Projet JarRet

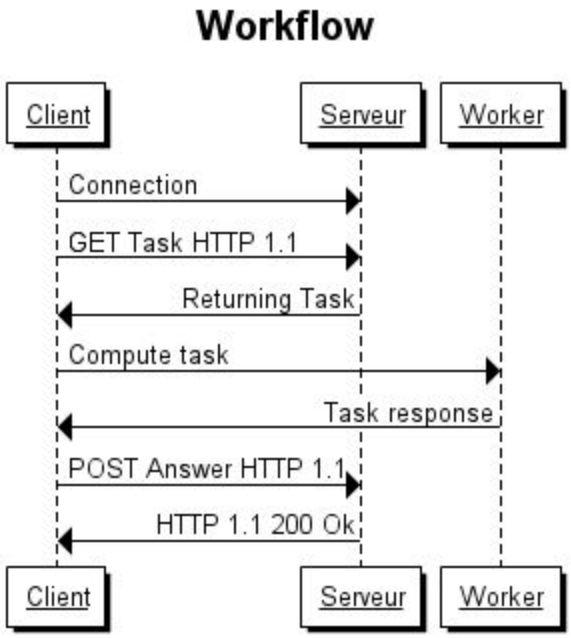
Université Paris-Est Marne la vallée

Manuel du développeur

Liazidi Amine
Huy Raymond

Le projet :

Le but est de distribuer des calculs sur plusieurs clients et collecter toutes les réponses au niveau du serveur.

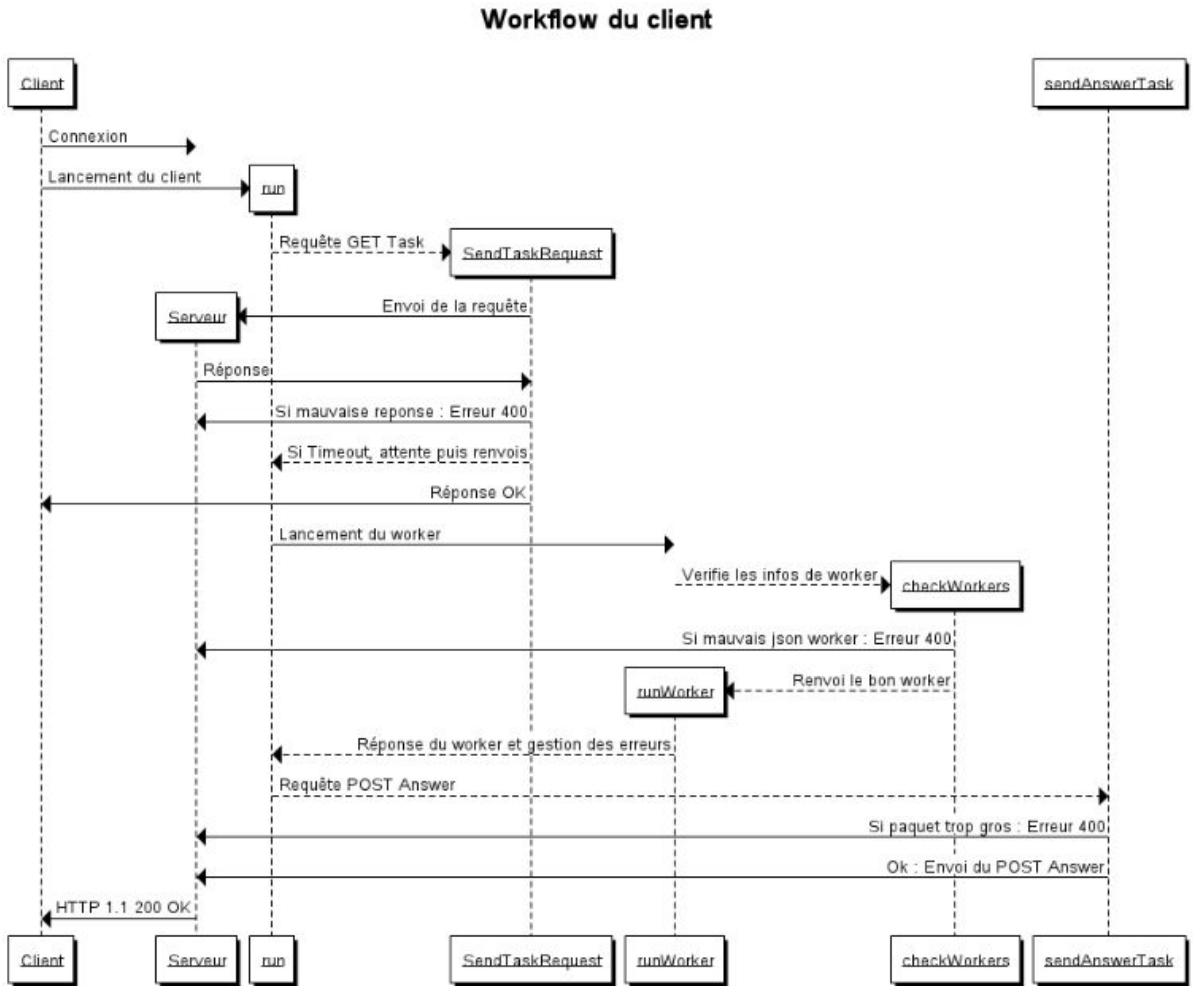


Le client :

On s'est basé sur l'architecture vus en TP.

Organisation du travail :

- Mise en place du code basique qui permet une connexion à un serveur
- Effectuer une requête GET dès la connexion du client
- Vérifier la réponse du serveur et sa validité
- Parser la réponse et la stocker
- Si le serveur répond qu'il n'a pas de job, retenter une nouvelle connexion.
- Récupérer le JAR
- Créer une instance du Worker
- Lancer le calcul
- Vérifier la réponse du worker
- Vérifier la taille du paquet à envoyer au Serveur
- Effectuer une requête POST en utilisant les résultats précédents



Ses méthodes internes :

- sendTaskRequest() : Optional<String>
    - Écrit sur la socket et envois un GET Task au Serveur, et retourne la réponse de celui-ci.
- checkWorkers(Map<String,String>) : Optional<Worker>
    - Prend en paramètre un job.
    - Vérifie la validité des informations du worker, vérifie si le Worker existe déjà et le retourne sinon on le crée.
- runWorker(Map<String,String>): Map<String,String>
    - Prend en paramètre un job.
    - Récupére le bon worker, lance un compute, et ensuite vérifie le résultat en effectuant la gestion d'erreur sur le JSON.
- checkWorkerResponse(String): boolean
    - Vérifie la validité du JSON renvoyé par la méthode compute du worker.
- sendAnswerTask(String, String, String) : void
    - Écrit sur la socket et envois un POST Answer au Serveur, contenant la réponse du worker.
    - Vérifie aussi la taille du paquet envoyé au Serveur. Une erreur 400 est envoyé au Serveur dans le cas d'erreure.
- run() : HTTPClient
    - Vérifie si le client est bien connecté.
    - Vérifie si la réponse au GET renvoyé par le serveur ne contient pas de Timeout, dans le cas contraire, le client se met en pause.
    - Lance le worker, récupère sa réponse et lance enfin la requête POST.

2. Ses méthodes externes :

- getTask(String, Charset) : ByteBuffer
    - Renvois une requête HTTP GET Task
- validGetResponse(String) : Optional<String>
    - Vérifie la réponse au GET envoyé au serveur. Et retourne un résultat en fonction de cette réponse.
- getPostHeader(String, Charset, String, int) : ByteBuffer
    - Renvois une requête HTTP POST Answer
- getTaskInfo(String) : ByteBuffer
    - Retourne les informations du job traité.
- getPostContent() : ByteBuffer
    - Retourne le contenu de la requête POST

3. Les problèmes soulevés lors de la soutenance beta qui ont été réglés :

- Gestion des exceptions : Envois d'un paquet HTTP 400 au serveur si l'encodage d'une réponse reçue est mauvais, ou bien si un paquet est trop gros à envoyer au serveur.
    - Avant la soutenance, on lançait une exception.
- Vérification des informations du worker récupéré.
    - Avant la soutenance, on ne vérifiais pas le JSON recuperé.
- Vérification du bon format des réponses du worker.
    - Avant la soutenance, on ne vérifiais pas le JSON recuperé.
- Reconnexion du client au Serveur :
    - Avant la soutenance, notre client ne se reconnectais pas automatiquement.
    - Voici la méthode run de notre client qui tourne dans une boucle while, et qui permet au client de se reconnecter au serveur :

```
public HTTPClient run() throws IOException {
        try {
                long start = System.currentTimeMillis();
                if(!sc.isOpen()){
                        this.sc = SocketChannel.open();
                }
                if(!sc.isConnected()){
                        this.sc.connect(server);
                }
                Optional<String> job = sendTaskRequest();
                if(job.isPresent()){
                        if(job.get().equals("ComeBackInSeconds")){
                                while(System.currentTimeMillis() - start <= TIMEOUT);
                                return this.run();
                        }
                ….
                }
        } catch (ClassNotFoundException | IllegalAccessException | InstantiationException |
IOException e) {
                        e.printStackTrace();
                } finally{
                        this.sc.close();
                }
        return null;
        }
```

Le serveur :

Structure interne :

```
private static class Context {

        private boolean inputClosed = false;
        private final ByteBuffer in = ByteBuffer.allocate(BUF_SIZE);
        private final ByteBuffer out = ByteBuffer.allocate(BUF_SIZE);
        private final SelectionKey key;
        private final SocketChannel sc;
        private long time;
        private HTTPHeaderServer head = null;

        public Context(SelectionKey key) {
                this.key = key;
                this.sc = (SocketChannel) key.channel();
                this.time = 0;

        }
```

- public void doRead() ;
  - lecture lorsque le client envoie un paquet.
- public void doWrite();
  - réponse du serveur au client lorsque celui ci à eu une requête de la part du client.
- private void process() ;
  - cette méthodes appelle celle du dessous si le paquet reçu est valide.
- private boolean processRequest(String request);
  - en fonction du contenu du paquet si c'est un post ou un get. on applique le protocole.
- private void resetInactiveTime() ;
  - méthodes permettant de réinitialiser le timer d'inactivité, en effet celui ci est appelé lorsqu'il vient d'émettre un message.
- private void addInactiveTime(long time, long timeout);
  - cette méthode permet de faire vieillir le timer du client.

```
}
```

La classe context représente le client au sein du serveur. Ils ont chacun un buffer d'entrée et un buffer de sortie. On délègue la responsabilité de lecture et écriture à cette classe interne.

Quant au serveur, il a la responsabilité de pouvoir gérer l'acceptation de client, sans lancer un autre thread gérant le serveur.

Structures annexes:

public class Job {

/*les champs*/

```
private final long jobId;
private final int jobTaskNumber;
private final String jobDescription;
private final int jobPriority;
private final String workerVersionNumber;
private final String workerURL;
private final String  workerClassName;
private final BitSet bitSet;
```

- public Job(long jobId, int jobTaskNumber, String jobDescription, int jobPriority, String workerVersionNumber,String workerURL, String workerClassName);
    - Constructeur pour créer un objet Job, dont les caractéristiques sont données en paramètres.
- public int getIndexOfFalseBitSet();
    - renvoie le premier bitset à faux.permettant de gerer les tasks des jobs
- public long getJobId();
    - getter du champs jobId;
- public int getJobPriority();
    - getter du champs JobPriority;
- public String getWorkerVersionNumber();
    - getter du champs workerVersionNumber;
- public String getWorkerURL();
    - getter du champs workerURL;
- public String getWorkerClassName() ;
    - getter du champs workerClassName;
- Optional<String> getTask() ;
    - renvoie le task d'un job au client
- public boolean jobIsFinished() ;
    -
- public boolean finishTask(int task);
    -

}

public class ResponseBuilder {

```
public static final String BAD_REQUEST = "HTTP/1.1 400 Bad Request\r\n"
        + "\r\n";
public static final String OK_REQUEST = "HTTP/1.1 200 OK\r\n\r\n";
private static ResponseBuilder builder = null;
private final Object lock = new Object();
```

- private ResponseBuilder(String url)
  - constructeur privée car l'on veut empêcher, d'instancier cette classe autre que par getInstance
- public static ResponseBuilder getInstance(String url)
  - renvoie une unique instance de réponseBuilder, si celui est déjà instancier on renvoie la même instance.
- public String initComeBack() throws JsonProcessingException;
  - retourne un objet Json contenant la valeur du comeBackInSeconds
- public ByteBuffer get(Optional<String> json);
  - retourne un ByteBuffer contenant le header d'une réponse HTTP 200 Ok.
- public ByteBuffer getContent(Optional<String> json);
  - retourne un ByteBuffer contenant le contenu de la réponse HTTP en fonction du comeback s'il est actif ou non.
- public String post(String content);
  - Vérifie le contenu de la réponse reçu par le client, avant de renvoyer un ByteBuffer contenant une réponse HTTP 200 Ok, ou une bad request.

}

public class ReadLineCRLFServer {

- public static Optional<String> readHeader(ByteBuffer buff);
  - méthodes permettant la lecture du header, permet la lecture en respectant le protocole TCP.
- public static Optional<ByteBuffer> readBytes(int size, ByteBuffer buff);
  - méthodes pour lire un plage de donnée, ici on récupère le json.

}

public class TaskReader {

/*les champs*/

```
public static TaskReader instance = null;
private final String url;
private final Map<Job,Integer> map=new HashMap<>();
```

- private TaskReader(String url) throws IOException
  - constructeur privée pour empêcher d'instancier cette classe excepté en passant par getInstance;
- public static TaskReader getInstance(String url) ;
  - singleton, renvoie une unique instance
- private void init();
  - méthodes d'initialisation des tâches, en les ajoutant toute à une map depuis le fichier json.
- public public void taskFinish(long jobId, int task, String msg);
  - parcours les tasks des jobs pour mettre à jour si elles sont finis le bitSet associé au job.

}

public class JSON {

- public static Server FactoryServer();
    - méthodes factory qui créeer le serveur avec le fichier de config
- private static Server parsingJsonConfig(JsonParser jParser) ;
  private static JsonParser createJsonParser(Path path) ;

}

Ses méthodes internes :

3. Les problèmes soulevés lors de la soutenance beta qui ont été réglés :

- Gestion des jobs: on avait une Blockingueue pour gérer les jobs, on ne prenait pas en compte les priorités du Jobs, du coup lorsqu'un client était disponible on poll de la queue pour donner le job à faire à un client. Ceci à été remplacé par un HashMap avec comme clé un job et value la priorité du job. Pour traiter les jobs on parcours la hashmap, pour chaque job, on l'envoie au client.
- Structure du job, on a créer un objet job, afin de stocker toutes les informations le concernant. De plus on a ajouté le Bitset à chaque job de taille job priorité -1, vu qu' un job peut avoir plusieur tasks (ce qui n'était pas pris en compte auparavant).
- Réglage du assertion error: on fait un test sur le champs des priorités des job afin de ne pas prendre en compte les priorité 0.

- Protocole TCP: Dans l'ancienne version dès que l'on recevait un paquet on le lisait directement, sans prendre en compte que l'on avait tout bien reçu ("\r\n\r\n"), désormais on prend en compte le fait que le header peut arriver en plusieur paquet.