|  |
| --- |
| UQAR, Campus de Lévis |
| Rapport TP2 |
| Systèmes distribués |

|  |
| --- |
| Nicolas Bouchard et Alexandre Croteau  16/04/2018 |

Table des matières

[Obtention de notre code 2](#_Toc511663392)

[Programme P1.js 2](#_Toc511663393)

[Problèmes rencontrés 3](#_Toc511663394)

[Démonstration de l’exécution 4](#_Toc511663395)

[Programme P2.java 5](#_Toc511663396)

[Problèmes rencontrés 6](#_Toc511663397)

[Démonstration de l’exécution 7](#_Toc511663398)

[Programme P3.java 8](#_Toc511663399)

[Problèmes rencontrés 9](#_Toc511663400)

[Démonstration de l’exécution 10](#_Toc511663401)

[Programme P4.java 10](#_Toc511663402)

[Problèmes rencontrés 12](#_Toc511663403)

[Démonstration de l’exécution 13](#_Toc511663404)

[DatabaseHelper.java 15](#_Toc511663405)

[Response.java 16](#_Toc511663406)

[TranslationResponse.java 16](#_Toc511663407)

[ResizeResponse.java 16](#_Toc511663408)

[Schéma de la base de données 17](#_Toc511663409)

[Script de création des tables 18](#_Toc511663410)

# Obtention de notre code

Pour avoir notre code, il suffit d’aller sur GitHub et de faire un clone de notre projet.

La page d’accueil de notre projet (public) est la suivante :

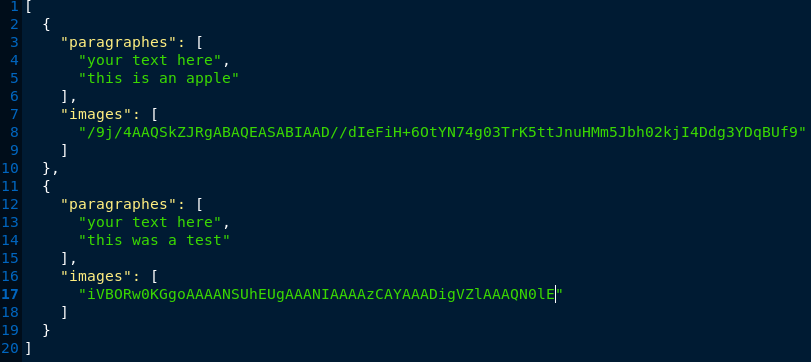
<https://github.com/NicolasBouchard14/DistributedSystemTP2>

# Programme P1.js

Pour le programme P1 qui prend en entrée le texte et les images qui seront traitées par P2 et P3, nous avons choisi d’utiliser JavaScript comme langage.

Étant donné que nous pouvions choisir entre télécharger le texte et les images directement depuis Internet dans le programme ou depuis une source locale, nous avons choisi de se construire un format de fichier JSON qui contient les entrées avec le texte et les images correspondantes.

Voici à quoi ressemble notre format de fichier JSON :



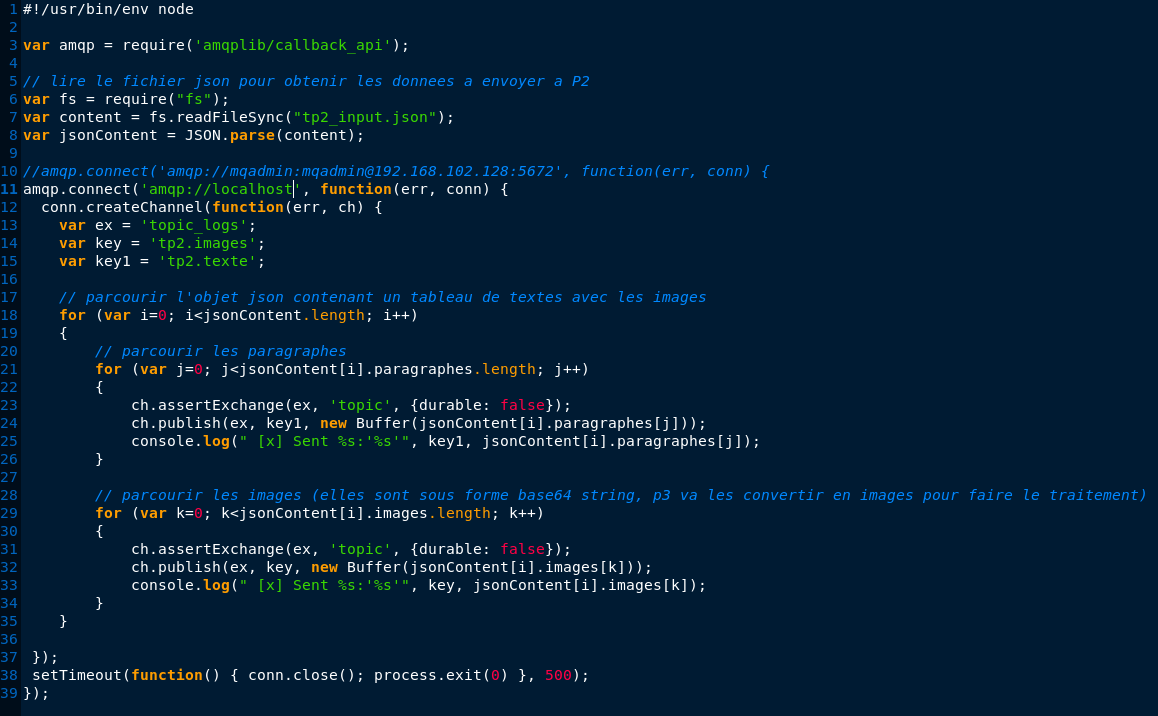
On peut voir que le contenu du fichier est principalement une liste d’objets qui contiennent eux deux listes, une liste de paragraphes et une liste d’images.

Les images sont encodées sous forme de chaîne de caractères en base64.

Le programme P1 est conçu pour lire tout le fichier et boucler sur tous les objets qui contiennent le texte et les images.

Ensuite, dans les deux boucles, une pour le texte et l’autre pour les images, on se connecte à un échangeur topic avec RabbitMQ, afin d’envoyer à la bonne file les images qui seront récupérées par le programme P3 écrit en java qui fera le redimensionnement et le texte qui sera récupéré par le programme P2 écrit en java aussi, qui fera la traduction du texte, tel que demandé.

Voici une capture du code de cette partie, la plus courte de notre application :



## Problèmes rencontrés

Les principaux problèmes rencontrés dans cette section sont liés à mon apprentissage de rabbitmq et de NodeJs, étant donné mon peu d’expérience avec ces technologies.

En effet, au départ, je ne savais pas comment lire mon fichier JSON avec NodeJs et comment parcourir la liste du texte et des images, j’ai donc dû commencer par me documenter avant de faire ma partie.

Ensuite, j’ai rencontré des problèmes en essayant de me connecter à la file de RabbitMQ dans chacune des deux boucles, celle du texte et celle des images. Je n’arrivais pas à envoyer le contenu de mon objet JSON représentant le fichier d’entrées.

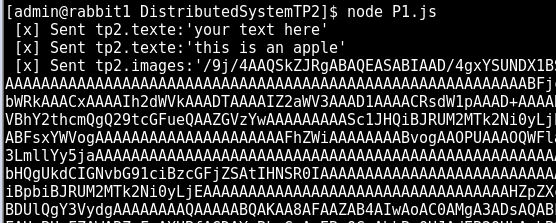
Ce problème est en rapport avec la durée de vie des variables en Java dans les expressions lambda et le fonctionnement du connecteur pour RabbitMQ. J’ai corrigé cela en commençant le programme par la connection à RabbitMQ et dans l’expression lambda de la fonction connect j’ai fait mes boucles pour lire mes objets qui proviennent du fichier JSON et les envoyer à P2 et P3.

## Démonstration de l’exécution

Pour démarrer le programme P1.js, il suffit de disposer de NodeJs sur votre machine et lancer la commande suivante :

Node P1.js

Vous aurez le résultat suivant :



Cette capture est un extrait du premier tour de la boucle de P1.

On voit que les deux paragraphes ont été envoyés et qu’une image sous forme Base64 a aussi été envoyée.

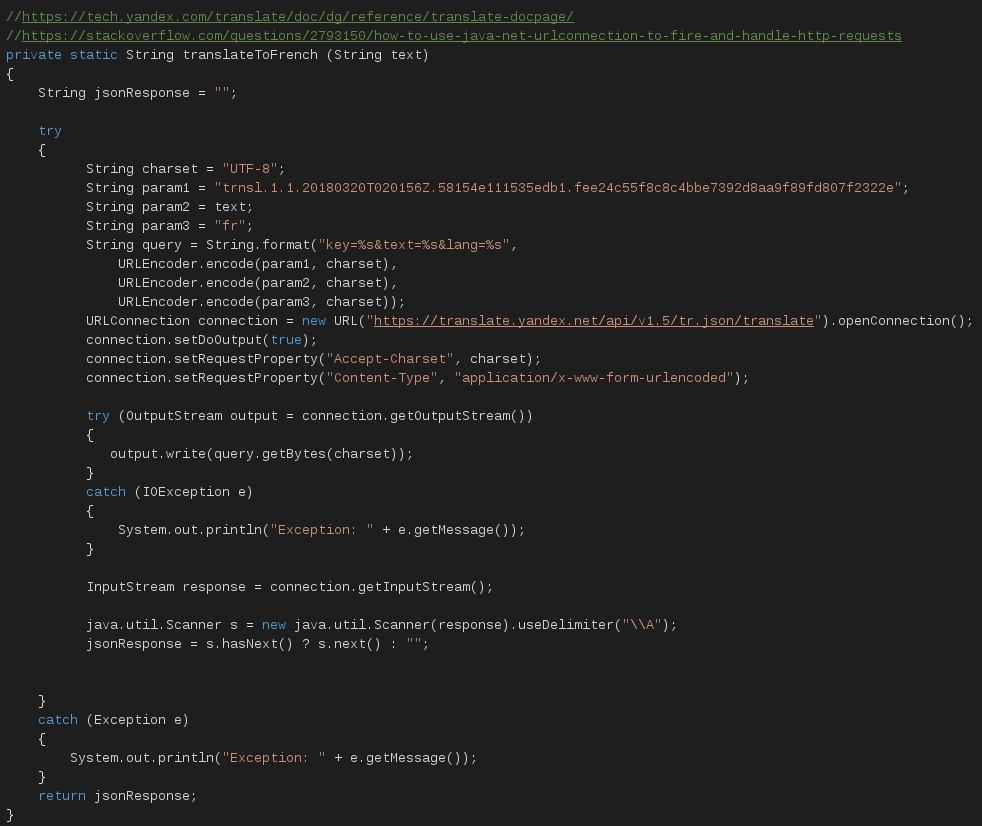
Nous remarquons également les clés de notre échangeur topic qui ici sont tp2.texte et tp2.images. P2 écoute sur la clé tp2.texte et P3 écoute sur la clé tp2.images.

# Programme P2.java

Pour le programme P2, nous avons choisi Java comme langage.

Une fois la connexion à RabbitMQ effectuée et un canal de communication pour la réception des messages établi, la classe P2 se met écoute sur la file d’attente « tp2.texte ».

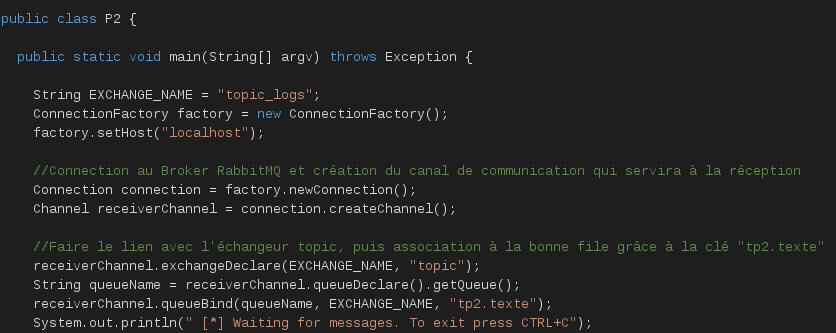
Lorsque du texte est reçu, celui-ci est envoyé à la méthode privée *translateToFrench*, dont voici une capture du code :

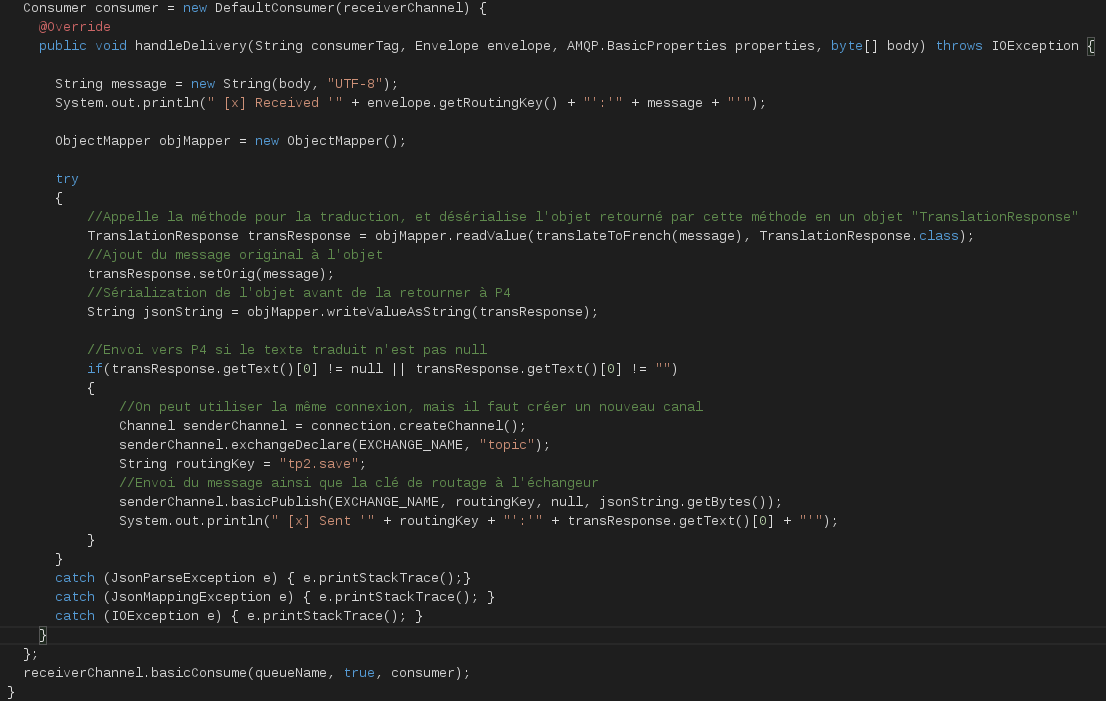


Cette méthode appelle une API gratuite du moteur de recherche Yandex qui s’occupe de faire la traduction du texte, que ce soit un seul mot ou un paragraphe entier.

La réponse en JSON reçue de cette API est ensuite désérialisée vers un objet de type TranslationResponse, puis on ajoute le texte original à cet objet. L’objet est ensuite resérialisé en JSON et prêt à l’envoi vers P4. La classe TranslationResponse est décrite plus loin dans ce document.

Pour l’envoi vers P4, un nouveau canal de communication est créé, puis on envoi le JSON créé précédemment vers l’échangeur avec la clé de routage « tp2.save ».

Voici un aperçu de la méthode *main*, séparée en deux captures d’écran : 



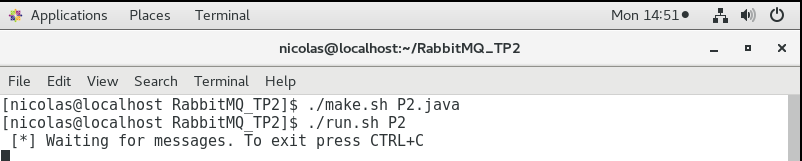
## Problèmes rencontrés

Un de mes problèmes avec le programme P2 a été de trouver un service gratuit qui offrait la traduction par une API. La plupart des solutions existantes sont payantes. Finalement, je suis tombé par hasard dur Yandex, qui correspondait exactement à nos besoins.

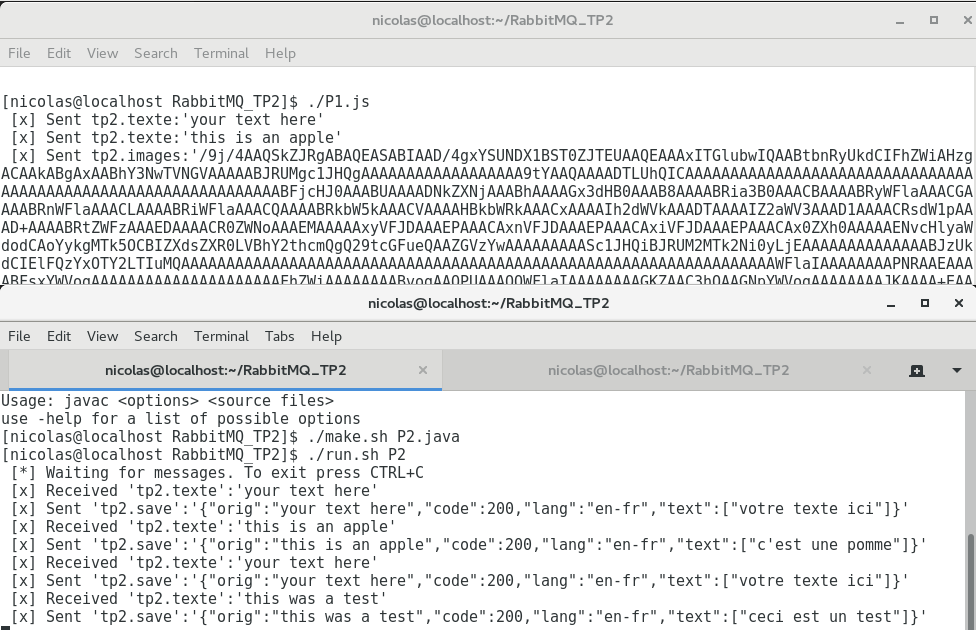
Le deuxième problème a été de savoir à quel moment ou à quel endroit dans le code je devais envoyer ma réponse vers P4 lorsque le texte était traduit. J’ai fini par comprendre que le code doit être placé directement dans la méthode « handleDelivery » afin d’être exécuter à la suite de la réception d’un message sur la file d’attente.

## Démonstration de l’exécution

Pour faciliter l’exécution des programmes Java de notre travail, un script de compilation et un script démarrage des programmes ont été créés. Donc pour démarrer P2, voici les 2 commandes à faire à partir d’un terminal :



Le programme est maintenant en écoute sur la file ayant la clé tp2.texte. Voyons ce qui arrive si nous partons P1 :



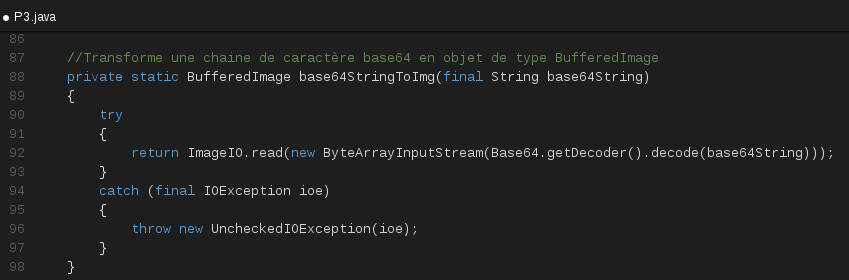
On peut observer que les messages envoyés de P1 sont bien reçus par P2. On voit aussi que P2 envoi une chaine de caractère JSON à la clé tp2.save la queue sur laquelle P4 est en écoute.

# Programme P3.java

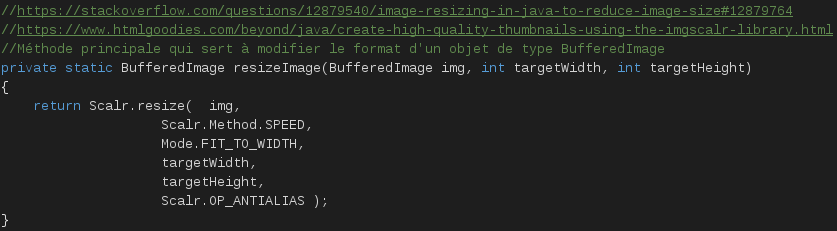
Pour le programme P2, nous avons choisi Java comme langage.

Une fois la connexion à RabbitMQ effectuée et un canal de communication pour la réception des messages établi, la classe P2 se met écoute sur la file d’attente « tp2.images ».

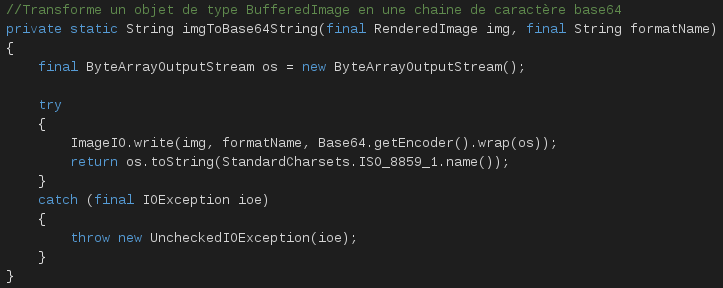
Lorsqu’une image sous format Base64 est reçue, cette image est convertie en objet de type BufferedImage à l’aide de la méthode privée *base64StringToImg*.



Une fois l’image dans ce format, la méthode privée *resizeImage* est appelée avec les paramètres de hauteur et de largeur voulus pour la transformation de l’image originale. Comme nous voulons une image rapetissée et une agrandie, elle est appelée deux fois.



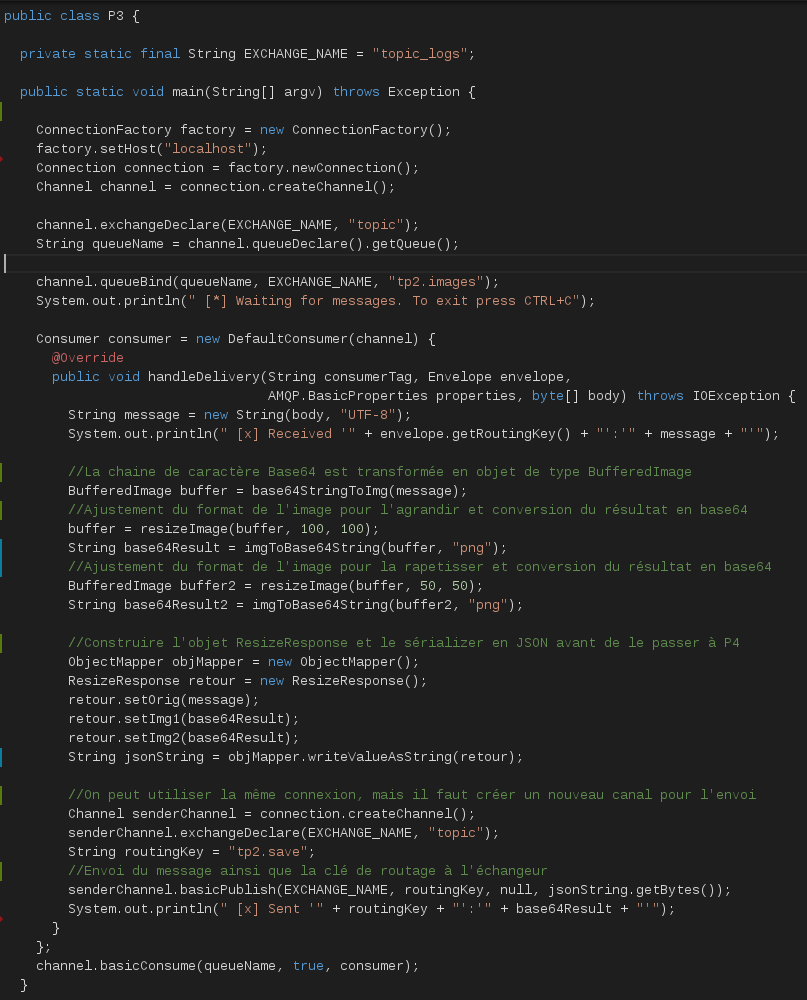
Les images obtenues sont ensuite reconverties en base64 grâce à la méthode privée *imgToBase64String*.



Nous avons maintenant trois chaines de caractère, correspondant chacune à une image de différentes tailles. Un objet de type ResizeResponse est alors construit avec ces chaines de caractère, puis cet objet est sérialisé en JSON avant d’être envoyé vers P4.

Pour l’envoi vers P4, un nouveau canal de communication est créé, puis on envoi le JSON créé précédemment vers l’échangeur avec la clé de routage « tp2.save ».

Voici une capture d’écran de la méthode *main* de P3 :

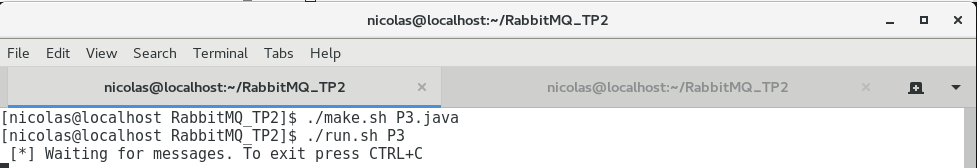


## Problèmes rencontrés

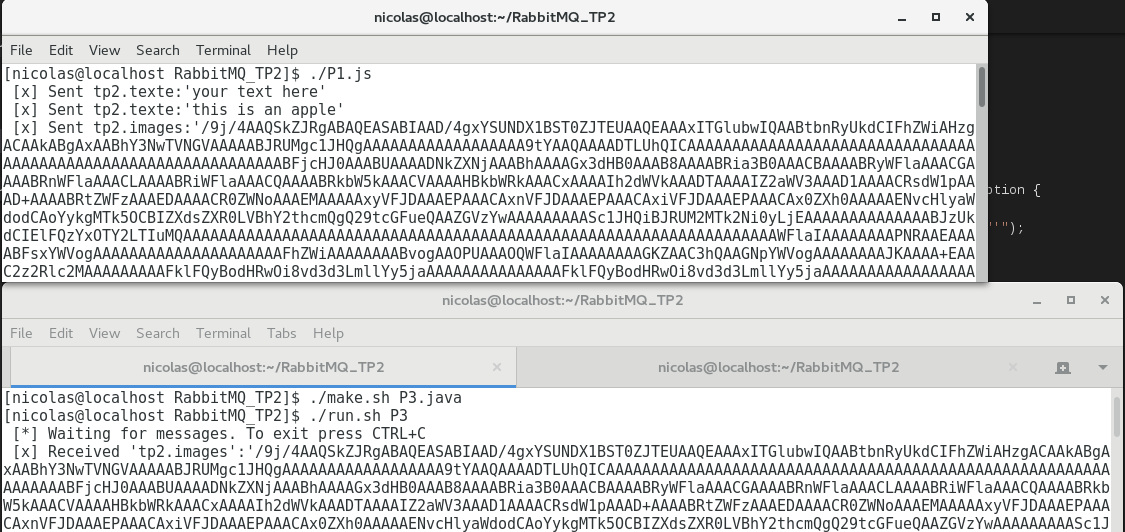
La modification de la taille des images m’a posé quelques problèmes, en commençant par trouver une librairie simple à utiliser qui offrait cette fonctionnalité. J’ai trouvé la librairie ImgScalr, mais très peu de documentation la concernant. Finalement son utilisation s’est révélée plutôt simple. Au niveau de RabbitMQ, j’ai eu peu de problèmes car j’avais déjà réalisé P3.

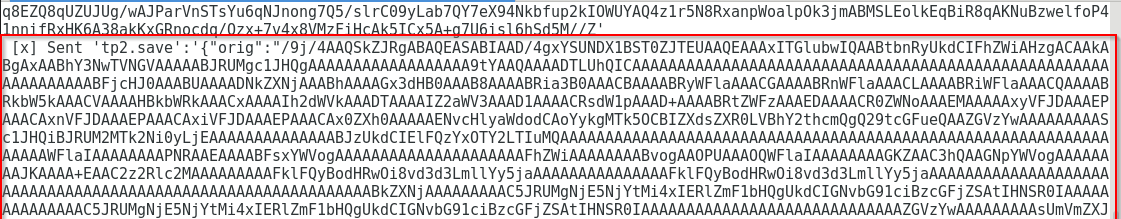
## Démonstration de l’exécution

Pour faciliter l’exécution des programmes Java de notre travail, un script de compilation et un script démarrage des programmes ont été créés. Donc pour démarrer P2, voici les 2 commandes à faire à partir d’un terminal :



Le programme est maintenant en écoute sur la file ayant la clé tp2.images. Voyons ce qui arrive si nous partons P1 :





On peut observer que les messages envoyés de P1 sont bien reçus par P3. On voit aussi que P3 envoi une chaine de caractère JSON à la clé tp2.save la queue sur laquelle P4 est en écoute.

# Programme P4.java

Le programme P4, comme déjà expliqué auparavant sert à recevoir les résultats de P2 et P3 (texte traduit et images redimensionnées) et enregistrer le tout dans une base de données sur un cluster Galera MariaDB.

Le programme P4 écoute donc sur la clé tp2.save dans notre échangeur sur la file file\_d\_attente02, car le programme P2 et le programme P3 envoient les deux leurs résultats à ce même emplacement.

Le truc pour réussir à savoir si le message reçu est une image ou du texte et enregistrer le tout dans la bonne table de la base de données est que les réponses de P2 et P3 sont sérialisées au format JSON depuis deux classes différentes, qui héritent toute les deux de response.java, que nous décrivons plus loin dans le présent document. Nous avons écrit un utilitaire de désérialisation personnalisé qui permet de choisir entre le type qui correspond au retour de P2 ou au retour de P3.

Selon le type d’objet reçu, nous appelons la bonne fonction de la classe DatabaseHelper qui permet de gérer l’enregistrement à l’aide d’un connecteur JDBC.

Voici une capture du code de P4 sur la page suivante :



## Problèmes rencontrés

Pour cette partie, notre principal problème fut notre difficulté à recevoir les données de P2 et P3 sur la même file et avec la même clé.

En effet, au départ, nous ne savions pas comment pouvoir recevoir les résultats du programme P2 et du programme P3, en même temps dans le programme P4.

Notre première tentative fut d’utiliser deux clés différentes dans le même programme, mais chaque fois que je tentais d’écouter sur la deuxième clé, la première cessait de fonctionner.

J’ai donc dû lire beaucoup de documentation sur le web pour voir comment s’y prendre pour rassembler les résultats de deux programmes dans le même avec RabbitMQ.

J’ai trouvé ma solution dans un exemple en ligne qui répondait indirectement à mes besoins et que j’ai dû adapter.

En effet, il suffit de recevoir tous les résultats sur la même clé de notre échangeur topic, ce qui est beaucoup plus simple en apparence, du moins. Ensuite, il faut donc avoir une classe de base compatible autant avec P2 qu’avec P3, afin de sérialiser nos réponses au format JSON et les lire avec P4.

C’est ce que nous avons fait avec les trois classes response, resizeresponse et traductionresponse.

À Partir de là, tout est plus simple, nous n’avons qu’à vérifier le type et insérer les résultats dans la base de données.

## Démonstration de l’exécution

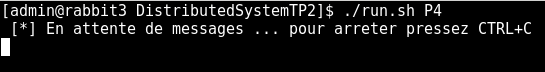
Pour lancer le programme P4, il suffit d’utiliser notre script run.sh qui permet de lancer nos programmes java avec le bon classpath, afin de pouvoir utiliser tous les jars du dossier courant.

Nous aurions pu ajouter le dossier du travail dans le classpath de nos machines virtuelles, mais comme elles servent aussi à d’autre chose, j’ai préféré faire un script. Ce script permet aussi de lancer nos programmes depuis d’autres machine dont nos ordinateurs portables qui sont configurés différemment.

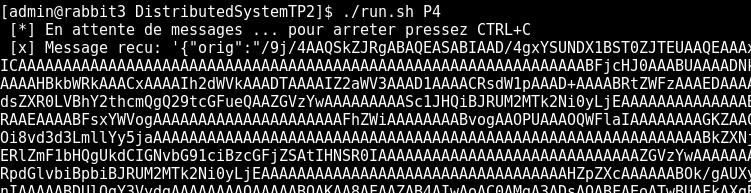
En bref, pour lancer P4, il faut taper la commande suivante :

./run.sh P4

Cela nous donne le résultat suivant :



Ensuite, lorsque P1 a envoyé ses messages à P2 et P3 et que leurs traitements sont complétés, P4 reçoit les résultats et affiche ceci :





On peut voir ici que pour le texte traduit P4 reçoit et affiche le texte original, ainsi que la traduction.

C’est le même principe pour les images, même s’il est impossible de le montrer dans la capture précédente. En effet, P4 reçoit l’image originale, ainsi que les deux images redimensionnées.

Après chaque réception, P4 enregistre les résultats dans la base de données sur notre cluster utilisant Galera cluster et MariaDB. Si l’enregistrement a été effectué avec succès, P4 affiche « ok ».

# DatabaseHelper.java

La classe DatabaseHelper permet de faire les opérations nécessaires à P4 sur la base de données.

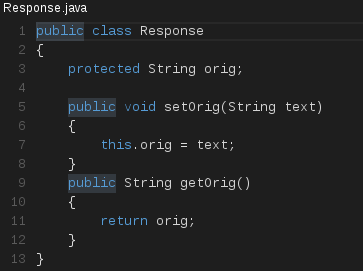
Il y a donc une fonction pour se connecter à la base de données, une pour insérer dans la table image avec trois chaînes de caractères en Base64 converties en tableaux d’octets et une fonction qui prend un texte en anglais et en français et l’insère dans la table texte.

Voici une capture du code de la classe :



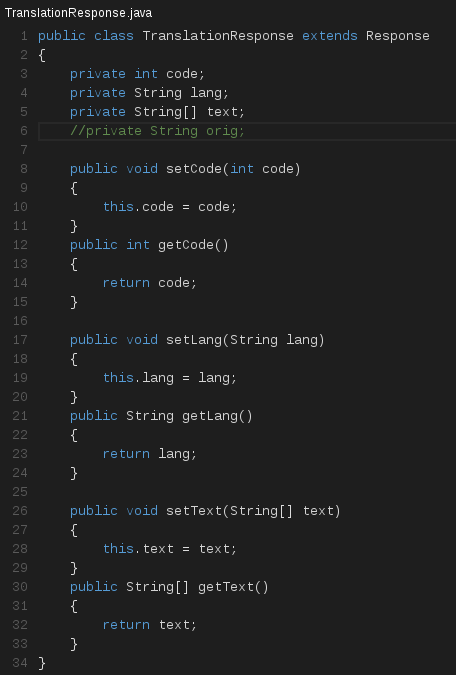
# Response.java

Classe mère des classes TranslationResponse et ResizeResponse, créée car ces deux classes contiennent la propriété « orig ». Effectivement, dans les deux cas l’image ou le texte original doit être réacheminé vers P4.



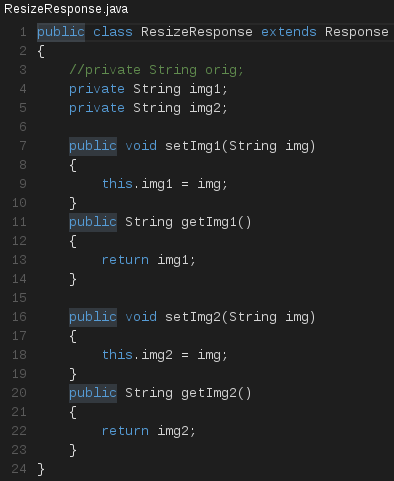
# TranslationResponse.java

Cette classe sert uniquement à la sérialisation/désérialisation JSON avec la libraire Jackson. Elle est utilisée dans la classe P2 et hérite de Response.java.



# ResizeResponse.java

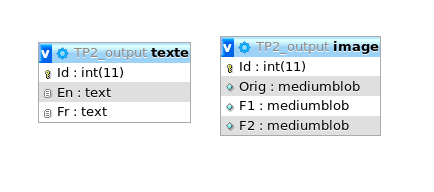
Cette classe sert uniquement à la sérialisation/désérialisation JSON avec la libraire Jackson. Elle est utilisée dans la classe P3 et hérite de Response.java.



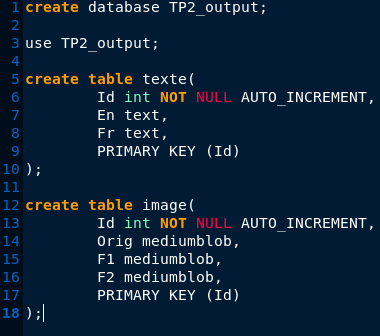
## Schéma de la base de données

Nous avons choisi d’utiliser deux tables. Une pour les résultats de la traduction et l’autre pour garder tous les formats des images reçues.

La table texte contient un id et une colonne pour le texte en français et l’autre pour le texte en anglais.

La table image contient un id, un blob pour l’image originale, un bloc pour le format 1 et un bloc pour le format 2.

## Script de création des tables

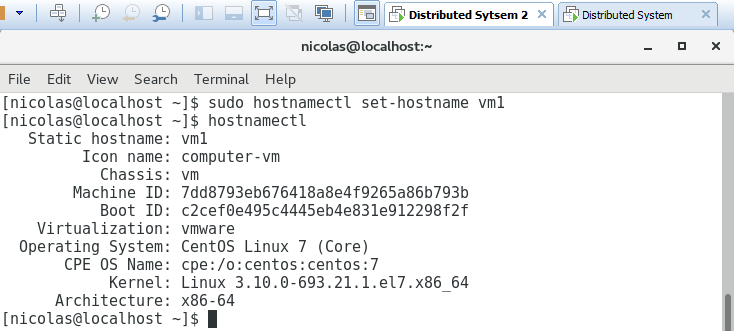
Ce fichier ne contient rien de plus, rien de moins que le code nécessaire à la création des tables pour enregistrer les résultats avec P4 dans notre cluster Galera MariaDB.

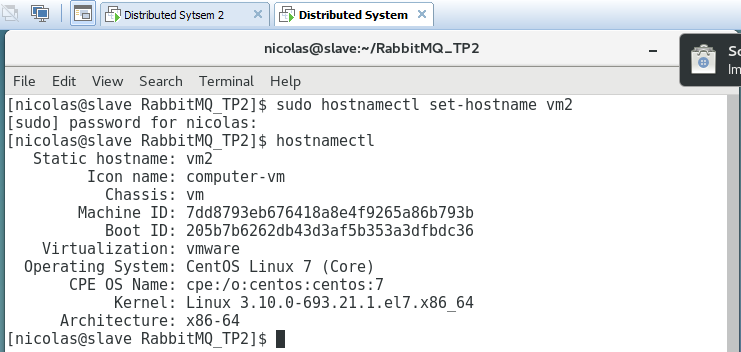
# Cluster RabbitMQ

Les étapes de configuration du cluster RabbitMQ sont inspirées de ce tutoriel : <http://devopspy.com/linux/rabbitmq-cluster-centos-7/>

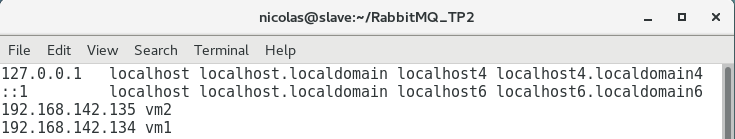
## Configuration du réseau

La première étape consiste à configurer la carte réseau sur les deux machines afin qu’elles puissent communiquer à l’aide de leur nom d’hôte sur le réseau. Comme nous n’avons pas de serveur DNS, nous configurons simplement le nom d’hôte sur chaque machine et le fichier /etc/hosts est ajusté pour faire la correspondance entre l’adresse IP et le nom d’hôte.

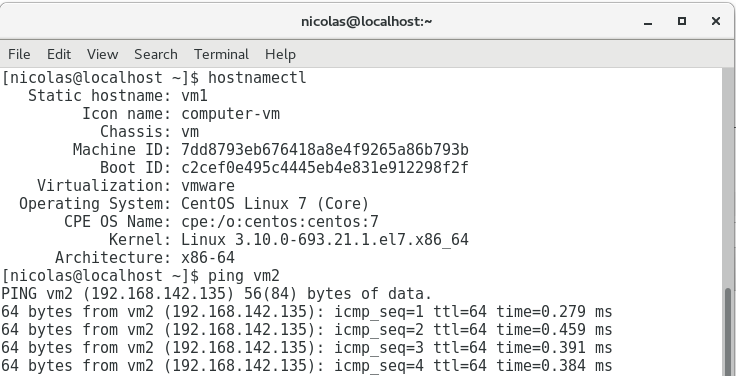
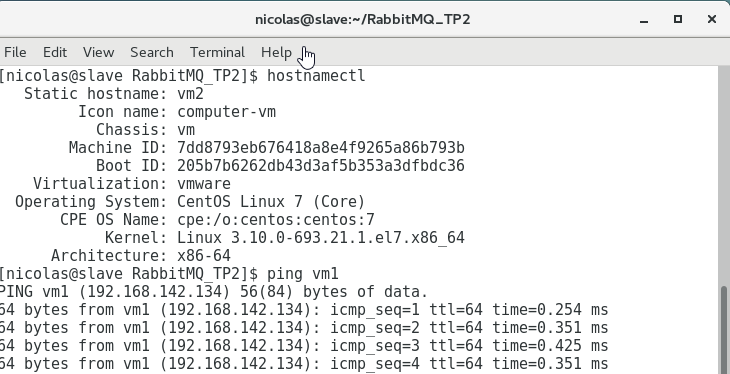




Fichier /etc/hosts sur les deux machines :

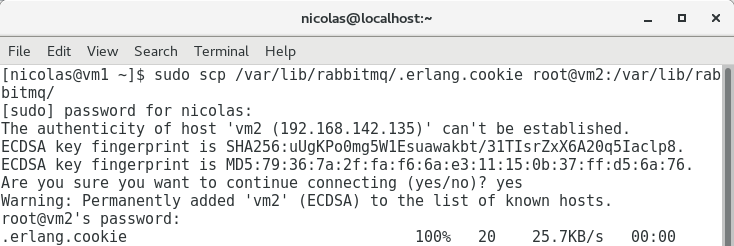


Les machines sont donc maintenant capables de se rejoindre sur le réseau avec leurs noms d’hôte :



## Authentification des nœuds

L’authentification des nœuds se fait à l’aide du cookie Erlang. Ce cookie doit être identique sur chaque machine. Il suffit de le copier d’un des nœuds vers les autres nœuds :



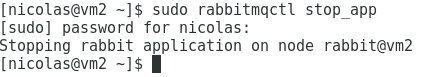
## Configuration de la haute-disponibilité

Par défaut la haut-disponibilité n’est pas active sur le cluster RabbitMQ. Il faut donc entrer cette ligne de commande sur tous les nœuds afin de l’activer :



## Démarrer le cluster

La première étape est de stopper un le(s) nœuds que nous voulons ajouter au cluster :



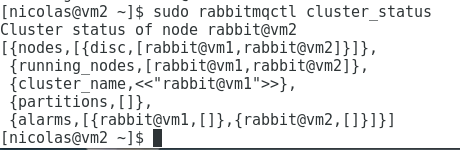
Ensuite nous allons ajouter ce nœud au cluster en pointant vers vm1.



Finalement, nous pouvons repartir le nœud. Il démarrera dans le cluster.

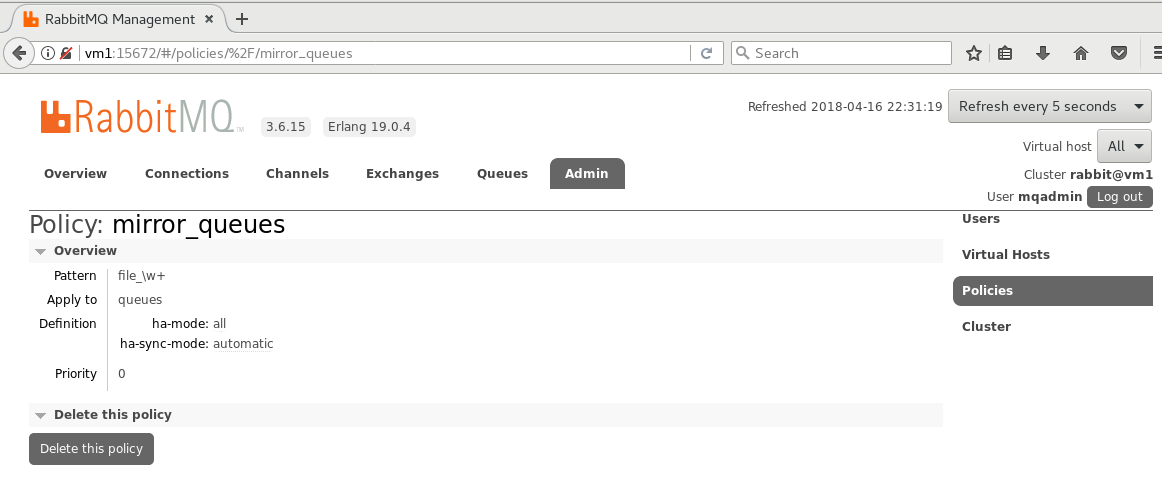


Pour vérifier l’état du cluster :

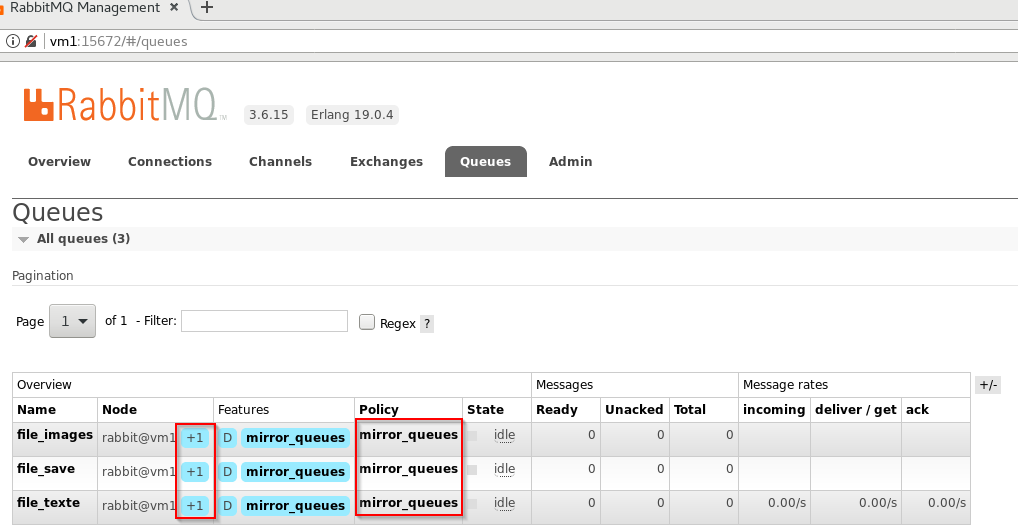


# Files d’attente en miroir

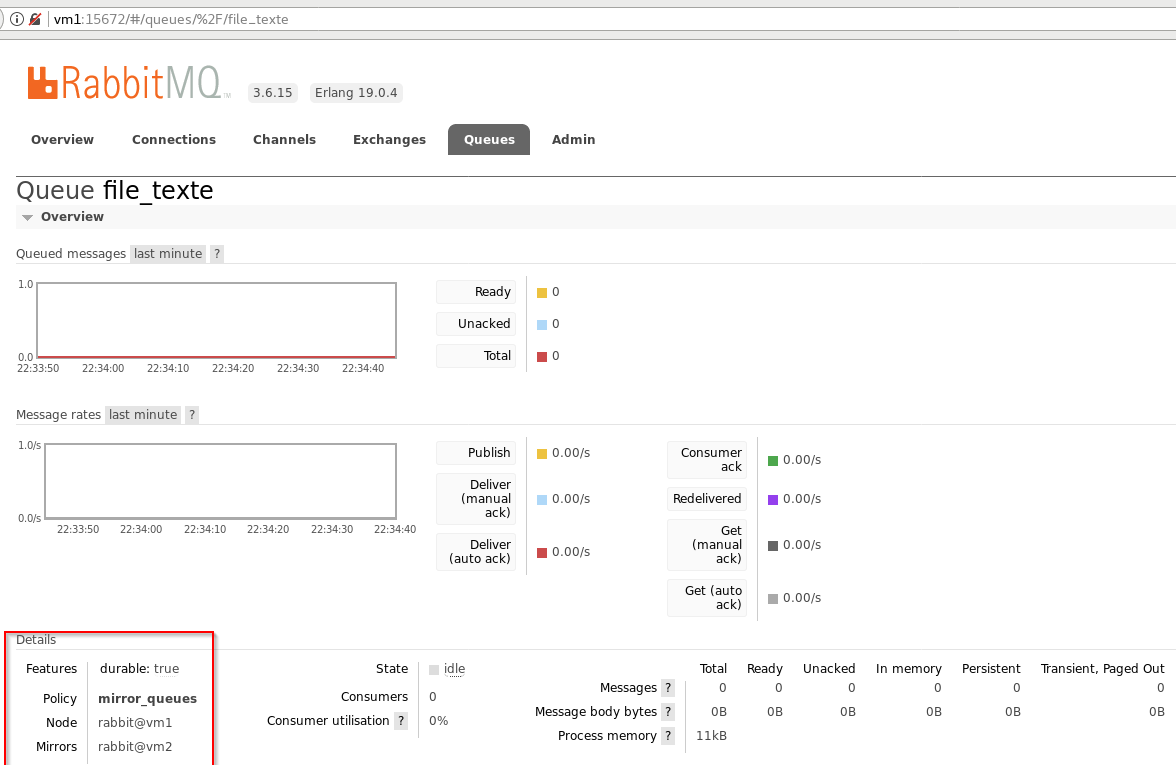
Une fois le cluster configuré et fonctionnel, il faut ajouter une règle dans la configuration de RabbitMQ pour spécifier les paramètres de réplication des queues et quelles queues nous voulons répliquer. Pour ce faire, nous avons utilisé l’interface de gestion :



En résumé, cette règle va répliquer toutes les files commençant par « file\_ » sur tous les nœuds disponibles. Toujours dans l’interface de gestion, on peut voir que la stratégie est bien appliquée et que les queues est hébergée sur deux nœuds.

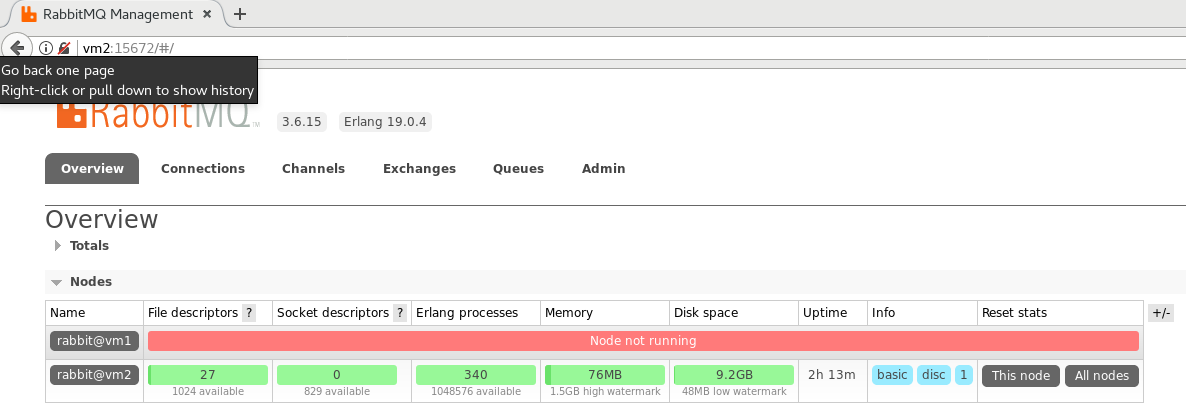


Enfin, si on regarde les informations détaillées de l’une des queues, on voit qu’un miroir est bien configuré :

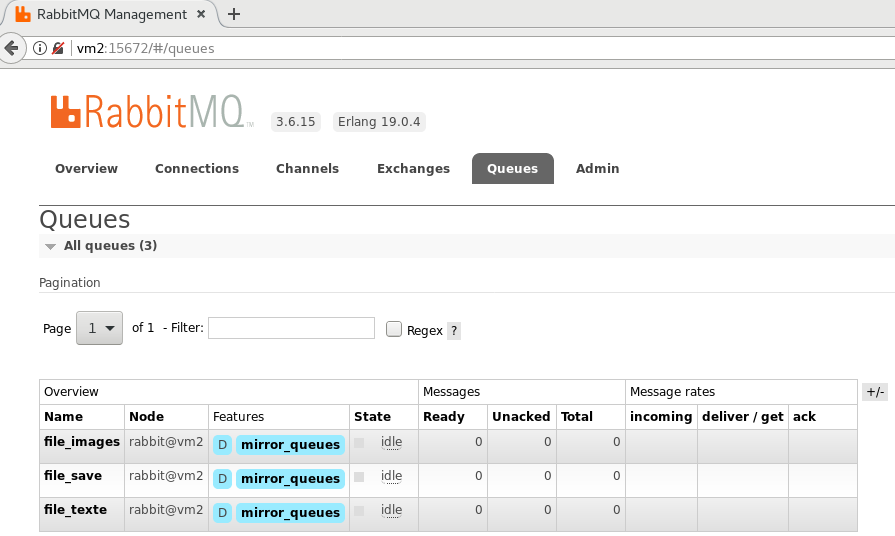


## Démonstration

Pour la démonstration, je vais stopper une des deux machines virtuelles(vm1) :



Dans l’interface de gestion, on voit bien que les queues sont encore actives sur le nœud restant :



Puis si on repart vm1, la reprise est automatique et la queue devient à nouveau répliquée, sauf que les rôles ont été échangés entre le nœud principal et le miroir.

