# Rapport ALGOREP 2021

Ségolène Denjoy
Etienne Sharpin
Laurélie Michielon
Laura Lacambra
Promo 2022 - SCIA - IMAGE


2 novembre 2021

# Table des matières

# 1 Introduction

Nous souhaitons mettre en place un système clients/serveurs avec un mécanisme permettant de contrôler ou d'injecter des fautes dans le système. L'idée générale est la suivante : les clients proposent des valeurs/commandes aux serveurs. Ces serveurs souhaitent ensuite se mettre d'accord sur l'ordre dans lequel ils vont accepter ces commandes. Une fois d'accord, il les écriront dans un fichier de log. Chaque serveur doit, à la fin de l'exécution, avoir le même fichier de log. Il s'agit donc d'une forme de réplication de logs.

# 2 Consensus

La première étape est le consensus. Cela consiste à construire un système dans lequel chaque client va posséder une valeur/commande originale (on utilisera premièrement leur uid). Les clients vont alors soumettre leurs valeurs aux serveurs et attendre leur confirmation de réception. Chaque serveur admet sa propre version du log, qui doit être identique lorsque le système se termine.
Pour ce faire, nous avons choisi d'implémenter l'algorithme de RAFT. Il nous permet de partager l'état des différentes machines et leurs transitions d'état. Cet algorithme peut se décomposer en 2 problèmes indépendants : l'élection des leaders et les réplications des logs que nous expliquons ci-dessous.

## 2.1 Election du Leader

Le principe de cette étape est de trouver un leader parmi la totalité des serveurs. Ainsi le leader pourra communiquer avec tous les serveurs tandis qu'eux ne pourront communiquer qu'avec le leader. Pour ce faire chaque serveur peut devenir candidat et donc se présenter en tant que leader. L'éléction du leader se fait uniquement si il a la majorité des votes des autres serveurs.
Rappelons donc qu'un serveur peut se positionner dans l'un de ses trois états :

— Leader : utilisé pour communiquer avec les clients et les autres serveurs

— Follower : utilisé pour répondre au leader

— Candidate : utilisé pour l'éléction d'un nouveau leader

Sachant qu'il ne peut y avoir qu'un leader.
Au début tous les serveurs sont dans l'état Follower. Premièrement il faut donc élir un leader. Pour ce faire, vient l'étape de l'élection.

Les serveurs les plus rapides à sortir de leur état de Follower, deviennent des Candidates. Ils se proposent donc à être leader et c'est le premier qui obtient la majorité qui le devient. Plus précisément, chaque candidat vote pour lui-même puis ils envoient une demande de vote aux Followers. Si un Follower n'a pas encore voté pour cette élection, alors il vote pour le premier candidat qui le contacte. Un candidat devient donc leader s'il obtient la majorité. Si aucun des candidats n'a la majorité, une nouvelle élection recommence.
Après cette étape, nous avons notre nouveau leader.

Il prévient alors tous les autres serveurs que c'est désormais lui le leader puis il commence à envoyer des heartbeats aux serveurs tous les X temps pour que les serveurs sachent qu'il est toujours en vie et reste en contact avec eux. Ces serveurs répondent alors au leader et ainsi il sait également quel server est en vie. On ne prend pas en considération lorsqu'un serveur ne répond pas à nos heartbeats. Cependant, si personne ne répond alors le Leader ne se considerera plus comme tel et repassera dans l'état Follower.
Un Follower considèrera son leader comme mort seulement si il ne reçoit pas de heartbeat pendant un laps de temps.

Précédemment, lorsque l'on utilisait un rapport avec la vitesse ou le temps (comme par exemple "plus rapide" ou "tous les X temps"), nous faisons en fait, référence à un timer. Chaque serveur a un timer associé qui est déterminé de manière aléatoire entre un intervalle de valeurs. Ce timer permet d'effectuer des actions différentes en fonction de la réception ou non de données.

1. Si l'on ne reçoit pas de donnée, cela nous permet principalement de changer d'état.
2. Si l'on en reçoit, la plupart du temps nous allons répondre à notre destinataire.

Dans le cas des Followers :

1. Si l'on n'en reçoit pas, cela permet de faire le passage avec l'état Candidate. Le Follower passera donc en Candidate et se présentera en tant que Leader.
2. Si l'on en reçoit il y aura 3 grands types de données :
   — Les données relatives au client et donc envoyé par le leader.
   — Les données liées à l'éléction d'un leader et aux heartbeats.
   — Les données liées à la présentation de candidats.

   Dans tous ces cas, le but du Follower est de répondre à ses messages.

Dans le cas du Leader :

1. Si l'on ne reçoit pas de donnée, il n'y a donc aucun client qui ne communique avec nous et aucun serveur qui ne répond à nos heartbeats. Dans ce cas, cela veut donc dire que nos derniers heartbeats n'ont pas été réceptionnés et donc que nous sommes mort à la vue des autres serveurs. Nous ne sommes donc plus Leader, nous changeons en état Follower.
2. Si l'on en reçoit, alors il y aura 2 grands types de données :
   — Les données relatives au client et donc que l'on doit relayer aux serveurs.
   — Les données liées aux heartbeats.

Pour récupérer des données, nous avons tout d'abord eu l'idée d'utiliser irecv qui est non bloquant puis de vérifier si la requête récupérait un message et ainsi utiliser la fonction wait pour récupérer ce message. Cependant, nous avions des petits problèmes de réception. Des fois nous ne recevions aucun message et des fois oui. On a donc changé de méthode et nous nous sommes rabattus sur la fonction iprobe qui permet, par un booléen, de connaître le serveur qui veut nous envoyer un message. De là, nous enchaînons avec un recv qui est certes bloquant mais seulement au serveur qui souhaite nous envoyer un message.

## 2.2 Réplication des logs

Les données des clients sont envoyées au Leader. Celui ci stocke les infos dans une liste des logs non-commits. Il envoit ensuite ces données aux autres serveurs. Dès que la majorité des serveurs répondent au commit, les données sont sauvegardées sur disque dans un fichier de log *log_replication_N.txt* où $N$ correspond au rank du serveur.

Ces fichiers sont présents dans le dossier *logs_server*.

# 3 REPL

Pour cette partie, le but est de créer un processus que nous appelerons le processus REPL qui va artificiellement modifier le comportement de notre système. Ce processus ne fait donc pas parti de notre système, il sert à envoyer des messages afin d'interagir avec les processus du système.
Notre REPL est maintenant capable d'envoyer les ordres suivants : START, END, CRASH, RECOVERY et SPEED.

## 3.1 START

Cette commande permet le démarrage des processus clients. Chaque client va donc attendre de recevoir cette commande par le REPL avant d'effectuer leurs demandes.

## 3.2 END

Cette commande permet d'arrêter l'exécution du programme proprement sans avoir besoin de faire Ctrl+C.

## 3.3 CRASH & RECOVERY

La commande CRASH sert à simuler la mort d'un processus tandis que RECOVERY permet la résurection d'un processus mort, c'est-à-dire un processus qui a reçu CRASH précédemment.

### 3.3.1 CRASH

Lors de la mort du processus, tous les messages reçus seront alors ignoré à l'exception des commandes du REPL et plus précisément celle du RECOVERY. C'est donc à l'aide de la fonction recv permettant de bloquer le serveur concerné, que l'on fait attendre le serveur jusqu'à ce qu'il reçoive la commande RECOVERY du REPL. Cette dernière permettra au serveur de revivre et donc de réecouter tous les autres processus.

Voyons le cas où on applique un CRASH puis un RECOVERY au leader. Lors de sa mort, des éléctions ont lieu et un nouveau leader est alors élu, puisque les serveurs ne recevaient plus de heartbeat *[cf. Test 5.1.2]*. Lorsque l'ancien leader reçoit RECOVERY, il renait et reprend donc son poste de Leader. Il envoie alors ses heartbeats mais les serveurs ayant enregistrés le nom du nouveau leader savent que les heartbeats que l'ancien Leader envoie ne sont pas ceux qu'ils doivent considérer : ils

ne répondent pas. L'ancien leader ne reçoit donc aucun retour de ses heartbeats, il se retire alors de sa position de Leader et devient Follower.

### 3.3.2 RECOVERY

La commande RECOVERY sert à simuler le retour d'un processus dans le système. Elle ne fait rien de particulier si ce n'est break une boucle infinie où était coincé le processus.

Dans le cas d'un serveur, quand le processus enverra son heartbeat avec le nombre de logs commits au leader, le leader se rendra compte que le follower n'est plus à jour et lui enverra la bonne version des logs.

## 3.4 SPEED

Cette commande sert à impacter la vitesse d'execution d'un processus. Nous avons 3 modes de vitesse auquelles nous affectons différents intervalles :

— LOW : qui se situera entre 450 et 600ms

— MEDIUM : qui se situera entre 300 et 450ms

— HIGH : qui se situera entre 150 et 300ms

Initialement, chaque processus s'exécute à la vitesse MEDIUM.
Le timer sera donc dans l'un de ses intervalles. Rappelons donc que plus la vitesse d'exécution est lente, plus le timer sera grand. Et inversement, lorsque la vitesse d'exécution est rapide, le timer sera petit.

Sachant cela, si l'on applique SPEED LOW au leader et un SPEED HIGH à l'un des serveur, il se peut que le serveur ne reçoive pas les heartbeats du leader et procède donc à une élection *[cf. Tests 5.1.3]*. Dans notre programme, nous ne verrons jamais ce problème arriver puisque l'on considère le leader comme mort seulement si l'on ne reçoit pas ses heartbeats pendant 5 tours de notre timer. Il faudrait donc diminuer le nombre de tours où accentuer la différence des intervalles des modes de vitesse pour tomber dans ce cas.

# 4 Liste de commandes

Le script principal qui demande le nombre de clients et de serveurs va également générer un nombre aléatoire de données uniques et de tailles aléatoires pour chaque client.
Les clients vont ensuite lirent leur fichier de données à envoyer aux serveurs et envoyer chaque commandes régulièrement. Chaque client attend un temps aléatoire entre 2 et 4 secondes entre chaque commande afin de permettre à l'utilisateur de mieux tester les différentes commandes du REPL. Ce temps d'attente peut être accéléré ou ralentit grace à la fonction SPEED du REPL.

Après l'envoie d'une commande, le client attend que le leader renvoit une confirmation de la replication du log. Si le leader ne répond pas au bout de 10 heartbeats du client, le client va donc s'assurer que le leader n'a pas changé et renvoyer ses données.

# 5 Tests

## 5.1 Tests sur les serveurs

Voici notre pipeline de notre algorithme.

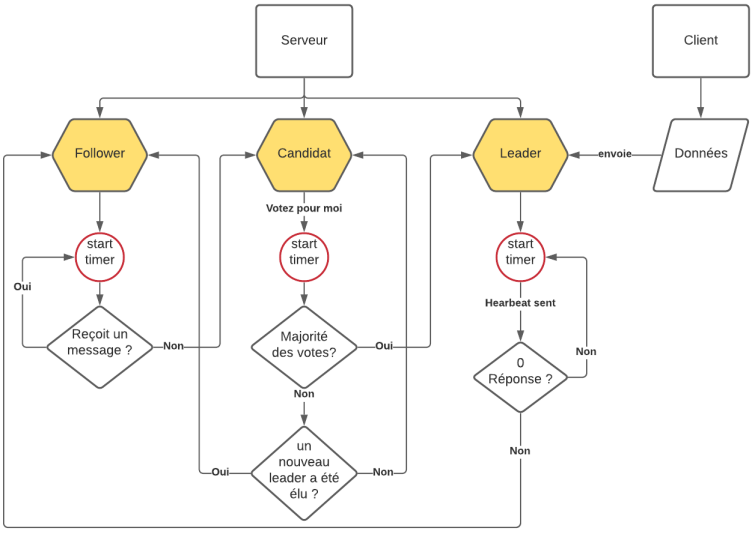


FIGURE 1 – Pipeline

### 5.1.1 Tests dans son ensemble sans injecter des fautes

Testons notre programme en faisant varier le nombre de serveurs, afin de voir qu'il n'y a pas de problème. Regardons nos logs.

```
Number of clients: 0
Number of servers: 3
--DEBUG FOLLOWER 0 TERM: 0
--DEBUG FOLLOWER 1 TERM: 0
--DEBUG FOLLOWER 2 TERM: 0
--DEBUG CANDIDAT 2 TERM: 1
--DEBUG FOLLOWER 0 TERM: 1 VOTE FOR 2
--DEBUG FOLLOWER 1 TERM: 1 VOTE FOR 2
--DEBUG CANDIDAT 1 TERM: 2
--DEBUG FOLLOWER 0 TERM: 2 VOTE FOR 1
--DEBUG FOLLOWER 1 TERM: 2
--DEBUG LEADER_LOOP 2 TERM: 1
```

Chaque serveur est bien initialisé à l'état Follower.

— Tout d'abord le serveur 2 se présente en tant que Candidate durant le term 1, les 2 Followers votent alors pour lui.

— Cependant le serveur Candidate 2 n'a pas encore fini de receptionner ses votes et il ne sait pas encore qu'il a obtenu la majorité.

— Durant ce laps de temps, le Follower 1 ne voyant pas la présence d'un Leader, il se présente alors à son tour en tant que Candidate au term 2.

— Le Follower 0 vote pour lui. Pendant ce temps, le serveur Candidate 2 a fini de comptabiliser ses votes, il envoie un message pour informer la réussite de son éléction.

— Le serveur Candidate 1 reçoit alors le message du Leader avant de pouvoir comptabiliser ses votes.

— Il repasse donc Follower et le Leader est bien le serveur 2.

On remarque que chaques serveurs n'ont pas forcément le même term. Ici le serveur 0 et 1 sont donc au term 2 tandis que le Leader est au term 1 puisqu'il n'est pas au courant de l'élection qui a eu entre-temps. Lors d'une future élection, le serveur 0 et 1, ne pourront voter que pour les élections se déroulant à un term plus élévé. Ils ignoreront les élections à term plus petit ou égale.
Testons maintenant pour 50 serveurs. On s'épargnera quelques lignes de logs.

```
Number of clients: 0
Number of servers: 50
—DEBUG FOLLOWER 25
—DEBUG FOLLOWER 17
—DEBUG FOLLOWER 21
[...]
—DEBUG FOLLOWER 19
—DEBUG CANDIDAT 17
—DEBUG CANDIDAT 25
—DEBUG CANDIDAT 25
—DEBUG LEADER_LOOP 17
—DEBUG FOLLOWER 25
```

Chaque serveur est bien initialisé, les serveurs 17 et 25 se présentent en même temps. Le serveur 25 n'ayant pas eu la majorité se représente de nouveau mais est arrêté par l'éléction du serveur 17 qui a eu la majorité, comme pour l'exemple précédent.

### 5.1.2 Tests de la commande CRASH

```
Number of clients: 0
Number of servers: 4
—DEBUG FOLLOWER 0 TERM: 0
—DEBUG FOLLOWER 2 TERM: 0
—DEBUG FOLLOWER 1 TERM: 0
—DEBUG FOLLOWER 3 TERM: 0
—DEBUG CANDIDAT 1 TERM: 1
—DEBUG CANDIDAT 3 TERM: 2
—DEBUG LEADER_LOOP 1 TERM: 1
—DEBUG FOLLOWER 3 TERM: 2
Enter a command:
```

```
CRASH 1
—DEBUG Receive CRASH
—DEBUG CANDIDAT 3 TERM: 3
—DEBUG CANDIDAT 0 TERM: 4
—DEBUG FOLLOWER 0 TERM: 4
—DEBUG LEADER_LOOP 3 TERM: 3
```

Tout se passe comme prévu. Le Leader meurt et une nouvelle éléction est effectuée.

Regardons maintenant le cas où la majorité des serveurs meurent.

```
Number of clients: 0
Number of servers: 5
—DEBUG FOLLOWER 3 TERM: 0
—DEBUG FOLLOWER 0 TERM: 0
—DEBUG FOLLOWER 2 TERM: 0
—DEBUG FOLLOWER 1 TERM: 0
—DEBUG FOLLOWER 4 TERM: 0
—DEBUG CANDIDAT 4 TERM: 1
—DEBUG CANDIDAT 2 TERM: 2
—DEBUG FOLLOWER 2 TERM: 2
—DEBUG LEADER_LOOP 4 TERM: 1
Enter a command:
CRASH 2
Enter a command:
CRASH 1
Enter a command:
CRASH 4
—DEBUG CANDIDAT 0 TERM: 3
—DEBUG CANDIDAT 0 TERM: 4
—DEBUG CANDIDAT 3 TERM: 5
—DEBUG CANDIDAT 0 TERM: 5
—DEBUG CANDIDAT 3 TERM: 6
[...]
```

Nous faisons donc mourir la majorité des serveurs, ici les serveurs 1 2 et 4 : le Leader meurt à la fin. Les 2 Followers restants se présentent alors Candidate mais aucun des 2 ne peut avoir la majorité des votes (2 votes max sur 5). C'est un combat infini.

### 5.1.3 Tests de la commande SPEED

Testons notre programme en faisant varier la vitesse d'exécution des serveurs. Regardons nos logs. On ralentit la vitesse du Leader et on augmente celle d'un Follower.

```
Number of clients: 0
Number of servers: 4
—DEBUG FOLLOWER 3 TIME_OUT: [300, 450]
—DEBUG FOLLOWER 2 TIME_OUT: [300, 450]
```

```
—DEBUG FOLLOWER 1 TIME_OUT: [300, 450]
—DEBUG FOLLOWER 0 TIME_OUT: [300, 450]
—DEBUG CANDIDAT 3 TIME_OUT: [300, 450]
—DEBUG CANDIDAT 1 TIME_OUT: [300, 450]
—DEBUG FOLLOWER 1 TIME_OUT: [300, 450]
—DEBUG LEADER_LOOP 3 TIME_OUT: [300, 450]
Enter a command:
SPEED LOW 3
—DEBUG Receive SPEED_LOW - new TIME_OUT [450, 600]
Enter a command:
SPEED HIGH 1
—DEBUG Receive SPEED_HIGH - new TIME_OUT [150, 300]
```

L'initialisation est correcte, chaque serveur a sa vitesse d'exécution en MEDIUM. Grâce à notre variable qui nous permet de considérer mort le Leader que si le Follower ne reçoit pas 5 heartbeats d'affilés, nous n'avons aucun problème lorsque nous effectuons ces 2 commandes. En revanche si l'on modifie la valeur de cette variable et qu'on la met égale à 2. Alors cela nous donnerait les logs suivant.

```
Number of clients: 0
Number of servers: 4
—DEBUG FOLLOWER 3 TERM: 0 TIME_OUT: [300, 450]
—DEBUG FOLLOWER 2 TERM: 0 TIME_OUT: [300, 450]
—DEBUG FOLLOWER 0 TERM: 0 TIME_OUT: [300, 450]
—DEBUG FOLLOWER 1 TERM: 0 TIME_OUT: [300, 450]
—DEBUG CANDIDAT 2 TERM: 1 TIME_OUT: [300, 450]
—DEBUG CANDIDAT 0 TERM: 2 TIME_OUT: [300, 450]
—DEBUG LEADER_LOOP 2 TERM: 1 TIME_OUT: [300, 450]
—DEBUG FOLLOWER 0 TERM: 2 TIME_OUT: [300, 450]
Enter a command:
SPEED LOW 2
—DEBUG Receive SPEED_LOW - new TIME_OUT [450, 600]
Enter a command:
SPEED HIGH 0
—DEBUG Receive SPEED_HIGH - new TIME_OUT [150, 300]
—DEBUG CANDIDAT 0 TERM: 3 TIME_OUT: [150, 300]
—DEBUG FOLLOWER 0 TERM: 3 TIME_OUT: [150, 300]
—DEBUG CANDIDAT 0 TERM: 4 TIME_OUT: [150, 300]
—DEBUG FOLLOWER 0 TERM: 4 TIME_OUT: [150, 300]
[...]
—DEBUG CANDIDAT 0 TERM: 32 TIME_OUT: [150, 300]
—DEBUG FOLLOWER 0 TERM: 32 TIME_OUT: [150, 300]
—DEBUG CANDIDAT 0 TERM: 33 TIME_OUT: [150, 300]
—DEBUG LEADER_LOOP 0 TERM: 33 TIME_OUT: [150, 300]
—DEBUG FOLLOWER 2 TERM: 1 TIME_OUT: [450, 600]
```

Le Leader 2 a sa vitesse d'exécution plus lente que le Follower 0 qui, lui, a une vitesse élevée. Les envois des heartbeats du Leader sont donc plus espacés. Le Follower 0 a encore moins de chance de réceptionner ces heartbeats dû à sa forte vitesse d'exécution.
Sachant qu'à partir de 2 heartbeats manqués, il considère le Leader comme mort, le Follower 0 se

présente donc plusieurs fois en tant que Candidate. Cependant, durant son éléction, il est possible qu'il reçoit un heartbeat du Leader actuel, dans ce cas il repasse Follower.
Si durant ce laps de temps où il se présente, il ne reçoit aucun heartbeat du Leader (term 33 - ligne 24) : il obtient la majorité des votes par les autres Followers, et devient alors Leader.
Le Leader subit alors un coup d'état et repasse Follower. On remarque que ce nouveau Follower reste au term 1 puisqu'il n'est pas au courant des élections passées.

Nous nous sommes pas intéressés à remettre à jour le term puisqu'en modifiant notre variable avec un nombre plus élevé, nous n'arriverons jamais dans ce cas. En effet, vu que nous n'avons que 3 modes de vitesse que nous avons choisi au préalable, cela ne se produira pas. Cependant, en réalité, si on pouvait réduire davantage la vitesse d'exécution du Leader, il faudrait mettre à jour le term de l'ancien Leader devenu Follower, afin qu'il puisse voter pour le bon term et éviter tout problème lors d'une future élection.
De plus, grâce à l'augmentation du nombre de heartbeats réceptionnés (maintenant on attend 5 heartbeats avant de considérer comme mort le Leader), on réduit le nombre de messages qui transitent dû aux éléctions.

## 5.2 Testsuite sur les clients

Afin de vérifier que les commandes envoyées par les clients soient bien tous receptionnés et inscrits et que la réplication de logs serveurs soit bien faite, une testsuite compare le contenu des fichiers de logs et des clients.

Pour lancer la testsuite :

```
make check
```

Pour lancer la testsuite avec les informations de debug :

```
make check_debug
```

Un test de la testuite est aussi présent et peut être lancé avec :

```
make check_testsuite
```

# 6 Mesure de performances

Pour cette partie, on va chercher à étudier la performance de notre algorithme pour la partie concernant les éléctions.
Pour ce faire, chaques serveurs va crée un fichier répertoriant ces performances. On va s'intéresser au term, aux envoies et aux récéptions des messages. Ensuite, on regroupe les fichiers de performances des logs en resortant la moyenne, le minimum et le maximum de chaques catégories.

Regardons les performances pour 0 client et 50 serveurs.

| | | RECV | SEND | RECV | SEND | RECV | SEND | RECV | SEND | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| STATUS | TERM | IWANTTOBECANDIDATE | IWANTTOBECANDIDATE | VOTE | VOTE | IMTHELEADER | IMTHELEADER | HEARTBEAT | HEARTBEAT | TOTAL RECV | TOTAL SEND | TOTAL MSG |
| LEADER | 2 | 0 | 2 | 35 | 0 | 0 | 1 | 848 | 18 | 848 | 19 | 867 |

FIGURE 2 – Analyse du fichier performance du Leader

On remarque que le Leader a du faire 2 éléctions pour avoir la majorité. On observe que l'éléction du Leader se fait avec peu de message. Il reçoit pratiquement autant de heartbeats qu'il en envoie.

| | | RECV | SEND | RECV | SEND | RECV | SEND | RECV | SEND | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TERM | IWANTTOBECANDIDATE | IWANTTOBECANDIDATE | VOTE | VOTE | IMTHELEADER | IMTHELEADER | HEARTBEAT | HEARTBEAT | TOTAL RECV | TOTAL SEND | TOTAL MSG |
| mean | 2.0 | 7.47 | 0.122 | 0.22 | 1.88 | 1.0 | 0.0 | 17.33 | 17.33 | 25.80 | 19.20 | 45.0 |
| min | 2.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 17.0 | 17.0 | 19.0 | 17.0 | 36.0 |
| max | 2.0 | 8.0 | 2.0 | 7.0 | 2.0 | 1.0 | 0.0 | 18.0 | 18.0 | 27.0 | 20.0 | 47.0 |

FIGURE 3 – Analyse et regroupement des fichiers performances des Followers

On remarque qu'il y a eu d'autres candidats vu que le term est de 2. On peut également le voir grâce à la varibale *SEND IWANTTOBECANDIDATE* qui a été envoyé au plus 2 fois par un Follower. Il y a donc au plus 2 Followers qui se sont également présentés, ou seulement 1 qui s'est présenté durant les 2 terms. De plus, cela se visualise aussi avec la variable *RECV VOTE* qui atteint son maximum avec 7 votes. on observe également que les candidats ne votent pas puisque le minimum de *SEND VOTE* est de 0.
Chaque Follower a bien répondu à tous les heartbeats du Leader puisque *RECV HEARBEAT* est égale à *SEND HEARTBEAT*. Le petit écart entre le *SEND HEARTBEAT* du Leader qui est de 18 est surement dû à l'arrêt du programme.
On remarque également que le nombre total de messages échangés maximum pour un Follower est de 47. Cependant, il est seulement de 19 si on ne prend pas en compte les échanges d'heartbeats.

# 7 Conclusion

Dans notre programme, l'algorithme RAFT ne gère pas la mort de plus de la moitié des serveurs. En effet, si l'on se retrouve dans ce cas, la moitié des serveurs étant morts, nous n'aurions plus de majorité possible. Pour gérer ce cas, l'une des solutions possibles serait d'ajouter une variable globale servant à calculer le nombre de serveur mort et ainsi connaître le nombre de server vivant à l'instant t. Cependant, nous ne chercherons pas à implémenter cette solution car cela ne serait pas réaliste puisque les processus ne sont pas censés savoir quand un de leur congénère meurt.

Nous avons également rencontré des problèmes sur l'utilisations des variables globales dans un système distribué lors de leur modification. En effet, si l'on souhaite modifier la valeur de cette variable, la nouvelle valeur n'est pas mise à jour pour tous les processus. Chaque variable est propre à chacun et donc le seul moyen de propager celle-ci est d'envoyer des messages.

Une optimisation envisageable de notre programme est d'envoyer nos messages à travers les heartbeats. En effet, nous distinguons chaque message alors qu'il pourrait être envoyé à travers le heartbeat puisque le heartbeat sert uniquement à savoir si un serveur est en vie et qu'il possède le bon nombre de logs.

Pour plus de réalisme, nous pourrions vider la RAM du processus lorsqu'il reçoit la commande CRASH pour mieux simuler un crash. Voir même effacer le disque afin d'également simuler un disk failure. Cela pourrait être une possible amélioration.

# 8 Bibliographie


— https://mpi4py.readthedocs.io/en/stable/tutorial.html#running-python-scripts-with-mpi

— https://raft.github.io

— https://lipn.univ-paris13.fr/~coti/doc/tutopympi.pdf

— http://thesecretlivesofdata.com/raft/

— https://research.computing.yale.edu/sites/default/files/files/mpi4py.pdf

— https://arxiv.org/pdf/1808.01081.pdf