
**INF4410 – Systèmes répartis et infonuagique**

**Automne 2017**

# TP2 : Services distribués et gestion des pannes

**Groupe 1**
**Dominik Courcelles**
**Dan Vatnik**

**Soumis à : Houssem Daoud**

**[2017 / 11 / 10]**

**Spécifications sur l'implémentation**

Le répartiteur fonctionne de la façon suivante pour répartir les calculs. Au démarrage, il crée un RMIRegistry. Il lit ensuite le fichier de calculs à faire en entier et crée une pile d'opérations à exécuter. Par la suite, il choisit calculateur de façon aléatoire parmi ceux qui sont dans le RMIRegistry et il demande au calculateur le nombre d'opérations qu'il peut exécuter. Le répartiteur prend une opérations de plus que le nombre d'opérations qu'il peut exécuter et les envoie au calculateur. Le répartiteur s'assure ainsi de ne pas avoir de refus du calculateur trop souvent. Lorsque le répartiteur envoie une série d'opérations vers un calculateur, il crée d'abord un Thread et c'est lui qui s'occupe de faire l'appel à distance. Le Thread est créé pour éviter que le répartiteur ne bloque en attendant la réponse du serveur de calculateurs. Pour traiter les serveurs de calculs malicieux, le même calcul est envoyé à deux serveurs de calculs. Le nombre d'opérations envoyées correspond au plus petit nombre parmi les deux serveurs plus 1. Si la réponse n'est pas identique pour les deux serveurs de calculs, les opérations sont remises sur la pile et le répartiteur recommence.

Si un calculateur meurt en court de route, le répartiteur s'en aperçoit par une RemoteException et il l'enlève du RMIRegistry. Si jamais le répartiteur meurt, les calculateurs s'arrêteront après un certain délai puisque les calculateurs vérifient périodiquement s'ils sont encore présents dans le RMIRegistry. Si le répartiteur est reparti avant ce délai, alors les calculateurs se reconnectent automatiquement au RMIRegistry.

Pour exécuter le tout, nous partons d'abord le répartiteur avec la commande suivante :

```
java -jar -Djava.rmi.server.codebase=file:repartitor.jar -Djava.security.policy=policy
-Djava.rmi.server.hostname="(Adresse IP de la machine)" repartitor.jar
```

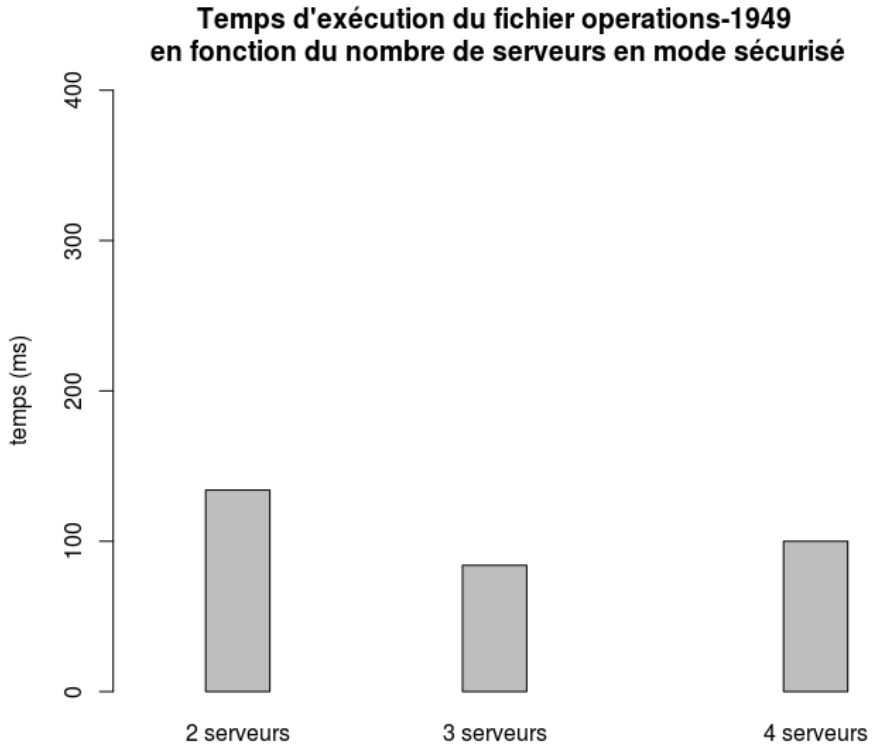
Nous partons ensuite les calculateurs avec la commande suivate :

```
java -jar -Djava.rmi.server.hostname="(Adresse IP de la machine)"
-Djava.security.policy="policy" calculator.jar (Nombre d'opérations que le calculateur peut
accepter) (Pourcentage entre 0 et 100 où le calculateur retourne un mauvais résulat)
(Adresse IP du répartiteur)
```

**Tests de performance – mode sécurisé**

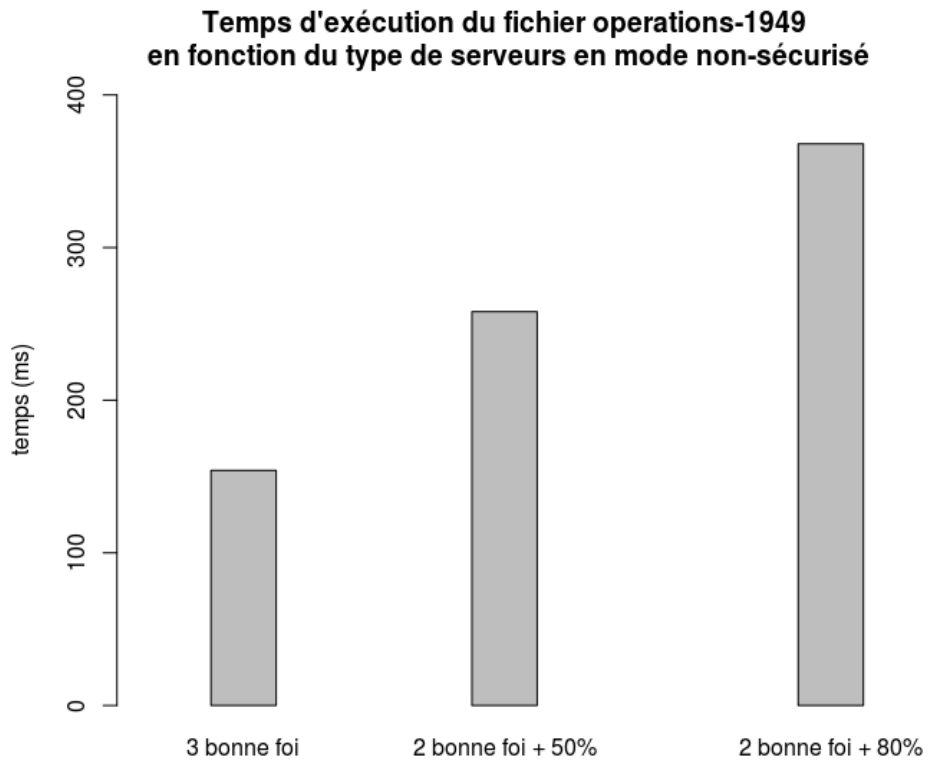
En mode sécurisé, nous n'avons pas vu à peine une petite différence lorsqu'il avait 2, 3 ou 4 serveurs. D'une exécution à l'autre, entre les 2, 3 ou 4 serveurs, le temps était plus court parfois pour 2 serveurs, mais plus long d'autres fois. Le graphique ci-dessous représente le temps d'exécution d'une moyenne de 3 exécutions pour chacun des 2, 3 et 4 serveurs. On remarque tout de même que le temps d'exécution pour 2 serveurs est légèrement plus long tandis que le temps d'exécution pour 3 serveurs

serait légèrement plus court que les autres. Comme le temps de calcul est très court sur les calculateurs, le répartiteur n'a pas terminé de répartir les calculs que déjà les calculateurs ont terminé. Ainsi, que nous ayons 2 ou 100 serveurs de calculs, ça ne change rien, car le répartiteur ne peut pas répartir les calculs plus vite. Nous verrions sûrement une différence si le temps pour effectuer les calculs était plus long, car il y aurait plus de calculs qui s'exécuteraient en même temps selon le nombre de calculateurs.




**Tests de performance – mode non sécurisé**

En mode non sécurisé, nous voyons une petite différence dans le temps lorsqu'il y a des serveurs de calculs malicieux. Ceci est dû au fait qu'il y a plus de calculs qui sont exécutés sur les calculateurs en bout de ligne, car il faut exécuter à nouveau certains calculs. Le graphique ci-dessous représente aussi le temps d'exécution d'une moyenne de 3 exécutions pour chacun des essais. Ici, on remarque que le serveur de calculs malicieux est celui qui fait la différence. Plus il est malicieux, plus il faut exécuter à nouveau des opérations, donc plus ça prend de temps.




**Question 1**

Une architecture possible serait d'ajouter un deuxième répartiteur qui ne ferait que vérifier que le premier répartiteur est en vie et qui récupère l'état du premier répartiteur. Le deuxième répartiteur indique aussi périodiquement au premier répartiteur qu'il est là. Si jamais le deuxième répartiteur ne reçoit plus de signal du premier après un certain délai, il redirigera toutes les requêtes à lui et prendra le relais. Il pourra repartir le premier répartiteur. Si le deuxième répartiteur s'arrête, le premier répartiteur peut s'en rendre compte et le repartir. Si jamais les deux meurent en même temps, alors le système s'écroule quand même. L'avantage est d'avoir une résilience plus élevée aux pannes, car il y a une redondance. Les inconvénients sont d'ajouter de la complexité au système et que le deuxième répartiteur ne soit pas exactement au même niveau que le premier répartiteur lorsqu'il meurt, ce qui veut dire que des calculs seront exécutés à nouveau.