



5/17/2015

Dev

Architecture



PEROUMALLE Mourougan et TANG Thanh Lam

Sommaire

Client.....	2
Serveur	3
Conclusion	4

Client

ClientJarRet :

Principale classe du Client, il y a tous les champs de base d'une client (SocketChannel, port, nom du serveur) ainsi que l'id du client et un Map<ClientInfo, Worker> pour mémoriser les worker et ne pas avoir à les recréer.

Méthode query : plusieurs étapes :

- création d'un **requestTask** qui va demander une task au serveur.

- récupération d'un **worker** si possible (si le worker est déjà dans la map de Worker on le reprend sinon on recrée ce worker)

- le worker est stocké/créer dans la classe **RequestAndAnswer**, qui nous permet de compute, envoyé et lire la réponse final du serveur (Bad Request 400 ou HTTP OK)

Si lors d'un task on lit un comeBackInSeconds on sleep et on boucle tant que ce ComebackInSeconds revient.

Les protocoles HTTP voulu sont respecter grâce à la méthode des fichiers **HTTPHEADER** et **HTTPREADER** (des méthodes ont été ajoutés/ajusté pour le serveur qui reçoit des GET TASK et POST ANSWER)

RequestTask:

Méthode getJson : envoie d'une requête GET TASK au serveur et on retourne la réponse dans une Map<String, Object>

ClientInfo:

Clé de la Map de worker, avec une fonction de hachage et equals pour l'insertion et récupération correct dans la Map. Cette classe contient les informations nécessaires à la reconnaissance d'un worker.

ResponseAndAnswer:

Se créer de deux manières :

- Aucun worker connu : on crée le worker dans la classe ResponseAndAnswer

- Worker connu : on l'ajoute en paramètre pour construire la classe

ResponseAndAnswer

On compute le worker, ajoute le résultat obtenu dans le buffer, en faisant les vérifications nécessaires (il y a-t-il une des erreurs prévue ? si oui on remplace le champ answer par l'erreur associé).

On envoie le buffer rempli au serveur et affiche la réponse du serveur : success si HTTP OK, bad request sinon.

Serveur

ListJob: Contient une liste de Job et des méthodes associés. Cette classe est créée à partir d'un fichier de configuration et une instance de cette classe se trouve dans Server.

L'algorithme de poll de task est expliqué plus bas. Mais il est composé des méthodes `getHighestCurrentPriorityJob`, `GetTask` essentiellement.

Méthode `getHighestCurrentPriorityJob` : retourne le job avec la **currentPriority** (cf. Job et algo) la plus élevée.

`GetTask()` on prend le `HighestJobPriority` retourné et on retourne sa **currentTask**

Méthode `createFiletoJobs(FILE)` : parse un fichier de configuration des jobs et créer une instance de `ListJob`

`IsDone ()` : retourne vrai si tous les tâches de tous les jobs ont été reçues.

`update (JobId, Task, Answer)` : méthode utilisée à la réception d'une réponse valide d'un client, on update le `bitSet` et le fichier `Answer` du Job associé.

Job:

Représentation d'un Job. Possède tous les paramètres d'un job (`JobID`, description, priorité, url...) on ajoute aussi un `currentPriorité`, un `currentTask` et un `BitSet` de la taille de `TaskNumber`.

Lors du poll d'un Job à envoyer, on va se servir de ces données pour poll le bon Job:

ALGO DE POLL DE TASK :

Soit une liste de job dans `ListJob` :

0) initialisation de tous les Job : `currentPriorité = MaxPriorité` ; `currentTask=1`.

Parcourir tous les jobs et retourner le job avec le plus HAUT **currentPriorité**

Pour ce job on retourne le task de `currentTask` et on après l'avoir retourné on affecte `currentTask` à `set.nextClearBit (currentTask+1)` (prochain indice d'une tâche dont on n'a pas reçu de réponse)

Remarque : à chaque réception d'un "answer" on met à jour le `BitSet` d'indice Task reçu. (On ne renvoie pas ainsi de tâche inutile).

Si **currentTask** dépasse le **taskNumber** on initialise **currentTask** à `set.nextClearBit(0)` (premier indice de Task non envoyé) et on descend **currentPriorité** de 1.

On réitère.

CAS SPÉCIAL :

-Tous les jobs renvoie `isDone = true` (Tous les tasks ont eu leur réponse) : c'est bon on renvoie des buffers `ComeBackinSeconds` jusqu'à nouvel ajout de worker.

-Tous les **currentPriorité** des jobs sont à 0 mais des jobs n'ont pas reçu toute la réponse de leur tâche : on reset tous les jobs `isDone=false` (**currentTask** = `set.nextClearBit(0)` && **currentPriorité** = `MaxPriorité`) et on réitère l'algo tant qu'on arrive pas au cas où tous les tasks sont reçus.

PRO: -Tous les tasks sont envoyés tant qu'on n'a pas eu de réponses

-Les priorités sont respectées : on commence par la plus grande priorité, on respecte le nombre d'envoi grâce au décrémentation de `currentPriorité` progressif (pour un job prio 1 et 2, j'aurai fait 3 envois total).

-On ne néglige pas tous les jobs avant la fin du plus prioritaire : soit le cas ou job A avec prioA = 2 et job B avec PrioB prioB = 3, l'ordre d'envoi est celui-ci :

Bprio=3-- | Aprio=2-- | Bprio = 2-- | Aprio = 1-- | Bprio=1--

Tous les jobs sont à prio 0. (Bien évidemment il est possible que toutes les réponses soit reçu dès les premiers envoie de B et A et on s'arrête)

CONS :

-l'algo est un peu itératif, même si il est possible d'alterner après des priorités qui diminue, on doit finir toute les task de B avant de passer à l'envoi de A.

LoggerServer:

La classe loggerServer contient deux loggers, un pour les informations du serveur et l'autre pour les erreurs. Il prend en paramètre un nom de dossier pour enregistrer les deux fichiers de logs.

La classe est composée :

-logError(String error,long jobId, int numberTask, String client) : enregistre dans le fichier logError.log un message d'erreur avec le job et le numéro de tâche.

-logWarning(String message, String sourceClass, String method, Throwable t) : enregistre dans le fichier de logError.log un message d'erreur avec le nom de classe, la méthode et l'exception.

-logInfos(String message): enregistre un message d'information dans le fichier logInfos.log

ConfigServer:

La classe ConfigServer prend un nom de fichier et initialise les variables port, le dossier des logs, le dossier des réponses, le temps de reconnexion du client et la taille max d'un fichier. On utilise des gets pour récupérer ses variables.

Conclusion

Ce projet nous à permit de mettre un exemple d'échange Serveur/client basé sur le protocole HTTP et en mode TCP non bloquant.

Nous avons pu appréhender les notions d'échange contrôle par un protocole strict et optimisation des requêtes.

Nous regrettons de ne pas avoir eu le temps de faire une méthode d'ajout de worker pendant que le serveur est toujours en cours.