Génération de spécification formelle pour l’algorithme Egalitarian Paxos

PFE
Auteur : Alexandre Siret
Tuteur : Pierre Sutra

---

1. Introduction

Les systèmes distribués modernes reposent très largement sur des algorithmes de consensus tels que Paxos afin de garantir la cohérence des données malgré la présence d’erreurs ou de délais réseaux.

Ce projet à pour objectif de rajouter une couche de certitude sur la correction d’un algorithme de consensus particulièrement complexe : **Egalitarian Paxos (EPaxos).** Le projet part de l’article Making Democracy Work: Fixing and Simplifying Egalitarian Paxos, écrit par : Fedor Ryabinin; Alexey Gotsman; et Pierre Sutra. Cet article à pour objet de corriger un bug trouvé dans Egalitarian Paxos, ainsi que de simplifier l’algorithme, ce qui justifie la volonté de rajouter une couche de certitude sur le fait que l’algorithme est maintenant correct.

Le projet consiste donc à produire une **spécification formelle de l’algorithme Egalitarian Paxos (EPaxos) en TLA+**, puis de vérifier certaines de ses propriétés fondamentales à l’aide d’un **model checker.**

---

2. Objectifs et phases du projet

2.1 EPaxos

Un premier objectif du projet consistait à comprendre l’algorithme dont le projet est question, en particulier ce que EPaxos peut apporter de plus, et ces défauts, par rapport à un algorithme de consensus plus classique.

En quelque mots : Egalitarian Paxos (EPaxos) est une variante de Paxos visant à améliorer les performances en supprimant l’élection d’un leader (d'où l'appelation “Egalitarian”) et en exploitant le fait que certaines commandes sont **commutatives**.

L’algorithme repose sur la construction (et le partage) d’un graphe de dépendances entre les commandes, puis l'exécution des commandes se fait localement en fonction du graphe de dépendances obtenu.

2.2 TLA+

Un deuxième objectif du projet était de m'acclimater au langage TLA+ un langage de spécification formelle qui m'était étranger en commençant le projet.

Cet objectif consistait à lire le livre *Specifying Systems* de Leslie Lamport pour comprendre des concepts clés de TLA+.

2.3 Développement de la spécification

Le troisième objectif était d'implémenter la spécification formelle de l'algorithme. Un objectif secondaire pendant cette phase était de voir si des outils d'intelligence artificielle (copilot/chatgpt) peuvent être utiles pour générer de la spécification formelle.

2.4 Utilisation de la spécification

Le quatrième objectif est d'obtenir des résultats à partir de la spécification en utilisant le model checker pour tester certaines configurations.

2.5 Étapes du projet

Les 3 phases principales du projet sont les suivantes :

1. Phase d'apprentissage :
    a. Comprendre le fonctionnement de l'algorithme EPaxos.
    b. Apprendre les bases de TLA+.
2. Phase d'implémentation :
    a. Produire une spécification formelle de Epaxos.
    b. Évaluer l'apport des outils d'IA dans ce type de travail.
3. Phase de test : Utiliser le model checker pour obtenir la confirmation que l'algorithme est correct dans certaines configurations.

---

3. Phase d'apprentissage

3.1 Compréhension de Epaxos.

Dans un premier temps Il a fallu mieux comprendre EPaxos.

3.1.a Absence de leader

Contrairement à Paxos classique, EPaxos ne nécessite pas l'élection d'un leader. Chaque processus peut initier des propositions.

3.1.b Ordre partiel et commandes commutatives

Paxos impose un **ordre total** sur toutes les commandes. EPaxos, au contraire, exploite le fait que certaines commandes ne sont pas en conflit et peuvent donc être exécutées dans un **ordre partiel**.

Cela permet d'emprunter un *fast path* lorsque les commandes sont indépendantes. C'est une amélioration en performance en exploitant cette information supplémentaire.

3.1.c Complexité accrue

Ces avantages ont un coût : l'algorithme est plus complexe. La gestion des dépendances, des confirmations partielles et des phases de récupération rend l'implémentation délicate, ce qui justifie une volonté de faire de la vérification formelle.

3.1.d Décomposition du protocol

Le protocol à se décompose bien en 3 parties, par la suite je vais appeler ces 3 parties :

- Commit Protocol
- Recovery Protocol
- Execution Protocol

Le commit protocol crée le graphe de dépendances, le recovery protocol vient s'entremêler avec le commit protocol et permet de corriger les erreurs qui peuvent survenir. L'exécution protocol utilise le graphe de dépendance pour exécuter localement les commandes, il n'interagit pas avec les 2 autres parties.

3.1.e Invariants

Un dernier point important à soulever avant de passer à la suite est les invariants. C'est les propriétés que l'on souhaite vérifier avec le model checker. Les invariants des parties commit et recovery sont les suivants :

- **Agreement** : Si un id de commande est comité dans deux processus différents, alors c'est la même commande
- **Visibilité** : Si deux id de commandes conflictuelles id et id' sont comité, alors id est dans les dépendances de id', ou id' est dans les dépendances de id.
- **Liveness :** Une commande soumise sera toujours comité au bout d'un certain temps.

3.2 Apprentissage de TLA+

Pour cette partie, j'ai lu régulièrement le livre *specifying systems* de Leslie Lamport. Plus spécifiquement, j'ai lu la Part I : Getting Started. Cela à été utile, d'une part je me suis habitué à la syntaxe de TLA+ et la structure générale d'un fichiers de spécification formelle, d'autre part les exemples présents dans le livre m'ont servi de template pour mes premières tentatives en TLA+. J'ai aussi lu certaines sections de Part II : More advanced topics.

4. Phase d'implémentation

Cette phase d'implémentation à été faite étape par étape. Les trois parties de EPaxos ont été implémentée individuellement, dans l'ordre suivant de difficulté croissante :

- Execution Protocol
- Commit Protocol
- Recovery Protocol

4.1 Spécification de SMR

Mais avant même de commencer par l'exécution protocol, j'ai d'abord fait, pour m'entrainer, une implémentation de **SMR (State machine replication)** dans le cas général, c'est à dire que j'ai pris en compte le fait que les commandes peuvent commuter, comme Epaxos le fait. L'ordre imposé est donc partiel et pas total.

Une question naturelle à avoir est : comment j'ai pu faire pour implémenter une solution au problème SMR général qui à besoin d'un protocol aussi complexe que EPaxos pour le résoudre? La réponse est simple, j'ai triché. J'ai utilisé des variables globales accessibles par tout le monde.

Avec une première spécification, complétée avec les invariants correspondants, j'ai ensuite réellement commencé l'implémentation de EPaxos en TLA+.

4.2 Execution Protocol

Premièrement, le protocole d'exécution contient aussi de la triche. Pour soumettre les commandes et construire et partager les dépendances qu'on aurait eu à la fin du protocole de commit de EPaxos, j'ai utilisé des variables globales. À partir de ces objets, j'ai implémenté le protocole d'exécution.

```
1 while true
2     let G ⊆ dep be the largest subgraph such that ∀id ∈ G. phase[id] = COMMITTED ∧ dep[id] ⊆ G
3     for C ∈ SCC(G) in topological order do
4         for id ∈ C in the order of command identifiers do
5             if id ∉ executed ∧ cmd[id] ≠ Nop then
6                 execute(cmd[id])
7                 executed ← executed ∪ {id}
```

La difficulté de cette implémentation était la traduction du pseudo code écrit en langage naturel de haut niveau en TLA+. Le concept difficile à implémenter était :

- for C \in SCC(G) in topological order

Il fallait d'abord construire l'objet SCC(G) (les composantes fortements connexes de G)

On définit d'abord une fonction récursive avec un ensemble des nœuds visités pour gérer les cycles, pour obtenir l'ensemble des éléments atteignables à partir d'un élément c de G.

```
RECURSIVE ReachableRec(_, _, _)
ReachableRec(G, c, visited) ==
    IF c \in visited THEN {}
    ELSE
        LET deps == {d \in dep[c] : d \in G}
            newVisited == visited \cup {c}
        IN {c} \cup UNION {ReachableRec(G, d, newVisited) : d \in deps}

Reachable(G, c) == ReachableRec(G, c, {})
```

Ensuite, on définit la composante fortement connexe de c à partir de Reachable, en prenant tous les éléments de G qui peuvent atteindre c et qui sont atteignables à partir de c.

```
MutuallyReachable(G, c) == {c} \cup { d \in G : c \in Reachable(G, d) /\ d \in Reachable(G, c) }
```

Finalement, on peut définir SCC(G).

```
SCCs(G) == {MutuallyReachable(G, c) : c \in G}
```

Ensuite, on veut ordonner cet ensemble dans l'ordre topologique.
Ma solution à été de construire l'ensemble de toutes les permutations de SCC(G), puis de choisir celle qui est ordonnée topologiquement.

```
RECURSIVE Permutations2(_)
Permutations2(S) ==
    IF S = {} THEN
        { <<>> }
    ELSE
        UNION { { Append(seq, x) : seq \in Permutations2(S \ {x}) } : x \in S }

TopoOrder(SCCSet) ==
    CHOOSE seq \in Permutations2(SCCSet) :
        \A i, j \in 1..Len(seq) :
            i < j => \A c \in seq[j], d \in seq[i] : d \notin dep[c]
```

Ce n'est pas une solution élégante, je pense que c'est en partie dû à mon manque de confort avec TLA+, et en partie dû au fait qu'il n'est pas facile de représenter dans sa tête une notion comme l'odre topologique des composantes fortement connexes d'un graphe.

Quoi qu'il en est, j'obtiens une spécification fonctionnelle du protocole d'exécution.

4.3 Commit Protocol

Pour le commit protocol, la difficulté et tout autre, les lignes du pseudo code sont plus simples à retranscrire individuellement, mais il faut trouver une manière de simuler la communication entre les processus du protocole.

Il y a deux aspects à simuler, le premier est qu'il faut un état local pour chaque processus.

Pour se faire, chaque variable devient une matrice de valeurs qui map à chaque couple processus et identifiant de commande la valeur de la variable. Par exemple, une operation du pseudo code comme celle-ci :

$$\text{cmd}[id] \leftarrow c$$

devient en TLA+ : cmd[p][id] = c

```
/\ cmd' = [cmd EXCEPT ![p][id] = c]
```

Ensuite, en m'interdisant d'accéder à ou modifier par exemple cmd[q][id] quand je suis le processus p, j'ai une simulation artificielle de processus qui ont chacun leur état local.

Le deuxième aspect a simuler est la communication entre les processus.

L'algorithme repose sur des événements, une opération s'éxécute quand un processus reçoit un message d'un autre processus:

$$\textbf{when received } \texttt{PreAccept}(id, c, D) \textbf{ from } q$$

Pour modéliser cet aspect j'utilise une variable *msgs*, accessible par tout le monde, quand un message est envoyé, je le rajoute à msgs, en ajoutant bien le sender et le destinataire.

Pour modéliser la réception d'un message, je permet l'exécution d'une opération seulement quand le message correspondant est dans msgs, puis, après l'opération, je consume le message en le supprimant de msgs.

4.4 Recovery Protocol

Cette partie utilise le même modèle que le commit protocole. Un point à noter est que pour une des opérations, j'ai dû la diviser en 3 sous opérations.

```
50 when received RecoverOK(b, id, abal_q, c_q, dep_q, initDep_q, phase_q) from all q ∈ Q
51     pre: bal[id] = b ∧ |Q| ≥ n − f
52     let b_max = max{abal_q | q ∈ Q}
53     let U = {q ∈ Q | abal_q = b_max}
54     if ∃q ∈ U. phase_q = COMMITTED then send Commit(b, id, c_q, dep_q) to all
55     else if ∃q ∈ U. phase_q = ACCEPTED then send Accept(b, id, c_q, dep_q) to all
56     else if initCoord(id) ∈ Q then send Accept(b, id, Nop, ∅) to all
57     else if ∃R ⊆ Q. |R| ≥ |Q| − e ∧ ∀q ∈ R. (phase_q = PREACCEPTED ∧ dep_q = initDep_q) then
58         let R_max be the largest set R that satisfies the condition at line 57
59         let (c, D) = (c_q, dep_q) for any q ∈ R
60         send Validate(b, id, c, D) to all processes in Q
61         wait until received ValidateOK(b, id, I_q) from all q ∈ Q
62         let I = ⋃_{q∈Q} I_q
63         if I = ∅ then
64             send Accept(b, id, c, D) to all
65         else if (∃(id', COMMITTED) ∈ I) ∨ (|R_max| = |Q| − e ∧ ∃(id', _) ∈ I. initCoord(id') ∉ Q) then
66             send Accept(b, id, Nop, ∅) to all
67         else
68             send Waiting(id, |R_max|) to all
69             wait until
70                 case ∃(id', _) ∈ I. phase[id'] = COMMITTED ∧ (cmd[id'] ≠ Nop ∧ id ∉ dep[id']) do
71                     send Accept(b, id, Nop, ∅) to all
72                 case ∀(id', _) ∈ I. phase[id'] = COMMITTED ∧ (cmd[id'] = Nop ∨ id ∈ dep[id']) do
73                     send Accept(b, id, c, D) to all
74                 case ∃(id', _) ∈ I. (p received Waiting(id', k')) ∧ k' > n − f − e do
75                     send Accept(b, id, Nop, ∅) to all
76                 case p received RecoverOK(b, id, _, cmd, dep, _, phase) from q ∉ Q with
                    phase = COMMITTED ∨ phase = ACCEPTED ∨ q = initCoord(id) do
77                     if phase = COMMITTED then send Commit(b, id, cmd, dep) to all
78                     else if phase = ACCEPTED then send Accept(b, id, cmd, dep) to all
79                     else send Accept(b, id, Nop, ∅) to all
80     else send Accept(b, id, Nop, ∅) to all
```

Les 2 moments où il y a écrit wait until ne peuvent pas être modélisés tels quels dans la spécification. En effet, malgré l'illusion que je travaille avec plusieurs processus, en réalité, j'ai un seul processus qui exécute séquentiellement des opérations. Je l'ai donc divisé en 3 opérations séparées.

D'autre part, le protocole de recovery n'a pas de sens si il est seul, il vient se greffer au protocole de commit.

4.5 Commit + Recovery Protocol

Après une tentative de faire les choses proprement et d'utiliser des modules, j'ai choisi de copier coller les codes de commit et de recovery dans un seul fichier de spécification. Comme j'avais initialement créé les spécifications comme des spécifications à part entière, c'était difficile d'essayer de les intégrer en tant que modules dans un troisième fichier de spécification qui réunit les deux.

J'obtient finalement la spécification commit + recovery avec les invariants suivants:

```
Agreement ==
  \A id \in Id :
    \A p, q \in Proc :
      /\ phase[p][id] = "committed"
      /\ phase[q][id] = "committed"
      => /\ dep[p][id] = dep[q][id]
         /\ cmd[p][id] = cmd[q][id]

Visibility ==
  \A id, id2 \in Id : \E p, q \in Proc :
    /\ id # id2
    /\ phase[p][id] = "committed"
    /\ phase[q][id2] = "committed"
    /\ Conflicts(cmd[p][id], cmd[q][id2])
    => \/ id \in dep[q][id2]
       \/ id2 \in dep[p][id]
```

Et la propriété de Liveness :

```
Liveness ==
    \A id \in Id :
      id \in submitted
      => \E p \in Proc :
           phase[p][id] = "committed"
```

---

5. Phase de test

5.1 Model Checking

Un *model checker* explore exhaustivement l'espace des états possibles d'un système à partir d'une configuration initiale, en appliquant toutes les transitions autorisées.

Chaque état est défini par la valeur de l'ensemble des variables du modèle.

Le model checker vérifie que les invariants sont satisfaits dans chacun des états atteignables. Cette exploration est exhaustive mais souffre d'une explosion rapide du nombre d'états quand le nombre de processus ou de commandes évolue.

5.2 Expériences locales

Les premières expériences (jusqu'à 2 processus et 2 commandes) ont pu être menées localement. Cependant, l'ajout d'un troisième processus rend l'exploration de l'espace d'états trop coûteuse pour ma machine.

5.3 Utilisation de Google Cloud Platform

Pour dépasser ces limites, une machine plus puissante a été déployée sur GCP. Plusieurs obstacles ont été rencontrés :

- Quotas de CPU par région. (cela m'a empêcher de prendre une machine plus performante)
- Limites de stockage par défaut (10 Go).

Initialement, en lançant le model checker avec 2 processus et 2 commandes sur la machine GCP, le model checker a tourné 10 fois plus vite.

Cependant, au moment de le lancer avec 3 processus, j'ai vite rencontré l'erreur : Error when writing the disk, no space left on device. Effectivement, le model checker garde une trace de chaque état distinct qu'il rencontre et le stocke pour éviter les cycles dans son parcours. Avec 10GB de stockage seulement sur toute la machine, ce n'est pas surprenant que le disque soit complètement rempli

Après extension à un disque de 500GB (encore une fois, je suis limité par le quota par région de GCP) , le model checker a pu explorer jusqu'à **1,5 milliard d'états distincts en environ 2h20**, remplissant de nouveau entièrement le disque.

Ainsi,

---

6. Apport et limites de l'intelligence artificielle

L'IA a été utilisée comme aide à la rédaction et à la génération de premiers brouillons de spécifications TLA+.

6.1 Points positifs

- Gain de temps pour démarrer.
- Aide à la structuration initiale.

6.2 Limites

- L'IA a du mal avec un langage de niche comme TLA+, elle fait beaucoup d'erreurs et invente beaucoup de choses qui n'existent pas en TLA+.
- En plus de ça, ce n'est pas simple de comprendre ou déboguer les erreurs. Des erreurs très simples à corriger dans du code généré par IA qui à l'air correct peuvent faire perdre beaucoup de temps.

L’IA reste un outil utile, mais je pense qu’il est facile de se retrouver à perdre du temps en utilisant l’IA sur un sujet aussi niche. Je pense qu’il est judicieux de l’utiliser plutôt moins souvent que plus souvent, au risque de perdre du temps et de devenir frustré.

Il faut partir du principe que tout ce qui est généré aura des erreurs (franchement de mon expérience, c’est le cas). De plus, il est inutile de s’acharner à essayer de faire régler un problème à l’IA où elle a déjà échoué une ou deux fois, même en comprenant exactement le problème en question et lui expliquant en détails, sa solution est presque toujours d’ajouter de la complexité à la chose qu’elle a déjà proposée, plutôt que de prendre une autre direction.

L’IA m’a été plus utile vers la première moitié du projet, dans la seconde partie, la plus compliqué, j’ai l’impression que essayer d’utiliser l’IA m’a fait perdre du temps.

En conclusion de cette section, je pense que l’IA peut servir pour générer de la spécification TLA+ tant que le projet reste plutôt simple.

---

7. Conclusion

Ce projet m’a permis d’apprendre des notions de model checking et de spécification formelle en TLA+. La rédaction de la spécification en TLA+ m’a apporté une meilleure compréhension des mécanismes de l’algorithme, ainsi qu’une compréhension de certaines de ses propriétés fondamentales.

Le projet a permis de vérifier la correction de l’algorithme dans des configurations de petites tailles.

Néanmoins, les limites pratiques observées montrent que la vérification exhaustive reste coûteuse voire impossible dans des configurations plus grandes, même avec des ressources importantes. Malgré cela, ce travail constitue une base solide pour de futures extensions.