

Rapport MRI : TP3, MOM

Antoine LEVAL, Antoine POSNIC

April 6, 2018

Abstract

Ce compte rendu concerne le TP3 de MRI, "Message-oriented middleware", source : <https://cours.4x.re/pr/rpc-et-mom/tp4-mom>. Vous pouvez retrouver la version finale du TP sur ce dépôt Github : https://github.com/Viinyard/IPR_TP3_maven

1 Introduction

Ce TP sur les MOM sera développé avec le broker RabbitMQ qui implémente le protocole AMQP. RabbitMQ se charge d'accepter, de stocker et de renvoyer les messages. On peut faire l'analogie de la poste, rabbitMQ est donc à la fois la boîte au lettre (qui réceptionne les message), le bureau de poste (qui stocke les messages, dans une queue), et le facteur, qui renvoie les messages.

RabbitMQ est donc composé de :

Producer : un programme qui envoie (produit) des messages.

Consumer : un programme qui reçoit (consume) des messages.

Queue : Une file qui stocke les messages (limité seulement par la mémoire du serveur et les limites du stockage).



Figure 1: Producer, queue, consumer

Sur ce schéma P représente le producer, C représente le consumer et en rouge est représenté la queue. Une queue n'est pas liée à un producer ou à un consumer, plusieurs producers peuvent envoyer des messages à une queue et plusieurs consumers peuvent recevoir des messages d'une queue.

RabbitMQ propose différents patterns d'envoi et de réception de messages, dans ce TP nous allons nous intéresser au pattern "publish/subscribe".

2 Mise en place de l'environnement

Pour la mise en place de ce TP nous avons 6 possibilités d'installation pour rabbitMQ, nous avons choisis de procéder avec Docker qui permet de lancer un serveur rabbitMQ en une commande sur un conteneur totalement indépendant du système d'exploitation.

Une fois docker installé (<https://store.docker.com/search?type=edition&offering=community>).

Il suffit de lancer la commande suivante dans un terminal :

```
sudo docker run -d --hostname my-rabbit --name some-rabbit -p 8081:15672 -p 8082:5672 -e
RABBITMQ_DEFAULT_VHOST="mri" -e RABBITMQ_DEFAULT_USER="mri" -e
RABBITMQ_DEFAULT_PASS="64GbL3k7uc33QCtc" rabbitmq:3-management
```

Une fois lancé, il est possible d'accéder à l'interface d'administration à l'adresse suivante : <http://localhost:8081>.

Afin de pouvoir utiliser rabbitMQ dans Eclipse nous avons ajouté la dernière version du client Java pour rabbitMQ dans notre buildpath : <https://www.rabbitmq.com/releases/rabbitmq-java-client/v3.6.14/>. Ensuite nous avons démarré les parties du TP à réaliser sur l'exemple fournis dans les tutoriels rabbitMQ "publish/subscribe" pour Java.

3 Le modèle publish/subscribe

Dans ce modèle un producer peut délivrer des messages à plusieurs consumers. Toute les instances de consumer recevront les messages. Dans rabbitMQ le producer n'envoie pas les messages directement à une queue, il envois ses message à un "exchange".

En publiant les messages à un exchange, cela permet d'obtenir différent comportement, car celui-ci sait comment traiter un message, à quelle(s) queue(s) l'envoyer en fonction de son type (parmis : direct, topic, headers, fanout). A noter qu'il est possible de ne pas déclarer d'échange car il en existe un par défaut, nommé par une chaîne vide et de type "direct".

3.1 L'échange fanout

L'échange fanout distribue les messages qu'il reçoit à toute les queues qu'il connaît.

On commence par créer un échange :

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

Maintenant pour que notre exchanger publie ses messages dans une queue il faut les lier entre eux, il faut les "binder" :

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

On bind donc une queue à un exchange grâce à leurs noms. Dans notre cas où on souhaite que le producer broadcast des messages à tout les consumer, il faut que chaque consumer bind sa propre queue à l'exchanger concerné, ici "logs". Car si plusieurs consumer consommaient une même queue, alors les messages de cette queue serait dispatché entre les différents consumer, ainsi les consumers ne recevraient pas tous les messages, on appelle cela le "load balancing".

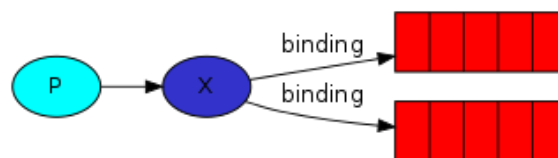


Figure 2: Queue binding

Afin de pouvoir déclarer une nouvelle queue sans risquer d'utiliser un nom de queue déjà utilisé, rabbitMQ propose une méthode pour créer une queue avec un nom aléatoire :

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

Cette méthode crée une queue non durable (qui ne survie pas à un restart du broker), exclusive (utilisé par une seule connexion, supprimée quand la connexion est fermée) et autodelete (queue supprimé lorsque le dernier consumer se désabonne).

Ensuite le producer peut envoyer des messages à l'exchanger qui se chargera de le dupliquer dans chaque queue.

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

3.2 Envoyer la date

Pour la première partie du TP le but était d'envoyer la date toute les secondes.

```
fr>>istic>>date>>EnvoyerDate.java
```

```
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
while(loop) {
    String message = getDate();
    channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));
    System.out.println(" [x] Sent '" + message + "'");
    Thread.sleep(1000);
}
```

fr » istic » date » RecevoirDate.java

```
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName, EXCHANGE_NAME, "");
Consumer consumer = new DefaultConsumer(channel) {...};
channel.basicConsume(queueName, true, consumer);
```

On lance alors :

- 1 Producer : EnvoyerDate.java
- 3 Consumer : RecevoirDate.java





Et on constate sur la console d'administration que l'échange "date" de type fanout a été créé.

date	fanout		1.0/s	3.0/s
-------------	--------	--	-------	-------

Figure 3: Exchange fanout

On constate aussi que quatre connexions sont actives (un producer et trois receiver). On peut observer que le taux de message publié coïncide avec le nombre de message délivré par connexion. Chaque receiver reçoit les mêmes messages. Ainsi qu'avec le nombre de message publié dans chaque queue.

On peut également remarquer dans les features que les queues sont bien en AD (AutoDelete) et Excl (Exclusive).

Overview			Message rates	
Channel	User name	State	publish	deliver / get
172.17.0.1:50666 (1)	mri	 running	1.0/s	
172.17.0.1:50668 (1)	mri	 running		1.0/s
172.17.0.1:50670 (1)	mri	 running		1.0/s
172.17.0.1:50672 (1)	mri	 running		1.0/s




Overview			Message rates	
Name	Features	State	incoming	deliver / get
amq.gen-LcDdSWg1qCGbYgHN36Ahsw	AD Excl	 running	1.0/s	1.0/s
amq.gen-gby8_yYV77YqQ79OFYVvIg	AD Excl	 running	1.0/s	1.0/s
amq.gen-iRLnOJK65Z_P_20oOW2J2w	AD Excl	 running	1.0/s	1.0/s

Figure 4: Connection et queues

Le producer produit 1 message par seconde, les trois consumers chacun 1 message par seconde également. Les messages sont bien reproduit sur chacune des queues. On peut vérifier cela sur une trace d'exécution :







```
// EnvoyerDate
[x] Sent 'Thu Apr 05 23:37:05 CEST 2018'
// RecevoirDate 1
[x] Received 'Thu Apr 05 23:37:05 CEST 2018'
// RecevoirDate 2
[x] Received 'Thu Apr 05 23:37:05 CEST 2018'
```

3.3 Load Balancing

Pour le load balancing le seul code qui change sur situe dans le consumer `fr>>istic>>date>>lb` `RecevoirDate.java` au moment où on déclare la queue.

```
String queueName = "file_date";
channel.queueDeclare(queueName, false, false, false, null);
channel.queueBind(queueName, EXCHANGE_NAME, "date");
```

En effet le but est de partager une même queue entre plusieurs consumer, voyons alors les conséquences en instanciant 2 consumer sur une même queue `"file_date"`. Vérifions grâce à l'interface d'administration le résultat.

Overview			Message rates		
Channel	User name	State	publish	deliver / get	
172.17.0.1:50666 (1)	mri	 running	1.0/s		
172.17.0.1:50668 (1)	mri	 running		1.0/s	
172.17.0.1:50670 (1)	mri	 running		1.0/s	
172.17.0.1:50672 (1)	mri	 running		1.0/s	
172.17.0.1:50714 (1)	mri	 running		0.40/s	
172.17.0.1:50716 (1)	mri	 running		0.60/s	











Overview			Message rates			
Name	Features	State	incoming	deliver / get	ack	
amq.gen-LcDdSWg1qCGbYgHN36Ahsw	 	 running	1.0/s	1.0/s	0.00/s	
amq.gen-gby8_yYV77YqQ790FYVvIg	 	 running	1.0/s	1.0/s	0.00/s	
amq.gen-iRLnOJK65Z_P_20oOW2J2w	 	 running	1.0/s	1.0/s	0.00/s	
file_date		 running	1.0/s	1.0/s	0.00/s	

Figure 5: Load Balancing

L'exchanger publie bien un message par seconde dans chaque queue, peu importe le nombre de consumer. Et la queue `"file_date"` delivre bien un message par seconde, alors qu'il y a deux instances de consumer sur cette même queue.

Dans les connexions cela se vérifie par le fait que les deux derniers consumers ne consomment qu'environ un demi message par seconde. Sur la trace d'exécution on voit que chaque consumer ne reçoit qu'un message sur deux :

```
[x] Received 'Fri Apr 06 01:54:17 CEST 2018'
[x] Received 'Fri Apr 06 01:54:19 CEST 2018'
[x] Received 'Fri Apr 06 01:54:21 CEST 2018'
```

4 Routing

Le but du routing est de permettre aux consumers de s'abonner seulement aux queues qui les intéressent. Pour cela il faut déjà configurer l'échanger en mode direct :

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

En effet le système précédent distribue tout les messages à toutes les queues. Le fonctionnement de l'échanger en mode direct fonctionne très simplement, il envoie les messages dans la queue qui correspond parfaitement à la clé de routage du message.

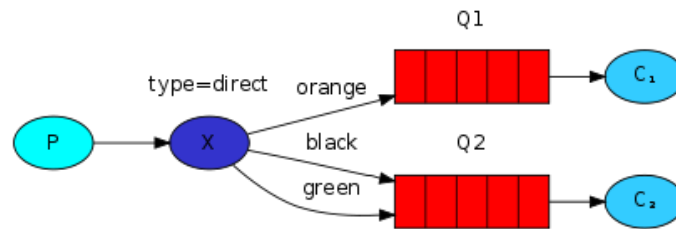


Figure 6: Direct exchange

Une queue peut se binder (s'abonner) à plusieurs routes, par exemple Q2 est abonné à black et green. Et plusieurs queue peuvent s'abonner à une même route.

En fait on retrouve à peu près le même mécanisme qu'en mode fanout, l'échanger va envoyer le message à toute les queues qui sont bindé à lui et pour lesquels les clés de routage du message et de ou des queues correspondent. Si plusieurs consumers partagent une même queues le résultat sera identique au mode fanout.

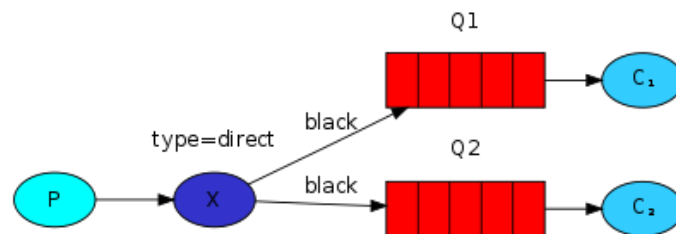


Figure 7: Direct exchange multiple

Ici les messages avec la clé de routage "black" sont bien dupliqué entre Q1 et Q2.

Pour cette partie le but est de configurer le producer pour envoyer deux types de date, les dates normale, mais aussi les dates GMT, avec chacune sa clé de routage pour permettre aux consumers de ne recevoir que celles qui les intéressent. Voyons les changement à opérer :

```
fr>>istic>>date>>route>>EnvoyerDate.java
```

```
channel.basicPublish(EXCHANGE_NAME, "locale", null, getDate().getBytes("UTF-8"));
channel.basicPublish(EXCHANGE_NAME, "gmt", null, getDateGMT().getBytes("UTF-8"));
```

```
fr>>istic>>date>>route>>RecevoirDateGMT.java
```

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

```
fr>>istic>>date>>route>>RecevoirDate.java
```

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

On a gardé les queues unique, mais il aurait été possible de faire du load balancing ici aussi évidemment.

Overview			Message rates	
Channel	User name	State	publish	deliver / get
172.17.0.1:42514 (1)	mri	running	2.0/s	
172.17.0.1:42516 (1)	mri	running		1.0/s
172.17.0.1:42518 (1)	mri	running		1.0/s

Figure 8: Direct exchange connexions

Ici on peut vérifier le fonctionnement attendu par l'échanger direct, il reçoit bien deux messages par secondes, la date locale et GMT, mais chaque consumer ne reçoit qu'un message par seconde, car l'un est abonné à "locale" et l'autre à "GMT" uniquement.

Voyons la trace d'exécution :

```
// EnvoyerDate.java
[x] Sent 'Fri Apr 06 16:23:20 CEST 2018'
[x] Sent '6 Apr 2018 14:23:20 GMT'
// RecevoirDate.java
[x] Received 'Fri Apr 06 16:23:20 CEST 2018'
// RecevoirDateGMT.java
[x] Received '6 Apr 2018 14:23:20 GMT'
```

5 Routage par topic

Le routage par topic est très similaire au mode direct, l'échanger va distribuer le message dans toute les queues dont la clé de routage correspond à celle du message. Sauf que le routage par topic ajoute une fonctionnalité supplémentaire.

Le routage est composé de mots, séparé par des points, on peut créer une clé de routage avec autant de mots que l'on souhaite, dans la limite des 255 octets.

Il y a deux cas spéciaux :

- "*" : peut substituer un mot.
- "#" : peut substituer zero ou plusieurs mots.

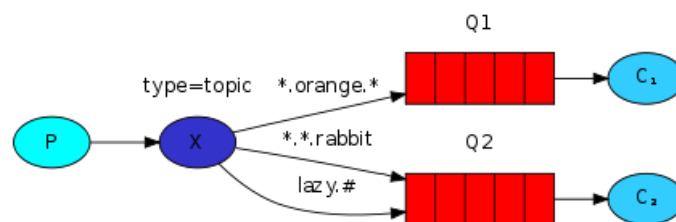


Figure 9: Routage par topic

Par exemple Q1 peut matcher "quick.orange.rabbit" ou encore "lazy.orange.elephant" mais pas "quick.orange.male.rabbit" ni "quick.orange" car "*" ne peut substituer qu'un seul mot.

Par contre Q2 peut matcher "lazy" ou "lazy.pink.male.rabbit" puisque "#" peut matcher zero ou plusieurs mots.

Ici le but était donc d'adapter notre routage simple en un routage par topic de façon à permettre aux consommateurs de s'abonner soit aux dates GMT, soit aux dates simples, ou aux deux à la fois.

Voici les changements qu'il a fallu effectuer :

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

```
fr >> istic >> date >> topic >> EnvoyerDate.java
```

```
channel.basicPublish(EXCHANGE_NAME, "date.locale", null, getDate().getBytes("UTF-8"));
channel.basicPublish(EXCHANGE_NAME, "date.gmt", null, getDateGMT().getBytes("UTF-8"));
```

```
fr >> istic >> date >> topic >> RecevoirDate.java
```

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





```
fr >> istic >> date >> topic >> RecevoirDateGMT.java
```

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

```
fr >> istic >> date >> topic >> RecevoirToutesDates.java
```

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

On peut vérifier le résultat sur l'interface d'administration :

Overview			Message rates	
Channel	User name	State	publish	deliver / get
172.17.0.1:42618 (1)	mri	 running	2.0/s	
172.17.0.1:42620 (1)	mri	 running		1.0/s
172.17.0.1:42622 (1)	mri	 running		1.0/s
172.17.0.1:42624 (1)	mri	 running		2.0/s




Overview				Message rates		
Name	Features	Consumers	State	incoming	deliver / get	ack
amq.gen-P1VDtcyXesoUDKyI0z3DCw	AD Excl	1	 running	1.0/s	1.0/s	0.00/s
amq.gen-mUWrSEXuXxEFD_wG3BdKcA	AD Excl	1	 running	2.0/s	2.0/s	0.00/s
amq.gen-wrMfWAE17bsGrSAlpx04wQ	AD Excl	1	 running	1.0/s	1.0/s	0.00/s

Figure 10: Topic

On observe bien un producteur qui publie 2 messages par secondes, un pour la date locale, un pour la date GMT. Et trois consumers, deux qui reçoivent un message par seconde, pour chaque type de date, et le dernier qui reçoit deux message par secondes, car il reçoit les deux types de dates en même temps.

Voyons maintenant sur une trace d'exécution :

```
// EnvoyerDate.java
[x] Sent 'Fri Apr 06 17:18:15 CEST 2018'
[x] Sent '6 Apr 2018 15:18:15 GMT'
// RecevoirDate.java
[x] Received 'Fri Apr 06 17:18:15 CEST 2018'
// RecevoirDateGMT.java
[x] Received '6 Apr 2018 15:18:15 GMT'
// RecevoirToutesDates.java
[x] Received 'Fri Apr 06 17:18:15 CEST 2018'
[x] Received '6 Apr 2018 15:18:15 GMT'
```

6 Un chat

Le but est de programmer un mini chat semblable à IRC, fonctionnant avec des topics. Au lancement le programme demande un nom et un topic.

Le code est très similaire à la partie précédente, mis à part le fait qu'il faille fournir le topic et un pseudo au lancement du programme. Et que le publisher et le receiver sont dans la même classe désormais.

Le programme utilise la librairie apache commons-cli pour parser les arguments.

```
> java -jar target/mom-1.0-SNAPSHOT-jar-with-dependencies.jar -h
usage: -h | -p <pseudo> -t <topic>
-p,--pseudo <pseudo> Votre pseudo sur le chat
-t,--topic <topic> Topic auquel se connecter, exemple : chat.rmi
```

Dans la trace d'exécution suivante l'utilisateur démarre le programme avec comme pseudo "User1" et comme topic "chat.rmi" :

```
> java -jar target/mom-1.0-SNAPSHOT-jar-with-dependencies.jar -p User1 -t chat.rmi
Welcome User1 you are now connected to chat.rmi !
> Hello World !
chat.rmi#User1>Hello World !
chat.rmi#User2>Hello User1 !
> exit
```

Si on veut utiliser notre programme comme un programme espion d'administration qui permet de regarder ce qu'il se passe sur tous les canaux il suffit de s'abonner à tous les topics comme ceci :

```
java -jar target/mom-1.0-SNAPSHOT-jar-with-dependencies.jar -t chat.# -p Espion
Welcome Espion you are now connected to chat.# !
chat.rmi#User1>Hello World !
chat.rmi#User2>Hello User1 !
```

Voici en résumé la seule partie du code intéressante sur la partie chat qui résume bien le rôle du publisher et du receiver :

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

System.out.println("Welcome " + pseudo + " you are now connected to " + topic + " !");

Consumer consumer = 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(envelope.getRoutingKey() + "#" + message);
    }
};
channel.basicConsume(queueName, true, consumer);

Scanner sc = new Scanner(System.in);
String message;
do {
    message = sc.nextLine();
    if (message.length() > 0) {
        channel.basicPublish(EXCHANGE_NAME, topic, null, (pseudo + ">" +
            message).getBytes("UTF-8"));
    }
} while(!message.equals("exit"));
```