|  |
| --- |
| IDMC - Miage |
| OpenRedis |
| Réseaux |

|  |
| --- |
| Lucas Chartier  Yassmine Elbouhmi  Kevin Colin  Bilal Darouich |

# Architecture

Pour ce qui est de l’architecture nous avons utilisé une architecture Maven classique. Nous avons mis dans un même répertoire 2 projets Maven différents, un client et un serveur. Nous savons que le serveur sera probablement testé avec le client redis officiel et nous avons pris soin d’imiter ses valeurs de retour ainsi que son fonctionnement malgré le fait d’avoir développé notre propre client.

Nous n’avons cependant pas utilisé de dépendances externes.

La structure des fichiers est montré dans la figure 1 ci-dessous.

Une image contenant texte, capture d’écran

Description générée automatiquement

Figure 1 : structure de fichier

# Fonctionnalités développées

Dans un premier temps nous avons créer un serveur classique. Dans un classe main on ouvrait un SocketServer sur le port 6379, puis dans une boucle while sans fin ou récupérer tous les sockets arrivants sur le serveur. Le client permettait de créer un socket sur ce même port et d’envoyer une requête.

Une fois cette base posée nous avons pu commencer à développer chacune des fonctionnalités requises.

## Les commandes

Les commandes n’était pas la fonctionnalité la plus difficile à implémenter, pas de difficulté particulière n’a été rencontré.

### Choix techniques

Les requêtes sont réceptionnées dans l’InputStream du socket, pour la lire nous utilisons la plupart du temps un BufferedReader. La requête est ensuite traitée manuellement.

Pour traiter la requête on découpe la chaîne de caractères sur le caractère « espace » et on vérifie qu’elle ne contient pas un message entre guillemet. Dans le tableau obtenu le premier élément est la commande, on utilise un switch pour effectuer différentes actions en fonction de la commande lancée. Une partie de l’implémentation est montré en figure 2.

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure 2 : implémentation de la méthode manageRequest

## Pipelining et multiligne

### Difficultés rencontrées

Plusieurs difficultés ont été rencontrés pour la fonctionnalité de pipelining et de multiligne et sont listées ci-dessous

* Plusieurs requêtes dans un chaîne de caractères : la méthode manageRequest n’était plus suffisante pour ce besoin. Il fallait dorénavant être capable de séparer les différentes requêtes reçues et dans le socket, de les traiter une à une et de renvoyer les réponses.
* L’opérateur de pipelining : le problème principal est que l’opérateur utilisé pour faire du pipelining par le client redis est le caractère pipe : « | ». Mais la plupart des terminales ne possèdent pas le même jeu de caractères que le l’interpréteur java. Donc lorsque ce caractère était reçu par le socket, il n’apparaissait tout simplement pas ou n’était pas formatté correctement, il n’était pas non plus possible de séparer la chaîne de caractère sur ce caractère ci, on a donc malheureusement dû changer d’opérateur vers l’opérateur « > »

### Choix techniques

Pour répondre à ces deux besoins nous avons mis en place une nouvelle méthode. Celle-ci permet de récupérer toutes les requêtes présentes dans une chaîne de caractères en fonction des séparateurs requis, donc « > » et « \n ». Pour chaque requête identifiée on l’exécute avec la fonction manageRequest et on concatène le résultat à une nouvelle chaîne de caractères. En figure 3 l’implémentation de la méthode manageRequests et en figure 4 un exemple d’utilisation du pipelining.

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure 3 : implémentation de la méthode manageMultipleRequest

Une image contenant texte, capture d’écran, Police, noir

Description générée automatiquement

Figure 4 : exemple de pipelining

## Le pattern publish / subscribe

### Difficultés rencontrées

Pour ce pattern nous avons choisis de stocker les sockets « abonnées » à un canal dans une map où la clé représente ce canal. Lorsqu’un client publie sur un canal on transmet à toutes les sockets correspondants.

### Choix techniques

Lorsqu’un client passe un subscribe on doit lui permettre de saisir certaines commandes uniquement puisque lui-même est figé dans un état de souscription. On utilise un while pour continuer à écouter sur le socket en flux entrant et sortant, ainsi on peut envoyer au client de la donnée et récupérer ses requêtes. En figure 5 et 6 l’implémentation des méthodes permettant au serveur de gérer la souscription.

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement

Figure 5 : envoie de la requête vers la fonction

Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement

Figure 6 : implémentation de la méthode manageSubscription

## La gestion de multiple client

### Difficultés rencontrées

Encore une fois plusieurs problèmes ont été rencontrés et sont listés ci-dessous

* Bien que la boucle while qui permettait auparavant de gérer les requêtes réceptionnées permettaient déjà de gérer un nombre convainquant de client sans soucis tant que le traitement est léger, ce n’était pas suffisant pour permettre de gérer énormément de client.
* De plus, il y avait de gros problème quant à la façon dont pouvait être gérer certaines commandes. Par exemple pour le subscribe, il faut garder la connexion avec le socket du client ouverte sur le flux entrant et le flux sortant, pour pouvoir afficher les nouvelles informations mais aussi récupérer les requêtes éventuelles du client. Le problème est que faire ceci avec un autre while bloque l’exécution du Thread et le serveur n’est alors plus capable de récupérer les requêtes entrantes.

### Choix techniques

Pour gérer efficacement de multiple client la solution était donc de créer un nouveau thread pour chaque nouvelle connexion entrante. La boucle while a donc été changé pour permettre la création du Thread à chaque connexion entrante comme montré en figure 7.

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement

Figure 7 : boucle d’écoute des nouveaux sockets

Il fallait également que les Threads puissent accéder et modifier la même donnée, pour cela on a rendu nos structures de données et collections statiques dans la classe principale.

Le Thread créé permet de gérer l’entièreté de la connexion avec un client ainsi que la gestion de ses requêtes. En figure 8 un extrait du thread qui permet cela.

Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement

Figure 8 : extrait de la classe ServerHandler.java

## La réplication master / slave

### Difficultés rencontrées

Deux grosses difficultés ont été rencontrés de nouveaux à cause des Threads.

* Il fallait trouver un moyen de permettre à un serveur d’être reconnu par un autre en tant que esclaves, ce qui est un problème puisque par défaut un serveur se lance et écoute les clients
* Il fallait faire en sorte qu’un serveur ne se lance pas tout le temps sur le même port puisqu’il n’est pas possible d’ouvrir deux serveurs sur le même port sur la même machine.
* Pour permettre à un serveur de se connecter à un autre on a permis à l’utilisateur d’un serveur maître d’entrer une commande pour se connecter à un esclave. Cependant il fallait également déplacer l’écoute des sockets entrantes sur un autres Threads pour ne pas stopper l’écoute quand il souhaite entrer une commande.

### Choix techniques

Pour répondre aux problèmes rencontrés on a effectué plusieurs modifications. On a commencé par déplacer l’écoute des sockets entrantes sur un Thread, un RequestListener dont l’implémentation est montrée en figure 9, et on a permit au Thread principale de lancer un serveur en fonction de paramètre demandé à l’utilisateur au lancement de l’application.

Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

Description générée automatiquement

Figure 9 : implémentation du RequestListener

L’utilisateur peut donc désormais lancer un serveur sur le port qu’il souhaite comme montré en figure 10.

Une image contenant texte, capture d’écran, Police, noir

Description générée automatiquement

Figure 10 : interface de lancement du serveur

Lors du lancement du serveur on vérifie s’il s’agit d’un serveur maître ou d’un serveur esclave.

* S’il s’agit d’un serveur maître on lance le Thread RequestListener.java et on attend pour l’utilisateur serveur d’entrer une commande. La figure 11 montre la méthode qui permet de stopper l’exécution a cet endroit et d’écouter les messages de l’utilisateur.
  + L’utilisateur dispose de deux commandes : « quit » et « connect <port> »
* S’il s’agit d’un serveur esclave on attend la connexion d’un serveur maître. Lorsqu’un serveur mapître se connecte on synchronise la Map avec le serveur Maître et on attend pour récupérer les requêtes suivantes. Tout cela s’effectue dans un Threads différents, le MasterListener, semblable au RequestListener et qui permet d’écouter les requêtes envoyées par le serveur, son implémentation est donnée en figure 12. La séparation du Listener dans un autre Thread permet de permettre à l’utilisateur de saisir des requêtes tout en continuant à recevoir les données du serveur maître. La figure 13 montre la méthode qui permet de stopper l’exécution pour un serveur slave.
  + L’utilisateur dispose ensuite de 2 commandes : « quit » et « make master ». Lorsque qu’un esclave devient un Master on exécute alors le RequestListener et on peut connecter se serveur à des esclaves.

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure 11 : implémentation de la méthode handleClientConnection pour le serveur maître

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure 12 : implémentation de MasterListener.java

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure 13 : implémentation de la méthode handleServerConnection pour le serveur esclave

## Le clustering

Le clustering ne fait pas partie des fonctionnalités que nous avons eu le temps de développer cependant nous avons pu réfléchir à quelques problématiques que ceci représentait.

La connexion pose déjà un problème :

* Dans l’état actuel des choses un Thread écoute tous les sockets entrants sur le serveur, sauf si le serveur est un esclave puisque dans ce cas on sait alors que le premier socket qui tentera de se connecter au serveur sera forcément un serveur maître. Mais si le serveur doit être à la fois maître et esclave il faut qu’il puisse différencier les requêtes qui proviennent d’un serveur qui tente de se connecter et de se synchroniser et celles qui proviennent d’un client qui voudrait effectuer des opérations sur les données.
  + Solution possible : Peut-être que l’objectMapper et l’utilisation de instanceof aurait permit de faire une différence entre les requêtes en permettant au serveur d’envoyer un type d’objet que lui seul peut envoyer. Les requêtes seraient traitées comme des string et les serveurs comme des objets différents.

Une fois cette différence effectuée il aurait été assez facile d’ajouter les serveurs à des listes de serveurs associés et de les synchroniser entre eux. Cependant on ne sait pas s’il existe un mécanisme autre que les verrous pour assurer l’intégrité des données entre tous les serveurs du cluster et à tout moment.

# Quelques précisions

* On assure l’intégrité des données dans l’application en posant des verrous sur les fonctions qui peuvent être appelées par les différents threads et pour lesquels on écrit dans une variable, un exemple en figure 14
* Les données qui doivent être partagées sont rendu static dans la classe OpenServer, un exemple en figure 15
* Pour être sûr du bon fonctionnement de toutes les fonctionnalités
* Les souscriptions à des canaux ainsi que les canaux en eux-mêmes ne sont pas synchronisés avec le serveur pour les raisons suivantes :
  + Même avec en utilisant un ObjectOutputStream où un ObjectInputStream on ne sait pas comment aurait réellement été transmise la connexion avec ce socket et s’il était encore possible de les utiliser.
  + Lorsque le serveur maître s’arrête il est supposé mettre fin à toutes les connexions, donc en cas d’interruptions du serveur maître tous les sockets auraient été fermé et le client n’aurait pas été capable de continuer son utilisation normale.
* Une erreur a été faite et les channels ont été implémentés en tant que Map et non avec les ServerSocketChannel.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Figure 14 : verrou posé sur la fonction sendMessage

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Figure 15 : variables statiques de la classe OpenServer.java

# Tests

Pour tester nos clients nous n’avons pu utiliser qu’une machine locale. Comme nous avons également développé un client nous avons utilisé ce client pour effectuer la plupart de nos tests. Pour tester nous lancions un ou plusieurs serveurs ainsi que plusieurs clients, nous les connections entre eux en effectuant différentes actions et on vérifiait que tout se déroulait correctement et que l’intégrité des données était préservé.

Lorsque nous avons compiler le serveur et le client en .jar nous créer des scripts de lancement puis avons effectuer les mêmes tests qui se sont déroulés sans problèmes.

# Conclusion

Ce projet nous à permit de manipuler des mécanismes de réseaux ainsi que de développement intéressants et innovants dans nos études. Il nous a permit de comprendre plus en profondeur le fonctionnement des sockets et de pouvoir les manipuler.