Compte Rendu Projet Réseau :

UGEGREED

Par

Agonse Cesaire

et

Atrax Nicolas

Sommaire :

Manuel Utilisateur :

Présentation :

Le but de ce projet est de réaliser un système de traitement de calculs via un partage des tâches au sein d'un réseau dépendant du protocole TCP.
L'objectif est de permettre de vérifier des conjectures sur des nombres importants de cas de manière optimisée.
Notre application a donc pour rôle de se connecter à d'autres applications similaires puis, à l'aide de commandes entrées par l'utilisateur, distribuer au sein du réseau les calculs à réaliser et en récolter les résultats.

Utilisation :

Les fichiers et commandes nécessaires à l'utilisation de cette application sont détaillés plus précisément dans le fichier README.txt du projet afin de faciliter les différents tests et la prise en main par l'utilisateur.
Les différentes fonctionnalités présentes ou manquantes sont toutefois présentées ci-dessous.

Fonctionnalités présentes :

- **Lancement d'une application en mode ROOT :**
  Il est possible de lancer une application en indiquant son port sans préciser d'application à laquelle se connecter au démarrage.

- **Connexion d'une application à une autre :**
  Il est possible de lancer une application en indiquant son port ainsi que de préciser l'adresse et le port d'une autre application à laquelle se connecter. La table de routage sera alors mise à jour afin de garder en mémoire les liens entre les différentes applications.

- **Déconnexion d'une application au sein du réseau :**
  L'utilisateur peut déconnecter une application si celle-ci y est autorisée. Les applications qui étaient connectées à elle se connectent alors à l'application père de celle-ci et la table de routage est mise à jour pour enregistrer les changements apportés.

- **Lancement des calculs via téléchargement du Jar :**
  L'utilisateur peut lancer des calculs en entrant l'adresse du jar, le nom de la classe, la plage sur laquelle réaliser les calculs et le nom du fichier texte ou enregistrer le résultat. L'application peut alors télécharger le jar correspondant, effectuer les calculs et enfin enregistrer le résultat au sein du fichier texte indiqué.

- **Répartition des tâches au sein du réseau :**
  Si plusieurs applications sont présentes et fonctionnelles au sein du réseau, l'application en charge des calculs peut envoyer des requêtes aux autres applications (celle à qui elle est connectée si elle n'est pas en mode ROOT et celles qui sont connectées à elle) afin de répartir les tâches.
  Une fois leurs calculs effectués, celles-ci peuvent envoyer le résultat à l'application responsable de la demande initiale à l'aide de la table de routage puis , si tout les résultats ont bien été reçus/effectués, ceux-ci sont enregistrés au sein du fichier texte indiqué.

Fonctionnalités manquantes :

- **Redirection des calculs propre à une application lors d'une déconnexion :**
  Nous souhaitions que les calculs effectués par une application lors de sa déconnexion soient attribués à d'autres applications du réseau.
  Malheuresement, en raison du nombre de cas différents à traiter pour ce cas(avancement dans la plage de calcul, résultats déjà envoyés, …) , nous n'avons pas implémenté cette fonctionnalité.

Problèmes notables :

- **Bug lors de la succession de conjecture :**
  Nous avons noté un bug lors de l'utilisation successive de la commande START.
  En effet, certaines fois après que les calculs aient été effectués par une autre application, une partie d'entre eux n'est pas reçue par l'application à l'origine de la demande.
  Après de nombreux tests nous en avons déduits que l'application à l'origine de la demande de calcul ne lance plus la méthode treatKey malgré l'utilisation de la méthode wakeup, empêchant ainsi le lancement de la méthode doRead et donc la récupération de la requête de résultat.
  De plus, le bug décale le traitement des requêtes (dans la query queue) et peut donc rendre l'application dans un état incohérent.
  L'apparition de ce bug n'étant pas régulière, n'avons malheureusement pas pu trouver de moyen de le régler.

Choix d'architecture :

ClientServer :

Objet principal de notre application, il contient les différents objets nécessaires au fonctionnement de celle-ci telles que les Thread,Selector et Channel client et serveur, la table de routage, le Context ou encore le CalcExecutor dont les rôles sont définis ci-dessous.
Il possède donc différentes méthodes afin d'accéder à ceux-ci , traiter les commandes de l'utilisateur et gérer les requêtes reçues.

Context :

Objet permettant de gérer la lecture et l'écriture au sein des différents channels lors de l'appel de la méthode treatKey.
Il contient les buffers de lecture et d'écriture, une queue contenant les requêtes à envoyer et un Reader permettant de convertir le contenu d'un buffer en requête.
Il possède donc les méthodes permettant de convertir les différentes requêtes pour les insérer dans le buffer d'écriture ou encore celles permettant d'ajouter des requêtes au sein de la queue.

Query :

Objet représentant une requête, il s'agit d'une superclasse pour les classes des différents types de requête (allant de 1 à 8) .
Chaque requête dispose d'un id ainsi que de différents objets variant selon le type de requête (par exemple un id de calcul, un url ou encore une adresse).

QueryReader :

Objet implémentant l'interface Reader et permettant la conversion du contenu d'un buffer en objet Query.
Il contient 8 Reader différents qu'il utilise en fonction de l'id de requête lu dans le buffer afin d'appliquer la conversion correspondante.

CalcExecutor :

Objet permettant de traiter les différents calculs d'une plage en les répartissant au sein du réseau, et d'envoyer les résultats à l'application à l'origine de la demande de calcul.
Il contient un executorService afin d'effectuer des calculs différents dans différents threads, un map pour contenir les résultats des calculs propres à l'application, une map pour enregistrer les sources des différentes demandes de calculs, ainsi que le clientServer afin de récupérer les informations nécessaires à la répartition du calcul et à l'enregistrement des résultats.

Rooter :

Objet permettant l'enregistrement des différents nœuds dans le réseau et leur connexion.
Cet objet contient une map Context,Context qui permet d'identifier par quel application passer pour accéder à l'application dont l'adresse est contenue dans le Context en clé.
Chaque application possède un objet Rooter qui sera mis à jour en fonction des modifications au sein du réseau. L'objet sera couramment appelé table de routage ou routeur.

Gestion des évolutions attendues :

Construction de la table de routage à base de Context :

La table de routage utilisait uniquement ,lors de la béta, des InetSocketAddress en tant que clé et valeur. Il nous a été demandé de préférer des Context.
En effet, lorsque l'on utilise la table de routage pour envoyer une requête à une certaine adresse et donc une clé bien précise, il fallait parcourir toutes les clés du père et du fils.
Nous utilisons donc des Contexts pour corriger cela et ainsi accéder directement à la clé qui y est attachée. De plus, nous pouvons avoir accès directement à d'autres informations, notamment pour la méthode toString de la table de routage qui amène un certain confort utilisateur.

```
public class Rooter {
  private final HashMap<Context, Context> table;

  public Rooter() {
    this.table = new HashMap<>();
  }
```

Appliquer un switch sur les query plutôt que sur un int :

Afin d'améliorer notre méthode processQuery au sein de clientServer, il nous a été demandé d'effectuer un switch sur le type de Query plutôt que sur un int id présent au sein de Query.
Pour se faire nous avons donc dû transformer Query en sealed class et appliquer un permits sur les différents types de requête :

```
public sealed class Query permits Query0, Query1, Query2, Query3, Query4, Query5, Query6, Query7, Query8{
```

```
public void processQuery(Query query, SelectionKey key) throws InterruptedException {
  switch (query) {
  //départ d'une plage de calcul
  case Query0 queryf ->{
    //System.out.println("traitement de la requete 0");
```

Gestion des calculs via un objet dédié et un executorService :

Lors de la présentation béta, notre application gérait le traitement des calculs au sein du ClientServer. Il nous a donc été demandé, afin de simplifier la lecture du code et rester dans

la philosophie de la programmation objet, de gérer ces tâches au sein d'un objet dédié.
De plus, la boucle sur la plage se lançait au sein d'un nouveau Thread, il nous a donc été demandé de remplacer cela par l'utilisation d'un executorService possédant ses propres Thread.
Pour répondre à ses 2 demandes, nous avons créé un objet CalcExecutor qui contient un executorService et les différentes méthodes de traitement de calcul (répartition des tâches, téléchargement du jar, exécution des calculs sur une plage, envoie des résultats).
La méthode processCalc qui lançait initialement un nouveau Thread effectue maintenant un submit à l'executorService afin d'effectuer l'exécution des calculs sur une plage :

```
public void processCalc(Query query, int start, int end) throws InterruptedException {
  if (clientServer.getSemaphore().availablePermits() == 0 ) {
    //renvoyer false pour signaler que ca va pas
    return;
  }
  clientServer.getSemaphore().acquire();
  executorService.submit(() -> execCalc(query, start,end));

}
```

```
private void execCalc(Query query, int start, int end) {
    var queryf = (Query0) query;
    if(end > queryf.getEndRange()) {
        return;
    }
    System.out.println("Start calc : " + start + " to " + end);
    var calcId = queryf.getCalcId();

    for (int i = start; i < end; i++) {
      //calcul du check
      var checkRes = getCheck(queryf.getUrl(), queryf.getClassName(), i);
      //ajout au noeud local ou envoie du resultat
      if (calcId.origin().equals(clientServer.getID())) {
        addRes(calcId, checkRes,i);
      } else {
        sendRes(calcId, checkRes, i);
      }
    }
    writeRes(calcId);
    clientServer.getSemaphore().release();
}
```

Gestion de la taille du buffer dans le Context :

Dans le Context d'une clé lors de la gestion d'une requête. Il se peut que dans le buffer un dépassement de buffer ou une exception "bufferOverFlow" arrive car les requêtes ne sont pas bornées.
Au départ, la vérification de la taille restante du buffer se faisait dans la méthode processOut juste avant de retirer une requête (query) dans la queue. La méthode n'était pas bonne car on ne verrifiait pas la taille de la requête. Pour corriger cela, nous avons décidé de donner une valeur de retour boolean pour la méthode processQuery qui arrive juste après .Chaque processQueryN peuvent ainsi connaître la taille de la requête avant de la mettre dans le buffer. Celle-ci réponds true si il est possible de mettre tout le contenu. Si ce n'est pas le cas alors elle renvoie false et la méthode processOut multipli la taille du buffer par 2 grâce à la méthode resizeBufferOut et ceci jusqu'à ce que processQuery renvoie true grâce une boucle while.

```java
public void processOut() {
  var query = queryQueue.poll();
  if(query == null) {
    logger.info("no query in the queue");
    return;
  }
  while (query != null) {
    synchronized(bufferOutLock) {
      while (!processQuery(query)) {
        //System.out.println("agrandissement");
        resizeBufferOut();
      };
    }
    query = queryQueue.poll();
  }

}
```

Par exemple, voici la vérification de processQuery4.
rappel: processQuery appel processQueryN sur le bon type via un switch sur query.

```java
public boolean processQuery4 (Query query) {
    var queryf = (Query4) query;
    var address = queryf.getNewDest();
    if(bufferOut.remaining() < Integer.BYTES + getNbByteAddress(address)) {
      logger.warning("no enough space in buffer");
      return false;
    }
    bufferOut.putInt(query.id());
    processOutAddress(address);
    return true;
}
```