PATRONS ET MODÈLES

TP#1 : Communications S2S

UQAC

LEYE Mame Ramatoulaye

RAHARIJAONA Anja

10/02/2020

Explication du Design :

Notre architecture se décompose en 4 grosses parties pouvant être gérées indépendamment les unes des autres. Nous citons :

- ***Tâches*** : Notre système de tâches a été illustré en utilisant le patron Command. En effet , nou pouvons créer différentes types de tâches découlant de l'interface Task. Nous avons implémenté "SendMsgTask" comme nous aurions pu implémenter "CallTask" ou "DrawSomething" , qui sont également des tâches qui diffèrent sur le comportement dans leur exécution, mais auront le même squelette.
- ***Communication*** :
  Chaque protocole de communication est créé à partir du pattern Factory.
- ***Builder*** : la construction de notre API se base sur le pattern Builder et fera l'initialisation nécessaire.
- ***Traffic*** : c'est dans cette partie que nous gérons les flux grâce au pattern Decorator qui nous permettra de traiter des flux sous différentes formes.

Nous avons une liste de tâches et de threads qui sont présents dans une classe dite "Executor" qui se chargera du travail de notre Job System.
Nous avons choisi de ne pas utiliser le pattern Singleton et le remplaçons par des injections de dépendance par constructeur.

Utilisation de la librairie:

Pour l'initialisation de notre librairie, nous passons en paramètres de notre Builder les paramètres suivants :

- le protocole de communication (UDP, TCP ..)
- la propriété du flux (clair, compressé, encrypté ..)
- le nombre de threads

Les 2 premiers paramètres sont issus de classes Enum que nous pouvons agrandir si besoin. Les différentes classes faisant fonctionner l'API sont donc créés en fonctions des précédents paramètres.
Pour rester dans une optique de simulation, nous créons pendant l'exécution un certain nombre de tâches (ici des messages) que nous envoyons par le biais du package Traffic de notre design.
Les résultats de la communication peuvent être suivies dans la console avec un suivi complet de chaque étape que traverse le flux ainsi que le travail des threads.
Afin de stopper l'exécution, la touche "Echap" doit être appuyée.

Aspect critique et améliorations :

Nous avons utilisé un certain nombre de mots clés se référant au fonctionnement asynchrone du traitement de nos tâches (à savoir *async* et *await*) dans la fonction Work() qui est exécuté par chaque thread de notre pool.
Ainsi, pendant l'exécution, nous pouvons constater que les threads ne s'occupent jamais 2 fois de la même tâche et semblent avoir un caractère de traitement en série. Cela peut s'expliquer par le laps de temps introduit après la création de la tâche afin de simuler son traitement.

La solution étant un prototype, des pistes d'amélioration seraient de :

- commenter le code selon les normes
- ajouter des tests unitaires
- ajouter différents types de tâches : dans notre implémentation, nous avons utilisé la tâche SendMsgTask qui envoie des messages. Les types de tâches vont différer les unes des autres en modifiant la lambda associée aux classes qui implémentent l'interface TaskCommand. Ainsi la création de ces tâches nécessite uniquement la modification du comportement de la Lambda.