TEA : Architecture distribuée (RMI)

Janos Falke / Université de La Rochelle (Semestre 6, 2018/19)

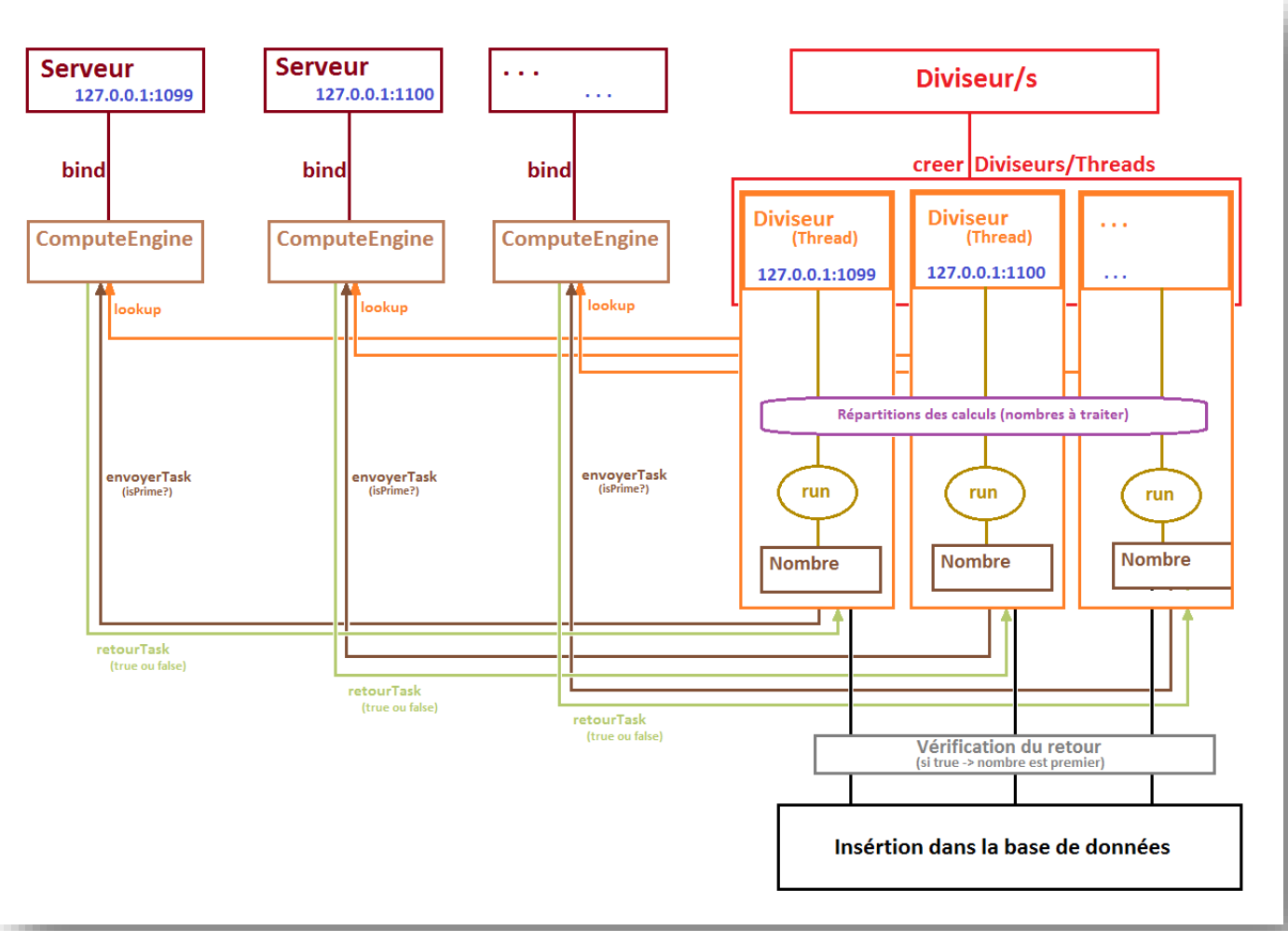


Figure 1: Représentation de l'objectif (Image plus lisible dans le dossier source)

Sommaire

Présentation

Le but de ce projet était de mettre en place une infrastructure distribuée pour le calcul de nombres premiers. Cette infrastructure a été réalisé avec RMI (*Remote Methode Invocation*) sur Java.

Création de classes

Je suis parti sur la création de plusieurs classes/interfaces différentes :

- Serveur
- Diviseur
- *Compute (Interface)*
- ComputeEngine
- *Task (Interface)*
- Nombre

Serveur

Cette classe va créer le serveur qui va permettre de créer un ComputeEngine sur des ports (différents pour chaque Serveur/ComputeEngine, commençant par 1099 (Port de RMI par défaut)). Cette classe crée un serveur et donc un ComputeEngine pour faire les opérations (ici du calcul des nombres premiers demandé par Diviseur), évidemment on peut choisir l'adresse et aussi le numéro de port mais également si on veut afficher les détails lors du traitements des calculs/nombres.

Diviseur

Cette classe va créer les différentes instances de Diviseur. Le main de cette classe va premièrement regarder les options passé en paramètres sur ligne de commande et traiter les entrées éventuelles. De suite il va permettre de créer une connexion, ici, à postgresql. Après on va créer une base de données avec le nom passé en paramètre (ou par defaut 'TEAArchi'). Ensuite on va créer la table 'diviseurs' dans laquelle on va trouver les nombre_premier(BigInt). Après cela, on va créer une liste de diviseurs qui vont être défini de suite comme des Threads car on hérite de Runnable, évidemment ici on peut aussi choisir le nombre de diviseurs souhaités. Maintenant on pourra lancer les instances de diviseurs crées.

Ces diviseurs vont se connecter aux ports et adresses correspondants (ceux des Serveurs/ComputeEngine, soit des adresses et port par défaut ou donné par l'utilisateur en paramètre) et après boucler sur les nombres à traiter. Ces nombres vont être envoyé un par un à la ComputeEngine qui va renvoyer un booléen (nombre est premier ou pas). De suite on va insérer seulement les nombres premiers (donc qui sont true) dans la base de données et dans note table 'diviseurs'.

Chaque Diviseur va traiter les nombres comme suit : Il commencera à son numéro d'identification et à chaque boucle incrémenta du nombre total des diviseurs.

Exemple :
Avec 2 diviseurs (de 0 à 10) cela nous donne ici:
Diviseur/Thread 0 va traiter les nombres 0,2,4,6,8,10 et Diviseur/Thread 1 va traiter les nombres 1,3,5,7,9

Cette réparation est la même qu'on avait vu en TP pour le calcul distribué de Pi.

(Exemple plus bas) : L'utilisateur peut ici donner plusieurs informations sur ligne de commande. Le login de la base de données (postgres par défaut), le mot de passe (pgAdmin par défaut), le nom de la base de données (TEAArchi par défaut), le nombre de diviseurs (1 par défaut) et le début (1 par défaut) et la fin (100 par défaut) des nombres entiers à vérifier/traiter et finalement 'verbose', donc si on souhait des détails par rapport aux traitements effectués (par défaut true). La base de données passé en paramètre va être recrée à chaque exécution comme également la table 'diviseurs'. Finalement on peut donner les ou la adresse/s du diviseurs avec le port correspondant (si cela est fait on n'a plus besoin de fournir le nombre de diviseurs).

Compute (Interface)

Cette interface qui va être utilisé par ComputeEngine permet l'implémentation de la méthode de *executeTask* qui retourne un élément de type Object et hériter de la classe Remote.

ComputeEngine

Cette classe qui est appelé par la classe Serveur et qui va exécuter la tâche à traiter (Task, ici calcul des nombres premiers) et le serveur.

Task (Interface)

Cette interface hérite pour rendre sérialisable les opérations / taches.

Nombre

Cette classe nous donne la création d'un nombre (à partir d'un BigInteger). Cette création va être appelée dans le diviseur qui va créer des instances de nombres avant de pouvoir envoyer les calculs à faire au ComputeEngine (Vérification si un nombre premier) car pour cela il lui faut un objet qui implémente de la classe Task et ceci est le cas pour les objets de Nombre.

Donc ComputeEngine pourra exécuter la tâche à faire de Nombre qui est décrit dans la méthode implémenté par l'interface Task dans laquelle on va vérifier si le nombre d'entier donné est un nombre premier ou pas. Ensuite on pourra retourner un booléen pour signaler cela. Avec cette information qui va être récupéré par le diviseur on pourra stocker les données obtenues dans la table 'diviseurs' de la base de données.

La méthode de vérification pour les nombres premiers se base sur les explications de l'exercice 3 du TP2 avec des améliorations d'accélérer le traitement (par exemple de retourner false toute de suite s'il s'agit d'un multiple de 2 (modulo 2)).

Aide pour l'exécution sur la ligne de commande

Pour exécuter il faut se mettre dans le répertoire racine 'TEA_Archicture' !

S'il faut compiler *(normalement pas besoin)* :

- javac Serveur/Serveur.java Diviseur/Diviseur.java Operations/Nombre.java Compute/Compute.java Compute/Task.java Compute/ComputeEngine.java

Pour le serveur / ComputeEngine :

On va passer la java.policy pour obtenir les droits aux sockets (ports) → à cause du RMISecurityManager.

- **java -Djava.security.policy=java.policy Serveur/Serveur -p [numéro port] -a [adresse] -v [verbose]**

Exemple:

- java -Djava.security.policy=java.policy Serveur/Serveur
- java -Djava.security.policy=java.policy Serveur/Serveur -a 127.0.0.1 -p 1100 -v false (sans -p → par défaut 1099 & sans -a → par défaut l'adresse locale & -v → true par défaut)

Pour le diviseur :

On va passer la java.policy pour obtenir les droits aux sockets (ports) → à cause du RMISecurityManager.

On va passer un driver postgres du dossier 'lib' pour pouvoir se connecter à une base de données postgres.

- **java -Djava.security.policy=java.policy -cp lib/postgresql-42.2.5.jar; Diviseur/Diviseur -l [login] -m [motDePasse] -n [nomBD] -t [NbDiviseurs] -b [Debut] -e [Fin] -v [verbose] -a [{adresses :ports}]**

Exemple :

- java -Djava.security.policy=java.policy -cp lib/postgresql-42.2.5.jar; Diviseur/Diviseur -l postgres -m pgAdmin -n TEAArchi -b 214784312147 -e 214784312999 -v false -a {127.0.0.1 :1099;127.0.0.1 :1100}
  (Ici on va se connecter à postgresql avec l'utilisateur postgres et le mot de passe pgAdmin, puis on va créer et se connecter à la base de données TEAArchi. On va créer 2 diviseurs sur les adresses 127.0.0.1 :1099 et 127.0.0.1 :1100 et ils vont traiter les entiers entre 214784312147 et 214784312999 (répartis et BigInteger) et on ne va pas afficher les détails (verbose = false))

Prises des choix

Multithread

J'ai décidé de mettre la classe Diviseur qui hérite de la classe Thread pour pouvoir créer du multithreading. Avec ceci il est maintenant possible de répartir le travail entre ces threads. C'est pour ça que dans la classe diviseurs va créer les diviseurs avec des méthodes statique, ces méthodes statiques vont être appelées dans le main et avant le traitement des nombres entiers.

Dans la classe des diviseurs j'ai choisi de répartir les calculs des nombres premiers comme vu dans le TP1 avec le calcul de Pi, donc chaque thread va commencer à début+son_id et s'incrémente avec le nombre de thread totale.

Pour créer plusieurs serveurs, il faut lancer plusieurs terminaux et chaque terminal va être associé à un serveur (exemple plus bas).

Classe Serveur

J'ai choisi de créer cette classe pour gérer les connexions entre les ComputeEngine et leurs liens.

Classe Diviseur

Ici j'ai pris plusieurs choix ici. Premièrement les arguments passés en paramètres par utilisateurs ne sont pas obligatoires et j'ai créer des valeurs par défaut pour avoir un lancement standard. Puis j'ai décidé de lancer une exception lorsque les données saisies ne correspondantes pas (par exemples le début des nombres supérieur à la fin des nombres). De suite j'ai décidé de également afficher le temps d'exécution du traitement qui peut être très intéressant si on teste avec des nombres de diviseurs / serveurs différents.

Classe Nombre

Ici on va créer une classe qui va simplement créer un BigInteger et dans la méthode execute (implementé par Task) vérifier si c'est un nombre premier ou pas. J'ai choisi de créer une fonction sqrt que j'ai trouvé sur l'internet (https://stackoverflow.com/questions/4407839/how-can-i-find-the-square-root-of-a-java-biginteger) pour pouvoir appliquer la méthode de vérification de nombre premier du TP2 car dans Java 8 la fonction sqrt de BigInteger n'existe pas, elle est venue dans Java 9. Ainsi de suite j'ai choisi de traiter d'abord les nombres par rapport à la multiplicité de 2 ou 3 pour accélérer le traitement. On a également grâce au booléen verbose la possibilité d'afficher des détails du traitements des calculs/nombres.

Java.policy

Car dans les deux classes et pour chaque diviseur/serveur on crée des RMISecurityManager on est obligé de créer un java.policy qui nous donne les permissions aux sockets. J'ai décidé de ne pas lui fournir un lien (*FilePermission*) mais de lui donner la permission de n'importe quel chemin avec *permission java.security.AllPermission;*. Cela évite de devoir changer de lien/chemin dès qu'on change de machine.

Base de données

La base de données n'a pas besoin d'être existante ici car on va créer une nouvelle au lancement de la commande avec le nom passé en paramètre ou le nom par défaut. Ceci évite de devoir créer une base de données avant, il suffit d'avoir postgresql d'installé.

J'ai choisi de mettre seulement les nombres premiers dans la base de données et car dans diviseur on insère dès qu'on trouve un nombre premier, il est très possible que les nombres ne vont pas être inséré dans l'ordre croissant. Si on souhait les voir dans l'ordre croissant on a deux possibilités, soit on fait un 'order by' dans notre base de données ou on va changer le code de diviseur qu'il va seulement stocker les nombres trouvés dans un tableau/ArrayList et puis à la fin de tout les traitements (après les join des threads) on va pouvoir tous insérer dans l'ordre. Moi j'ai décidé de mettre les nombres premiers toute de suite et de les trier après si on souhait par un 'order by' dans la base de données, comme ca on peut même voir comment les diviseurs différents ont travaillé.

Options en ligne de commande

J'ai choisi de mettre des options sur ligne de commande (par exemple : -t pour le nombre de diviseurs si on ne précise pas d'adresses sinon avec -a pour diviseur, -a et -p pour serveur) car tous les arguments ont des valeurs par défaut donc comme ça, l'utilisateur a plus besoin de s'occuper de l'ordre ou de mettre tous s'il veut simplement changer le nombre de threads, ceci nous donne alors une facilité d'utilisation.

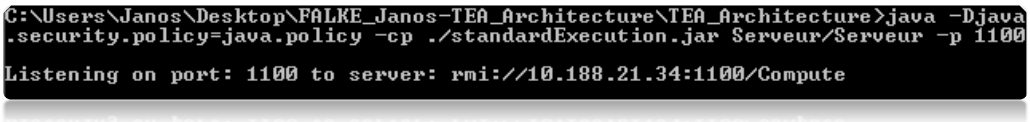
*(Toutes les options de ligne de commande expliquées dans le **readme** !)*

Exécutable (*standardExecution*.jar)

S'il faut compiler *(normalement pas besoin)* :

- jar -cvf standardExecution.jar Serveur/Serveur.class Diviseur/Diviseur.class Operations/Nombre.class Compute/Compute.class Compute/Task.class Compute/ComputeEngine.class

Serveur



Ici on pourra comme pour l'exécution standard préciser les paramètres/options (explications dans le **readme !**).

Commande : java -Djava.security.policy=java.policy -cp ./standardExecution.jar Serveur/Serveur OPTIONS

Diviseur

Il faut également préciser le driver pour postgresql donc on va devoir lancer deux jars simultanément.

Commande : java -Djava.security.policy=java.policy -cp lib/postgresql-42.2.5.jar;standardExecution.jar Diviseur/Diviseur OPTIONS

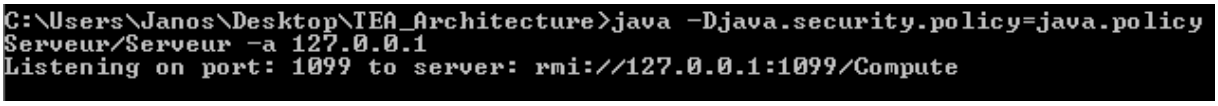
Exemple de gain de temps d'exécution :

Vérification des entiers de 0 à 50000 :

***Rouge** = serveur/s | **bleu** = diviseur → Images plus lisibles dans le dossier 'Captures_de_ecran' !*

Avec 1 serveur(ComputeEngine)/thread :

Avant l'exécution du diviseur (serveur en attente):

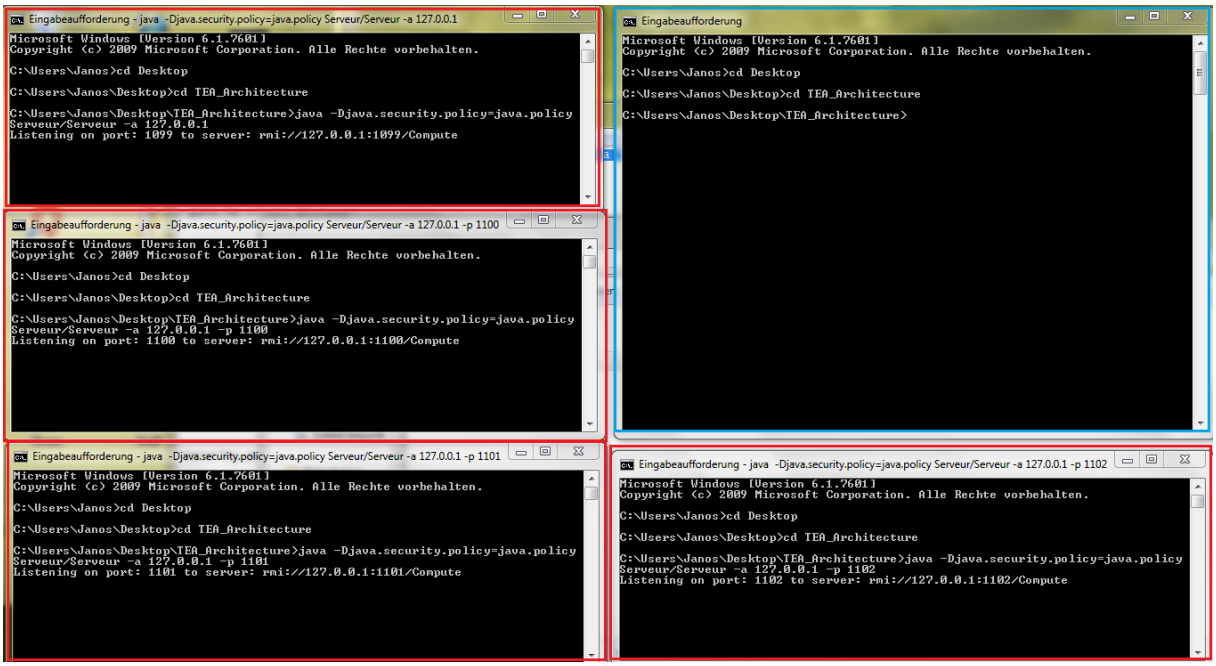


Après l'exécution du diviseur :

Figure 2: Temps d'exécution -> 22.35 secondes
(avec l'affichage du traitement du côté de serveur sinon on passe à ~ 17 secondes ! → -v false)

Avec 4 serveurs(ComputeEngines)/threads :

Avant l'exécution du diviseur (serveurs en attente):



Après l'exécution du diviseur :

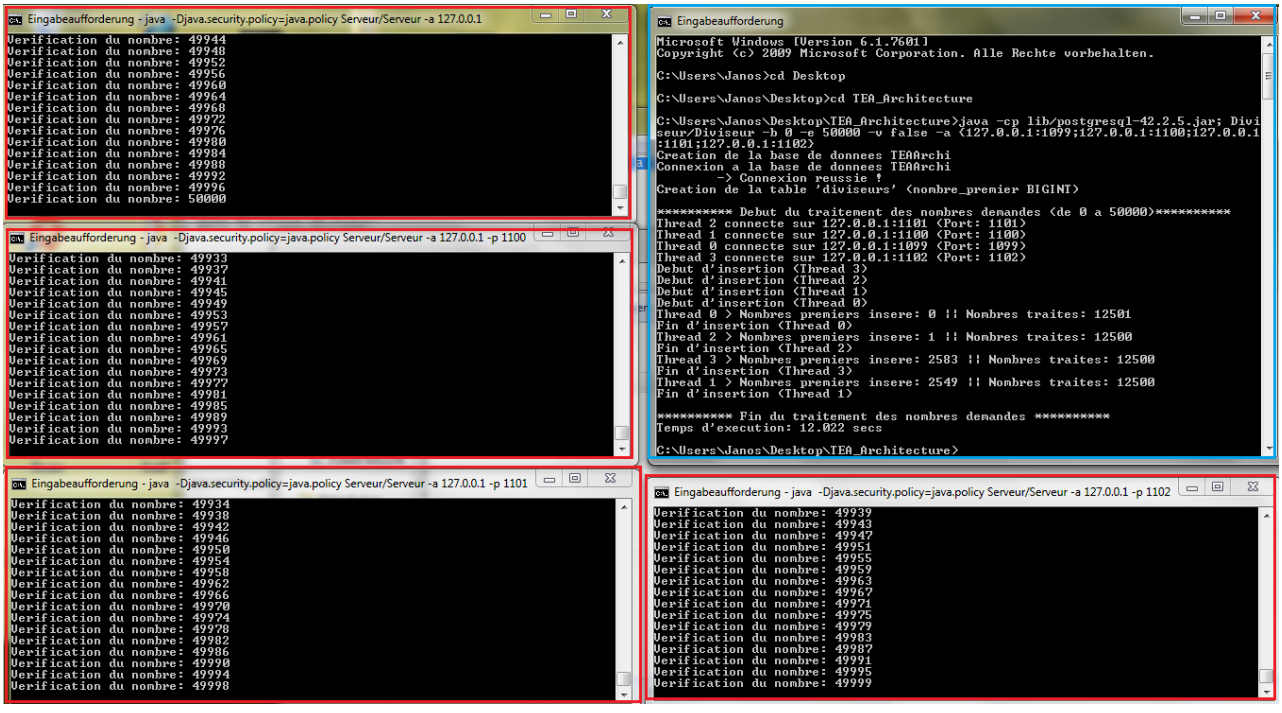


Figure 3: Temps d'exécution -> 12.02 secondes
(avec l'affichage du traitement du côté de serveur sinon on passe à ~ 3.5 secondes ! → -v false)

Conclusion : Dans cet exemple on a gagné 10.33 secondes avec la repartions des calculs sur 4 serveurs différents ! **(Sans affichage du traitement / détails des nombres côté serveur on gagne même 13.5 secondes !)**