TRAVAUX DIRIGÉS
AGENTS DE MESSAGES

I. introduction

le but de ce TD est d'utiliser un agent de messages pour permettre à des micro-services de dialoguer entre eux. Pour cela nous allons utiliser l'outil NATS (https://nats.io)

II. Environnement

Le service d'agent de message est constitué d'un serveur qui va assurer le transit des message. Les micro-services vont publier sur ce serveur ou s'abonner sur des sujets pour réceptionner des messages. Le serveur NATS peut être installé nativement sur votre machine ou être déployé dans un conteneur. Dans un premier temps nous allons procéder à l'installation sur votre Vm

II.1. **installation.** Pour cela, vous aller vérifier quelle est la dernière version disponible de nats-server sur https://github.com/nats-io/nats-server/releases/ puis vous aller utiliser les commandes suivantes pour installer le soft

```
curl -L https://github.com/nats-io/nats-server/releases/download/vX.Y.Z/nats-server-vX.Y.Z-linux-amd64.zip -o nats-server.zip
unzip nats-server.zip -d nats-server
```

ou X.Y.Z correspond à la version de release. Enfin, vous déplacer le binaire du serveur dans un répertoire du path

```
sudo cp nats-server/nats-server-vX.Y.Z-linux-amd64/nats-server /usr/bin
```

II.2. **lancement du serveur.** Vous pouvez ensuite lancer le serveur à l'aide de la commande

```
nats-server
```

qui lance le serveur avec le port d'écoute par défaut 4222. La commande nats-server admet les options suivantes :

- **-p, –port** : pour préciser le port d'écoute
- **-a, –addr** : pour préciser l'adresse de l'hote, par défaut 0.0.0.0 pour écouter toutes les interfaces
- **-m, -http_port** : pour préciser le port de monitoring du serveur en http
- **-ms, -https_port** : pour préciser le port de monitoring du serveur en https

III. publish-subscribe

III.1. **actions.** Le principe d'action de l'agent de messages est le suivant. Des clients vont :

- publier des messages
- publier un message nécessitant une réponse
- s'abonner à des sujets et récupérer les messages les concernants

III.2. **sujet.** La gestion des messages se fait à l'aide du sujet. Le sujet est constitué d'une suite de caractères formant des mots. Ces noms sont sensible à la casse, ne doivent pas contenir d'espace ni les caractères spéciaux > et *. Les sujets peuvent être hiérarchisé, le séparateur . dans la chaine de caractère du sujet permet de créer cette hiérarchie. Par exemple on peut définir les sujets suivants dans le cas d'une messagerie liée aux horloges mondiales

→ time.us

→ time.us.east

→ time.us.east.atlanta

→ time.eu.east

dans cet exemple, un client peux s'abonner sur tout ou partie des sujets. Par exemple, le client 1 peut s'abonner au sujet **time.us.east** et recevra les messages de ce sujet uniquement, alors que le client 2 s'abonnant à **time.*.east** recevra les messages des sujets **time.us.east** et **time.eu.east**. L'étoile sert de joker pour un mot. Le > permet quand a lui de prendre en compte tout les formats qui suivent. Par exemple, le client 3 s'abonnant à **time.us.>** recevra les messages des sujets **time.us.east** et **time.us.east.atlanta**. On peux bien sur mixer ces caractères pour constituer une pile de sujets récupérer.

IV. code des clients

La publication et l'abonnement à des sujets vont être traité en asynchrone dans vos codes python. pour pouvoir développer pour NATS, il faut installer le package nats-py dans votre projet python

```
pip install nats-py
```

Toute la programmation va ensuite s'effectuer dans un boucle événementielle de gestion asynchrone des requêtes. Votre code doit donc importer les packages asyncio et nats. Le premier permet de gérer des actions asynchrones et de ne pas bloquer le fil d'exécution en utilisant des fonctions de callback. Pour plus d'info, je vous conseille la lecture de cet article

https://zestedesavoir.com/articles/1568/decouvrons-la-programmation-asynchrone-en-python/

présentant les principes de la programmation asynchrone.

IV.1. **connexion au serveur NATS.**

Votre code doit être défini dans une fonction précédée du mot clé *async* pour indiquer qu'elle utilise un fonctionnement asynchrone. Ensuite on utilise la méthode **connect** du package nats en précisant comme premier argument le serveur de messagerie que nous utiliseront. Ici, c'est le serveur '127.0.0.1 :4222' Vous n'êtes pas obligé de préciser le port, ni même le serveur car par défaut la fonction **connect** utilise ces valeurs. L'appel de cette méthode renvois la socket de connexion. Cet appel doit être précédé de *await*

```
import asyncio
import nats

async def mafonction():
    #connexion au serveur
    nc = await nats.connect("http://127.0.0.1:4222")

    # souscription a un sujet ou publication

    #fermeture de la connexion
    await nc.close()
if __name__ == '__main__':
  asyncio.run(mafonction())
```

IV.2. **publication d'un message.**

lorsqu'un service doit envoyer un message, on utilise la fonctionnalité de publication

```
import asyncio
import nats

async def mafonction():
    #connexion au serveur
    nc = await nats.connect("http://127.0.0.1:4222")

    # souscription a un sujet ou publication
    await nc.publish('sujet',b'message')
    #fermeture de la connexion
    await nc.close()
if __name__ == '__main__':
  asyncio.run(mafonction())
```

le b situé devant la chaine de message permet de la transformer en binaire. On peux aussi utiliser la méthode encode() pour transformer en binaire comme dans le cas d'une socket. À la suite de l'instruction publish,

S'il peut être intéressant qu'un autre service réponde, nous pouvons préciser une boite de réponse qui servira de sujet pour le service répondant.

```
    inbox = nc.new_inbox()
    await nc.publish('sujet',b'message',reply=inbox)
```

lorsqu'un message est publié avec une demande de réponse le service répondant va utiliser la boite de réponse comme sujet et publier ses données sur ce sujet.

IV.3. **souscription à un sujet.** lorsqu'un service doit s'abonner à un sujet, nous allons utiliser la méthode subscribe pour s'abonner. La réception de message peut être synchrone ou asynchrone. Si elle est synchrone, on s'abonne a un sujet et on reste en attente de l'arrivée d'un message que l'on traite (ou alors on récupère une erreur de timeout si cela a été défini)

```
import asyncio
import nats

async def mafonction():
    #connexion au serveur
    nc = await nats.connect("http://127.0.0.1:4222")

    # souscription a un sujet synchrone
    sub = nc.subscribe('sujet')
    msg = await sub.next_msg()
    print(f"message recu {msg.data}")
    #fermeture de la connexion
    await nc.close()
if __name__ == '__main__':
  asyncio.run(mafonction())
```

Si on souhaite traiter les messages de manière asynchrone, il faut définir une fonction de callback qui sera appelée à la réception d'un message

```
import asyncio
import nats

async def cb(msg):
     print(f"message recu {msg.data}")

async def mafonction():
    #connexion au serveur
    nc = await nats.connect("http://127.0.0.1:4222")

    # souscription a un sujet synchrone
    sub = await nc.subscribe('sujet', cb=cb)

    #fermeture de la connexion
    await nc.close()
if __name__ == '__main__':
  asyncio.run(mafonction())
```

Dans ce cas, la fonction **cb** sera appelé lorsqu'un message arrive sur le service

IV.4. **Gérer une requête.** On peux aussi gérer une requête et sa réponse d'une autre façon qu'avec la publication et l'abonnement à une inbox. On peux utiliser la méthode **request**

```
import asyncio
import nats

async def mafonction():
    #connexion au serveur
    nc = await nats.connect("http://127.0.0.1:4222")

    # souscription a un sujet synchrone
    msg = await nc.request('sujet', b'data')
    print (f" reponse recue {msg.data}"}
    #fermeture de la connexion
    await nc.close()
if __name__ == '__main__':
  asyncio.run(mafonction())
```

V. Pratique

V.1. **compteur.** Vous allez mettre en place un seveur nats et créer 2 services. Un qui publie toute les 10 secondes, un message avec le sujet "compteur" et comme donnée une valeur incrémentée de 2 et un second service qui s'abonne à ce sujet et qui affiche les données du compteur.

V.2. **sujet hierarchisé.** Créer 4 services :

(1) un qui va publier des messages avec des sujets hiérarchisés de la forme fr.région.département.ville ;
(2) le second abonné aux message du sujet fr.grand_est.67.colmar
(3) le suivant abonné à tous les sujets de type fr.grand_est, quelque soit le département ou la ville ;
(4) enfin le dernier qui est abonné au sujet sur Strasbourg, même si cela n'est pas dans le grand_est ou dans le bas Rhin

Tester votre structure de micro-services avec plusieurs sujets différents, voir si la hiérarchisation fonctionne.

V.3. **requêtes avec réponse.** vous allez créer deux services, le premier qui s'abonne au sujet "hotline", et le second qui publiera sur ce sujet en demandant un retour du service. Dans les data, prévoyer un code permettant d'avoir un retour en fonction de ces données. Les data peuvent correspondre à une chaine Json issue d'un dictionnaire. il faudra décoder dans le premier service la chaine, traité la donnée et renvoyer une réponse en correspondance. Vous pouvez utiliser une publication sur l'inbox définie implictement ou explicitement par le publiant, ou utiliser la méthode **respond** associée au message reçu pour renvoyer les données de la réponse.

V.4. **chat multi client.** Vous allez faire un logiciel de chat ou chaque client choisi le ou les sujets aux quels il souhaite s'abonner. Pour chaque message reçu, vous indiquerez si vous souhaitez renvoyer un message en retour ou non sur le même sujet ou transférer le message sur un autre sujet. l'abonnement peux se faire à l'aide des wildcards (* et >)