Algorithme de Lamport et RabbitMQ

FIG 3A- F2B 204

Yacine Ihadadene

yacine.ihadadene@telecom-bretagne.eu

Jinhai Zhou

jinhai.zhou@telecom-bretagne.eu

2017-2018

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

SOMMAIRE

1. INTRODUCTION

RabbitMQ est une implémentation open source du protocole Advanced Message Queuing Protocol (AMQP). AMQP est un protocole et une standardisation qui définit la manière d'échanger les messages.

Dans ce projet nous avons implémenté l'algorithme de Lamport Leslie (un algorithme synchronisation distribuée avec des horloges logiques) en utilisant RabbitMQ.

2. ALGORITHME DE LAMPORT

L'algorithme de Lamport est un algorithme d'exclusion mutuelle utilisé dans les systèmes distribués, il utilise l'attente active avant d'entrer en section critique.

L'algorithme utilise des horloges logiques pour ordonner les événements. Pour faire cela chaque site (processus) incrémente son horloge entre deux événements locaux ou distants. Lorsqu'un site "Si" reçoit un message avec une horloge H d'un autre site, l'horloge de "Si" devient alors max(Hi,H)+1.

Quand un site veut rentrer dans une Section Critique (SC), il incrémente son horloge et envoie un message "REQUEST" de demande de SC à tous les autres sites et dépose ce message dans une file d'attente de requêtes. Dans le cas, ou un site reçoit un message "REQUEST", il synchronise son horloge, met le message dans la file de requêtes et envois message de type "REPLY" daté de bonne réception à l'expéditeur.

Un site "Si" rentre en SC si et seulement si les deux conditions suivantes sans satisfaites:

- ➔ Il existe un message de type "REQUEST" (Hi, Si) dans la file de requête du site qui est ordonnée devant toute les autres requêtes (le plus ancien dans le temps)

- ➔ Le site "Si" reçoit un message de tous les autres sites ayant une date inférieur à Hi

Quand un site quitte la SC, il enlève sa requête de demande de SC de sa propre file de requêtes, incrémente son horloge et envois un message daté de type "RELEASE(Hi, Si)" à tous les autres sites afin de signaler qu'il quitte la SC.

Quand un site reçoit le message RELEASE(Hj, Sj), il synchronise son horloge et enlève REQ(Hj,Sj) de la file de requête.

3. PRÉSENTATION DE LA SOLUTION ET DE L'IMPLÉMENTATION

Notre solution contient 4 modules :

1- le module **requestQ.py** implémente la base de la file d'attente de chaque site, nous avons utilisé **heapq** un algorithme qui organise une liste sous forme d'un tas binaire. Le module permet une utilisation très efficace du type "list" pour retirer un élément de telle sorte qu'ils soient toujours ordonnés. (La complexité pour retirer un élément est O(log(N))

2- le module **publisher.py** implémente la partie demande d'entrer en SC de l'algorithme de Lamport. Il permet à un site d'effectuer une demande de type "REQUEST" qui représente la demande d'entrée en section critique envoyé aux autres sites. Cette implémentation a été inspirée du **tutoriel** d'appel de procédure à distance (RPC) de RabbitMQ.

3-le module **consumer.py** implémente la partie d'entrée, d'exécution et de la libération de la SC de l'algorithme de lamport. il permet à un site d'être en écoute des messages envoyés sur une queue qui lui est associée.

Lorsque le consumer reçoit un message de type "REQUEST" il augmente son horloge logique et ajoute le message dans sa file de requêtes. Par la suite, il envoie un message de type "REPLY" directement à la queue associé au demandeur. Le consumer envois un message de type "RELEASE" aux autres sites lorsqu'il sort de la section critique.

L'implémentation du consumer est basé sur **Pika's Asynchronous consumer example**.

4-le module **site.py** reproduit le comportement d'une machine, un site peut faire une demande pour rentrer en section critique. Les étapes pour lancer un site sont ci-dessous :

- ➔ Pour instancier un site nous avons besoin de deux paramètres, "site_id" qui représente l'id du site ainsi que "total_site" qui représente le nombre total de site.
- ➔ Le site lance le consumer process
- ➔ Le site lance le publisher process pour demander de rentrer en section critique

Dans notre implémentation, chaque site peut envoyer et recevoir des messages. Donc chaque site est **subscriber** et **publisher** en même temps. Il existe un dernier processus **manager** qui gère le partage des données entre les deux premiers processus. Pour faire cela, nous avons utilisé le module python **multiprocessing**.

À partir du **site_id** et du **total_site**, chaque site crée un **exchange** et une **queue** (X_1 et Q_1 pour un site_id=1). l'exchange crée une connection avec toutes les autres queues appartenant aux sites restant. (voir schéma)

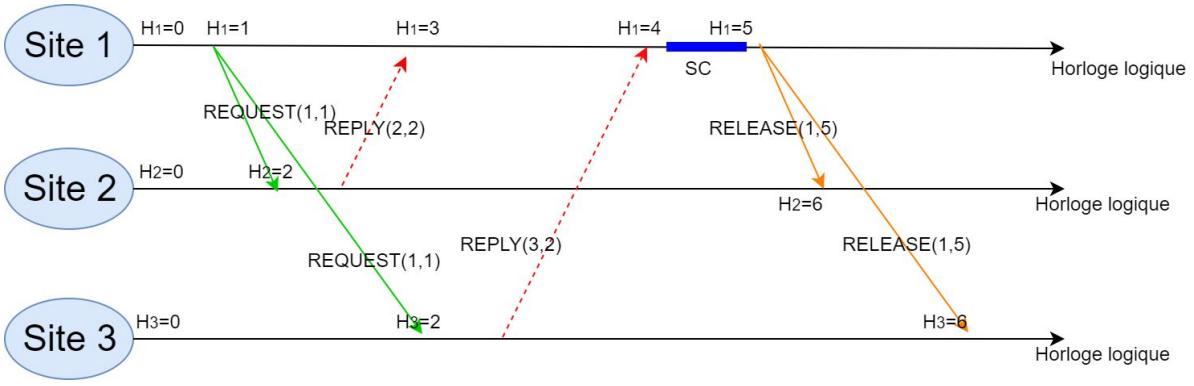
Chaque **exchange** est de type **fanout** (Broadcast). C'est-à-dire que pour chaque message envoyé par un site, tous les autres sites recevront ce message.

3. EXEMPLES D'UTILISATIONS

Pour les exemples d'utilisation nous avons trois exemples d'utilisations avec 3 sites : S1, S2 et S3.

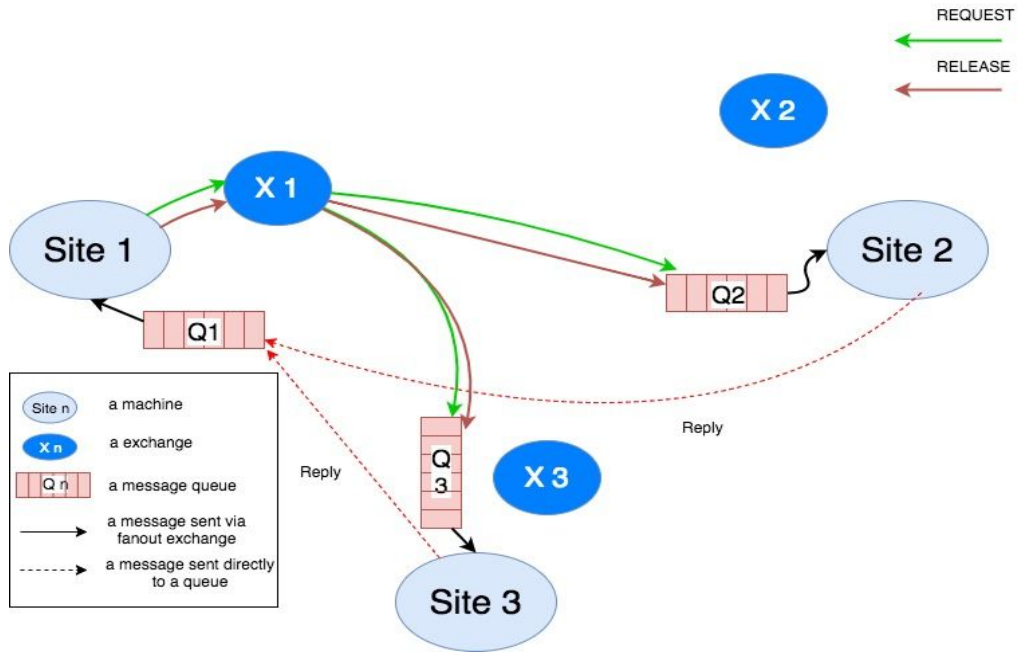
Exemple 1 : S1 demande à entrer en section critique SC

Pour cette exemple, nous avons 3 sites (S1,S2 et S3) nous allons dérouler l'algorithme pour S1 jusqu'à ce qu'il sorte de la SC.(demande de rentrer en SC, entre en SC, utilise une ressource et qui la libère ensuite)



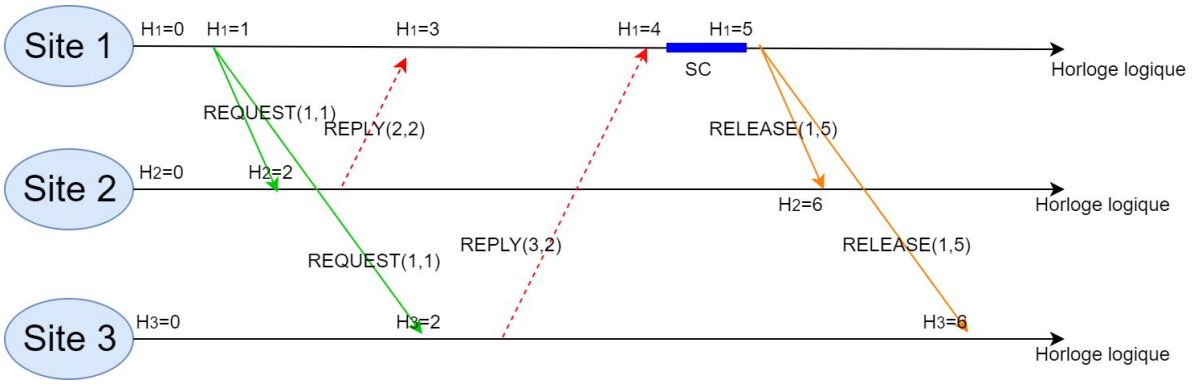
S1 augmente son horloge logique, dépose le message dans sa file de requêtes et envois un message de type "Request" sur l'exchange X1. X1 broadcast le message sur Q2 et Q3, les site S2 et S3 consomment la requête depuis leur queue respective (Q2 et Q3). Le site S2 et S3 remettent à jours leur files de requête et synchronisent leurs horloges. S2 et S3 envois des messages directes vers Q1 avec comme type "REPLY" daté.

Une fois que S1 a reçu tous les messages "REPLY" de tous les autres sites et que leur horloge est inférieur à son horloge il rentre en SC. Par la suite, quand S1 libéré la ressource il envoie un message de type "RELEASE" à tous les autres sites. Lorsque S2 et S3 reçoivent le message "RELEASE" ils synchronisent leurs horloges et enlève "REQUEST" de leur files de requêtes.



Exemple 2 : S2 demande à entrer en section critique

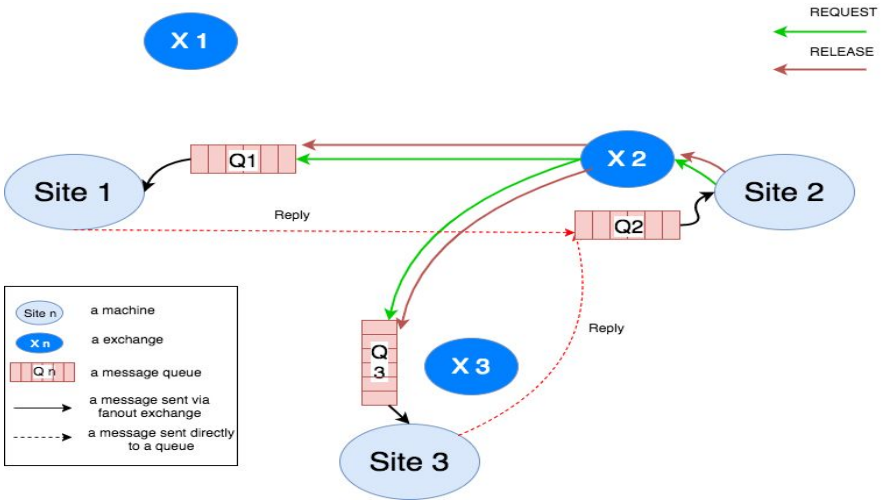
Pour cette exemple, nous avons 3 sites (S1,S2 et S3), nous allons dérouler l'algorithme pour S2 jusqu'à ce qu'il sorte de la SC. (demande de rentrer en SC, entre en SC, utilise une ressource et qui la libère ensuite)



S2 augmente son horloge logique, dépose le message dans la file de requêtes et envois un message de type "Request" sur l'exchange X2. X2 broadcast le message sur Q1 et Q3, les site S1 et S3 consomment la requête depuis leur queue respective (Q1 et Q3). Les sites S1

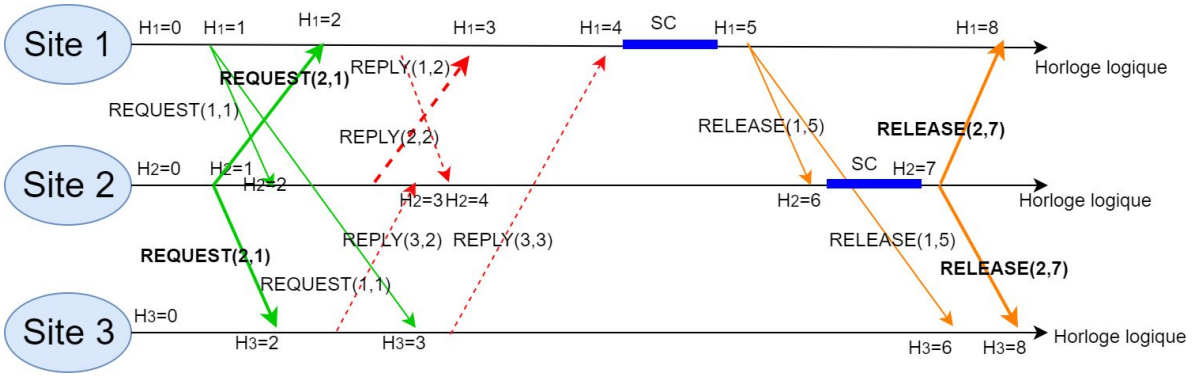
et S3 remettent à jours leur file de requête et synchronisent leurs horloges. S1 et S3 envois des messages directes vers Q2 avec comme type “REPLY” daté.

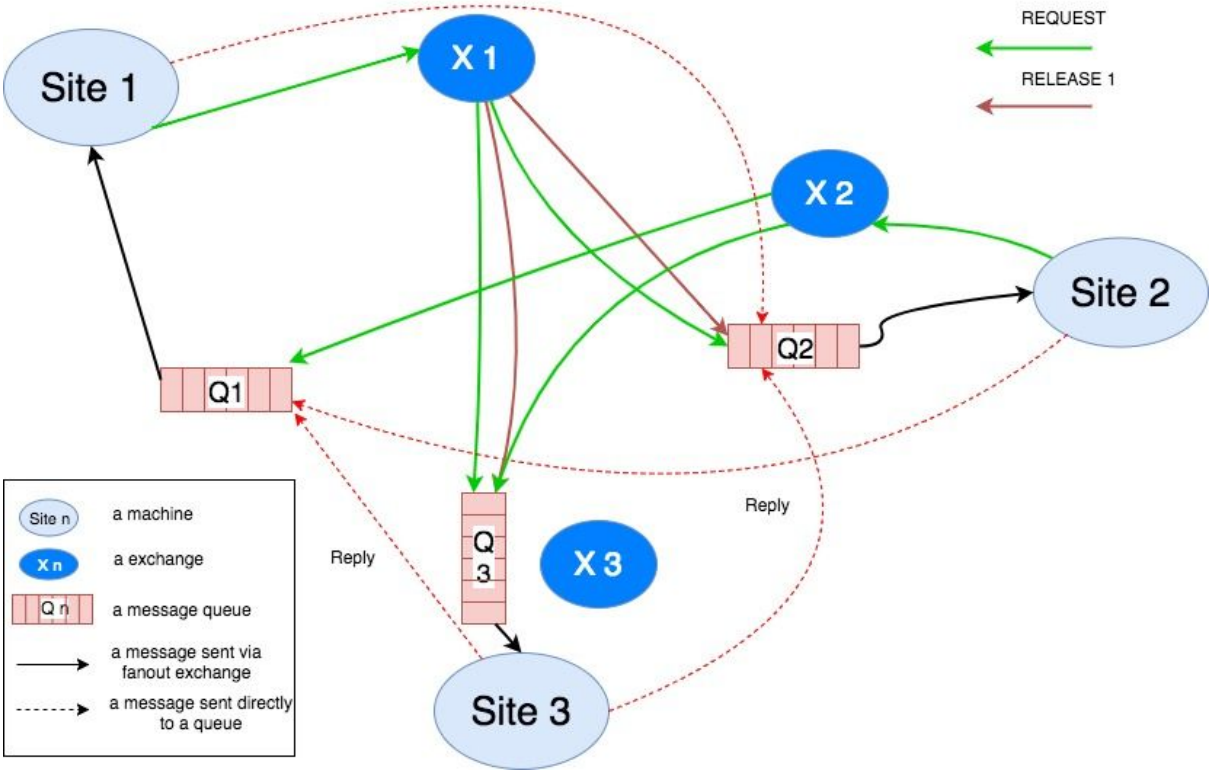
Une fois que S2 a reçu tous les messages “REPLY” de tous les sites et que leur horloge est inférieur à son horloge, il rentre en SC. Une fois qu’il a libéré la ressource il envoie un message de type “RELEASE” à tous les autres sites. Par la suite, quand S2 libéré la ressource il envoie un message de type “RELEASE” à tous les autres sites. Lorsque S1 et S3 reçoivent le message “RELEASE” ils synchronisent leurs horloges et enlève “REQUEST” de leur files de requêtes.



Exemple 3 : S1 et S2 demandent la SC en même temps

Dans le cas où S1 et S2 demandent la SC nous avons implémenté une solutions dans le module **requestQ** de sorte à donner la priorité au site ayant le plus petit identifiant donc S1 est prioritaire sur S2.





Dans le schéma nous avons la représentation des deux demandes simultanées, les deux sites ont reçu les messages de type “REPLY”. Le site 1 entre en SC et le site 2 ne rentre pas dans la SC car dans sa file de requêtes la requête du site 1 est prioritaire. Une fois le message “RELEASE” reçu du site 1, il supprime le message de type “REQUEST” de sa file de requête et rentre en SC.

4. EXTRAIT DU CODE

Module publisher.py :

```
import pika
import logging
import sys
import time

LOG_FORMAT = ('%(levelname) -10s %(asctime)s %(name) -30s %(funcName) '
              '-35s %(lineno) -5d: %(message)s')
LOGGER = logging.getLogger(__name__)


class Publisher(object):
```

```
  def __init__(self, exchange_name, queue_name):
    self._exchange_name = exchange_name
    self._queue_name = queue_name


    connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
    self._connection = connection
    channel = connection.channel()


    channel.exchange_declare(exchange=exchange_name,
                             exchange_type='fanout')
    self._channel = channel
    # lamport
    self._site_id = int(exchange_name[1:])

  def send_REQUEST(self, time):
    message = "{!s},{!s}".format(self._site_id,time)
    self._channel.basic_publish(exchange=self._exchange_name,
                                routing_key='',
                                body=message,
                                properties=pika.BasicProperties(reply_to=self._queue_name,type="REQUEST"))
    LOGGER.info('Broadcasted message : %s type REQUEST', message)


  def close_connection(self):
    self._connection.close()
```

Module consumer.py : Pour ce module les extrait les plus importants sont ci-dessous :
L'initialisation des variables de la classe consumer:

```
# -*- coding: utf-8 -*-

import logging
import pika
import sys
import json
import time
from multiprocessing import Process, Pipe
import os

LOG_FORMAT = ('%(levelname) -10s %(asctime)s %(name) -30s %(funcName) '
              '-35s %(lineno) -5d: %(message)s')
LOGGER = logging.getLogger(__name__)


class ExampleConsumer(object):
    """This is an example consumer that will handle unexpected interactions
    with RabbitMQ such as channel and connection closures.

    If RabbitMQ closes the connection, it will reopen it. You should
    look at the output, as there are limited reasons why the connection may
    be closed, which usually are tied to permission related issues or
    socket timeouts.

    If the channel is closed, it will indicate a problem with one of the
```

```
    commands that were issued and that should surface in the output as well.

    """

    EXCHANGE_TYPE = 'fanout'
    ROUTING_KEY = ''

    def __init__(self, queue_name, exchange_names, lock, logical_time, requestQ,
replys):
        """Create a new instance of the consumer class, passing in the AMQP
        URL used to connect to RabbitMQ.

        :param str amqp_url: The AMQP url to connect with

        """
        self._connection = None
        self._channel = None
        self._closing = False
        self._consumer_tag = None
        self._url = "localhost"
        self.QUEUE = queue_name
        self.exchange_bindings = exchange_names
        LOGGER.info('Queue is %s', self.QUEUE)

        # lamport
        self._site_id = int(queue_name[1:])
        self._lock = lock
        self._logical_time = logical_time
        self._requestQ = requestQ
        self._replys = replys
        self._number_of_REPLY = 0
        self._exchange_name = "X{!s}".format(self._site_id)

    def connect(self):
        """This method connects to RabbitMQ, returning the connection handle.
        When the connection is established, the on_connection_open method
        will be invoked by pika.

        :rtype: pika.SelectConnection

        """
        LOGGER.info('Connecting to %s', self._url)
        return pika.SelectConnection(pika.ConnectionParameters(host='localhost'),
                                     self.on_connection_open,
                                     stop_ioloop_on_close=False)

    def on_connection_open(self, unused_connection):
        """This method is called by pika once the connection to RabbitMQ has
        been established. It passes the handle to the connection object in
        case we need it, but in this case, we'll just mark it unused.

        :type unused_connection: pika.SelectConnection

        """
        LOGGER.info('Connection opened')
        self.add_on_connection_close_callback()
        self.open_channel()

    def add_on_connection_close_callback(self):
        """This method adds an on close callback that will be invoked by pika
        when RabbitMQ closes the connection to the publisher unexpectedly.

        """
        LOGGER.info('Adding connection close callback')
        self._connection.add_on_close_callback(self.on_connection_closed)
```

```
    def on_connection_closed(self, connection, reply_code, reply_text):
        """This method is invoked by pika when the connection to RabbitMQ is
        closed unexpectedly. Since it is unexpected, we will reconnect to
        RabbitMQ if it disconnects.

        :param pika.connection.Connection connection: The closed connection obj
        :param int reply_code: The server provided reply_code if given
        :param str reply_text: The server provided reply_text if given

        """
        self._channel = None
        if self._closing:
            self._connection.ioloop.stop()
        else:
            LOGGER.warning('Connection closed, reopening in 5 seconds: (%s) %s',
                           reply_code, reply_text)
            self._connection.add_timeout(5, self.reconnect)

    def reconnect(self):
        """Will be invoked by the IOLoop timer if the connection is
        closed. See the on_connection_closed method.

        """
        # This is the old connection IOLoop instance, stop its ioloop
        self._connection.ioloop.stop()

        if not self._closing:

            # Create a new connection
            self._connection = self.connect()

            # There is now a new connection, needs a new ioloop to run
            self._connection.ioloop.start()

    def open_channel(self):
        """Open a new channel with RabbitMQ by issuing the Channel.Open RPC
        command. When RabbitMQ responds that the channel is open, the
        on_channel_open callback will be invoked by pika.

        """
        LOGGER.info('Creating a new channel')
        self._connection.channel(on_open_callback=self.on_channel_open)

    def on_channel_open(self, channel):
        """This method is invoked by pika when the channel has been opened.
        The channel object is passed in so we can make use of it.

        Since the channel is now open, we'll declare the exchange to use.

        :param pika.channel.Channel channel: The channel object

        """
        LOGGER.info('Channel opened')
        self._channel = channel
        self.add_on_channel_close_callback()
        self.setup_exchange(self.exchange_bindings)

    def add_on_channel_close_callback(self):
        """This method tells pika to call the on_channel_closed method if
        RabbitMQ unexpectedly closes the channel.

        """
        LOGGER.info('Adding channel close callback')
        self._channel.add_on_close_callback(self.on_channel_closed)
```

```
    def on_channel_closed(self, channel, reply_code, reply_text):
        """Invoked by pika when RabbitMQ unexpectedly closes the channel.
        Channels are usually closed if you attempt to do something that
        violates the protocol, such as re-declare an exchange or queue with
        different parameters. In this case, we'll close the connection
        to shutdown the object.

        :param pika.channel.Channel: The closed channel
        :param int reply_code: The numeric reason the channel was closed
        :param str reply_text: The text reason the channel was closed

        """
        LOGGER.warning('Channel %i was closed: (%s) %s',
                       channel, reply_code, reply_text)
        self._connection.close()
```

#La fonction qui permet de setup tous les exchanges :

```
    def setup_exchange(self, exchange_names):
        """Setup the exchange on RabbitMQ by invoking the Exchange.Declare RPC
        command. When it is complete, the on_exchange_declareok method will
        be invoked by pika.

        :param str|unicode exchange_name: The name of the exchange to declare

        """

        for exchange_name in exchange_names:
            self._channel.exchange_declare(self.on_exchange_declareok,
                                           exchange_name,
                                           self.EXCHANGE_TYPE)
            LOGGER.info('Declaring exchange %s', exchange_name)

    def on_exchange_declareok(self, unused_frame):
        """Invoked by pika when RabbitMQ has finished the Exchange.Declare RPC
        command.

        :param pika.Frame.Method unused_frame: Exchange.DeclareOk response frame

        """
        LOGGER.info('Exchange declared')
        self.setup_queue(self.QUEUE)

    def setup_queue(self, queue_name):
        """Setup the queue on RabbitMQ by invoking the Queue.Declare RPC
        command. When it is complete, the on_queue_declareok method will
        be invoked by pika.

        :param str|unicode queue_name: The name of the queue to declare.

        """
        LOGGER.info('Declaring queue %s', queue_name)
        self._channel.queue_declare(self.on_queue_declareok, queue_name)

    def on_queue_declareok(self, method_frame):
        """Method invoked by pika when the Queue.Declare RPC call made in
        setup_queue has completed. In this method we will bind the queue
        and exchange together with the routing key by issuing the Queue.Bind
        RPC command. When this command is complete, the on_bindok method will
        be invoked by pika.

        :param pika.frame.Method method_frame: The Queue.DeclareOk frame

        """
```

```
        LOGGER.info('Binding %s to %s with %s',
                    self.exchange_bindings, self.QUEUE, self.ROUTING_KEY)
        for ex in self.exchange_bindings:
            self._channel.queue_bind(self.on_bindok, self.QUEUE,
                                     ex)

    def on_bindok(self, unused_frame):
        """Invoked by pika when the Queue.Bind method has completed. At this
        point we will start consuming messages by calling start_consuming
        which will invoke the needed RPC commands to start the process.

        :param pika.frame.Method unused_frame: The Queue.BindOk response frame

        """
        LOGGER.info('Queue bound')
        self.start_consuming()

    def start_consuming(self):
        """This method sets up the consumer by first calling
        add_on_cancel_callback so that the object is notified if RabbitMQ
        cancels the consumer. It then issues the Basic.Consume RPC command
        which returns the consumer tag that is used to uniquely identify the
        consumer with RabbitMQ. We keep the value to use it when we want to
        cancel consuming. The on_message method is passed in as a callback pika
        will invoke when a message is fully received.

        """
        LOGGER.info('Issuing consumer related RPC commands')
        self.add_on_cancel_callback()
        self._consumer_tag = self._channel.basic_consume(self.on_message,
                                                         self.QUEUE)

    def add_on_cancel_callback(self):
        """Add a callback that will be invoked if RabbitMQ cancels the consumer
        for some reason. If RabbitMQ does cancel the consumer,
        on_consumer_cancelled will be invoked by pika.

        """
        LOGGER.info('Adding consumer cancellation callback')
        self._channel.add_on_cancel_callback(self.on_consumer_cancelled)

    def on_consumer_cancelled(self, method_frame):
        """Invoked by pika when RabbitMQ sends a Basic.Cancel for a consumer
        receiving messages.

        :param pika.frame.Method method_frame: The Basic.Cancel frame

        """
        LOGGER.info('Consumer was cancelled remotely, shutting down: %r',
                    method_frame)
        if self._channel:
            self._channel.close()
```

#Fonction on_message gère la réception d'un message (3 types de message) par un site :

```
    def on_message(self, unused_channel, basic_deliver, properties, body):
        """Invoked by pika when a message is delivered from RabbitMQ. The
        channel is passed for your convenience. The basic_deliver object that
        is passed in carries the exchange, routing key, delivery tag and
        a redelivered flag for the message. The properties passed in is an
        instance of BasicProperties with the message properties and the body
        is the message that was sent.
```

```
        :param pika.channel.Channel unused_channel: The channel object
        :param pika.Spec.Basic.Deliver: basic_deliver method
        :param pika.Spec.BasicProperties: properties
        :param str|unicode body: The message body

        """
        LOGGER.info('Received message # %s: %s properties %s',
                    basic_deliver.delivery_tag, body, properties)
        self.acknowledge_message(basic_deliver.delivery_tag)
        if properties.type == "REQUEST":
            site, logical_time = body.split(',')
            site, logical_time = int(site), int(logical_time)
            self._requestQ.add_request(site, logical_time)
            self._logical_time.value = max(self._logical_time.value, logical_time)+1
            LOGGER.info('Added request from site[%s] at time[%s]', site, logical_time)
            LOGGER.info('Request queue size:{!s}, logical time:
{!s}'.format(self._requestQ.size(), self._logical_time.value))
            # send REPLY msg
            self.send_REPLY(properties.reply_to)
        elif properties.type == "REPLY":
            site, logical_time = body.split(',')
            site, logical_time = int(site), int(logical_time)
            self._logical_time.value = max(self._logical_time.value, logical_time)+1
            self._replys[site-1] = logical_time
            LOGGER.info('Received REPLY from site[%s] at time[%s]', site, logical_time)
            LOGGER.info('Request queue size:{!s}, logical time:
{!s}'.format(self._requestQ.size(), self._logical_time.value))
            # try to enter Critcal Section
            self._number_of_REPLY += 1
            if self.can_enter_crtical_section():
                self.enter_crtical_section()
        elif properties.type == "RELEASE":
            site, logical_time = body.split(',')
            site, logical_time = int(site), int(logical_time)
            peek_site, peek_time = self._requestQ.peek_request()
            if peek_site != site:
                LOGGER.error('RELEASE site[%s] not equal to site[%s]', peek_site, site)
            else:
                pop_site, pop_time = self._requestQ.pop_request()
                self._logical_time.value = max(self._logical_time.value, logical_time)+1
                LOGGER.info('Deleted request from site[%s] at time[%s]', pop_site,
pop_time)
                LOGGER.info('Request queue size:{!s}, logical time:
{!s}'.format(self._requestQ.size(), self._logical_time.value))
                # try to enter Critcal Section
                if self.can_enter_crtical_section():
                    self.enter_crtical_section()
```

Fonction qui teste la possibilité d'entrer en section critique :

```
    def can_enter_crtical_section(self):
        try:
            peek_site, request_time = self._requestQ.peek_request()
        except KeyError as err:
            return False
        if self._number_of_REPLY != len(self._replys) - 1 or self._site_id != peek_site:
            return False
        for i in range(1, len(self._replys)+1):
            if self._site_id != i:
                if self._replys[i-1] <= request_time:
                    return False
        return True
```

La fonction qui permet de rentrer en section critique :

```
    def enter_crtical_section(self):
```

```
        LOGGER.info('Site[{!s}] enters crtical section'.format(self._site_id))
        time.sleep(2)
        self._number_of_REPLY = 0
        self._logical_time.value +=1
        peek_site, peek_time = self._requestQ.peek_request()
        if peek_site != self._site_id:
            LOGGER.error('RELEASE site[%s] not equal to site[%d]', peek_site,
self._site_id)
        else:
            self._requestQ.pop_request()
            LOGGER.info('Deleted request from site[%d] at time[%s]', self._site_id,
self._logical_time.value)
            LOGGER.info('Request queue size:{!s}, logical time:
{!s}'.format(self._requestQ.size(), self._logical_time.value))
        self.send_RELEASE()



    def acknowledge_message(self, delivery_tag):
        """Acknowledge the message delivery from RabbitMQ by sending a
        Basic.Ack RPC method for the delivery tag.

        :param int delivery_tag: The delivery tag from the Basic.Deliver frame

        """
        LOGGER.info('Acknowledging message %s', delivery_tag)
        self._channel.basic_ack(delivery_tag)

    def stop_consuming(self):
        """Tell RabbitMQ that you would like to stop consuming by sending the
        Basic.Cancel RPC command.

        """
        if self._channel:
            LOGGER.info('Sending a Basic.Cancel RPC command to RabbitMQ')
            self._channel.basic_cancel(self.on_cancelok, self._consumer_tag)

    def on_cancelok(self, unused_frame):
        """This method is invoked by pika when RabbitMQ acknowledges the
        cancellation of a consumer. At this point we will close the channel.
        This will invoke the on_channel_closed method once the channel has been
        closed, which will in-turn close the connection.

        :param pika.frame.Method unused_frame: The Basic.CancelOk frame

        """
        LOGGER.info('RabbitMQ acknowledged the cancellation of the consumer')
        self.close_channel()

    def close_channel(self):
        """Call to close the channel with RabbitMQ cleanly by issuing the
        Channel.Close RPC command.

        """
        LOGGER.info('Closing the channel')
        self._channel.close()

    def run(self):
        """Run the example consumer by connecting to RabbitMQ and then
        starting the IOLoop to block and allow the SelectConnection to operate.

        """
        self._connection = self.connect()
        self._connection.ioloop.start()
```

```
    def stop(self):
        """Cleanly shutdown the connection to RabbitMQ by stopping the consumer
        with RabbitMQ. When RabbitMQ confirms the cancellation, on_cancelok
        will be invoked by pika, which will then closing the channel and
        connection. The IOLoop is started again because this method is invoked
        when CTRL-C is pressed raising a KeyboardInterrupt exception. This
        exception stops the IOLoop which needs to be running for pika to
        communicate with RabbitMQ. All of the commands issued prior to starting
        the IOLoop will be buffered but not processed.

        """
        LOGGER.info('Stopping')
        self._closing = True
        self.stop_consuming()
        self._connection.ioloop.start()
        LOGGER.info('Stopped')

    def close_connection(self):
        """This method closes the connection to RabbitMQ."""
        LOGGER.info('Closing connection')
        self._connection.close()

    def send_REPLY(self, dest_queue):
        message = "{!s},{!s}".format(self._site_id, self._logical_time.value)
        self._channel.basic_publish(exchange='',
                            routing_key=dest_queue,
                            body=message,
                            properties=pika.BasicProperties(type="REPLY"))
        LOGGER.info('Sent message : %s, type REPLY', message)
        LOGGER.info('Request queue size:{!s}, logical time:
{!s}'.format(self._requestQ.size(), self._logical_time.value))
```

La fonction qui permet d'envoyer une RELEASE (sortie de SC) :

```
    def send_RELEASE(self):
        message = "{!s},{!s}".format(self._site_id,self._logical_time.value)
        self._channel.basic_publish(exchange=self._exchange_name,
                            routing_key='',
                            body=message,
                            properties=pika.BasicProperties(type="RELEASE"))
        LOGGER.info('Broadcasted message : %s type RELEASE', message)
        LOGGER.info('Request queue size:{!s}, logical time:
{!s}'.format(self._requestQ.size(), self._logical_time.value))



def main(queue_name, exchange_names):
     logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
     example = ExampleConsumer(queue_name, exchange_names)
     try:
         example.run()
     except KeyboardInterrupt:
         example.stop()


if __name__ == '__main__':
     if len(sys.argv) != 3:
         print("usage: python consumer.py its_queue_name
binding_exchange_name1,binding_exchange_name2")
     else:
         main(sys.argv[1], sys.argv[2].split(','))
         print(sys.argv[1], sys.argv[2].split(','))
```

Module site.py :

```
#!/usr/bin/env python
import logging
import pika
import sys
import json
import time
from multiprocessing import Process, Pipe, Value, Array, Lock
from multiprocessing.managers import BaseManager
from consumer import ExampleConsumer
from publisher import Publisher
from requestQ import RequestQueue

LOG_FORMAT = ('%(levelname) -10s %(asctime)s %(name) -30s %(funcName) '
              '-35s %(lineno) -5d: %(message)s')
LOGGER = logging.getLogger(__name__)

class requestQManager(BaseManager):
    pass

class Site(object):

    def __init__(self, site_id, site_number):
        'parse parameter, create sharing object between process'
        if type(site_id) is str:
            site_id = int(site_id)
        if type(site_number) is str:
            site_number = int(site_number)
        self._site_id = site_id
        self._its_queue_name = "Q{!s}".format(site_id)
        self._its_exchange_name = "X{!s}".format(site_id)
        exchange_names = []
        for i in range(1, site_number+1):
            if i != site_id:
                exchange_names.append("X{!s}".format(i))
        self._binding_exchange_names = exchange_names
        print(self._its_queue_name, self._its_exchange_name,self._binding_exchange_names)
        # declare sharing object
        requestQManager.register('RequestQueue', RequestQueue, exposed = ['add_request', 'pop_request',
'peek_request', 'size'])
        self._lock = Lock()
        self._logical_time = Value('i', 0)
        self._replys = Array('i', [0 for i in range(site_number)], lock=self._lock)
        self._mymanager = requestQManager()
        self._mymanager.start()
        self._requestQ = self._mymanager.RequestQueue()

    def start_consumer(self):
        example = ExampleConsumer(self._its_queue_name,
            self._binding_exchange_names, self._lock, self._logical_time,
            self._requestQ, self._replys)
        try:
            example.run()
        except KeyboardInterrupt:
            example.stop()

    def run_consumer_process(self):
        self._p = Process(target=self.start_consumer, args=())
        self._p.start()

    def start_publisher(self):
        self._publisher = Publisher(self._its_exchange_name, self._its_queue_name)
```

```
    def request_for_critical_section(self):
        self._requestQ.add_request(self._site_id, self._logical_time.value)
        self._logical_time.value += 1
        self._publisher.send_REQUEST(self._logical_time.value)

if __name__ == '__main__':
    if len(sys.argv) != 3:
        print("usage: python site.py its_id total_count_of_sites")
    else:
        logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
        site = Site(sys.argv[1], sys.argv[2])
        site.run_consumer_process()
        site.start_publisher()
        # scenario begins here
        if sys.argv[1] == '1':
            time.sleep(1)
            site.request_for_critical_section()
        # scenario ends here
        site._p.join()
```

module requestQ.py:

```
#!/usr/bin/env python
from heapq import heappush, heappop

class RequestQueue(object):

    def __init__(self):
        self._pq = []                       # list of entries arranged in a heap
        self._entry_finder = {}             # mapping of tasks to entries
        self._REMOVED = '<removed-task>'    # placeholder for a removed task

    def add_request(self, site, time):
        'Add a new request from a site, no duplicate request from a same site'
        if type(site) is str:
            site = int(site)
        if type(time) is str:
            time = int(time)
        if site in self._entry_finder:
            return
        entry = [time, site]
        self._entry_finder[site] = entry
        heappush(self._pq, entry)

    def remove_request(self, site):
        'Mark an existing request from a site as REMOVED. Raise KeyError if not found.'
        entry = self._entry_finder.pop(site)
        entry[-1] = self._REMOVED

    def pop_request(self):
        'Remove and return the lowest logical time request. Raise KeyError if empty.'
        while self._pq:
            time, site = heappop(self._pq)
            if site is not self._REMOVED:
                del self._entry_finder[site]
                return site, time
        raise KeyError('pop from an empty priority queue')

    def peek_request(self):
        'Peek the lowest logical time request. Raise KeyError if empty.'
        while self._pq:
```

```
            if len(self._pq) < 1:
                raise KeyError('pop from an empty priority queue')
                return
            time, site = self._pq[0]
            if site is self._REMOVED:
                heappop(self._pq)
                del self._entry_finder[site]
            else:
                return site, time
        raise KeyError('peek from an empty priority queue')

    def size(self):
        return len(self._pq)

if __name__ == "__main__":
    requestQ = RequestQueue()
    requestQ.add_request(3,0)
    requestQ.add_request(1,1)
    requestQ.add_request(2,1)
    for i in range(3):
        print requestQ.peek_request()
        print requestQ.pop_request()
```

5. SYNTHÈSE DE LA SOLUTION

Nous avons dans un premier lieu créer un Producer et Consumer en RabbitMQ, notre choix c'est porté sur une **connection** de **type** SelectConnection pour le consumer, car sinon le consumer ne peut pas envoyer et recevoir des messages en même temps. Le choix de connection à été difficile car lors des tutoriels de RabbitMQ, les exercices on été effectué avec des connecteur de type blocking, nous avons donc lu la documentation de pika pour trouver une solution asynchrone. Une fois le producer et le consumer implémenté nous avons implémenté le module site.

Pour **combiner** le **Publisher** et **consumer** nous avons eu 3 choix : **thread**, **process** et une **connection purement asynchrone**. Concernant le choix entre les thread et les processus pour le site. les threads possèdent un mémoire partagé ce qui n'est pas le cas pour les processus. Sauf que les processus sont plus robust que les threads, nous avons donc décidé d'utiliser les processus.

Concernant le **partages de données entres les processus** nous avons aussi eux 3 différentes alternatives : **Queue**, **Pipe** et un **processus gérant**. Les deux premières solutions sont limité car elle ne sont pas conçu pour le partage de structures de données complexes (Queue et pipe sont efficace pour des solutions primitive et non des objets). Nous avons donc utilisé un processus gérant.

RabbitMQ nous a facilité l’implémentation car nous n’avons pas eu à gérer la communication entre les différents site. Nous avons donc pu communiquer facilement et rapidement entre plusieurs sites grâce à ce Middleware orienté messages.