Antoine GUCKER

Projet RabbitMQ : implémentation d'un jeu de Loups-garous

Langage utilisé : Python 3

Introduction

Ce projet permet de jouer aux jeu Loup-garous depuis des terminaux, sans avoir besoin qu'un joueur soit meneur de jeu.
Il est composé de 2 fichiers :

- **host.py**, qui fait office de meneur de jeu. Il gère l'inscription des joueurs à une partie, la répartition des rôles et organise les votes.
- **player.py**, qui gère la réception des messages envoyés par host.py et qui permet aux joueurs d'interagir avec le jeu.

Une instance de host.py est démarrée pour la partie, et chaque joueur démarre une instance de player.py.

RabbitMQ est utilisé pour la communication entre l'hôte et les joueurs.

Utilisation de RabbitMQ

Les queues

L'hôte dispose de 2 queues :

- **add_players**, sur laquelle les joueurs s'inscrivent à la partie lors du démarrage. Elle ne sert plus une fois le bon nombre de joueurs atteint.
- **host_queue**, qui sert à la réception de messages des joueurs au cours de la partie.

Les joueurs disposent chacun d'une queue. Son nom est le même que celui du joueur.
Ces queues sont liées à la clé "**player**" au début de la partie.

```
channel.queue_bind(
        exchange='host_to_players', queue=Player_name,
routing_key="player")
```

De plus, celles des loups-garous sont liées à la clé "**werewolf**" lors de la répartition des rôles.

```
channel.queue_bind(
                exchange='host_to_players', queue=Player_name,
routing_key="werewolf")
```

Traitement des messages

Les messages sont collectés à l'aide de **start_consuming()**.
Lors du traitement d'un message envoyé par l'hôte à un joueur, les 4 premiers caractères du message servent d'étiquette :

- "info" : seulement afficher le contenu
- "ack " : renvoyer un accusé de réception après avoir lié sa queue avec la clé "werewolf". L'accusé de réception permet de synchroniser les joueurs et l'hôte.
- "vote" : demander au joueur de voter, et renvoyer sa réponse
- "elim" : éliminer le joueur en terminant son programme.

Architecture du code

host.py

La première phase consiste à inscrire le nombre de joueurs voulu à la partie.

```
# WAITING FOR PLAYERS

# Queue for inscription only
channel.queue_declare(queue='add_players')

print( ' [*] Waiting for messages. To exit press CTRL+C')

def callback(ch, method, properties, body):

    Body = body.decode()
    print(Body + " has joined.")

    list_of_players.append(Player(Body, None))

    if len(list_of_players) == Number_of_players :
        channel.stop_consuming()

channel.basic_consume(on_message_callback=callback, queue='add_players', auto_ack=True)

channel.start_consuming()
```

Pour cela, chaque joueur envoie son nom à l'hôte sur la queue "**add_players**".
L'hôte les inscrit dans list_of_players, et démarre la partie dès le bon nombre de joueurs atteint.
La partie commence par la répartition des rôles.

La phase de jeu consiste en la boucle suivante :

```
# GAME LOOP

while True :
    murder_by_werewolves()

    if check_victory(Number_of_werewolves, Number_of_players) :
        return

    village_vote()

    if check_victory(Number_of_werewolves, Number_of_players) :
        return
```

- Les loups-garous désignent un joueur à éliminer
- Si un camp est éliminé, la partie s'arrête
- Le village élit un joueur à éliminer
- Si un camp est éliminé, la partie s'arrête.

Ainsi, la partie s'arrête dès qu'à l'issue d'une élimination, un camp est décimé.

Lors des votes, la réception des messages des votants se fait lors d'une boucle **start_consume()**.
Un compteur est utilisé pour compter les votes. Il est incrémenté lors l'appel à la méthode on_message_callback=**vote_callback**. Une fois le total atteint, la boucle cesse à l'aide de **stop_consume()**.

```
def vote_callback(ch, method, properties, body):#

    self.count += 1

    Body = body.decode()

    try :
        k = int(Body)
        self.result[k] += 1
    except :
        pass

    if self.count == len(self.voters) :
        channel.stop_consuming()
        print("Vote finished ")
```

player.py

Pour le joueur, après l'envoi de son nom à l'hôte, la phase de jeu consiste en une boucle dans laquelle le joueur attend de recevoir des messages :

```python
def callback(ch, method, properties, body):

    Body = body.decode()
    header = Body[:4] # The first 4 characters are beyond : "info" / "ack " / "vote" / "elim"
    message = Body[4:]

    if header == "info" : ···


    elif header == "ack " : ···


    elif header == "vote" :···

    elif header == "elim" : ···

    else : ···


channel.basic_consume(
    queue=Player_name, on_message_callback=callback, auto_ack=True)


channel.start_consuming()
```

Conclusion

RabbitMQ est adapté à la mise-en-place de systèmes distribués. Il est facile d'utilisation, et permet des échanges asynchrones de messages.
En connectant plusieurs ordinateurs au même serveur RabbitMQ, il devrait être possible de faire fonctionner mon jeu avec des appareils distants.

L'utilisation de la méthode **start_consume()** n'est pas très pratique, étant donné qu'elle oblige à utiliser des variables globales pour accéder aux données depuis son **on_message_callback**. Il aurait été plus pratique d'utiliser une méthode qui ne consomme que le prochain message dans la queue, sans rentrer dans une boucle.