

# Rapport Réseaux Informatiques

*par HAMRI Nabil, KIRBACH Hugo, ADAM Thomas, BOURREL Adrien*

<b>Introduction.....</b>	<b>3</b>
<b>Mode de travail et répartition des tâches.....</b>	<b>3</b>
<b>Choix techniques et technologiques utilisés.....</b>	<b>3</b>
Langage de programmation.....	3
Modèle de programmation.....	3
Communication réseau.....	4
Nos tests.....	4
<b>L'architecture du projet.....</b>	<b>4</b>
<b>Fonctionnalités implémentées.....</b>	<b>5</b>
Commandes classique.....	5
Formatage Redis-cli.....	8
Relation client ⇒ serveur.....	8
Relation serveur ⇒ client.....	8
Publish/Subscribe.....	9
Réplication.....	9
La compréhension.....	9
Ce que nous voulions faire.....	10
Ajout de la commande.....	10
Définition du Master.....	10
Enregistrement des commandes saisies pour réplication.....	10
Définition du Slave.....	10
Maintenabilité.....	10
Le résultat.....	10
<b>Difficultés rencontrées et solutions trouvées.....</b>	<b>10</b>
Publish/Subscribe.....	11
Réplication.....	11
Problème de récupération des données.....	11
Solution envisagée.....	11
<b>Conclusion.....</b>	<b>11</b>

## Introduction

Notre rapport expose le projet de mise en œuvre d'un serveur Redis, réalisé par notre équipe de quatre personnes. Notre objectif principal était d'implémenter un serveur Redis fonctionnel, accompagné d'un dossier détaillé décrivant toutes les phases de conception, de développement et de livraison du projet. Ce dossier comprendra une documentation approfondie sur l'architecture du projet, les choix techniques effectués, les difficultés rencontrées ainsi que les solutions apportées et les fonctionnalités implémentées.

Notre projet tourne autour de la communication client-serveur. Le serveur Redis que nous avons développé doit être capable de gérer plusieurs clients simultanément en gardant une certaine cohérence pour tous les clients. Nous détaillerons plus en détails les fonctionnalités que nous avons dû développer dans la partie "Fonctionnalités implémentées".

## Mode de travail et répartition des tâches

Afin de se répartir de manière équitable le travail nous avons commencé par mettre en place un backlog sur Trello. Nous avons donc ensuite commencé par faire une réunion "d'initialisation" du projet durant laquelle nous avons mis en place le répertoire Git, complété le Trello avec les tâches à faire ainsi que les erreurs à éviter aussitôt que nous en avons en tête. Par la suite nous avons discuté de qui allait faire quoi, et nous sommes donc attribué les tâches correspondantes.

## Choix techniques et technologiques utilisés

### Langage de programmation

Nous avons réalisé ce projet en **Java 17** ([lien](#)) avec une forte utilisation du package "java.io.OutputStream" et "java.io.InputStream". Cela permet une communication réseau.

### Modèle de programmation

Nous avons décidé d'utiliser les threads pour gérer les connexions clients en simultanées. Chaque client est géré par un thread dédié, permettant par conséquent une exécution des requêtes en parallèle.

La gestion des clients unique permet de manipuler la classe serveur qui comporte la structure de données qui stocke les données. C'est le "**DisqueDur**".

## Communication réseau

La communication client-serveur s'effectue avec le protocole TCP/IP. Nous avons utilisé les classes Java basiques telles que `ServerSocket` et `Socket` pour mettre en place la communication réseau. Cela nous a permis de bénéficier d'une communication fiable et d'une bonne gestion des connexions et des flux de données.

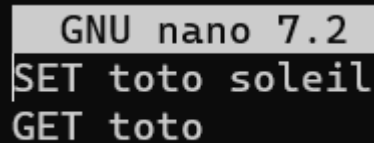
## Nos tests

Pour tester notre serveur Redis, nous avons utilisé un terminal "redis-cli". Nous l'avons donc ensuite utilisé afin de tester toutes les commandes que nous devons implémentées telles que l'enregistrement d'une valeur dans une clé, la récupération de cette valeur ou encore la vérification de l'existence d'une clé.

Pour gérer le pipeline, c'est-à-dire réaliser plusieurs commandes à la suite. Nous séparons la donnée au `\r\n`, cela permet d'exécuter une par une les commandes.

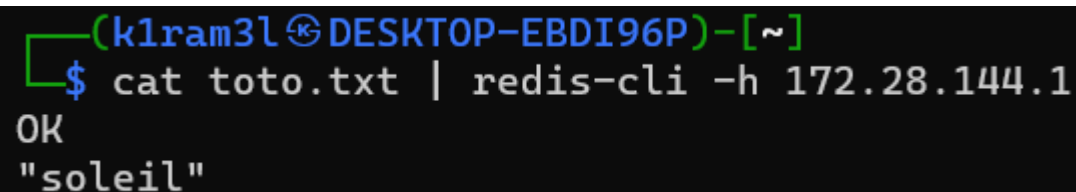
### Exemple :

1. nano toto.txt



```
GNU nano 7.2
SET toto soleil
GET toto
```

- 2.



```
(k1ram3l@DESKTOP-EBDI96P)~$ cat toto.txt | redis-cli -h 172.28.144.1
OK
"soleil"
```

- 3.

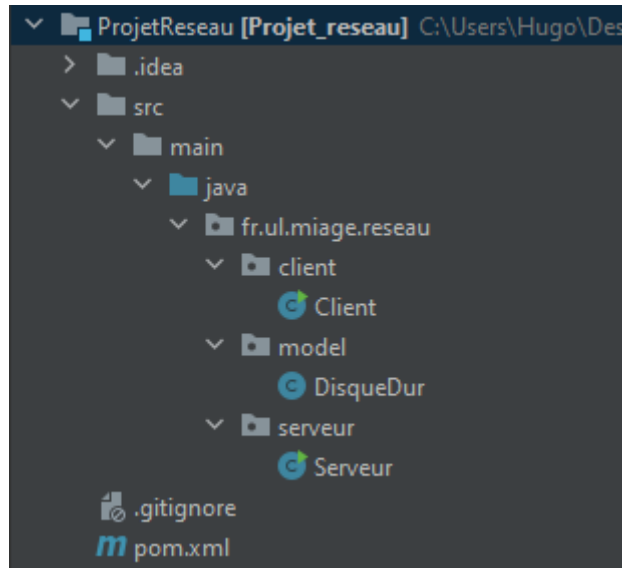
Pour les commandes classiques, il y a un script dans le git "script\_commande.txt" :

```
SET nabil "Hugo aime les pizzas"
GET nabil
STRLEN nabil
EXPIRE nabil 60
SET toto tata
EXISTS toto
SET toto "tata"
APPEND toto nana
INCR hugo
INCR hugo
DECR hugo
```

## L'architecture du projet

Notre projet est sous forme d'un projet Maven, décision qui a été prise à l'initiation du projet dans le cas où une librairie quelconque aurait pu nous servir dans le projet.

L'arborescence du projet est aussi simple que la suivante :



Où la classe `DisqueDur` correspond à notre stockage des données (clé/valeurs), `Serveur` est la classe principale servant à lancer le serveur et à analyser les requêtes effectuées par le client. Enfin, la classe `Client` nous ayant servis durant le début du projet, période pendant laquelle nous testions nos commandes via console d'IDE et non à travers un client Redis-cli. Cette classe nous a donc permis de tester une bonne partie de la version initiale du projet.

## Fonctionnalités implémentées

### Commandes classique

Nous avons effectué nos tests en local et pour faire cela nous nous sommes connecté à notre serveur avec notre adresse ipv4, d'où la commande "**redis-cli -h <IP>**"

Nous devons implémenter diverses commandes opérant sur les chaînes :

- SET key value / SETNX key value

La commande "SET" enregistre une valeur dans la clé mise en premier paramètre;

```
nabil@LAPTOP-J6IECHH3:~$ redis-cli -h 172.19.160.1
172.19.160.1:6379> SET cle test
OK
172.19.160.1:6379> |
```

```
172.19.160.1:6379> SETNX cle2 valeur
OK
172.19.160.1:6379> GET cle2
"valeur"
172.19.160.1:6379> |
```

Si la clé est déjà enregistrée, elle sera alors réécrite avec la nouvelle valeur saisie :

```
172.20.144.1:6379> set a a
OK
172.20.144.1:6379> get a
"a"
172.20.144.1:6379> set a b
OK
172.20.144.1:6379> get a
"b"
```

- GET key

La commande "GET" récupère tout simplement la valeur de la clé que l'on souhaite et l'affiche :

```
nabil@LAPTOP-J6IECHH3:~$ redis-cli -h 172.19.160.1
172.19.160.1:6379> SET cle test
OK
172.19.160.1:6379> GET cle
"test"
172.19.160.1:6379> |
```

- STRLEN key

La commande "STRLEN" calcule la longueur de la valeur de la chaîne contenue dans la clé mise en paramètre :

```
172.19.160.1:6379> GET cle
"test"
172.19.160.1:6379> STRLEN cle
(integer) 4
172.19.160.1:6379> |
```

- APPEND key « chaîne à concaténer »

La commande "APPEND" récupère la valeur de la chaîne contenue dans une clé et concatène une valeur mise en second paramètre après la première valeur.

```
172.19.160.1:6379> APPEND cle " test2"
(integer) 10
172.19.160.1:6379> GET cle
"test test2"
172.19.160.1:6379> |
```

- INCR key / DECR key

Les commandes suivantes servent tout simplement à incrémenter ou décrémenter la valeur d'une clé si celle-ci est une valeur numérique.

```
172.19.160.1:6379> GET cle2
"2"
172.19.160.1:6379> INCR cle2
(integer) 3
172.19.160.1:6379> DECR cle2
(integer) 2
172.19.160.1:6379> DECR cle2
(integer) 1
172.19.160.1:6379> |
```

A savoir que si la clé n'existe pas, elle sera créée et initiée à 0, et si la clé n'est pas un integer, un message d'erreur sera renvoyé en console pour informer l'utilisateur.

- EXISTS key

Dans l'exemple suivant, nous initialisons une clé avec une valeur pour vérifier son existence par la suite. Puis nous exécutons la commande "EXISTS" qui nous renvoie bien "1", ce qui signifie que cette clé existe bien.

De plus, lorsque nous voulons vérifier l'existence de la clé "toto", cela nous indique bien "0", ce qui signifie qu'elle n'existe pas.

```
172.19.160.1:6379> SET cle test
OK
172.19.160.1:6379> EXISTS cle
(integer) 1
172.19.160.1:6379> EXISTS toto
(integer) 0
172.19.160.1:6379> |
```

- DEL key

Pour vérifier que la suppression d'une clé fonctionne bien, nous avons exécuté la commande correspondante puis nous avons relancé la précédente commande qui est "EXISTS" pour vérifier son existence et nous remarquons que cette clé n'est plus connue puisqu'elle a été supprimée.

```
172.19.160.1:6379> DEL cle
(integer) 1
172.19.160.1:6379> EXISTS cle
(integer) 0
172.19.160.1:6379> |
```

- EXPIRE key duration

Concernant la commande "EXPIRE", il faut renseigner le nom de la clé et le temps (en secondes) avant que la clé ne s'expire et par conséquent n'existe plus. Dans l'exemple ci-dessous, nous avons exécuté la commande EXPIRE sur cette clé avec un temps de 10 secondes. Au bout de ces 10 secondes, nous avons ré-exécuté la commande "EXISTS" et nous pouvons voir que la clé n'existe plus.

```
172.19.160.1:6379> EXPIRE cle 10
OK
172.19.160.1:6379> EXISTS cle
(integer) 1
172.19.160.1:6379> EXISTS cle
(integer) 0
172.19.160.1:6379> |
```

## Formatage Redis-cli

Le logiciel est utilisable avec le client redis-cli. Cela veut dire que le serveur doit traduire le formatage du client puis envoyer les données formatées.

### Relation client ⇒ serveur

Nous devons traduire ce type de données, \***3****\$7**publish**\$4**toto**\$7**message :

- \***3** : Cela veut dire qu'on a **3** parties
- **\$7** : C'est la longueur de la chaîne qui suit.
- **publish**, **toto** et **message** : Ce sont les éléments saisis par l'utilisateur

### Relation serveur ⇒ client

Nous avons différents retours :

- - **ERR** **message**\r\n ⇒ S'il y a une erreur
- +**OK**\r\n ⇒ Si la commande est validé
- **:Chiffre**\r\n ⇒ Si le retour est un nombre
- **\$LongueurChaine**\r\n**Chaine**\r\n ⇒ Si le retour est une chaîne de caractère



Avant le formatage pour redis-cli, on utilise un client réalisé par nous même. Il pouvait envoyer et recevoir les données. C'est le fichier client.java qu'on a parlé ci-dessus.

## Publish/Subscribe

Nous avons implémenté les méthodes SUBSCRIBE/PUBLISH.

### Etape 1 :

D'abord, nous lançons un "subscribe" qui permet de suivre et d'écouter un "channel"/canal.

```
172.28.144.1:6379> subscribe toto
Reading messages... (press Ctrl-C to quit)
OK
```

### Etape 2 :

Par la suite, on ouvre un autre terminal (client) et on publie un message sur le channel.

```
172.28.144.1:6379> publish toto message
OK
172.28.144.1:6379> |
```

### Etape 3:

On voit qu'on a reçu le message dans le premier client

```
172.28.144.1:6379> subscribe toto
Reading messages... (press Ctrl-C to quit)
OK
"message"
```

## Réplication

Malheureusement, l'étape de réplication n'a pas été entièrement implémentée, malgré de nombreuses tentatives non fructueuses.

### La compréhension

Le principe de réplication en lui-même n'est pas si complexe à comprendre, et semblait tout aussi simple à implémenter. Pourtant, nous ne savons si c'est dû aux "mauvaises pistes" que nous avons empruntées mais nous n'avons pas réussi à avoir un résultat satisfaisant concernant la requête de Réplication. Le détail de ce que nous voulions faire au moment du développement est expliqué dans l'étape suivante.

## Ce que nous voulions faire

### Ajout de la commande

Premièrement, il fallait ajouter la commande de replication à notre ensemble de commande. Pour se faire nous avons donc ajouter “RELPICAOF” à notre ensemble de commande.

### Définition du Master

Pour définir qui était le Master et le Slave nous avons décidé de définir un booléen “Master” qui était par défaut à False et passait à true dans le cas où l'utilisateur saisissait la commande “REPLICAOF NO ONE”, commande de redis-cli pour définir le Master.

### Enregistrement des commandes saisies pour réplication

Nous avons ensuite fait une fonction qui stock l'intégralité des commandes valides (celles qui ont un impact et ne renvoient pas un message d'erreur) dans une hashmap avec comme clé un indice croissant et unique commençant à 1, et comme valeur la commande, en byte, saisie par l'utilisateur.

Sur le principe, cet objet devait servir donc à stocker toutes les commandes du Master sur le serveur principal de tel sorte à ce que le slave puisse cloner ces commandes. Nous avons également essayé, en parallèle, d'envoyer directement le contenu de l'objet DisqueDur de manière à copier les données plutôt que les commandes.

### Définition du Slave

Pour se définir en tant que Slave, il faut donc que le nouveau client (sur un port différent à minima) tape “REPLICAOF [IP] [PORT]”. Ce client clone donc deux choses dans nos essais:

- La liste des commandes du serveur sur le PORT en question et les répliques dans l'ordre de la hashmap.
- Le contenu de l'objet DisqueDur pour avoir toutes les données d'un coups

Cependant, ces deux test nous ont posé soucis et seront détaillés dans la partie “[Réplication](#)” des Difficultés rencontrées et solutions trouvées.

### Maintenabilité

Le but initial était que lorsque le serveur Master s'interrompt, il passe un/le Slave en tant que Master. Puisque nous avons bloqué à l'étape précédente, nous n'avons pas implémenté cette feature.

## Le résultat

Finalement, nous n'avons donc pas réussi à implémenter la méthode de Réplication dû aux nombreux problèmes intervenus durant le procédé de développement. Le détail de ces problèmes et comment nous les avons traités se trouve dans la partie suivante.

## Difficultés rencontrées et solutions trouvées

### Publish/Subscribe

Tout d'abord avant de réaliser la compatibilité avec le client redis-cli, il y avait aucun problème. Effectivement avec l'utilisation du redis-cli, dès qu'on quitte la commande subscribe avec Ctrl + C, Cela créer un nouveau client alors qu'on demande de quitter la commande. Il y a aucun message qui est envoyé au serveur à la suite de la déconnexion de cette commande.

C'est peut-être dû à un problème du client redis ou un retour du client nous arrivé au serveur (utilisation wireshark).

### Réplication

#### Problème de récupération des données

Comme expliqué précédemment, nous avons bloqué à l'étape "[Définition du Slave](#)". Dans la partie définition de la commande REPLICAOFF [IP] [PORT] nous avons tenté d'instancier un nouveau serveur sur le port saisi dans la commande (correspondant au port du serveur master). C'est cette instanciation qui nous a posé problème puisqu'une fois le nouveau serveur instancié, il était impossible de se connecter à ce dernier.

Premièrement, si nous nous connectons à ce nouveau serveur avec les données normalement répliquées, lorsque nous tentions de récupérer ce jeu de commande ou l'objet disque dur, ils étaient tous deux vides (alors que supposé complété, au moment du test, par quelques commandes). D'un autre côté, lorsqu'on tentait de faire serveur.run() sur ce nouveau serveur, une erreur apparaissait disant qu'il était impossible d'utiliser ce bindAdress. Après quelques recherches, nous avons tenté toutes les solutions trouvées sur internet (Kill les tâches utilisant cette adresse, changer d'adresse, ...) mais rien n'y faisait. Nous avons donc décidé d'essayer de passer par une méthode plus simple.

#### Solution envisagée

Après ces tentatives ratées, nous avons décidé d'essayer de procéder de manière plus simple. Le principe cette fois-ci était de lancer deux connexions au moment de l'initiation du serveur, une sur le port que la personne choisissait à l'exécution et une sur un serveur avec un port fixe défini comme slave. Cette solution n'est pas exactement la manière dont fonctionne la réplication mais nous voulions essayer cette solution de "secours".

Une fois ces deux connexions lancées, nous voulions qu'à partir du moment où l'utilisateur se définisse Master (REPLICAOFF NO ONE), toutes ses commandes soient directement envoyés sur le 2eme serveur en parallèle, de tel sorte à ce que quand un nouvel utilisateur se connecte à ce port il ai accès a toute la BDD des valeurs.

Malheureusement cette solution n'était pas aussi simple puisqu'une fois de plus l'exécution du socket généré une erreur liée à la BindAdress.

Nous avons donc finalement décidé de ne pas intégrer ces débuts de solutions à notre produit final car il n'y avait pas de sens à laisser l'input des fonctions possible sans résultat cohérent. Nous avons cependant laissé ces tentatives sur notre GIT sur les branches "MasterSlave" et "feature/testMasterSlave" respectivement pour la 1ere et 2ème solution évoqué dans ce rapport.

## Conclusion

En conclusion, la mise en place de notre propre implémentation d'un serveur redis nous a permis de mettre en pratique les connaissances en programmation mais aussi en réseau. Nous avons réussi à développer un serveur fonctionnel avec un sous-ensemble de commandes Redis malgré le manque de la prise en charge du pattern de "master/slave".