi, una *soluzione di integrazione* deve poter interfacciare queste differenti tecnologie
- La messaggistica (messaging) è una tecnologia che abilita comunicazioni *asincrone* fra programmi con consegne *affidabili*. I programmi comunicano inviando fra loro pacchetti di dati chiamati *messaggi*
  - I canali, detti anche *code*, sono percorsi logici che connettono programmi e trasmettono messaggi

Obiettivi

- Il messaggio è qualche tipo di dato (stringa, byte, record, oggetto)
- Un messaggio contiene due parti un header e un body
  - L'header contiene informazioni sul messaggio (chi lo ha mandato, dove va, etc.)
  - Il body contiene i dati trasmessi
- Le architetture di messagistica asincrona sono potenti
  - Un middleware di messagistica coordina e gestisce l'invio e la ricezione di messaggi, trasmettendo messaggi in modo affidabile
  - Poiché la rete è inaffidabile e può non riuscire a mandare in modo appropriato i dati, un sistema di messagistica supera queste limitazioni ritrasmettendo il messaggio finché ci riesce

Vantaggi Dei Messaggi

- Abilita la comunicazione ed il trasferimento dati fra applicazioni
  - All'interno dello stesso processo, si possono condividere i dati in memoria, invece per inviare dati ad un altro host bisogna copiarli. Significa dover serializzare gli oggetti e mandarli attraverso la rete
- Integrazione di piattaforme e linguaggi
  - Un sistema di messagistica può essere un traduttore universale fra le applicazioni che si basano su linguaggi e piattaforme diverse
- Comunicazione asincrona
  - La messagistica permette un approccio *send and forget*, chi invia non deve aspettare che il destinatario riceva ed elabori il messaggio, non deve neppure aspettare che il sistema sottostante invii il messaggio
  - Una volta che il messaggio è stato memorizzato, chi invia può fare altre cose mentre il messaggio è trasmesso in background. La risposta è un altro messaggio che è rilevato con un meccanismo di *callback*

Vantaggi Dei Messaggi

- Temporizzazione variabile
  - Con comunicazioni sincrone, il chiamante deve aspettare che il destinatario completi l'elaborazione prima di poter ricevere il risultato e continuare. Il chiamante può fare chiamate con velocità pari alla velocità di elaborazione del destinatario
  - Con comunicazioni asincrone, il richiedente può mandare tante richieste alla propria velocità e il ricevente li consumerà con un ritmo diverso
- Tante Chiamate
  - Con le chiamate remote, troppe chiamate ad un singolo ricevente possono sovraccaricare il ricevente. La comunicazione asincrona abilita il ricevente a controllare la frequenza di consumo delle richieste, in modo da non sovraccaricarsi
- Comunicazione affidabile
  - La messagistica usa un approccio store and forward per i messaggi

RabbitMQ

- RabbitMQ è un broker di messaggi e un server di code che può essere usato da applicazioni per condividere dati attraverso un protocollo comune, o per accodare lavori per opportuni processi
- RabbitMQ implementa lo standard aperto AMQP (Advanced Message Queuing Protocol)
- RabbitMQ svolge il ruolo di broker fra l'app e il server con cui vuole dialogare l'app
- I Produttori creano messaggi e li pubblicano (ovvero li inviano) ad un servizio broker (RabbitMQ)
- Il messaggio ha due parti: payload, e label. Il payload è quel che si vuole trasmettere. La label descrive il payload, così che RabbitMQ determina chi dovrà avere una copia del messaggio
- AMQP descrive il messaggio con una label (nome dello scambio e opzionalmente un tag sull'argomento) e lascia che sia RabbitMQ a mandarlo ai riceventi interessati basandosi sulla label

RabbitMQ

- La comunicazione è fire-and-forget e unidirezionale (chi manda non si aspetta una risposta)
- I *Consumatori* si attaccano ad un broker e si sottoscrivono ad una coda. Quando un messaggio arriva in una coda, RabbitMQ lo manda a uno dei consumatori in ascolto
- Il consumatore riceve solo il payload e non l'etichetta. RabbitMQ non dice neppure chi è stato il *Produttore*
- Un *canale* AMQP è creato dall'applicazione ed è una *connessione* virtuale dentro la connessione TCP. Ogni canale ha un ID univoco assegnato ad esso
- La pubblicazione di un messaggio, la sottoscrizione ad una coda o la ricezione di un messaggio, sono tutte operazioni fatte su un canale
- Una connessione TCP è costosa per il sistema e richiede tempo per essere impostata, un canale è veloce da impostare e non vi è limite sul numero di canali

RabbitMQ

- Per poter mandare un messaggio si hanno tre parti: scambi (*exchange*), code (*queue*) e connessioni (*binding*)
- Gli scambi sono i punti dove i produttori pubblicano i messaggi
- Le code prendono messaggi, che saranno ricevuti dai consumatori. I messaggi in una coda aspettano di essere consumati. I consumatori ricevono messaggi: (i) da una coda su cui si sono iscritti, o (ii) se richiedono un singolo messaggio da una coda
- Le connessioni permettono ai messaggi di viaggiare da uno scambio a una certa coda
- Se una coda ha uno o più consumatori iscritti, i messaggi saranno mandati immediatamente ai consumatori
- Se un messaggio arriva ad una coda senza iscritti, il messaggio attende in coda e quando un consumatore si iscrive gli verrà inviato

Code di RabbitMQ

- Se una coda ha tanti consumatori, i messaggi ricevuti dalla coda sono serviti in modo round-robin ai consumatori. Ogni messaggio è inviato ad un solo consumatore
- Il consumatore che lo riceve manda un ack di ricezione messaggio e RabbitMQ rimuove il messaggio dalla coda
- Se un consumatore non manda un ack, RabbitMQ considera che il consumatore non è pronto e non gli manda altri messaggi
  - L'applicazione può rallentare gli ack, se sta facendo altro, e non verrà sovraccaricata di messaggi
- Se il consumatore si disconnette da RabbitMQ prima di mandare l'ack, RabbitMQ manda il messaggio ad un altro consumatore
- Se il consumatore manda un comando di reject del messaggio, e il parametro requeue è true, RabbitMQ manda il messaggio ad un altro consumatore, mentre se requeue è false il messaggio viene scartato

Code di RabbitMQ

- Sia i consumatori che i produttori possono creare code
- Un consumatore non può creare una coda se è iscritto ad un'altra coda nello stesso canale
- Il nome della coda, fornito al momento della creazione, permette ai consumatore di iscriversi ad essa
- Le code forniscono
  - Un posto dove i messaggi aspettano per essere consumati
  - Un mezzo per effettuare bilanciamento del carico. Agganciando alle code un certo numero di consumatori, RabbitMQ distribuisce in modo round-robin i messaggi ad essi
  - Il punto di arrivo dei messaggi

Esempio 1

- Un produttore (P) manda un messaggio ad una certa coda
- Un consumatore (C) riceve il messaggio arrivato sulla coda

Produttore

- Il produttore (`Send.java`) di un messaggio si connette a RabbitMQ, crea un canale, crea una coda, e manda un messaggio (tipo `String`)

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");              // nome della macchina del broker
Connection connection = factory.newConnection();          // crea connessione
Channel channel = connection.createChannel();             // crea canale

String QUEUE_NAME = "queue-hello";

channel.queueDeclare(QUEUE_NAME, false, false, false, null); // crea coda

channel.basicPublish("", QUEUE_NAME, null, "hello".getBytes());//invia hello
```

- `queueDeclare()` crea la coda con i parametri: `queue`, `durable` (sopravvive se si fa ripartire il server), `exclusive` (per questa connessione soltanto), `autoDelete` (cancellabile dal server se non usata)
- `basicPublish()` pubblica un messaggio, ha i parametri: `exchange`, `routingKey`, `props`, `body`

Consumatore

- Il consumatore (`Recv.java`), crea la connessione, il canale, e la coda, in modo analogo al produttore
- Il consumatore registra un metodo di callback che, quando un messaggio è disponibile, viene chiamato

```
Consumer consumer = new DefaultConsumer(channel) {
   @Override                                          // metodo di callback
   public void handleDelivery(String consumerTag, Envelope envelope,
         AMQP.BasicProperties properties, byte[] body) throws IOException {
      String message = new String(body, "UTF-8");
      System.out.println("Ricevuto '" + message + "'");
   }
};
channel.basicConsume(QUEUE_NAME, true, consumer); // registra consumatore
```

- `basicConsume()` registra un consumatore alla coda, ha i parametri: `queue`, `autoAck`, `callback`
- `handleDelivery()` è il metodo di callback, previsto nell'interfaccia `Consumer` ed implementato dallo specifico consumatore

Pattern Event-Driven Consumer

- Intento: Un'applicazione necessita di consumare messaggi appena questi sono consegnati
- Problema
  - Se i consumatori controllano lo stato del canale in un ciclo, quando il canale è vuoto i consumatori bloccano il thread o consumano tempo di processore
  - I consumatori possono controllare la frequenza di lettura messaggi, ma sprecano risorse se non vi è niente da leggere
- Soluzione
  - Anziché chiedere continuamente al canale, il canale informa i consumatori quando vi è un messaggio
  - Il messaggio è automaticamente consegnato non appena arriva sul canale. La consegna del messaggio è come un evento che fa avviare l'esecuzione del consumatore

Pattern Event-Driven Consumer

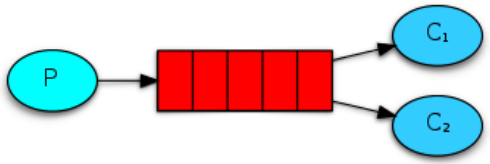
- Il consumatore event-driven è un oggetto che è invocato dal sistema di messaggistica quando un messaggio arriva sul canale del consumatore
- Il messaggio viene passato al consumatore tramite una callback
- L'API fornita dal sistema di messaggistica permette di far conoscere il metodo di callback
- Un consumatore Event-driven consiste di due parti
  - Inizializzazione: l'applicazione crea uno specifico consumatore e lo associa ad un canale di messaggi, quindi dopo l'esecuzione di questo codice il consumatore è pronto a ricevere messaggi
  - Consumo: il consumatore riceve ed elabora un messaggio. Il messaggio è passato come parametro di un metodo di callback del consumatore
- Il consumatore è dormiente fino a quando non arriva un messaggio

Avvio

- Scaricare le librerie amqp-client.jar e le librerie di log slf4j-api.jar
- Scaricare il broker (RabbitMQ server)
- Compilazione
  - javac -cp amqp-client-4.0.2.jar Send.java Recv.java
- Avvio server
  - sbin/rabbitmq-server
- Esecuzione
  - java -cp .:amqp-client-4.0.2.jar:slf4j-api-1.7.21.jar:slf4j-simple-1.7.22.jar Send
  - java -cp .:amqp-client-4.0.2.jar:slf4j-api-1.7.21.jar:slf4j-simple-1.7.22.jar Recv

Esempio 2

- Un produttore invia messaggi ad una coda
- Due consumatori sono avvisati a turno della presenza di un messaggio



Con Due Consumatori

- L'avvio di più consumatori collegati alla stessa coda, permetterà l'esecuzione in round-robin dei metodi di callback dei consumatori, quando i messaggi inviati dal produttore arrivano sulla coda

```
Consumer consumer1 = new DefaultConsumer(channel) {
   @Override
   public void handleDelivery(String consumerTag, Envelope envelope,
           AMQP.BasicProperties properties, byte[] body) throws IOException {
       String message = new String(body, "UTF-8");
       System.out.println("Ricevente uno '" + message + "'");
   }
};
channel.basicConsume(QUEUE_NAME, true, consumer1);

Consumer consumer2 = new DefaultConsumer(channel) {
   @Override
   public void handleDelivery(String consumerTag, Envelope envelope,
           AMQP.BasicProperties properties, byte[] body) throws IOException {
       String message = new String(body, "UTF-8");
       System.out.println("Ricevente due '" + message + "'");
   }
};
channel.basicConsume(QUEUE_NAME, true, consumer2);
```


Gestione Ack

- Per evitare di sovraccaricare consumatori lenti, vogliamo che un consumatore che manda ack lentamente, ricevi meno messaggi di un consumatore veloce
- Per non mandare in automatico ack al ricevimento di un messaggio, si passa false come secondo parametro di `basicConsume()`

```
channel.basicConsume(QUEUE_NAME, false, consumer);
```

- Inoltre, tramite `basicQoS(1)` si imposta il canale per far sì che il consumatore non riceva altri messaggi se non ha ancora mandato un ack per il precedente messaggio

```
channel.basicQos(1);
```

- Quindi, dopo aver concluso il lavoro relativo al messaggio ricevuto, il consumatore chiama `basicAck()` su `Channel`

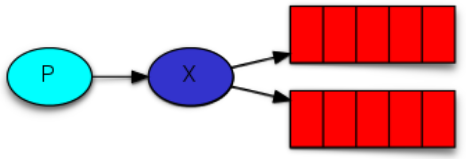
```
channel.basicAck(envelope.getDeliveryTag(), false);
```


Pattern Competing Consumers

- Intento: un'applicazione non riesce ad elaborare messaggi tanto velocemente da stare al passo con quanti ne arrivano sul canale
- Problema
  - Come fare ad elaborare i vari messaggi in parallelo?
  - Se il consumo dei messaggi è lento i messaggi si accumulano sul canale
  - Tanti messaggi potrebbero accumularsi per via di tanti trasmettitori sullo stesso canale o per un malfunzionamento della rete
  - Più canali equivalenti potrebbero non ricevere dati in modo uniforme
- Soluzione
  - Creare consumatori competitivi su un singolo canale che prendono ed elaborano messaggi in modo concorrente
  - In realtà, il sistema di messaggistica determina quale consumatore riceve il messaggio

Esempio 3

- Un produttore manda messaggi ad uno scambio
- Allo scambio sono collegate due code
- Ciascun consumatore collegato alle code riceve i messaggi



Scambi

- Il produttore non invia i messaggi direttamente alla coda, invece li invia ad uno *scambio* (*exchange*)
- Uno scambio da un lato riceve messaggi dai produttori e dall'altro lato li manda alle code
- Nei precedenti esempi si è usato uno scambio di default, identificato tramite la stringa vuota "", e i messaggi sono stati inviati alla coda indicata dal secondo parametro, se esiste

```
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
```

- Lo scambio sa cosa fare con un messaggio ricevuto tramite regole definite per il *tipo di scambio*
- Ci sono i tipi di scambio: *direct*, *topic*, *headers*, e *fanout*
- Il tipo *fanout* manda i messaggi che riceve a tutte le code che conosce

Creazione E Uso Di Scambi

- Per creare uno scambio

```
String EXCHANGE_NAME = "logs";
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
```

- Il produttore pubblica il messaggio su uno scambio

```
String message = "Hello World!";
channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));
```

- Il consumatore prende messaggi da una coda, quindi crea una coda vuota, non-durable, exclusive, auto-deleted

```
String queueName = channel.queueDeclare().getQueue();
```

- Il nome della coda sarà creato dal server (in modo random), non serve far conoscere il nome al produttore
- Per far sì che i messaggi dallo scambio arrivino alla coda bisogna creare una connessione (*binding*) fra la coda e lo scambio

```
channel.queueBind(queueName, EXCHANGE_NAME, "");
```

- Si registra il consumatore alla coda per fargli arrivare messaggi

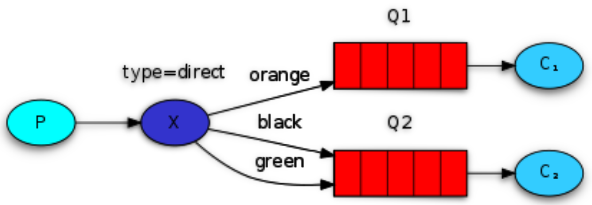
```
channel.basicConsume(queueName, true, consumer);
```


Design Pattern Publish-Subscribe

- Intento: usare i messaggi per annunciare eventi
- Problema
  - Un produttore deve inviare un evento a tutti i riceventi interessati
  - Ogni subscriber è notificato per ciascun evento una sola volta
  - L'evento sarà considerato consumato solo quando tutti i subscriber sono stati notificati, in tal caso l'evento può sparire dal canale
- Soluzione
  - Inviare l'evento al canale Publish-Subscribe, che manda una copia di ciascun evento a ciascun ricevente
  - Il canale in ingresso si divide in più canali di uscita, uno per ciascun subscriber. Quando un evento è pubblicato sul canale, quest'ultimo manda una copia del messaggio a ciascun canale in uscita
  - E' ottimo per intercettare messaggi, per es. per debug: si può creare un log di tutti messaggi senza dover cambiare l'applicazione

Esempio 4

- Un produttore manda vari tipi di messaggi
- Il consumatore collegandosi ad una coda può ricevere solo i messaggi a cui è interessato, che sono filtrati in base al tipo



Message Filter con RabbitMQ

- Per la creazione dello scambio, indicare il tipo direct (secondo parametro)

```
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
```

- Il produttore indica con `basicPublish()` una routing key (secondo parametro), in questo caso `cmd` è il valore della routing key

```
channel.basicPublish(EXCHANGE_NAME, "cmd", null, msg.getBytes("UTF-8"));
```

- Il consumatore crea una coda e la connette indicando la stessa routing key (terzo parametro di `queueBind()`)

```
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName, EXCHANGE_NAME, “cmd”);
```

- Quindi, come fatto in precedenza, si usa

```
channel.basicConsume(queueName, true, consumer);
```

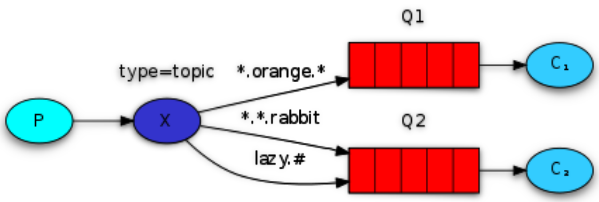
- Il consumatore riceverà dal canale e sulla coda indicata solo i messaggi filtrati secondo la routing key fornita

Design Pattern Message Filter

- Intento: far sì che un componente riceva solo messaggi a cui è interessato
- Problema
  - Il produttore manda vari messaggi di notifica ai consumatori, per comunicazioni di diversa natura
  - Alcuni consumatori potrebbero essere interessati solo a certi messaggi
- Soluzione
  - Usare un filtro di messaggi che elimina i messaggi indesiderati da un canale in base ad un certo criterio

Esempio 5

- Un produttore manda messaggi caratterizzati da varie chiavi
- Il consumatore C1 vuole ricevere solo i messaggi con chiave orange
- Il consumatore C2 vuole ricevere solo i messaggi con chiave rabbit e lazy



Topic Exchange

- Per la creazione dello scambio, indicare il tipo topic (secondo parametro)

```
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);
```

- Il produttore indica con `basicPublish()` una routing key (secondo parametro), di tipo String

```
channel.basicPublish(EXCHANGE_NAME, routingKey, null,
                                                message.getBytes("UTF-8"));
```

- La routing key ha un formato che consiste di tre parole e due punti
  - La prima parola descrive la velocità, la seconda il colore, la terza la specie
  - Es. un messaggio può avere routing key “quick.orange.rabbit”

Topic Exchange

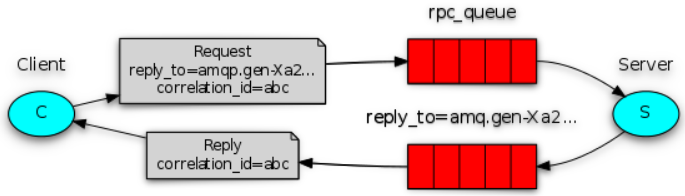
- Il consumatore C1 crea una coda e la connette indicando la binding key di tipo String "*.orange.*" (terzo parametro di `queueBind()`)
- Il consumatore C2 crea una coda e la connette indicando la binding key "*.*.rabbit" e "lazy.#"

```
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName, EXCHANGE_NAME, bindingKey);
```

- Sulla binding key, un * sostituisce una singola parola, ed un # sostituisce zero o più parole

Esempio 6

- Il client fa una richiesta e aspetta una risposta
- E' un esempio di implementazione di RPC su RabbitMQ



Lato Client: Coda Di Callback

- Il client manda un messaggio e vuole ricevere una risposta
- Per ricevere la risposta inviamo insieme alla richiesta un indirizzo di coda di callback

```
String replyQueueName = channel.queueDeclare().getQueue();

BasicProperties props = new BasicProperties
                            .Builder()
                            .correlationId(corrId)
                            .replyTo(replyQueueName)
                            .build();

channel.basicPublish("", reqQueueName, props, message.getBytes("UTF-8"));
```

- `corrId` è una stringa che contiene un identificatore unico
- Il client implementa un metodo di callback che registra sul canale con `basicConsume()`

```
channel.basicConsume(replyQueueName, true, consumer);
```


Lato Server

- Il server manda la risposta alla coda del client, indicata dal secondo parametro di `basicPublish()`, e quindi manda un ack al server

```
public void handleDelivery(String consumerTag, Envelope envelope,
            BasicProperties properties, byte[] body) throws IOException {

   BasicProperties replyProps = new BasicProperties
                .Builder()
                .correlationId(properties.getCorrelationId())
                .build();

   channel.basicPublish("", properties.getReplyTo(), replyProps,
                                                response.getBytes("UTF-8"));

   channel.basicAck(envelope.getDeliveryTag(), false);
}
```

Operazioni Sulla Coda

- E' possibile svuotare una coda del suo contenuto

```
channel.queuePurge("queue-name");
```

- E' possibile cancellare una coda

```
channel.queueDelete("queue-name");
```

- E' possibile cancellare una coda solo se è vuota

```
channel.queueDelete("queue-name", false, true);
```

- E' possibile cancellare una coda solo se non è usata, non ha consumatori

```
channel.queueDelete("queue-name", true, false);
```


Messaggi Che Scadono

- E' possibile pubblicare un messaggio che si autodistrugge dopo un timeout

```
channel.basicPublish("exchangeName", "routingKey",
          new BasicProperties.Builder()
                             .expiration("60000")
                             .build(),
          "message".getBytes());
```


Pull Di Messaggi

- Per prendere messaggi esplicitamente da una coda il consumatore può invocare `basicGet()`

```
GetResponse response = channel.basicGet(queueName, true);
```

- Dall'istanza di `GetResponse` restituita da `basicGet()` si possono leggere header e body del messaggio

```
if (null != response) {
   BasicProperties props = response.getProps();
   byte[] body = response.getBody();
   long deliveryTag = response.getEnvelope().getDeliveryTag();
   String message = new String(body, "UTF-8");
}
```


Recupero Da Fallimenti Della Rete

- Se la connessione fra i client e i nodi di RabbitMQ fallisce, i client Java supportano il recupero automatico della connessione e della topologia (ovvero code, scambi, connessioni e consumatori)
- Il processo di recupero è automaticamente abilitato
- Si avvia il processo di recupero se viene lanciata un'eccezione I/O, o se l'operazione di lettura dalla socket incontra un timeout