|  |
| --- |
| 25 Nov 2017 |
|  |

# Compte Rendu

**Implémentation de l’algorithme de Lamport avec RabbitMQ**

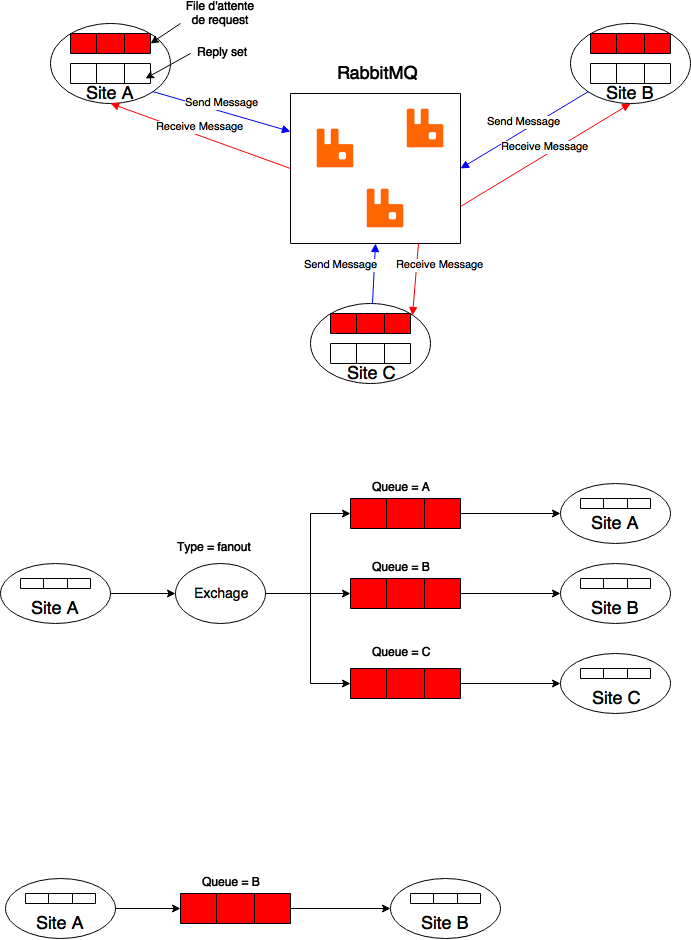
**Qian LI**

**Hanyuan ZHANG**

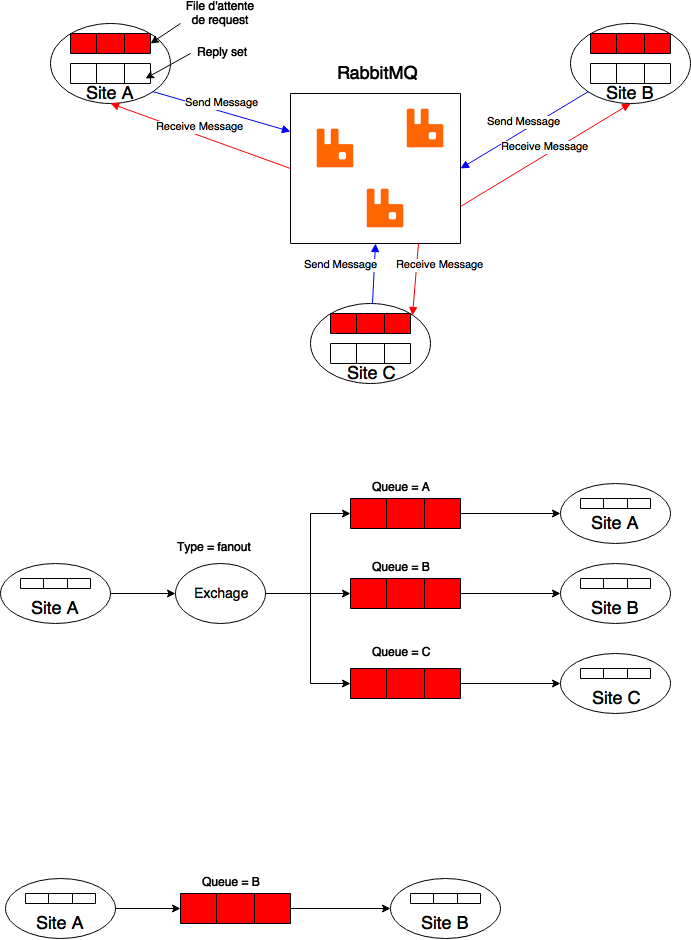
Description d’Architecture et Files d’Attente

architecture

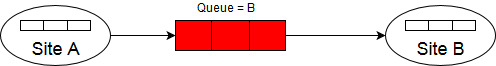
Pour réaliser l’exclusion mutuelle dans un système distribué en utilisant l’algorithme de Lamport et RabbitMQ, notre architecture est montrée ci-dessous. On a créé une file d’attente au côté de chaque site pour mémoriser les requêtes d’entrer dans la section critique et on a lié tous les sites par un middleware RabbitMQ pour qu’ils puissent communiquer entre eux. Grâce à RabbitMQ, l’ordre des messages est assuré et les différents types des messages (broadcast et unicast) sont transférés automatiquement en utilisant différents types d’exchanges.



Ensuite on déclare un exchange de type “fanout” permettant à chaque site ayant rôle de *publisher* d’envoyer les messages broadcast. Pour simplicité, la queue a le même nom que celui du site, par exemple, la queue du site A est nommé A comme *routing\_key*. Tous les sites dans notre architecture sont liés à cet exchange et chacun peut recevoir des messages publié par un des sites. Les détails sont au-dessous.



Quand site A est un publisher qui veut envoyer une requête d’entrer à la section critique ou un message de sortie de la section, il va utiliser exchange fanout pour envoyer le message à tous les autres sites qui sont rattachés à cet exchange. Et les autres sites comme consumers peuvent récupérer la requête d’A depuis leurs queues déclarées. Mais quand A veut répondre la demande d’un autre site B, la transmission de message est différente, il envoie le message directement avec le nom de queue, parce que le message de réponse est unicast.



files d’attente

Les files d’attente de RabbitMQ sont déjà présentés et ces sont des parties qu’on ne touche plus dans ce tp. Mais pour nous, il faut désigner les files d’attente du côté de chaque site pour mémoriser les requêtes, et cette queue doit être FIFO pour assurer l'ordre des demandes.

Dans l’algorithme de Lamport, tous les messages envoyés sont *timestamped*, et on peut comparer les valeurs d’horodatage pour déterminer l'ordre du site d’entrer dans la section critique. Du coup on a besoin de créer une file d’attente sur chaque site pour mémoriser les demandes et leurs ordres. Il y a trois types de cas qui vont modifier la queue.

* Quand le site i envoie la requête aux autres, sa propre file d’attente doit prendre en compte son demande avec timestamp (tsi, i).
* Quand une nouvelle requête depuis site j arrive, cette file d’attente doit ajouter la nouvelle demande (tsj, j).
* Quand un message de RELEASE arrive, la demande qui correspond à ce message (la demande au top de la queue) doit être supprimé.

Un point important dans cette partie est comment assurer la récupération du message dès que ce message arrive à la queue de RabbitMQ. Parce que toute les sites doivent être logiquement synchronisés en mettant à jour leurs horodatages suit à des horodatages reçus pour assurer l’ordre unique d’entrer dans CS dans tout le système. La queue de RabbitMQ assure l’ordre des messages arrivés donc le site doit assurer cette récupération du message sans garde.

Présentation de La Solution et Implémentations

Afin de réaliser des fonctionnalités demandées précédemment, au niveau de chaque site, on a défini deux listes : *request\_queue* et *reply\_set*.

* *request\_queue* : elle enregistre les demandes avec un tuple de format (tsi, i), où tsi est l’horodatage logique de la demande de site i, et i signifie l’index de la site qui a envoyé cette requête.
* reply\_set : elle enregistre les messages REPLY reçus avec un tuple de format (tsj, j), où tsj est l’horodatage logique du site j, et j indique l’index de site qui a envoyé cette réponse.

Comme indiqué dans l’algorithme de Lamport, on a utilisé l’horloge logique pour éviter le problème du décalage de l’horloge physique dans des différents processus. Chaque site a une horloge locale. Quand il envoie un message, il va augmenter l’horloge par un et quand il reçoit un message, il va comparer son horloge locale avec l’horloge reçue, puis mettre à jour son valeur.

Pour chaque site, il y a trois fonctionnalités principales :

* Demande la section critique :

REQUEST\_CS :

   If self not in REQUEST\_QUEUE

   then

       TIMESTAMPS := TIMESTAMPS + 1

       send REQUEST to EXCHANGE\_FANOUT

Quand un site reçoit une requête depuis le site X :

RECV\_REQUEST

   TIMESTAMPS : = max(TIMESTAMPS, RECV\_TIMESTAMPS)

   enqueue(REQUEST\_QUEUE, X)

   send REPLY to X

* Exécute la section critique :

          EXECUTE\_CS

   If self = dequeue(REQUEST\_QUEUE) ∧ length(REPLY\_SET) = NB\_QUEUES

              then

                  CRITICAL\_SECTION

* Libère la section critique :

          RELEASE\_CS

              TIMESTAMPS : = TIMESTAMPS + 1

              send RELEASE to EXCHANGE\_FANOUT

REPLY\_SET : = empty

Quand un site reçoit un message RELEASE de site X :

         RECV\_RELEASE

TIMESTAMPS : = max (TIMESTAMPS, RECV\_TIMESTAMPS)

dequeue(REQUEST\_QUEUE)

EXECUTE\_CS

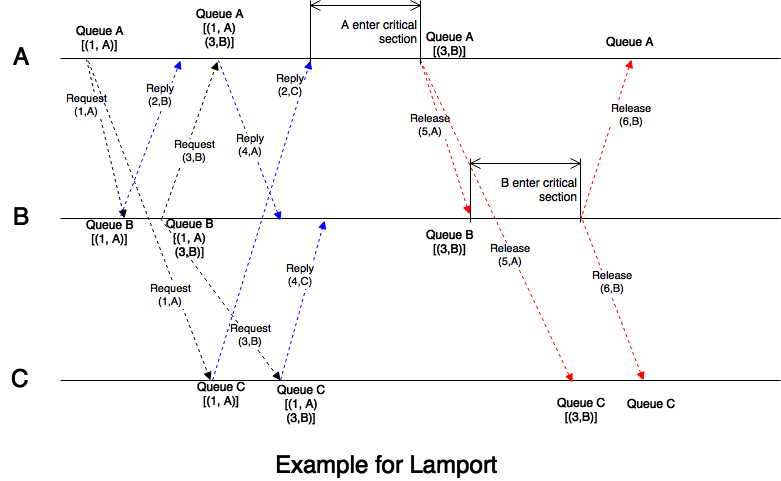
Pour tester les fonctionnalités, on a ajouté une fonction de l'interrogation qui va demander au site s’il voudrait entrer dans la section critique quand il n’est pas dans la liste *request\_queue*.

Les fonctions *start-consuming* et *publish* sont bloquants. Afin que le site puisse envoyer et recevoir les messages en même temps, puis lancer une fonction d’interrogation. Nous avons créé trois processus dans notre programme en utilisant le module *multiprocessing* du Python. De plus, pour que les variables puissent se communique, nous avons ajouté un *PIPE* entre des processus.

* Processus père : On initialise tous les canaux de communication dans le processus père et crée deux processus fils utilisant la fonction *Multiprocessing.Process().* En effet, le processus père contrôle le démarrage et l’arrêt du processus fils 1 qui demande le souhait d’entrer cs.
* Processus fils1 (interrogation) : On a mis la fonction d’interrogation dans un processus fils qui va se démarrer au début du programme et s’arrêter quand il a récupéré la réponse du site. Ensuite, ce processus va être relancé par le processus père s’il n’est plus dans la file d’attente.
* Processus fils2 (message consuming) : En effet, le temps de recevoir du message est aléatoire et le système a besoin de récupérer le message sans garde, donc on a besoin une fonction qui exécute en fond pour recevoir le message immédiatement quand il arrive à la queue de RabbitMQ.

exemple d’utilisation

On va présenter ici une exemple de trois sites et deux d'entre eux A et B veulent entrer dans la section critique. Le diagramme d’exemple est au-dessous.



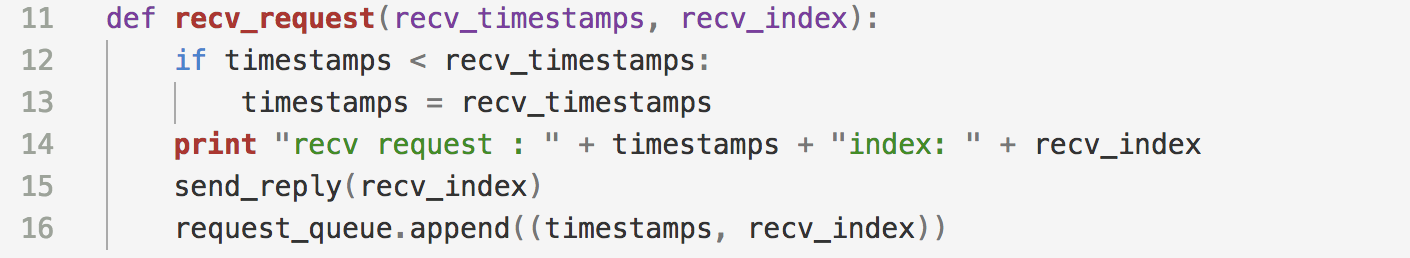
1. Au début de tout le processus, site A envoie une requête d’entrer à la CS (1, A) au site B et site C et met soi-même avec l’horodatage t=1 dans sa file d’attente.
2. Quand B ou C reçoit la requête d’A, ils mettent cet requête dans leurs files d’attente et synchronisent leur horloge logique avec l’horodatage d’A, ensuite ils retournent un REPLY (2, B) et (2, C) avec un horodatage synchronisé.
3. Site B envoie la requête (3, B) à site A et site C.
4. Pendant la durée d’attente des REPLY des autres sites, A reçoit la requête de B et l’ajoute dans sa file d’attente. Donc A envoie un REPLY (4, A) à B et C envoie un REPLY (4, C) aussi.
5. A entre zone critique dès qu’il confirme tous les réponse reçus.
6. A envoie un message de RELEASE dès qu’il sort de la section avec la requête supprimée depuis sa file d’attente.
7. B ou C reçoit le RELEASE d’A et supprime la demande d’A depuis leurs files d’attente. Ensuite, B confirme qu’il est au top de sa file d’attente et il a déjà reçu toutes les réponses des autres, donc B entre le zone critique.
8. B sort CS et envoie RELEASE aux autres. Enfin les file d’attente d’A, B, C sont tous vide.

extraits du code

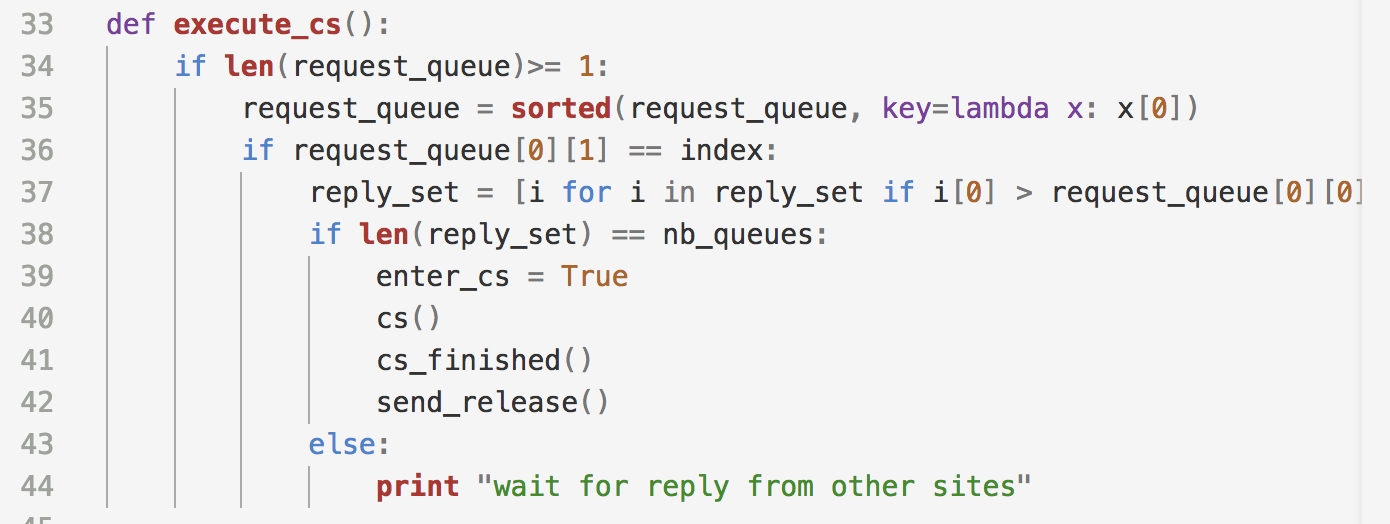
Dans cette partie, nous allons montrer les codes des fonctions principales , des utilisations des processus et des pipes.

Demande la section critique :

Quand un site reçoit une requête ：

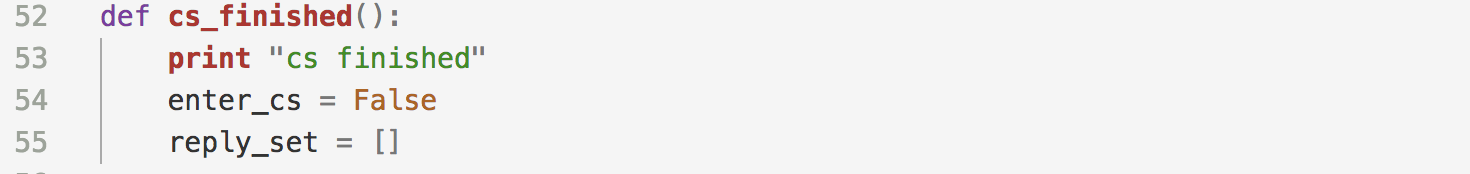


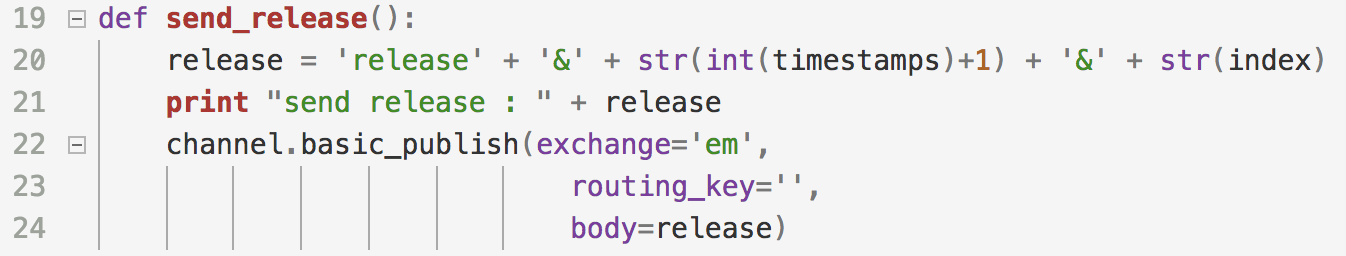
Exécute la section critique :



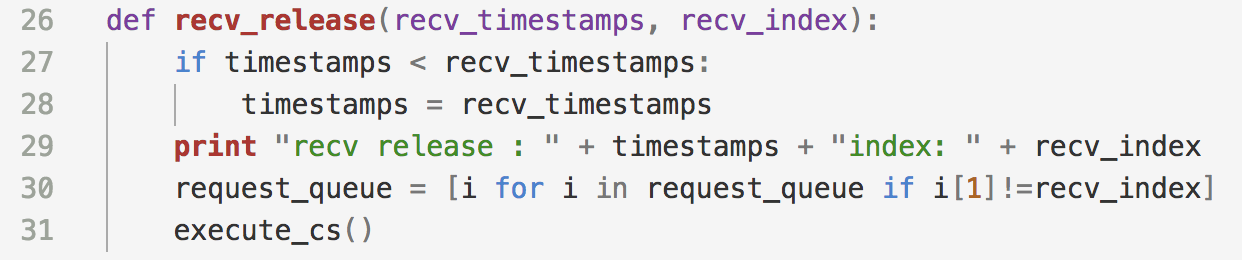


Libère la section critique :

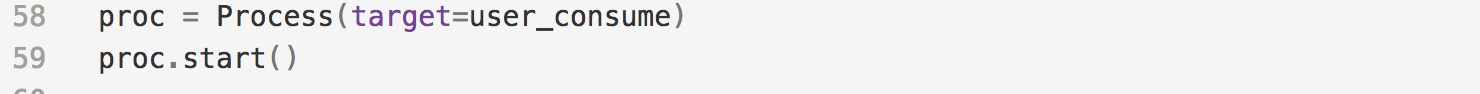




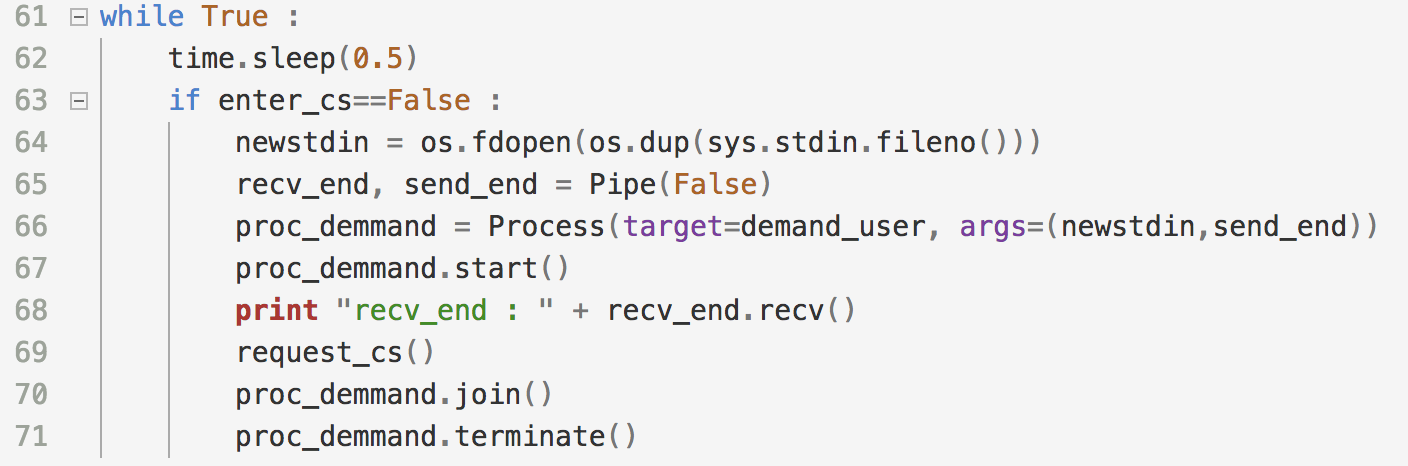
Quand un site reçoit un message RELEASE ：



Processus fils 2 (message consuming) :



Processus fils 1 (interrogation) et PIPE :



synthèse

Différentes étapes de la solution :

La première étape est de créer une queue pour chaque site, puis de les liées à un exchange de type *fanout*. Pour des messages REQUEST et RELEASE, nous utilisons cet exchange, et pour des messages REPLY nous les envoyons en utilisant le nom de la queue comme le *routing\_key*.

La deuxième étape est de créer deux processus fils, un pour demander au site s’il voudrait entrer dans la section critique, l’autre pour recevoir des messages simultanément.

Difficultés de programmation :

Les difficultés que nous avons rencontrées concernent plutôt de l’envoie et de la réception des messages en même temps. Enfin nous avons trouvé la solution en mettant l’envoie et la réception dans deux processus séparés. Un autre problème que nous avons rencontré est de partager l’entrée et la sortie entre des processus. Enfin nous avons résolu ce problème à l’aide un PIPE qui lie l’entrée et la sortie du processus père à celles de processus fils.

Avantages de RabbitMQ :

L’avantage de RabbitMQ est qu’il est un middleware et qu’il peut gérer la perte, l’ordre et la correction de l’erreur des messages. En utilisant RabbitMQ, nous pouvons simplifier le développement, garantir la communication entre des sites et considérer seulement la logique métier.