Tasdawit n'Bgayet
Université de Béjaïa
جامعة بجاية

Université A/Mira Béjaia

Faculté des Sciences Exactes
Master 2 Recherche

Mémoire de fin de cycle en vue d'obtention du diplôme Master en Informatique spécialité "Réseaux et Systèmes Distribués"

présenté et soutenu par

AMGHAR ABDESLAM

le 01 Juillet 2017

Le problème du consensus dans un système distribué dynamique ou multi-agents dynamique

Encadreur du Mémoire : **SADI MUSTAPHA**

Jurys

AISSANI Sofiane,	Enseignant	Président
TAHAKOURT Zineb,	Enseignante	Examinatrice
HAMZA Lamia,	Enseignante	Examinatrice

MÉMOIRE

Remerciements

On tient en premier lieu à remercier Le Premier, celui qui à créer tout et qui nous a laissé pour nous en ce bas monde des puzzles dont nous nous faisons qu'assemblé les briques pour un meilleur lendemain, notre Dieu tout puissant et miséricordieux qui nous a donné la force et la patience d'accomplir ce modeste travail.

Avec ma grande gratitude, je dédie ce modeste travail aux personnes les plus importantes pour moi, Mes parents et ma grand mère et mon frère bien-aimés, et ou reste de toute ma famille.

Nous tenons à remercier toute les personnes qui ont contribué, d'une manière ou d'une autre à ce que ce modeste travail puisse aboutir.

Évidement notre encadreur, Mr SADI Mustapha qui a suivi et encadré notre travail de mémoire, pour ces précieux conseils et son aide.

Nous exprimons notre gratitude aux membres de jury d'avoir consacré le temps qu'il fallait pour lire et corriger ce mémoire et de l'honneur qu'ils nous ont fait pour en participer à notre soutenance.

Table des matières

Table des matières i

Liste des figures iii

Introduction générale 1

1 Phare sur les principales définitions 3
- 1) Introduction . . . 4
- 2) Système Distribué Dynamique . . . 4
- 3) Système Multi-Agents . . . 5
- 4) Le consensus . . . 5
- 5) Conclusion . . . 7

2 Méthodes expérimentales 8
- 1) Introduction . . . 10
- 2) Problème et incertitude Algorithmique . . . 10
- 3) Modèles des Graphes . . . 16
- 4) État de l'art (Consensus) . . . 19
- 5) Conclusion . . . 30

3 Proposition d'un consensus pour un système dynamique 31
- 1) Problématique . . . 32
- 2) Objectifs . . . 32
- 3) Solution . . . 33
- 4) Cas d'étude . . . 59

4 Simulation de la solution apportée 60
- 1) Présentation de la simulation . . . 61
- 2) Vérification des propriétés . . . 68
- 3) Vérification des Objectifs . . . 73
- 4) Synthese . . . 74
- 5) Conclusion . . . 75

Conclusion générale et perspectives 76

Références **78**

Annexes **81**

1) Quorum slices 81

2) OpenGL 81

3) Intel Graphics Performance Analyzers 82

Liste des figures

2.1 Algorithmes fonctionnant par tours, synchronisés sur tous les processus. . . 13
2.2 Impossible de distinguer un processus lent d'un processus arrêté 13
2.3 Illustration d'un arbre dynamique par une suite de graphes qui représente l'évolution de l'arbre dynamique dans le temps 17
2.4 Illustration de la dynamique du modèle de graphe évolutif à travers une suite de graphes statiques 17
2.5 Illustration de graphe variant dans le temps où l'on ignore les latences ζ et ϕ 18

3.1 Cas d'adhésion d'un noeud solitaire à un groupe. 38
3.2 Les Quorums Slices. 40
3.3 Organigramme des changement d'état d'un noeud. 53
3.4 Organigramme des réactions d'un noeud aux différent message reçu. . . . 54
3.5 Organigramme des réaction d'un leader aux différent message reçu. 55
3.6 Diagramme de séquence de démarrage et de l'exécution d'un consensus. . 56
3.7 Diagramme de séquence des échanges des bulletins de vote entre trois noeuds et leurs leader (E : engager, A : avorter). 57

4.1 Fenêtre de manipulation de script Lua et d'affichage de Log. 62
4.2 Scène principale de la simulation. 62
4.3 Référendum 63
4.4 HistoriqueEchangeMessages.log 63
4.5 États des noeuds durant un référendum 64
4.6 Simulation.log 65
4.7 Durées des liaisons de communications entre les différents noeuds 65
4.8 Diagramme de class de l'interface. 66
4.9 Diagramme de class de la scène. 67
4.10 Exemple d'une boucle de noeuds dont un message peut tourner indéfiniment. 68
4.11 Graphe performance : Limite noeuds fils = 5, Délai d'attente pulsation leader = 2 69
4.12 Graphe performance : Limite noeuds fils = 10, Délai d'attente pulsation leader = 2 70

4.13 Graphe performance : Limite noeuds fils = 15, Délai d'attente pulsation leader = 2 70
4.14 Graphe performance : Limite noeuds fils = 5, Délai d'attente pulsation leader = 5 70
4.15 Graphe performance : Limite noeuds fils = 10, Délai d'attente pulsation leader = 5 71
4.16 Graphe performance : Limite noeuds fils = 15, Délai d'attente pulsation leader = 5 71
4.17 Graphe performance : Limite noeuds fils = 5, Délai d'attente pulsation leader = 8 71
4.18 Graphe performance : Limite noeuds fils = 10, Délai d'attente pulsation leader = 8 72
4.19 Graphe performance : Limite noeuds fils = 15, Délai d'attente pulsation leader = 8 72
20 Analyseurs de performances graphiques Intel 82

Introduction générale

Nous vivons aujourd'hui dans un monde où la majorité des systèmes sont distribuées. Le consensus est l'un des concepts de ces systèmes.

En permettant à un bloc de processus de s'entendre sur une des valeurs qu'ils proposent, le consensus peut être utilisé pour résoudre de nombreux problèmes d'accord dans les systèmes répartis en présence de fautes. Le consensus est un problème essentiel qui a été amplement étudiée et particulièrement dans les systèmes dont la topologie est connue (le nombre et la liste des participants est préalablement connue).

Cependant quand on passe à un système qui présente des variations au cours de temps au niveau de sa topologie ainsi que d'un nombre de noeud inconnue et dont la position l'est aussi, les hypothèses classiques d'une connaissance préalable de la composition du système, ou même seulement du nombre de participants, ne sont plus possibles.

Le problème du consensus peut être rapidement définit de la façon suivante : ńSoit un ensemble de processus, chacun proposant une valeur, trouver un protocole réparti afin de mettre tous les processus d'accord sur une des valeurs proposées initialementż.

L'intérêt d'un consensus vient du fait que, si nous étions capables de le résoudre, nous serions alors capables d'implémenter des services tolérants aux fautes. En pratique, pour mettre en place ce genre de système, il suffirait de répartir le calcul sur plusieurs ordinateurs.

La principale difficulté d'une telle approche, est qu'il est nécessaire de s'assurer de la cohérence au niveau du service. Pour cela tous les processeurs doivent effectuer les différents calculs dans le même ordre. Il a été démontré qu'un tel système distribué était implémentable si et seulement si le problème du consensus possède une solution. Malheureusement, dans le cas général des systèmes asynchrones, cela a été prouvé impossible si, ne serait-ce qu'un seul processus tombe en panne [6].

Le consensus est aussi l'un des défis importants dans les systèmes multi-agents, dans

lequel les agents doivent synchroniser certaines variables de contrôles, souvent en utilisant uniquement un graphique de communication incomplet et variable dans le temps.

De nombreuses techniques et protocoles ont été mises en oeuvre pour arriver à un consensus en un temps fini, tout en étant tolérant aux fautes, cela dit les solutions aux problèmes de disponibilité dans les environnements dynamiques demeurent encore au stade de l'aurore.

C'est dans ce contexte que nous souhaitons apporter notre modeste contribution concernant la thématique de consensus dynamique, dans une topologie où les noeuds sont mobiles et peuvent présenter une certaine indépendance.

Ce mémoire est organisé en quatre chapitres : Le premier chapitre est un rassemblement des principales définitions concernant les systèmes distribués, multi-agent et de consensus en lui-même.

Le deuxième chapitre décrit en premier lieu les problèmes rencontrés (synchronisme et asynchronisme, types de défaillances d'un processus) et les propriétés qui doivent être respecté lors de la conception d'un algorithme de consensus répartis, nous présenterons aussi les graphes les plus utilisés pour l'étude de ce genre de problème dans les systèmes statiques et ou dynamiques et nous finirons sur un état de l'art des consensus les plus connus et utilisés actuellement.

Nous présenterons une solution originale dans le chapitre trois, basée sur quatre protocoles décrit dans l'état de l'art, pour un algorithme de coordination des noeuds et un protocole de consensus dans un système qui est dynamique et variant dans le temps.

Dans le chapitre quatre, nous terminerons avec une présentation d'une simulation réalisée en C++, utilisant la bibliothèque graphique OpenGL et le langage de script LUA, dans l'environnement de développement Visual Studio. Nous traduirons aussi les mesures de performance récoltées, tout on vérifions les propriétés et les objectifs atteints.

Pour conclure cette étude, nous présenterons une conclusion avec des questions de perspectives.

Chapitre 1

Phare sur les principales définitions

« *La preuve de la valeur d'un système informatique est son existence même* »

Alan Jay Perlis

Sommaire

1) Introduction 4

2) Système Distribué Dynamique 4

2.1) Définition 4

2.2) Caractéristiques et Avantages d'un SDD 4

3) Système Multi-Agents 5

3.1) Définition 5

3.2) Domaines d'application et Caractéristiques 5

4) Le consensus 5

5) Conclusion 7

1) Introduction

Définir et comprendre le consensus ainsi que les systèmes dans lesquels ils seront implémenté et évolueront, sera la première étape de ce mémoire.

2) Système Distribué Dynamique

2.1) Définition

Un système distribué est un système ou les éléments de calcul sont autonomes dotés de capacité de communication et coordonnant leurs actions par transferts de messages. Les progrès rapide réalisés dans le domaine des objets connectés et dans les réseaux sans fils amènent graduellement à l'intégration de la dynamicité et à la possibilité que les composants et les communications à l'intérieur du système peuvent varier ou cours du temps [1, 3].

2.2) Caractéristiques et Avantages d'un SDD

Un système distribué peut être asynchrone, (la où les processus réagissent à l'événement réception plutôt que d'attendre un événement, d'où la communication est non bloquantes). Le problème est qu'on ne peut déduire l'ordre de réception des messages. D'autre part, les liens de communication sont assujettis à des variations dans la vitesse de transfert d'informations.
Un système distribue répartit les données et les taches sur plusieurs unités de calcul qui collaborent afin de résoudre un problème donné [1].

Ce procédé comporte certains avantages [1, 3] :

- **Le partage de ressources :** Le simple fait de distribuer les traitements sur les ordinateurs d'un réseau augmente les ressources disponibles.
- **Gain et amélioration des capacités :** Grande capacité de calcul et de mémoire dus à la répartition des ressources sur le système distribué.
- **Fiabilité :** En répartissant les charges entre les différents composants logiques du système, celui-ci va continuer à proposer tout ou partiellement ses services même en présence de fautes temporaires ou permanentes, qui peuvent être dus à certains des composants du système.
- **Évolution du système :** Les systèmes distribuées et les caractéristiques de son réseau peuvent évoluer avec des ajouts de nouveaux éléments, comme des machines avec différentes type d'architectures ou de puissances de calcul et de mémoire.

3) Système Multi-Agents

3.1) Définition

Un système multi-agents (SMA) et une organisation d'agents intelligents et interactifs qui sont des entités réelles ou virtuelles avec des comportements autonome (action, mouvement, etc.). Ils ont la capacité de percevoir et d'agir sur les environnements dans lesquelles ils évoluent, ils peuvent aussi communiquer avec d'autres agents et interagir avec eux avec des signaux physiques ou par voix d'échange de messages, afin d'atteindre les buts qu'ils poursuivent.

Les agents sont des machines programmées pour exécuter des algorithmes selon une dynamique d'interaction changeant leurs états et donc l'état du système, et selon les décisions locales ces états peuvent converger ou non. [2, 5].

3.2) Domaines d'application et Caractéristiques

Les systèmes multi-agents sont utilisés dans de nombreuses branches de l'électronique et de l'informatique (robotique, systèmes experts, systèmes distribués, commerce électronique, transport, logistique, etc.) on les retrouve le plus souvent dans les systèmes industriels complexes comme les entreprises en réseaux. On retrouvons également les systèmes multi-agents en médecine, en contrôle de processus, etc. [2, 5].

Les agents d'un système multi-agents présentent plusieurs caractéristiques importantes :

- **Autonomie :** les agents sont plus au moins et partiellement indépendants, conscients et autonomes.
- **Vue locale :** aucun agent n'a une vue globale du système, ou bien le système est trop complexe pour qu'un agent puisse utiliser ces connaissances de façon pratique.
- **Décentralisation** : il n'y a pas d'agent de contrôle désigné (ou le système est effectivement réduit à un système monolithique).
- **Hétérogénéité** : les données qui sont traitées et les décisions qui sont prises peuvent concerner différents domaines complètement différents et indépendants les uns des autres.
- **Flexibilité** : des entités peuvent s'insérer ou se retirer du système à tout moment en cours de fonctionnement.

4) Le consensus

Le problème du consensus est un problème fondamental dans les systèmes distribués. Un ensemble de processus informatique isolé ne peut communiquer qu'avec des

messages, et doit se mettre d'accord sur quelque chose. Le consensus est facile en soit en l'absence de défauts, mais devient difficile dans les scénarios de défaillance complexes avec la présence de canaux imparfaits, panne des participants, violation de la synchronisations ou même si certains d'entre eux peuvent comploter pour faire que ce consensus ne se produisent pas (comportement malveillant) [4, 12].

La solution à ces problèmes prend la forme d'un algorithme ou d'un protocole (consensus ou d'un algorithme de consensus) et est utilisée par tous les processus qui ont ou n'ont pas un comportement malveillant.

Les systèmes distribués sont généralement sujet à des échecs soit par des failles ou par des défauts intentionnels (problème du général byzantin). Pour obtenir un bon consensus en un temps fini le système doit être le plus fiable que possible et tolérant au échecs [4].

L'une des stratégies les plus communes pour y parvenir est de répliquer l'information. La réplication du système d'information est plus fiable parce que l'information se trouve sur plusieurs nuds. Cependant, il est difficile de maintenir la cohérence entre les nuds qui partagent des informations à un moment dans un environnement où il peut y avoir des défaillances [4, 6].

Par exemple, une base de données distribuée, pour reproduit un état commun elle doit maintenir un état cohérent de la réplication des informations, chaque réplique doit appliquer les mêmes opérations dans le même ordre pour une copie de la déclaration [4, 6].

Un algorithme de consensus doit remplir certaines conditions et exigences pour être utilisable [6] :

- **Achèvement (Terminaison) :**
  Tout processus correct doit se terminer en ayant attribué une valeur à la variable de décision.
- **Accord :**
  La valeur finale décidée par tous les processus corrects est la même.
- **Validité :**
  Si tous les processus corrects proposent la même valeur, alors tout processus correct dans l'état de décision choisissent cette valeur.
- **Intégrité :**
  Chaque processus correct décide au plus une valeur, et cette valeur doit avoir au moins été proposé par un autre processus.

5) Conclusion

De ces définitions, les systèmes distribués et multi-agents permettent l'interconnexion et l'interopération de plusieurs systèmes existants. Le consensus dynamique ou l'accord dynamique qui est une extension de problème de consensus basique pour ces systèmes, permet où décisions locales qui causent les états des agents de converger vers un état commun.

Chapitre 2

Méthodes expérimentales

« *Un programme informatique fait ce que vous lui avez dit de faire, pas ce que vous voulez qu'il fasse.* »

Troisième loi de Greer

Sommaire

1) Introduction 10

2) Problème et incertitude Algorithmique 10

2.1) Algorithme 10

2.2) Calcul et exécutions 10

2.3) Propriétés des algorithmes 11

2.4) Propriétés du Consensus dans les systèmes et algorithmes répartis 12

2.5) Synchronisme et Asynchronisme 13

2.6) Défaillances d'un processus 14

3) Modèles des Graphes 16

3.1) Graphes statiques 16

3.2) Graphes dynamiques 16

3.3) Graphes évolutifs 17

3.4) Graphes variant dans le temps 17

4) État de l'art (Consensus) 19

4.1) Le protocole de Paxos 19

4.2) Raft 20

4.3) Phase King 23

4.4) Nakamoto consensus (cas de Bitcoins) 23

4.5) Ripple Protocol Consensus Algorithm 24

4.6) Stellar Consensus Protocol 24

4.7) Preuve du temps écoulé (PoET) 26
4.8) Le protocole Lockstep comme protocole de consensus 28
4.9) Hashgraph Consensus 29
4.10) Autres travaux 29
5) Conclusion 30

1) Introduction

Les problèmes liés à l'implémentation d'un algorithme de consensus dans les systèmes distribues dynamiques ou multi-agents, englobe de manière approximative les problèmes rencontrés dans les algorithmes parallèles distribués, ainsi que celle liée à l'implémentation d'un réseau mobile. Dans ce chapitre nous décrirons ces incertitude algorithmique ainsi que les propriétés du consensus et les problèmes qui lui est sont inhérent, nous présenteront les graphes utilisé dans les systèmes dynamique, et nous termineront sur un état de l'art des protocoles de consensus les plus populaire du moment

2) Problème et incertitude Algorithmique

2.1) Algorithme

Les performances d'un ordinateur dépendent de la vitesse de communication entre ses composants électroniques (UAL, DD, Mémoire, Bus) et cette rapidité dépend du matériel (vitesse du signal lumineux : $3x10^8$m/s, vitesse du signal dans le cuivre : $2,7x10^8$m/s). Les dimensions d'une puce électronique : 0,25ţm et une unité de temps CPU : $1x10^{-9}$s et au delà (10^{-10}) cela provoque un effet quantum [1].

Les solutions sont freinées par ces limites matériels est un regroupement de plusieurs processeurs pour faire une illusion d'une seule machine peut résoudre certains problèmes complexes [1].

- Un algorithme : Description pas à pas des étapes à suivre pour atteindre un résultat
- Un algorithme séquentiel : son exécution se fera instruction par instruction (processus).
- Un algorithme parallèle : son exécution se fera plusieurs instructions à la fois.
- Un algorithme distribué : son exécution se fera plusieurs instructions sur plusieurs machines à la fois (donc parallèle aussi).

2.2) Calcul et exécutions

En informatique théorique, tout problème de calcul est un objet mathématique représentant une collection de questions que les ordinateurs pourraient résoudre.

Chaque processus possède un état local qui est un ensemble de valeurs stocké dans des variables locales et un compteur de programme. Un calcul au sein du système consiste en un pas d'un algorithme exécuté séquentiellement par les processus [7]. Chaque pas est soit :

1. Une invocation d'une opération de haut niveau ;

2. L'affectation d'une primitive sur un objet de base faisant ainsi changer l'état du processus et des calculs locaux;
3. La fin d'une opération de haut niveau avec le retour de la réponse.

2.3) Propriétés des algorithmes

a) Propriétés de vivacité

La vivacité se réfère à un ensemble de propriétés de systèmes, qui contraint le système à progresser (lié au futur) malgré le fait que des composants (processus) du système s'exécutant simultanément et peuvent à tout moment vouloir entrer dans des sections critiques. Les garanties de vivacité sont des propriétés importantes dans les systèmes d'exploitation et les systèmes distribués.
Une propriété de vivacité ne peut être violée lors d'une exécution en un temps finie d'un système distribué car l'événement "validé ou bon" ne peut se produire théoriquement qu'à un certain moment après la fin de l'exécution. La cohérence éventuelle est un exemple de propriété vivante [7].

Exemple : un événement souhaité arrivera nécessairement. Elle est liée à la progression du programme.

b) Propriétés de sûreté

Dans les systèmes distribués ou dans les algorithmes distribués, les propriétés de sécurité exigent de manière informelle d'un programme le fait de ne pas sortir d'un ensemble d'états certifié et irréfragable (lié au Passé). Contrairement aux propriétés de vivacité, les propriétés de sécurité peuvent être violé par une exécution achevé d'un système distribué [7].

Exemple : un événement indésirable ne s'est pas produit pendant l'exécution : le programme doit produire le résultat "exacte".

c) Propriétés d'avancement

Un algorithme est sans délai et sans interruption si dans ses exécutions, chaque processus correct termine ses opérations en un nombre fini de ses propres pas tout en s'exécutant suffisamment longtemps pour qu'il ait un résultat [7].

d) Propriétés d'équité

La notion d'équité se rapporte au principe même de l'équité dans le cadre des algorithmes d'ordonnancement ou de synchronisation.
Un algorithme équitable est un algorithme qui garantit que toutes ses threads concer-

nées par l'ordonnancement (ou la synchronisation) sont traitées de manière identique. Le non-respect de l'équité de ce contexte peut conduire à des cas de famine [7].

e) Propriétés des identifiants des processus

Un processus A est anonyme si l'algorithme ou les phase de communication et d'échange de message ne dépendent pas de l'identifiant de processus [7].

2.4) Propriétés du Consensus dans les systèmes et algorithmes répartis

Soit un ensemble de processus $p_1, \ldots, p_n$ reliés par des canaux de communication.
Initialement : chaque processus p_i propose une valeur v_i.
À la terminaison de l'algorithme chaque processus p_i décide d'une valeur d_i [12].

Les propriétés qui doivent être atteinte sont :

- Accord : la valeur décidée est la même pour tous les processus corrects
- Intégrité : tout processus ayant décidé au plus une fois, sa décision est définitive
- Validité : la valeur décidée est l'une des valeurs proposées
- Terminaison : tout processus correct décide au bout d'un temps n_i

Les valeurs énoncées ne sont pas assurément distinctes. De même pour les valeurs décidées, qui ne sont pas forcément des valeurs dominantes, ni celle d'un processus correct [9].

Le démarrage d'un algorithme de consensus peut être soit un démarrage simultané (à une heure donnée), soit un démarrage initié par l'un des processus(diffusion d'un message d'initialisation de l'algorithme), où bien sur réception d'un 1er message de l'algorithme [9].

Les variantes du consensus sont [11] :

- Le consensus uniforme (avec accord uniforme) : la valeur décidée est la même pour tous les processus qui décident (corrects ou ultérieurement fautifs "un processus peut décider, puis devenir incorrect ".)
- k-consensus (avec k Accord) : au plus k valeurs distinctes sont décidées pour l'ensemble des processus corrects (exemple de consensus basique : k=1)
- Consensus approximatif (avec ϵ -Accord) : les valeurs décidées par les processus corrects doivent être à distance maximale ϵ l'une de l'autre.

2.5) Synchronisme et Asynchronisme

Un lien est considéré comme étant synchrone, s'il existe une valeur connue initialement tel que le délai entre l'envoi et la réception d'un message soit borné par une constante Θ. Si le délai est borné au bout d'un certain temps par une valeur à priori non connue, on parle de lien partiellement synchrone. Si par contre, aucune borne n'existe, le lien est dit asynchrone.
On dit qu'un réseau est synchrone, si tous ses liens sont synchrones. On définit de même un réseau partiellement synchrone et un réseau asynchrone [6].

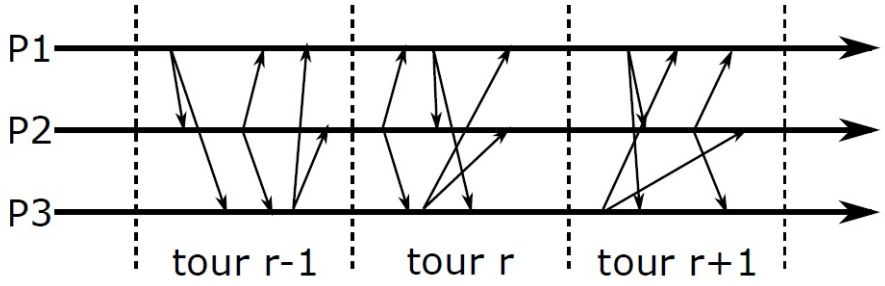


FIGURE 2.1 – Algorithmes fonctionnant par tours, synchronisés sur tous les processus.

Dans les consensus, les ordinateurs doivent effectuer les différents calculs dans le même ordre afin de s'assurer de la cohérence du service. Malheureusement, dans le cas général des systèmes asynchrones, cela a été prouvé impossible si, ne serait-ce qu'un seul processus tombe en panne [6].

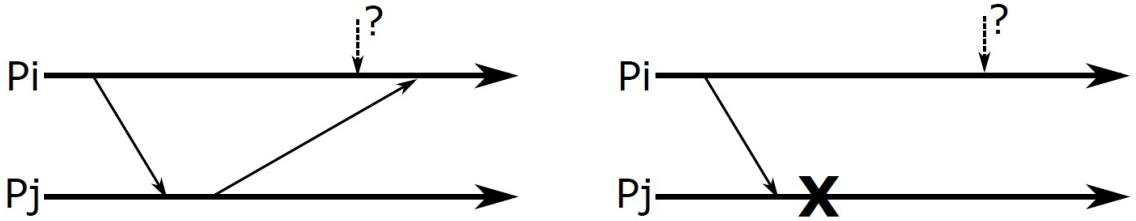


FIGURE 2.2 – Impossible de distinguer un processus lent d'un processus arrêté

Un cas particulier du problème de consensus appelé consensus binaire, restreint l'entrée et donc le domaine de sortie à un seul chiffre binaire 0,1. Lorsque le domaine d'entrée est grand par rapport au nombre de processus, par exemple un ensemble d'entrée de tous les nombres naturels, on peut montrer que le consensus est impossible dans un modèle de passage de message synchrone.

Alors que les communications du monde réel sont souvent intrinsèquement asynchrones, il est plus pratique et utile de modéliser les systèmes synchrones [9]. Dans un système distribué de transmission de message entièrement asynchrone dans lequel un processus peut avoir une défaillance d'arrêt, il a été prouvé que le consensus est impossible [6].

Cependant, ce résultat d'impossibilité découle d'un pire scénario d'un calendrier de processus qui est très improbable. En réalité, la planification des processus a un certain degré d'aléatoire [9].

Dans un modèle asynchrone, certaines formes de défaillances peuvent être traitées par un protocole de consensus synchrone. Par exemple, la perte d'un lien de communication peut être modélisée comme un processus qui a subi une défaillance byzantine.

Dans les systèmes synchrones, on suppose que toutes les communications se déroulent en rondes. Dans un tour, un processus peut envoyer tous les messages qu'il nécessite tout en recevant tous les messages d'autres processus. De cette manière, aucun message d'un tour ne peut influencer les messages envoyés dans le même cycle [13].

2.6) Défaillances d'un processus

Les modes de défaillances, comme les échecs byzantins qui sont les plus sévères ou bien les échecs d'arrêt qui sont moins sévères, montrent quel type d'hypothèses nous pouvons faire au sujet d'un processus. En outre, ils peuvent être considérés comme des pannes cohérentes ou des pannes incohérentes. Dans le premier cas, tous les utilisateurs perçoivent les mêmes défaillances. Tous les modes de défaillance de performance se trouvent dans cette catégorie. Ensuite, sous des échecs incohérents, nous avons les byzantins, où différents utilisateurs du service peuvent avoir des perceptions différentes de l'échec.

a) Échecs d'omission :

Lorsqu'un processus ou un canal ne parvient pas à faire quelque chose à laquelle on s'attend, il est qualifié d'Échec d'omission (Omission Failures), il s'agit d'un cas particulier du problème de performance où un serveur réponde "infiniment tard". [8].

Échecs d'omission des processus : un processus entraîne une omission lors d'une panne et ne fera aucun progrès supplémentaire sur son programme.

Un crash est considéré comme propre si le processus s'arrêté ou bien il continue de fonctionner correctement.

Un crash est qualifié d'échec si d'autres processus peuvent détecter avec certitude que le processus s'est crashé [8].

Échecs d'omission de communication : des pannes d'omission de communication peuvent survenir dans le processus d'envoi (send-omission failures), le processus de réception (receive omission failures) ou le canal (channel omission failures) [8].

b) Échecs Arbitraire :

Le terme "arbitraire" ou "défaillance byzantine" est utilisé pour désigner le type d'échec dans lequel toute erreur peut se produire. Dans un processus, un comportement arbitraire peut inclure la définition de valeurs de données incorrectes, le retour d'une valeur de type incorrect, l'arrêt ou la prise de mesures incorrectes [8].

Dans un canal, le comportement arbitraire peut inclure la duplication ou la corruption des messages. La plupart des protocoles incluent des mécanismes pour surmonter les défaillances arbitraires dans un canal. Par exemple, des sommes de contrôle pour détecter la corruption et les numéros de séquence pour détecter une reproduction [8].

Les échecs arbitraires dans un processus sont moins faciles à détecter et peuvent avoir un profond impact sur un système dans lequel plusieurs processus doivent coopérer. Par exemple, considérez le comportement des algorithmes de choix de leader si l'un des processus s'est comportés de manière erronée et na pas suivi le protocole de l'algorithme [8].

c) Systèmes synchrones et échecs temporels :

Chacun des éléments d'un système distribué synchrone est définis par [8] :

- Une limites supérieures et inférieures du temps d'exécution de chaque étape d'un processus ;
- Une liaison sur le temps de transmission de chaque message sur un canal ;
- Une limite sur le taux de dérive de l'horloge locale de chaque processus.

Si l'une de ces bornes n'est pas satisfaite dans un système synchrone, une erreur de synchronisation est dite avoir eu lieu. Peu de systèmes réels sont synchrones (ils peuvent être construits en cas de garantie à l'accès aux ressources) mais ils constituent un modèle utile pour raisonner sur les algorithmes du consensus. Les délais d'attente peuvent être utilisés pour détecter les processus défaillants. Dans un système asynchrone le délai d'attente peut seulement indiquer qu'un processus ne répond pas.

d) Tolérance de panne byzantine :

Une faute byzantine est une faute présentant des symptômes différents pour différents observateurs. Une défaite byzantine est la perte d'un service de système en raison d'un défaut byzantin dans les systèmes nécessitant un consensus [18].

L'objectif de la tolérance de panne byzantine est de pouvoir se défendre contre les défaillances byzantines, dans lesquelles des composants d'un système échouent avec des symptômes qui empêchent de s'entendre entre eux, la où un accord est nécessaire pour

le bon fonctionnement du système.

Le fonctionnement correct des composants d'un système tolérant aux pannes byzantine sera en mesure de fournir le service du système, en supposant qu'il n'y ait pas trop de composants défectueux [18].

3) Modèles des Graphes

3.1) Graphes statiques

On utilise traditionnellement des graphes statiques pour modéliser les capacités de communication d'un système distribué.

Un graphe orienté G est une paire (S, A) où S est un ensemble fini de sommets et A est une relation binaire sur S et constitue l'ensemble des arêtes de G [3].

Un graphe non-orienté G est une paire (S, A) où S est un ensemble fini de sommets et A est un ensemble de couples non ordonnés de sommets et ou les boucle sont interdites a l'intérieur du graphe [3].

3.2) Graphes dynamiques

Les réseaux modernes sans fils impliquent que le noeud faisant partie de l'architecture son mobile et peuvent être en constant mouvements ce qui peut amener à la dynamique du système.
Ces déplacements dans et hors de portée des rayons de couvertures font évoluer la topologie de réseaux à travers le temps [3].
La mobilité des noeuds implique une certaine autonomie au niveau de sa source d'énergie (utilisation de batterie), d'où un souci d'économie d'énergie, une approche consiste au cours de leur vie d'activer des cycles de repos pendant lesquels ils seront éteints. Ces cycles de repos sont une source de dynamicité qui influence le réseaux.

Selon les auteurs [10] on peut classifier la dynamicité d'un graphe selon des entités qui varie et peuvent être sujet au changement au cours du leurs vies [3] :

— Dans un graphe non-orienté (ou orienté) où l'ensemble des sommets varient dans le temps (ils peuvent rejoindre le graphe ou le quitter).

— Dans un graphe non-orienté (ou orienté) où l'ensemble des arêtes peuvent varier au fil du temps (ils peuvent rejoindre le graphe ou le quitter).

FIGURE 2.3 – Illustration d'un arbre dynamique par une suite de graphes qui représente l'évolution de l'arbre dynamique dans le temps [3].

3.3) Graphes évolutifs

L'objectif d'un graphe évolutif est de modéliser les réseaux dynamiques qui évoluent à travers le temps (l'évolution de l'ensemble des sommets et/ou de l'ensemble des arêtes). Un graphe évolutif peut être vu comme une suite ordonnée de graphes statiques où chaque graphe de cette suite exprime l'état du réseau à un moment donné. Il est exprimé par un couple G = (S, A) où S correspond à l'ensemble des sommets du graphe et A à une fonction dynamique sur les arêtes qui engendre à chaque étape t l'ensemble des arêtes qui sont disponibles figure 2.4 [3].

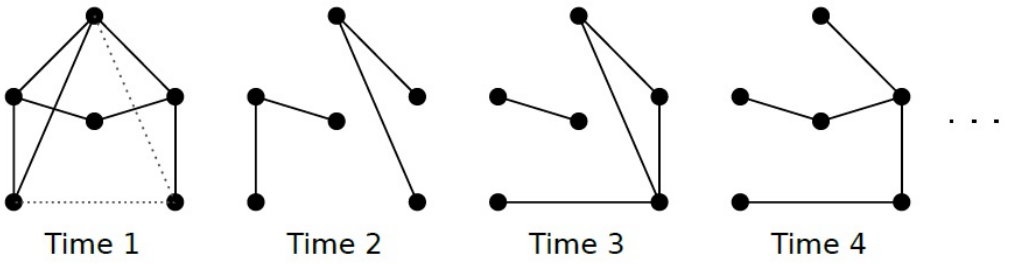


FIGURE 2.4 – Illustration de la dynamique du modèle de graphe évolutif à travers une suite de graphes statiques [3].

3.4) Graphes variant dans le temps

Un graphe variant dans le temps G = (S, A, T , ρ, ζ, ϕ) est un graphe avec un ensemble de sommets noté S = (p_1, p_2,..., p_n) qui interagissent entre eux. L'interaction entre ces sommets détermine l'ensemble des arêtes A ⊆ S ×S , dont les disponibilités peuvent être désordonnée [3].

La fonction de présence des arêtes ρ représente la disponibilités des arêtes au cours du temps [3].

La fonction de latences de chaque arête au cours du temps est représenté par ζ dans un graphe, elle est constante et connue par tous les processus [3].

La fonction ϕ représente le temps de traitement d'une action par chaque sommet, et permet de modéliser la variation de ce délai dans le temps [3].

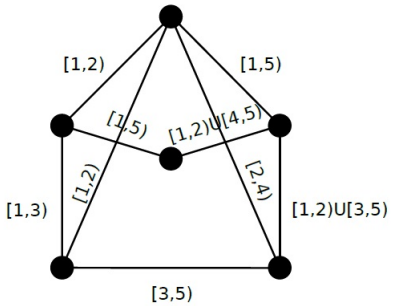


FIGURE 2.5 – Illustration de graphe variant dans le temps ou l'on ignore les latences ζ et ϕ [3].

4) État de l'art (Consensus)

4.1) Le protocole de Paxos

Le protocole de Paxos [20] a été publié pour la première fois en 1989 et a été nommé d'après un système fictif de consensus législatif utilisé sur l'île de Paxos en Grèce. Il a ensuite été publié comme article de journal en 1998.

La famille des protocoles Paxos comprend un éventail de compromis entre le nombre de processeurs, le nombre de retards des messages avant la connaissance de la valeur convenue, le niveau d'activité des participants individuels, le nombre de messages envoyés et les types de pannes. Bien qu'aucun protocole de consensus déterministe et tolérant aux pannes ne puisse garantir le progrès d'un réseau asynchrone, Paxos garantit la sécurité et la cohérence, et les conditions qui pourraient l'empêcher de progresser sont difficiles à provoquer.

Paxos est habituellement utilisé en cas de durabilité (par exemple, pour reproduire un fichier ou une base de données), dans lequel la quantité d'état durable peut être importante. Le protocole tente de progresser même pendant les périodes où certains nombres délimités de répliques ne répondent pas. Il existe également un mécanisme pour supprimer un réplica qui échoue en permanence ou pour ajouter un nouveau réplica.

Paxos définit plusieurs rôles pour ces acteurs : client, acceptor, proposer, learner, et leader. Généralement un seul processeur peut jouer plusieurs rôles au même instant. Ce n'est pas un problème pour le protocole, et c'est d'usage courant pour baisser la latence entre les messages.

- Le Client fait des requêtes aux systèmes distribués, et attend une réponse. Par exemple une requête d'écriture sur un système de fichiers distribué.
- Les Acceptors (Votants) servent de mémoire résistante aux pannes. Les Acceptors sont regroupés en groupes nommés Quorums. Chaque message envoyé à un Acceptor doit l'être au Quorum entier. Un message reçu sur un Acceptor qui n'a pas été reçu sur tous les autres Acceptors du Quorum est ignoré.
- Un Proposer pousse la requête du client, il a pour but de convaincre les Acceptors de tomber d'accord, et agît comme coordinateur pour avancer quand un conflit se présente.
- Les Learners servent à la réplication. Une fois qu'une requête d'un Client a été acceptée par les Acceptors, le Learner peut agir (i.e. : exécuter une requête et envoyer la réponse au client). Pour augmenter la disponibilité on peut ajouter des Learners.
- Leader : Paxos nécessite un Proposer différent (appelé le leader) pour avancer. Plusieurs processus peuvent croire être le leader, mais le protocole ne garantit l'avan-

cement uniquement si l'un d'eux est choisi. Si deux processus croient qu'ils sont leader, ils peuvent bloquer le protocole en envoyant continuellement des propositions conflictuelles. Mais l'intégrité des données est toujours préservée dans ce cas [20].

4.2) Raft

Raft est un algorithme de consensus conçu comme une alternative à Paxos . Il était censé être plus compréhensible que Paxos au moyen de la séparation de la logique, mais il est également formellement prouvé en toute sécurité et offre des fonctionnalités supplémentaires. Raft offre une manière générique de distribuer une machine d'état à travers un cluster de systèmes informatiques, en s'assurant que chaque noeud du cluster accepte la même série de transitions d'état [21].

Raft atteint un consensus par l'intermédiaire d'un leader élu. Un serveur dans un cluster est soit un leader, un candidat, soit un adepte. Le leader est responsable de la réplication des journaux aux adeptes. Il informe régulièrement les adeptes de son existence en envoyant un message de battement de coeur. Chaque suiveur a un délai d'attente (généralement entre 150 et 300 ms) dans lequel il attend le rythme cardiaque du leader. Le délai d'attente est réinitialisé lors de la réception des battements cardiaques. Si aucun rythme cardiaque n'est reçu, le suiveur change son statut en candidat, et commence une nouvelle élection de leader [21].

Élection du leader

Une élection de leader est lancée par un serveur candidat. Un serveur devient un candidat s'il ne reçoit aucun battement de coeur du leader dans le délai d'attente. Il commence les élections en augmentant le compteur de termes et en envoyant un message "RequestVote" à tous les autres serveurs. Les autres serveurs voteront pour le premier candidat qui leur envoie un message "RequestVote". Un serveur ne votera qu'une fois par trimestre. Si le candidat reçoit un message d'un leader avec un nombre de termes égal ou supérieur au terme actuel, son élection est vaincue et le candidat devient un adepte. Si un candidat reçoit une majorité de voix, il devient le nouveau leader. Si l'un d'eux ne se produit pas, par exemple, en raison d'un vote par division, une nouvelle élection de leader est lancée après un délai d'expiration [21].

Les valeurs de temporisation de chaque serveur doivent être réparties dans un intervalle raisonnable. Cela devrait réduire les chances de vote partagé car les serveurs ne deviendront pas candidats au même moment [21].

Algorithme 2.1 - L'algorithme de RAFT pour l'élection de leader [21]

```
1 : Leader :
2 : tant que (statut = Leader) faire
3 :    envoyer(BattementCoeur, toutLeMonde) ;
4 : fin tant que
5 : //- - - - - - - - - - - -
6 : Adepte :
7 : tant que (statut = adepte) faire
8 :    //Traitement des messages reçus
9 :    tant que (FileAttenteMessages.taille != 0) faire
10 :      //Verifier si c'est une demande de vote
11 :      Si (FileAttenteMessages[i].type = RequestVote) alors
12 :         statut ← candidat;
13 :         lastTime ← temps;
14 :         allez a Candidat ;
15 :      fin Si
16 :      //vérifier si c'est une pulsation leader
17 :      Si (FileAttenteMessages[i].type = battementCoeur) alors
18 :         lastTime ← temps;
19 :      fin Si
20 :      //vérifier le délai attente d'une pulsation leader
21 :      Si (LastTime + 300 < temps) alors
22 :         statut ← candidat;
23 :         lastTime ← temps;
24 :         idLeader ← 0;
25 :         allez a Candidat ;
26 :      fin Si
27 :   fin tant que
28 : fin tant que
29 : //- - - - - - - - - - -
30 : Candidat :
31 : tant que (statut = candidat) faire
32 :    //C'est un délai est dépassé réémettre la requete de vote
33 :    Si (lastTime + trimestre < temps) alors
34 :       engagerCandidateur ← vrai
35 :       compteurACK ← 1
36 :       compteurNACK ← 0
37 :       envoyer(RequestVote, toutLeMonde)
38 :    fin Si
```

Algorithme 2.1 - L'algorithme de RAFT pour l'élection de leader [21]

```
39 :  dejaVoter ← faux
40 :  //Traitement des messages reçus
41 :  tant que (FileAttenteMessages.taille != 0) faire
42 :    //vérifier si c'est une pulsation leader
43 :    Si (FileAttenteMessages[i].type = battementCoeur) alors
44 :      lastTime ← temps
45 :      idLeader ← FileAttenteMessages[i].Expediteur
46 :      statut ← adepte
47 :      allez a Adepte;
48 :    fin Si
49 :    Si (FileAttenteMessages[i].cible = monId) alors
50 :      //vérifier si on a voter pour moi
51 :      Si (FileAttenteMessages[i].type = voteACK) alors
52 :        compteurACK++
53 :      fin Si
54 :      //vérifier si on veut pas voter pour moi
55 :      Si (FileAttenteMessages[i].type = voteNACK) alors
56 :        compteurNACK++
57 :      fin Si
58 :    fin Si
59 :    //vérifier si on me demande de voter pour quelqu'un
60 :    Si (FileAttenteMessages[i].type = requestVote) alors
61 :      //si j'ai pas voter pour quelqu'un ou que je n'ai pas engager alore envoyer un
    ACK sinon envoyer un NACK
62 :      Si ((!engagerCandidateur) et (!dejaVoter)) alors
63 :        lastTime ← temps
64 :        envoyer(voteACK, FileAttenteMessages[i].Expediteur)
65 :      sinon
66 :        envoyer(voteNACK, FileAttenteMessages[i].Expediteur);
67 :      fin Si
68 :    fin Si
69 :  fin tant que
```

Algorithme 2.1 - L'algorithme de RAFT pour l'élection de leader [21]

```
//si je suis de ceux qui ont engager vérifier mon état d'avancement
Si ((compteurNACK > compteurACK) et (engagerCandidateur)) alors
    compteurACK ← 0
    compteurNACK ← 0
    engagerCandidateur ← faux
sinon
    statut ← Leader
    allez a Leader;
fin Si
```

4.3) Phase King

Phase King est un protocole de consensus binaire polynomial qui tolère les échecs byzantins [22].
L'algorithme résout le consensus dans un modèle de passage de message synchrone avec n processus et f échecs, à condition que n > 4f. Dans l'algorithme du roi de phase, il existe f + 1 phases, avec 2 tours par phase. Chaque processus surveille sa sortie préférée (initialement égale à la valeur d'entrée propre au processus). Au premier tour de chaque phase, chaque processus diffuse sa propre valeur préférée à tous les autres processus. Il reçoit ensuite les valeurs de tous les processus et détermine quelle valeur est la valeur majoritaire. Au deuxième cycle de la phase, le processus dont l'ID correspond au numéro de phase actuelle est désignée le roi de la phase. Le roi diffuse la valeur majoritaire observée au premier tour et sert de breaker. Chaque processus met ensuite à jour sa valeur préférée. Si le nombre de la valeur majoritaire observée au premier tour est supérieur à n/2 + f, le processus change sa préférence à cette valeur majoritaire; Sinon, il utilise la valeur de phase King. À la fin de f + 1 phases, les processus affichent leurs valeurs préférées.

4.4) Nakamoto consensus (cas de Bitcoins)

Le consensus de Nakamoto est un nom pour le protocole décentralisé de consensus de Bitcoin. Il est considéré comme l'innovation fondamentale de Bitcoin et sa clé du succès. Ce protocole de consensus ne requiert aucune partie de confiance ou identité présupposée parmi les participants [23].

Toute la conception de Bitcoin repose sur le principe selon lequel les agents rationnels externes et internes sont incités à détruire ou à tenter de détruire le réseau ou de le

renverser pour augmenter leurs profits ou éviter toute perte.

Approche :

- Élection d'un leader à travers une forme de "loterie" choisi parmi des mineurs qui décident de se concurrencer pour produire un nouveau bloc et gagner une récompense.
- Le premier participant à résoudre avec succès un puzzle cryptographique gagne la loterie des élections
- Le leader propose alors un bloc qui peut être ajouté à une chaîne de blocs précédemment engagés.
- Le leader élu diffuse le nouveau bloc au reste des participants qui votent implicitement pour accepter le bloc et ajoutant celui-ci à une chaîne de blocs acceptés et en proposant des blocs de transactions ultérieurs qui s'appuient sur cette chaîne.

Cela peut parfois conduire à des embranchements temporaires en raison de plusieurs solutions trouvées simultanément. Le plus long embranchement dans ce cas est considéré comme valide.

4.5) Ripple Protocol Consensus Algorithm

Un livre blanc a été publié par Ripple Labs, examinant comment les mécanismes décentralisés peuvent empêcher les mauvais acteurs dans un système de paiement en Blockchain de créer de fausses transactions.
L'algorithme de consensus du protocole Ripple (RPCA) utilise un système électoral de noeud pour établir la véracité des nouvelles transactions, en les ajoutant à une chaîne continue de transactions fermées considérées comme absolues.
L'objectif du RPCA est de permettre à un réseau globalement déconnecté de s'entendre sans compter sur l'infrastructure de démonstration de travail.
Chaque serveur du réseau Ripple est chargé de voter sur un nouveau lot de transactions candidat pendant les rondes qui se déroulent toutes les quelques secondes. Les transactions convenues par le réseau sont confirmées et rendues définitives une fois le cycle terminé [24].

4.6) Stellar Consensus Protocol

SCP est une construction pour l'accord byzantin fédérée, qui est une nouvelle approche du consensus.
Afin d'assurer le consensus, même lorsque les noeuds individuels agissent de manière arbitraire (échec byzantin), SCP est conçu pour ne pas exiger le consentement unanime de l'ensemble complet des noeuds pour que le système parvienne à un accord et tolérer les

noeuds qui représentent ou envoient des messages incorrects.
SCP prend le principal défi du consensus distribué : le système ne peut pas s'entendre sur une déclaration sans risque de se bloquer et perdre la vie.
Il est possible qu'une déclaration soit bloquée dans un état permanent indéterminé avant que le système n'arrive à un accord. L'objectif de SCP est de minimiser ce blocage et le potentiel de divergence. Le protocole utilise des bulletins de vote est un référendum sur la valeur associée aux bulletins de vote, cette approche signifie que, pour éventuellement externalisé une valeur, un noeud doit engager le bulletin de vote lié à cette valeur.
Chaque noeud peut soit engager ou avorter tout bulletin de vote.
Dans un système SCP avec intersection de quorum, il n'y a pas d'états bloqués pour les noeuds correct. Il y a toujours une séquence d'événements à travers laquelle les noeuds correct peuvent s'entendre et s'engager sur une valeur [25].

Algorithme 2.2 - Algorithme simplifier de consensus Stellar [25]

```
1 : contreMonVote ← 0;
2 : avecMonVote ← 0;
3 : tours ← 0;
4 : tant que (Referendum) faire
5 :   tours ← tours + 1;
6 :   envoyer(BulletinVote, engager, monVote, toutLeMonde);
7 :   tant que (FileAttenteMessages.taille != 0) faire
8 :     Pour (j=0; j< QuorumSlice.taille; j++) faire
9 :       Si (QuorumSlice[i].id = FileAttenteMessages[i].Expediteur) alors
10 :        Si (FileAttenteMessages[i].contenuEngager = monVote) alors
11 :          AvecMonVote ← AvecMonVote + 1;
12 :        fin Si
13 :      sinon
14 :        contreMonVote ← contreMonVote + 1;
15 :        ajouterALaListedesChoix(FileAttenteMessages[i].contenuEngager);
16 :      fin Si
17 :
18 :    fin Pour
19 :  fin tant que
20 :  Si ((tours != 1) et (contreMonVote > AvecMonVote)) alors
21 :    monVote ← autreChoieDominant;
22 :  fin Si
```

4.7) Preuve du temps écoulé (PoET)

Dans le but de parvenir à un consensus réparti de manière efficace, PoET [26] est conçu pour atteindre ces objectifs en utilisant de nouvelles instructions de CPU sécurisées qui sont largement disponibles dans les processeurs grand public et d'entreprise. PoET utilise ces fonctionnalités pour assurer la sécurité et le caractère aléatoire du processus électoral de leader sans nécessiter l'investissement coûteux d'énergie et de matériel spécialisé inhérent à la plupart des algorithmes "à l'épreuve".

PoET fonctionne essentiellement comme suit :

- Chaque candidat demande un temps d'attente à partir d'une enclave (une fonction de confiance).
- Le candidat avec le temps d'attente le plus court pour un bloc de transaction particulier est élu leader.
- Une fonction "CreateTimer", crée une minuterie pour un bloc de transaction dans l'enclave.
- Une autre fonction "CheckTimer", vérifie que la minuterie a été créé par l'enclave, et si elle a expiré, crée une attestation qui peut être utilisée pour vérifier que le candidat a, en fait, attendu le temps alloué avant de revendiquer le rôle de leader.

L'algorithme de choix de PoET leader répond aux critères d'un bon algorithme de loterie. Il distribue de manière aléatoire les élections de leadership dans toute la population de candidat avec une distribution similaire à celle fournie par d'autres algorithmes de loterie. La probabilité d'élection est proportionnelle aux ressources apportées (dans ce cas, les ressources sont des processeurs à usage général avec un environnement d'exécution approuvé). Une attestation fournit des informations pour vérifier que le certificat a été créé dans l'enclave (et que le candidat a attendu le temps alloué). En outre, le faible coût de la participation augmente la probabilité que la population des candidats soit grande, ce qui augmente la robustesse de l'algorithme de consensus.

Algorithme 2.3 - Algorithme simplifier de protocole PoET [26]

```
1 : //Aucun leader détecter
2 : //Préparation à la candidature pour le poste de leader
3 : idLeader ← 0;
4 : statut ← candidat;
5 : tempsAttente ← monTempsAttente;
6 : tant que (statut = candidat) faire
7 :    // Engager ma candidature
8 :    envoyer(candidat, tempsAttente, toutLeMonde);
9 :    // Traiter la file d'attente des messages
10 :   tant que (FileAttenteMessages.taille != 0) faire
11 :      Si (FileAttenteMessages[i].type < candidat) alors
12 :         Si (FileAttenteMessages[i].tempsAttente < tempsAttente) alors
13 :            Valide ← verifierAttestationEnclave(FileAttenteMessages[i].Expediteur);
14 :            Si (Valide) alors
15 :               tempsAttente ← FileAttenteMessages[i].tempsAttente;
16 :               idLeader ← FileAttenteMessages[i].Expediteur;
17 :               i ← i + 1;
18 :            fin Si
19 :         fin Si
20 :      fin Si
21 :   fin tant que
22 :   Si (i = nombreDuNoeud) alors
23 :      Si (idLeader = 0) alors
24 :         statut ← Leader;
25 :      sinon
26 :         leaderExiste ← vrai;
27 :         statut ← adepte;
28 :      fin Si
29 :   fin Si
30 : fin tant que
```

4.8) Le protocole Lockstep comme protocole de consensus

De nombreux jeux de stratégie en temps réel en ligne peer-to-peer utilisent un protocole Lockstep modifié comme protocole de consensus afin de gérer l'état du jeu entre les joueurs d'un jeu. Chaque action de jeu entraîne une diffusion delta de l'état de jeu à tous les autres joueurs avec un hash de l'état global du jeu. Chaque joueur valide le changement en appliquant le delta à son propre état de jeu et en comparant les hachages d'état de jeu. Si les hachis ne sont pas d'accord, un vote est exprimé et les joueurs dont l'état de jeu est en minorité sont déconnectés et supprimés du jeu (désynchronisé) [28, 29].

Algorithme 2.4 - Algorithme de protocole Lockstep [30]

```
1 : //- - - - Envoi de message
2 : hash_Msg ← hash(messageAEnvoyer);
3 : Envoyer :
4 : envoyer(demandeDelta, hash_Msg);
5 : attendreACK();
6 : Si (nonConfirmer) alors
7 :    allez a Envoyer;
8 : sinon
9 :    envoyer(messageAEnvoyer);
10 : fin Si
11 : //- - - - Réception Message
12 : Si (messageReception(demandeDelta)) alors
13 :    ajouterListeMessageEnAttenteReception(idExpediteur, hash_Msg);
14 : sinon
15 :    test ← faux;
16 :    hash_Msg ← hash(messageReu);
17 :    Pour (j=0 j< ListeMessageEnAttenteReception.taille; j++) faire
18 :       Si (idExpediteur = ListeMessageEnAttenteReception[j].idExpediteur) alors
19 :          Si (hash_Msg = ListeMessageEnAttenteReception[j].hash_Msg) alors
20 :             test ← vrai;
21 :          fin Si
22 :       fin Si
23 :    fin Pour
24 : fin Si
25 : Si (test) alors
26 :    traiterMessage(mesageReçu);
27 : fin Si
```

4.9) Hashgraph Consensus

Hashgraph consensus [27] est un algorithme qui fonctionne sur une liste d'événements contenus dans une structure de données du hashgraph et maintient un consensus sur l'ordre chronologique correct des événements parmi un nombre arbitraire de noeuds participants. Des cas d'utilisation peuvent être trouvés dans n'importe quel système distribué qui bénéficie du maintien d'un consensus parmi les participants sur l'état global du système et l'ordre des changements d'état qui lui sont appliqués. De bons exemples sont la réplication de journaux, les livres comptables distribués, les machines d'état partagées, etc.

4.10) Autres travaux

a) Chubby

Google a mis en place une bibliothèque de services de verrouillage distribué appelée Chubby [31]. Chubby maintient les informations de verrouillage dans de petits fichiers qui sont stockés dans une base de données répliquée pour obtenir une disponibilité élevée face aux pannes. La base de données est implémentée sur une couche logarithmique tolérante aux pannes qui repose sur l' algorithme de consensus de Paxos . Dans ce schéma, les clients Chubby communiquent avec le maître Paxos pour accéder ou mettre à jour le journal répliqué ; C'est-à-dire lire ou écrire sur les fichiers [32].

b) PageRank

Les problèmes de consensus ont également une relation étroite avec l'algorithme PageRank utilisé par le moteur de recherche Google pour classer les pages Web des résultats de la recherche. Étant donné que le nombre de sites Web jusqu'à présent est supérieur à 1 milliard, PageRank nécessite le calcul d'un vecteur propre correspondant à la plus grande valeur propre d'une grande mais faible matrice. Par conséquent, l'utilisation d'informations globales n'est pas possible dans ce cas, des systèmes distribués et parallèles mises en oeuvre sont obligatoires [33].

c) Sawtooth Lake Consensus

Le Sawtooth Lake met en oeuvre le PoET comme un mécanisme de consensus.
Le Sawtooth Lake Distributed Ledger fournit un mécanisme unique pour assurer l'équité dans la loterie des noeuds. Au lieu d'une compétition de preuve de travail entre les noeuds, Sawtooth Lake met en oeuvre un algorithme de preuve de temps écoulé (PoET) pour un consensus distribué. PoET s'appuie sur un environnement d'exécution approuvé, les extensions de Intel's Software Guard (SGX), pour générer des compteurs d'attente aléatoires justes et vérifiables et des certificats signés d'expiration de la minuterie. Ce mécanisme

réduit considérablement le calcul et le coût de l'énergie pour assurer un consensus distribué équitable.

La mise en oeuvre de PoET dans le Sawtooth Lake s'exécute dans une enclave simulée, et non dans un environnement d'exécution confiant [26].

d) Soft Computing and intelligent Information Systems de l'UGR

Des scientifiques des universités de Grenade et de Cadix ont dessiné un nouveau système automatique qui permet de prendre des décisions ou de résoudre des problèmes de la vie quotidienne lorsque plusieurs agents impliqués ne se mettent pas d'accord.
Ce travail a reçu le prix de meilleur article scientifique de l'année 2014 dans le contexte du Congrès Annuel de l'Association IEEE System, Man and Cybernetics Society

Il s'agit d'un modèle de consensus entre experts hétérogènes à différents niveaux dans le groupe. Les chercheurs proposent un système qui combine le prestige ou le poids spécifique de chaque expert avec un modèle de négociation avec les experts, qui suggère la modification des préférences en fonction de leur importance dans le groupe afin d'atteindre le consensus avec peu d'itérations (acte de répéter un processus dans l'intention d'atteindre un objectif désiré) [34].

5) Conclusion

On peut remarquer que les différent protocole du consensus proposer sont implementable de façon universel pour différent cas. On remarque aussi que deux types d'approches différentes sont utilisés pour tenir compte de l'identité d'un processus (ou agent) ou de son anonymat. La différence finale réside au niveau de la tolérance ou fautes, de la sécurité offerte, et de la rapidité d'exécution. Cependant ces protocoles ne sont implémenté que sur des réseaux supposé étant connexe. Conséquence, ils se retrouvent insuffisants et incomplets pour une adaptation à une architecture système qui est dynamique et variante dans le temps.

Chapitre 3

Proposition d'un consensus pour un système dynamique

« *Le but de l'informatique est l'émulation de nos facultés de synthèse, pas la compréhension des facultés analytiques.* »

Alan Jay Perlis

Sommaire

1) Problématique **32**

2) Objectifs **32**

3) Solution **33**

3.1) Consensus de base 33

3.2) Protocole résultat 35

3.3) Paramètres de l'algorithme de consensus 43

3.4) Algorithme 45

3.5) Organigramme 52

3.6) Scénario d'execution 56

4) Cas d'étude **59**

1) Problématique

Le consensus dans les systèmes dynamique est un domaine de recherche situé à l'intersection de la théorie des systèmes et de la théorie des graphes.
Il est l'exigence d'un certain accord, entre un certain nombre de processus (ou agents) pour une valeur unique. Pour cela les protocoles étudier atteignent cette état de consensus en un temps fini. Mais quand le système se trouve sujet à des variations et ou à des évolutions au niveau de son architecture au cours de temps (topologie dynamique, noeuds mobiles, etc.), on se retrouve devant des problèmes d'organisation et de coordination. Et les protocoles précédants s'en retrouvent réduit et déficient.

La plupart des protocoles qui prennent en compte l'identité des noeuds, agissent comme si le réseau était connexe et ne font qu'ignorer les processus incorrectes jusqu'à un certain taux. Les protocoles utilisant l'anonymat se servent qu'ont à eux de la réplications des journaux, comme les Blockchains, et de se fait ils sont sujets à moins de problème dans les réseaux dynamiques, car une même copie de l'ensemble des informations des transactions réalisées au sein du réseaux, est synchronisée entre les noeuds, le problème majeur de cette approche est l'énorme quantité des flux de données engendrées et la quantité exponentielle d'énergie consommée par ces algorithmes par rapport aux nombre de noeuds, sachant que ces dernier dans un réseau dynamique sont mobiles et sont souvent alimentés par des batteries.
On peut être aussi contraint d'adjoindre à l'algorithme des paramètres, pour des questions d'économie d'énergie, de vitesse de convergence ou de la robustesse du système au bruit, ce qui aura pour conséquence d'affadir les performances de l'algorithme.

Comment peut-on alors parvenir à concevoir ou à adapter un protocole qui permettrait une collaboration équitable dans le but d'atteindre un consensus en un temps fini, tout en s'assurant des contextes particuliers comme la tolérance aux fautes, la performance des accès, la coordination et l'organisation dans une topologie qui ne cesse d'évoluer, et de la cohérence des données, dans un système qui est dynamique et variant dans le temps?.

2) Objectifs

Nous pouvons résumer les contraintes à satisfaire dans les points suivants :

- Offrir une alternative à la coordination et à l'organisation des noeuds dynamiques pour un consensus;
- Améliorer le système de candidature et d'élection des leaders;
- Collaborer de manière équitable afin d'atteindre un consensus dans un temps fini;
- S'assurer que le système est tolérant aux fautes.

- Utiliser un système de paramètres pour une meilleure optimisation de l'algorithme qui sont variables et qui évoluent avec le système.

3) Solution

Comme dans n'importe quel protocole, notre algorithme de consensus devra suivre certaines étapes durant son exécution.
Pour cela nous nous baserons sur des protocoles existant, tout en appliquant les ajouts et modifications nécessaires.

3.1) Consensus de base

Nous prendrons comme protocoles de consensus de base Raft, Stellar, PoET et Lockstep pour former un nouveau consensus.

a) PoET :

L'algorithme de choix de PoET leader [26] répond aux critères d'un bon algorithme de loterie. La probabilité d'élection est proportionnelle aux ressources existantes (nombre de noeuds participants, puissance de calcul des processeurs, etc.). Une attestation fournit des informations pour vérifier que le noeud a attendu le temps alloué, pour prétendre participer aux élections et devenir leader.
L'avantage est que le faible coût pour une participation augmente la probabilité que la population des noeud candidat soit grande, ce qui augmente la robustesse de l'algorithme de consensus.

b) Raft :

Raft [21] est très avantageux pour maintenir un cluster. Il atteint un consensus par l'intermédiaire d'un leader élu. Le leader est responsable de la réplication des journaux aux noeuds fils. Il informe régulièrement ces fils de son existence en envoyant un message de pulsation (un battement de coeur). Si aucune pulsation n'est reçue, les noeuds fils changent leurs statuts de candidats et commencent une nouvelle élection de leader.

c) Stellar :

Le protocole Stellar [25] est un protocole de consensus générique et décentralisé, qui utilise des bulletins de vote est un référendum sur la valeur associée aux bulletins de vote, cette approche signifie que, pour éventuellement externaliser une valeur, un noeud doit engager le bulletin de vote lié à cette valeur. Un noeud peut soit engager ou avorter tout bulletin de vote.

Chaque noeud décide en qui il peut avoir confiance (le Quorum Slice, voir l'annexe), c'est-à-dire l'ensemble des noeuds qui vont le convaincre s'ils sont tous d'accord sur quelque chose. Chaque noeud peut décider lui-même du compromis entre la Vivacité :

- Est-ce qu'on peut progresser dans le consensus ?
- On ne reste pas bloqué si on arrive pas à communiquer avec les autres ;

et la sécurité :

- Est-ce qu'on est bien d'accord avec le consensus ?
- On ne croit pas à n'importe quoi et à n'importe qui ;

qui sont des caractéristique de chaque noeud et non du réseau entier.

L'avantage est qu'il n'y a pas d'états bloqués pour les noeuds correct. Il y a toujours une séquence d'événements à travers laquelle les noeuds corrects peuvent s'entendre et s'engager sur une valeur. Si les noeuds dépendent trop fortement des mauvais noeuds, ils sont appelés bloqués.
Dans ce protocole, les noeuds forment un ensemble dispensable, ce qui signifie que les noeuds corrects peuvent ratifier les déclarations sans la coopération des noeuds défectueux et les noeuds bloqués ne peuvent pas compromettre l'accord entre les noeuds corrects. Si aucun noeud intact n'a voté pour un bulletin de vote, ils passent à un bulletin de vote supérieur que ceux qu'ils ont promis d'avorter. Le manque de réponse des noeuds bloqués ne bloquera pas les noeuds corrects de l'assemblage des quorums et des progrès.

d) Lockstep :

Le protocole Lockstep [28, 29] est une solution partielle au problème de tricherie utilisé dans les jeux multijoueurs d'architecture peer-to-peer , dans lequel un client trompeur retarde ses propres actions pour attendre les messages d'autres joueurs. Un client peut le faire en agissant comme s'il souffrait d'une forte latence ; Le paquet sortant est forgé en fixant un horodatage avant le moment réel où le paquet est envoyé.
Pour éviter cette méthode de tricherie, le protocole de verrouillage nécessite que chaque joueur annonce d'abord un "engagement" (par exemple, une valeur de hash de l'action) ; Cet engagement est une représentation d'une action qui ne peut pas être utilisé pour inférer l'action, et ainsi comparez si celle-ci correspond à un engagement.
Une fois que tous les joueurs ont reçu les engagements, ils révèlent leurs actions, qui sont comparées aux engagements correspondants afin de s'assurer que l'engagement est effectivement l'action envoyée.

3.2) Protocole résultat

a) Élection des Leaders

Afin d'atteindre un certain niveau de stabilité dans la topologie dynamique du réseau en terme de coordination et d'organisation des noeuds, la première étape serait de créer un groupe de leaders proportionnel aux nombres de noeuds, élus grâce à l'algorithme de choix de PoET leader. Pour ce faire, chacun des noeuds va demander un temps d'attente à partir d'une enclave (un noeud de confiance, une mémoire partagée, un serveur de temps, un leader préexistant, une fonction de confiance...). Les noeuds avec les temps d'attente les plus courts sont élus leaders.

b) Les différents états d'un noeud :

Il existe cinq états dans lequel peut se retrouver un noeud :

- **Solitaire :** c'est quand le noeud n'a aucun leader et n'appartient à aucun groupe.
- **Leader temporaire :** il peut être soit un noeud solitaire qui vient de créer un groupe avec d'autre noeuds solitaire, au bien un leader élu dont le nombre de noeuds fils est inférieur à un certain chiffre.
- **Leader :** un noeud leader est un leader temporaire dont le nombre de noeuds fils est supérieur ou égale à un certain quota.
- **Noeud membre intermédiaire :** un noeud membre d'un groupe dans la profondeur est inférieur à la limite maximale, émet des pulsations pour garder les noeuds trop éloignés de la couverture du leader dans le groupe.
- **Noeud membre :** un noeud d'un groupe dans la profondeur a atteint la limite maximale n'émet plus de pulsation.

c) Propriété des noeuds

- Un noeud leader maintien une arborescence virtuelle dynamique qui le relie avec les noeuds qui se sont attachés à lui ;
- Pour se faire le leader informe régulièrement ses membres de son existence en envoyant des battements de coeur à des intervalles réguliers ;
- Une limite de profondeur pour un noeud peut être imposé, ainsi qu'une limite de noeuds fils pour un leader ;
  Remarque : si la limite de profondeur est égale à 1, alors le leader maintien une topologie en étoile.
- Un leader garde un historique des modifications faites, ainsi que les hash calculés correspondantes ;

- Un leader compare ces informations actuelles avec le hash contenu dans la pulsation d'un autre leader, et envoie une demande de mise à jour à ce leader si celle-ci est différente ou n'est pas présente dans son historique ;
- Pour qu'un leader approuve un nouveau leader celui-ci doit faire partie de son groupe de confiance (Quorum Slice).
- Un leader peut écouter les noeuds qui n'appartiennent pas à son groupe, au cas où il se retrouve éloigner des autres leaders ;
- Si un noeud est nouveau, il devient automatiquement solitaire.
- Chaque noeud connait sa profondeur dans l'arborescence ;
- On suppose qu'à chaque instant t, un noeud mobile connait ses coordonnées (x,y,z) dans l'espace ;
- Tout les noeuds possèdent un rayon de couverture limité ;
- Un noeud membre qui intercepte un message destiné à son leader, le fait remonté ;
- Un noeud membre qui intercepte un message de son leader, le fait propagé ;
- Chaque noeud ne fait confiance qu'à son Quorum Slice ;
- Chaque noeud possède une liste des leaders actives ;
- Un leader propage à des intervalles réguliers des messages à travers le réseau informant les autres de son existence.
- Chaque noeud garde une pile d'enregistrement d'une certaine taille, qui représente l'historique des noeuds rencontrés et de leurs statuts, qui sera utilisé pour bâtir son Quorum Slice ;

d) Gestion de l'ajout et de la coordination des noeuds

Le groupe présent de noeuds dans un réseau est devisé en un ensemble de clusters diriger chacun d'eux par un leader, et qui suis les mêmes règles que le protocole Raft pour maintenir sa position.

Un leader pour édifier son propre cluster de noeuds, est amené à construire sa propre arborescence couvrante dont il sera la racine :

- La diffusion d'une pulsation par un leader attire les noeuds solitaires les plus proche ;
- Chaque noeud fils a un délai d'attente dans lequel il attend la pulsation du leader. Le délai d'attente est réinitialisé lors de la réception d'une pulsation de leader, et le niveau de profondeur dans l'arborescence est mis à 1 . Si aucune pulsation n'est reçue, le noeud change son statut en solitaire ;

- Un noeud membre intermédiaire dans le niveau de profondeur dans l'arborescence est inférieur à la limite, fait propager des messages de synchronisation pour maintenir les noeuds éloignés du leader dans le groupe, contenant son niveau de profondeur et l'id de son leader ;
- Un neoud membre quelconque qui intercepte un message de synchronisation précédant d'un noeud membre intermédiaire, vérifie qu'ils ont le même leader, et si c'est le cas, il réinitialise le délai d'attente de la pulsation de leader et mit à jour son niveau de profondeur ;
- À des intervalles réguliers, un leader peut emmètre un message de synchronisation qui contient une requête de vérification et de mise à jour de la liste des noeuds qui lui sont encore associés ;
- Si le nombre de noeuds d'un groupe est inférieur à un certain chiffre, le leader change son statut en leader temporaire et les noeud se détacheront et rejoindront le premier autre leader rencontré. Cette option peut être désactivé ;
- Un noeud membre trop éloigné de son groupe et des autres leaders, change son statut en solitaire et émet une pulsation propre qui attirent les autres noeuds ;
- Si un noeud solitaire reçoit une pulsation d'un autre noeud solitaire celui-ci compare les temps d'attente et envoie une demande d'ajout si le sien est inférieur.
- Si un noeud solitaire reçoit une demande d'ajout d'un autre noeud solitaire, celui-ci confirme la demande d'ajout avec un ACK ;
- Si un noeud solitaire reçoit une ACK de demande d'ajout d'un autre noeud solitaire ou d'un leader, vérifie les informations du message, confirme son adhésion au groupe avec un ACK, et change son statut en noeud membre ;
- Un noeud solitaire qui reçoit un ACK d'adhésion au groupe, vérifie la demande, change son statut en leader temporaire et ajoute le noeud émetteur à la liste des noeuds fils, diagramme figure 3.1A ;
- Un leader qui reçoit un ACK d'adhésion au groupe, vérifie la demande, et ajoute le noeud émetteur à la liste des noeuds fils si son groupe est non plein, sinon il envoie NACK pour l'annulation de la demande d'ajout ;
- Un noeud membre qui reçoit un NACK d'annulation de demande d'ajout de son leader, change son statut en solitaire, diagramme figure 3.1B ;

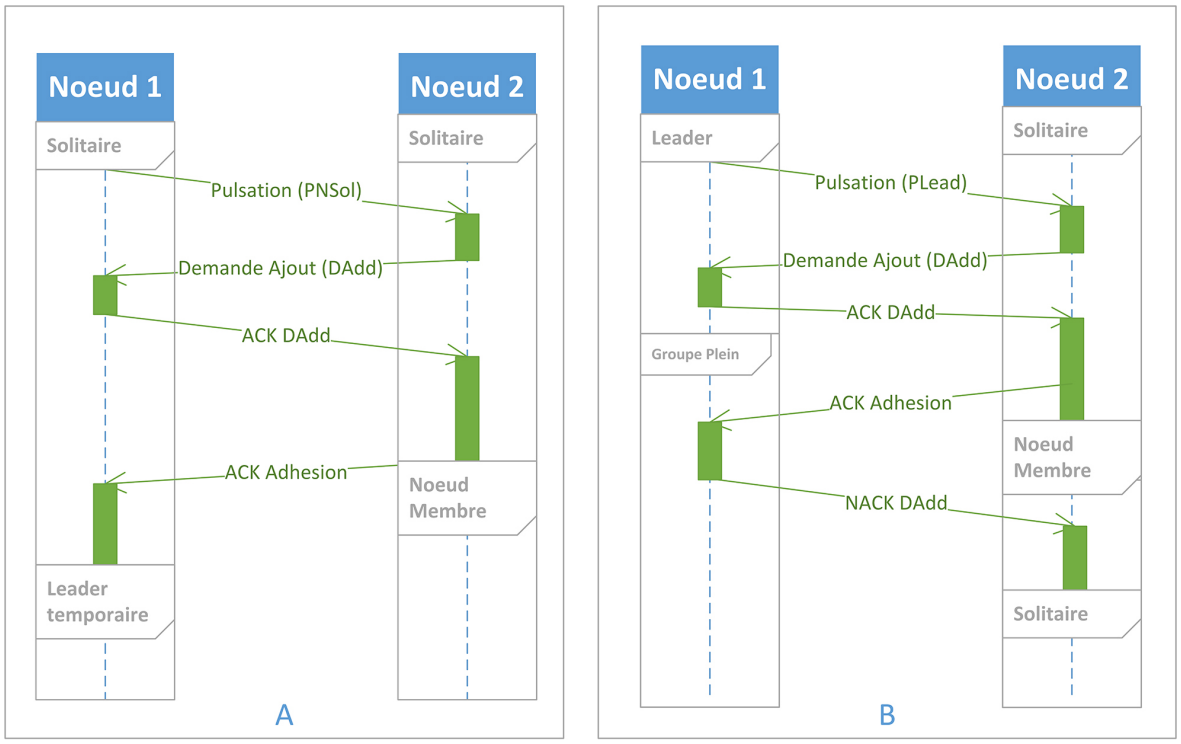


FIGURE 3.1 – Cas d'adhésion d'un noeud solitaire à un groupe.

- Un leader temporaire a une durée de vie prédéfinie par un paramètre, si celle-ci est atteinte et que le leader temporaire n'a pas atteint le quota de noeud membre, son groupe se disloquera ;
- Deux modes de comportement peuvent être adopté pour gérer un noeud qui est nouveau ou qui a perdu son groupe (noeud solitaire) et qui n'a aucun leader a proximité :
- **Mode NonForcé :**
  - Un noeud solitaire, vérifie constamment si des leaders non plein (limite de noeuds fils non atteinte) sont présents dans son rayon de couverture, et envoie une demande d'ajout au plus proche qui sera validée par un ACK ;
  - Si un noeud appartenant à un groupe reçoit une pulsation ou une demande d'ajout d'un noeud solitaire, il fait remonter le message à son leader si il sait que ce dernier est non plein, qui validera son ajout par un ACK, si la limite de noeud fils n'est pas atteinte entre temps.
- **Mode Forcé :**
  - Un noeud appartenant à un groupe qui intercepte une pulsation ou une demande d'ajout d'un noeud solitaire destiné à son leader, fera remonter le message à son leader qui confirmera son ajout par un ACK ignorant les limites de noeuds fils et de profondeur.

e) Le consensus

Les noeuds doivent maintenir un équilibre entre la vivacité et la sécurité. Nous voulons que le système soit réactif, mais pas au détriment de l'exactitude.

- Les noeuds manquent de vivacité lorsqu'ils se bloquent sur le chemin de l'accord, ralentissant ainsi le système.
- Les noeuds manquent de sécurité lorsqu'ils externalisent des valeurs incompatibles avec celles externalisées par d'autres noeuds, compromettant l'accord à l'échelle du système. Ces noeuds sont divergents.
- Un état divergent se produit lorsque les informations détenues par différents noeuds stockent des états contradictoires et inconciliables. Un système bloqué est moins dangereux qu'un divergent.

Notre consensus est atteint suite à un référendum (basé sur le protocole Stellar) entre les leaders, et dans ces derniers feront propager les informations vers les noeuds de leurs groupes, la gestion des envois des bulletins de vote aux niveaux des noeuds suivra le protocole Lockstep.

- Un noeud qui souhaite lancer un référendum doit obligatoirement passer par un leader qui dirige le groupe dans lequel il fait partie par un message DRef.
- Un noeud leader peut engager directement un référendum.
- Un leader qui veut lancer un référendum, informe les autres leaders en propageant un message Ref dans toutes les directions, et attend au minimum les ACKs des leaders dont il a connaissance.
- Un leader qui reçoit une demande de référendum, renvoie un ACK.
- Quand le leader qui a lancé le référendum n'a pas reçu tous les ACK attendus après un délai, il relance le référendum.
- Quand le leader qui a lancé le référendum reçoit tous les ACK attendu, il fait propager à nouveau un message ACK contenant une variable qui indique que le réseau est connexe, et commence la session de vote avec son groupe.
- Un leader qui reçoit l'ACK confirmant un réseau connexe, démarre la session de vote avec son groupe.
- La session de vote est ouverte par des messages SYN changeant l'état des noeuds fils.
- Des envois de message BVote entre noeuds fils et leader s'ensuit, jusqu'à ce qu'une majorité atteigne un certain pourcentage (définit comme paramètre dans l'algorithme).
- Chaque leader qui a fini sa session envoie son résultat et son poids qui indique le nombre de noeuds qu'il représente dans un message BVoteL aux autres leaders et attend le leurs.

- Les calculs se font localement pour chaque leader, il peut soit engager ou avorter un vote jusqu'à ce qu'il y ait consensus.

Chaque noeud est responsable de s'assurer de bon choix de son Quorum-Slice. Un bon quorum, partage les noeuds et aboutit à des quorums qui se chevauchent voir figure 3.2B. On appellera cette intersection "quorum de chevauchement". Lorsque les quorums ne se croisent pas, nous finissons avec des quorums disjoints voir figure 3.2A. Si les quorums sont disjoints, ils peuvent s'entendre de manière indépendante sur des déclarations contradictoires, les quorums disjoints peuvent miner le consensus.

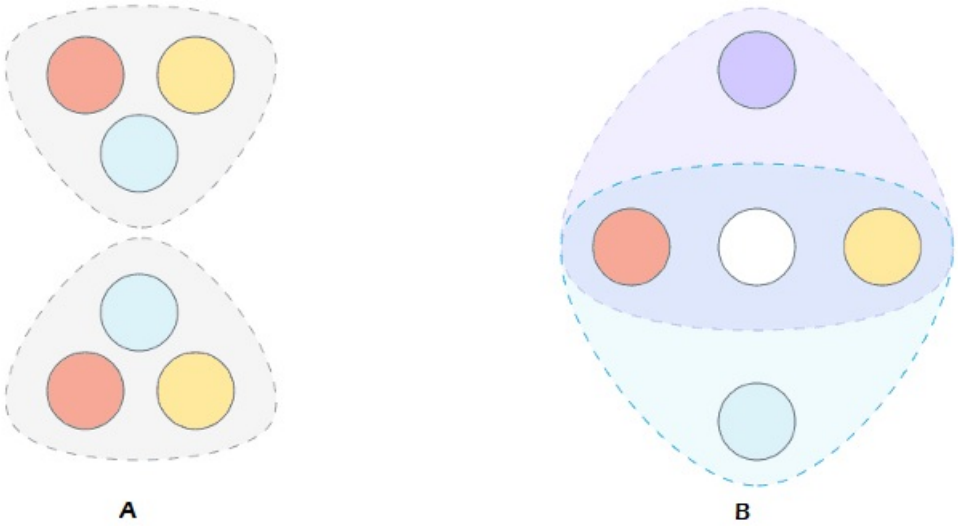


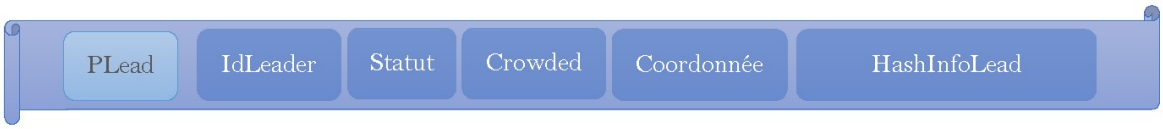
FIGURE 3.2 – Les Quorums Slices. [25]

Faire le bon choix c'est généralement de s'assurer que les tranches sont assez grandes et que les noeuds qu'elles contiennent sont suffisamment importants pour ne pas risquer l'intégrité de leur réputation en nourrissant des informations différentes pour différents processus.

f) Types trames de messages échangés

Différent type de message seront utilisé selon les événements dont un noeud pourrait être confronté :

i) Battement de coeur d'un Leader : Ce message est envoyé à des fréquences régulières permettant à un noeud de maintenir sa position de leader. Il contient un numéro de séquence unique identifiant cette requête, ainsi que son identifiant, un bit indiquant son statut (leader ou leader temporaire) et un autre indiquant si son groupe est plein ou non, ses coordonnées spatial (x,y,z), et un hash de l'ensemble des information propre à un leader.



D'autres messages peuvent être utilisé pour une meilleurs optimisation en termes d'économie d'énergie, qui s'alterne avec le premier modèle.

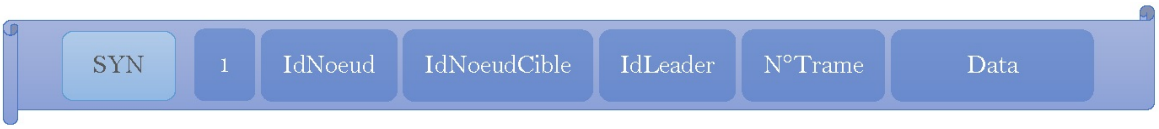
ii) Pulsation d'un noeud solitaire : Ce message est envoyé à des fréquences régulières pour attirer les autres noeuds. Il contient un numéro de séquence unique identifiant cette requête, ainsi que son identifiant, et son temps d'attente.

iii) Demande d'ajout à une arborescence : Ce message est envoyé pour une demande d'ajout, vers un leader ou un noeud solitaire dans le temps d'attente est inférieur. Il contient un numéro de séquence unique identifiant cette requête, ainsi que son identifiant, l'identifiant du noeud cible, ainsi que son temps d'attente.

iv) N-ACK : Ce message est un acquittement à un message qui le précède ou à son rejet. Il contient un numéro de séquence unique identifiant cette requête, un bit pour le type (0 pour NACK, 1 pour un ACK), l'identifiant du noeud émetteur, l'identifiant de noeud cible et les informations de message.

v) Lock : Ce message précède chaque envoie d'un autre message important, qui représente un engagement pour une action. Il contient un numéro de séquence unique identifiant cette requête, l'identifiant du noeud émetteur, l'identifiant du noeud cible et le hash du prochain message.

vi) Synchronisation : Ce message est utilisé pour échanger des données, ou comme demande ou réponse à une synchronisation ou une mise à jour. Il contient un numéro de séquence unique identifiant cette requête, ainsi qu'un bit pour indiquer le type de synchronisation (0 une seul trame, 1 plusieurs trames), son identifiant, l'identifiant de noeud cible, l'id du leader de l'émetteur, le Nř de la trame si le bit du type de synchronisation est à 1, et les données, (si le noeud émetteur est un leader et les bits de noeud cible sont à 0, alors il s'agit d'une synchronisation destiné à tout son groupe).



vii) Demande d'un référendum : Ce message est envoyé par un noeud vers son leader pour demander le démarrage d'un référendum. Il contient un numéro de séquence unique identifiant cette requête, ainsi que son identifiant, l'identifiant de son leader, et le sujet de référendum.

viii) Référendum : Ce message est envoyé par un leader pour avertir touts les leaders connus d'un éventuel référendum. Il contient un numéro de séquence unique identifiant cette requête, ainsi que l'identifiant du leader, l'identifiant du noeud cible (0 pour tous les noeuds), le Nř de référendum et le sujet de référendum.

ix) Bulletin de vote d'un noeud : Ce message est envoyé comme bulletin de vote pour engager une décision ou avorter un autre bulletin de vote par un noeud. Il contient un numéro de séquence unique identifiant cette requête, ainsi que son identifiant, l'identifiant de son leader, le type de décision (Avorter/Engager) et la réponse.

x) Bulletin de vote d'un leader : Ce message est envoyé comme bulletin de vote pour engager une décision ou avorter un autre bulletin de vote, par un leader vers un autre leader. Il contient un numéro de séquence unique identifiant cette requête, ainsi que l'identifiant du leader émetteur, l'identifiant de leader cible, son poids et la réponse prise par son groupe.

3.3) Paramètres de l'algorithme de consensus

Comme dans n'importe quel protocole, des paramètres peuvent être appareillé à l'algorithme de consensus (comme dans notre cas : le délai d'attente entre deux pulsations d'un leader, la profondeur maximale dans une arborescence, le nombre max de noeuds dans un groupe...). Par conséquent, l'optimisation du choix de ces paramètres conduit à une meilleure performance dans les termes de l'économie d'énergie, la vitesse de convergence, le nombre de messages engendré ou de la robustesse du système au bruit. En plus de l'optimisation, le contrôle des états des agents est très important. Dans certains cas, le mauvais choix des paramètres peut entraîner la divergence des états et la déstabilisation du système. En fait, la conception des règles locales d'interaction pour les agents est l'un des principaux objectifs de l'implémentation de notre protocole du consensus.

a) La vie d'un leader :

Pour pallier aux problème d'un trop grand nombre de leader, on utilisera des paramètres que ces derniers devront respecter pour garder leurs statuts.

1. **Nombre minimum de noeuds fils :** si le nombre de noeuds fils en est inférieur à ce chiffre, il change son statut en leader temporaire ;

2. **Délai de vie d'un leader temporaire :** chaque leader temporaire à une limite de temps dans le quelle il devra atteindre le quota de noeuds fils pour devenir ou redevenir leader, sinon son statut changera en solitaire.

Algorithme 3.1 - Surveillance de l'état d'un leader temporaire ou d'un leader

```
1 : Si (monStatut = LEADER) alors
2 :   Si ((noeudsFils.taille < minNoeudsFils) et!referendumEnCours) alors
3 :     monStatut ← LEADER_TEMPORAIRE;
4 :   fin Si
5 : fin Si
6 : //- - - - - - - - - - - - - -
7 : Si (monStatut = LEADER_TEMPORAIRE) alors
8 :   Si (noeudsFils.taille >= minNoeudsFils) ) alors
9 :     monStatut ← LEADER;
10 :  sinon Si (lastTimeStatutChangeToLeaderTmp + delaiVieLeaderTmp < temps) alors
11 :    monStatut ← SOLITAIRE;
12 :  fin Si
13 : fin Si
```

b) Purge de la liste des noeuds fils d'un leader :

Un leader peut être amené à mettre à jour sa liste de noeuds fils encore membre de groupe si celui-ci se retrouve plein (nombre de noeuds fils max atteint) ou bien avant une session de vote.

Deux variables sont utilisées :

- Le délai à attendre entre chaque purge ;

- Le délai maximum d'attente d'une réponse des noeuds encore membre, avant la mise à jour des listes ;

Algorithme 3.2 - Fonction de purge de la liste des noeuds fils qui ne font plus partie de groupe

```
1 : Si (monStatut = LEADER) alors
2 :   Si ( ( purgeNoeudFilsLastTime + delaiPurgeNoeudFils < temps ) ou purgeNoeudFilsEnCours ) alors
3 :     envoyerMessage(SYN, Tout_Les_Membre, "noeud membre") ;
4 :     Si (!purgeNoeudFilsEnCours) alors
5 :       purgeNoeudFilsEnCours ← vrai ;
6 :     sinon Si (purgeNoeudFilsLastTime + delaiPurgeAttenteNoeudFils < temps) alors
7 :       noeudsFils ← noeudsFilsPrsent ;
8 :       purgeNoeudFilsEnCours ← faux ;
9 :       purgeNoeudFilsLastTime ← temps ;
10 :    fin Si
11 :  fin Si
12 : fin Si
```

c) Changer pour un leader plus proche :

Un noeud membre qui possède un leader et qui reçoit une pulsation d'un autre, vérifie si le groupe de ce dernier n'est pas plein, compare les différentes distances, et change de leader si l'ancienne et plus longue de n×B (n étant définie à l'avance et B l'unité de mesure). Cette option peut être désactivé.

Le noeud membre suis le même protocole que pour un noeud solitaire pour la demande d'ajout destiné au leader.

3.4) Algorithme

Notre algorithme de consensus additionné à celui de la coordination des noeuds est une suite de réactions à des événements qui est la réception des messages.
Selon le statut d'un noeud, ces messages peuvent être traité différemment, voir même ignoré

a) Surveiller l'état d'un noeud

Avant chaque traitement de la file d'attente des messages reçus, le noeud mit à jour son statut selon les états de ses variables. Si c'est un noeud membre il vérifie en premier lieu si la pulsation de son leader est présente dans la file d'attente des messages.

Algorithme 3.3 - Surveillance et changement de statut selon les états des paramètres

```
1 : Si ((monStatut = LEADER) ou (monStatut = LEADER_TEMPORAIRE)) alors
2 :   Si (lastTimePulsation + tempEntre2Pulsation <= temps) alors
3 :     envoyerMessage(PLead, ToutLeMonde);
4 :     monLeader.lastTimePulsation ← temps;
5 :   fin Si
6 : fin Si
7 : //- - - - - - - - - - -
8 : Si ((monStatut = NOEUD_MEMBRE) ou (monStatut = NOEUD_MEMBRE_INTERMEDIAIRE)) alors
9 :   Pour (i = 0; i < messagesRecu.taille; i++) faire
10 :    Si ((messageRecu[i].type = PLead) et (messageRecu[i].idNoeud = monLeader.id)) alors
11 :      monLeader.lastPulsation ← temps;
12 :      monLeader.groupePlein ← messageRecu[i].groupePlein;
13 :      monNiveauProfondeur ← 1;
14 :      monLeader.coordonne ← messageRecu[i].coordonne;
15 :    fin Si
16 :    Si ((messageRecu[i].type = SYN) et (monStatut = NOEUD_MEMBRE) alors
17 :      Si ((messageRecu[i].infoMessage = "Intermediaire") et (messageRecu[i].idLeader = monLeader.id)) alors
18 :        monLeader.lastPulsation ← temps;
19 :        monNiveauProfondeur ← messageRecu[i].niveauProfondeur + 1;
20 :      fin Si
21 :    fin Si
22 :  fin Pour
23 :  Si ((monLeader.lastPulsation + delaiAttentePulLeader < temps) ou (lastTimeStatutChange + delaiReinitLienLeader < temps )) alors
24 :    monStatut ← SOLITAIRE;
25 :  fin Si
26 : fin Si
27 : Si (monStatut = NOEUD_MEMBRE) alors
28 :  Si (monNiveauProfondeur < limitProfondeur) alors
29 :    monStatut ← NOEUD_MEMBRE_INTERMEDIAIRE;
30 :  fin Si
31 : fin Si
32 : //- - - - - - - - - - -
33 : Si (monStatut = NOEUD_MEMBRE_INTERMEDIAIRE) alors
34 :  Si ((limitProfondeurArbre!= 1) et (monNiveauProfondeur < limitProfondeur)) alors
35 :    envoyerMessage(SYN, ToutLeMonde, monLeader.id, "Intermidiaire");
```

```
36 :    sinon Si (limitProfondeur == 1) alors
37 :      monStatut ← NOEUD_MEMBRE;
38 :      monNiveauProfondeur ← 1;
39 :    sinon Si (monNiveauProfondeur >= limitProfondeur) alors
40 :      monStatut ← NOEUD_MEMBRE;
41 :    fin Si
42 : fin Si
43 : //- - - - - - - - - - -
44 : Si (monStatut = SOLITAIRE) alors
45 :    Si (lastTimePulsation + tempEntre2Pulsation <= temps) alors
46 :      envoyerMessage(PNSol, ToutLeMonde);
47 :      lastTimePulsation ← temps;
48 :    fin Si
49 :    monLeader.id ← 0;
50 :    monLeader.existe ← faux;
51 : fin Si
```

b) Noeud Solitaire

Si le noeud est solitaire il ne traite que les messages PLead, PNSol, DAdd et ACK.

Algorithme 3.4 - Adhésion ou création d'un groupe par un noeud Solitaire

```
1 : Si (monStatut = SOLITAIRE) alors
2 :   Si (messageReçu[i].type = ACK) alors
3 :     Si ((messageReçu[i].etat = vrai) et (messageReçu[i].idNoeudCible = monId)) alors
4 :       Si (messageReçu[i].infoMessage = "ACK->DAdd") alors
5 :         envoyerMessage(ACK, vrai, messageReçu[i].idNoeud, "ACK->DAdd<-ACK");
6 :         monStatut ← NOEUD_MEMBRE_INTERMEDIAIRE;
7 :         monLeader.id ← messageReu[i].idNoeud;
8 :         monLeader.lastPulsation ← temps;
9 :         monLeader.coordonnee ← messageReu[i].coordonnee;
10 :        niveauProfondeur ← messageReu[i].niveauProfondeur;
11 :        break;
12 :      fin Si
13 :    sinon Si (messageReçu[i].infoMessage = "ACK->DAdd<-ACK") alors
14 :      listeNoeudsFils.ajouter(messageReçu[i].idNoeud);
15 :      monStatut ← LEADER_TEMPORAIRE;
16 :      niveauProfondeur ← 0;
17 :      break;
18 :    fin Si
19 :    //- - - - - - - - - - -
20 :  sinon Si (messageRecu[i].type = PNSol) alors
21 :    Si (messageRecu[i].tempsAttente <= monTempsAttente) alors
22 :      envoyerMessage(DAdd, messageRecu[i].idNoeud);
23 :    fin Si
24 :    //- - - - - - - - - - -
25 :  sinon Si (messageRecu[i].type = Plead) alors
26 :    Si ((!messageRecu[i].groupePlein) ou modeParainageForcé) alors
27 :      envoyerMessage(DAdd, messageRecu[i].idNoeud);
28 :    fin Si
29 :    //- - - - - - - - - - -
30 :  sinon Si (messageRecu[i].type = DAdd) alors
31 :    Si (messageReçu[i].idNoeudCible = monId) alors
32 :      envoyerMessage(ACK, true, messageRecu[i].idNoeud, "DAdd");
33 :    fin Si
34 :  fin Si
```

c) Noeud Leader ou Leader Temporaire

Si le noeud est un leader ou un leader temporaire, il parcourt les messages concernant les demandes d'ajout, de demandes de référendum, des bulletins de vote et de leurs traitements, selon le protocole concernant l'ajout de noeud et de consensus décrit précédemment.

Algorithme 3.5 - Traitement des événements par un leader selon les messages reçus

```
1 : Si ((monStatut = NOEUD_MEMBRE) ou (monStatut = NOEUD_MEMBRE_INTERME-
    DIAIRE)) alors IF(messageRecu[i].type = Plead)
2 :   leaderEnActivite(true, messageRecu[i].idNoeud);
3 : sinon Si (messageReçu[i].type = PNSol) alors
4 :   Si ((noeudsFils.size() < limiteNoeudFils) ou (modeParainageForcé)) alors
5 :     envoyerMessage(ACK, true, messageRecu[i].idNoeud, "ACK->DAdd");
6 :   fin Si
7 : sinon Si (messageRecu[i].type = DAdd) alors
8 :   Si (messageReçu[i].idNoeudCible = monId) alors
9 :     Si ((noeudsFils.size() < limiteNoeudFils) ou (modeParainageForcé)) alors
10 :      envoyerMessage(ACK, true, messageRecu[i].idNoeud, "ACK->DAdd");
11 :    fin Si
12 :  fin Si
13 : sinon Si (messageRecu[i].type = ACK) alors
14 :  Si (messageReçu[i].idNoeudCible = monId) alors
15 :    Si (messageReçu[i].infoMessage = "ACK->DAdd<-ACK") alors
16 :      Si ((noeudsFils.size() < limiteNoeudFils) ou (modeParainageForcé)) alors
17 :        listeNoeudsFils.ajouter(messageReçu[i].idNoeud);
18 :      sinon
19 :        envoyerMessage(ACK, faux, messageReçu[i].idNoeud, "ACK->DAdd");
20 :      fin Si
21 :    fin Si
22 :    Si (messageReçu[i].infoMessage = "ACK->Ref") alors
23 :      test ← faux;
24 :      Pour (j = 0; j < listeLeaderConnu.taille; j++) faire
25 :        Si (listeLeaderConnu[j] = messageReçu[i].idNoeud) alors
26 :          test ← vrai;
27 :        fin Si
28 :      fin Pour
29 :      Si (test) alors
30 :        ACKLeadersEnAttente++;
```

```
31 :          fin Si
32 :        fin Si
33 :      fin Si
34 : sinon Si (messageRecu[i].type = LOCK) alors
35 :      ajouterListeMessageEnAttente(messageReçu[i].idNoeud, hash);
36 : sinon Si (messageReçu[i].type = SYN) alors
37 :      Si ((messageReçu[i].idNoeudCible = monId) et purgeNoeudFilsEnCours) alors
38 :        Si (messageReçu[i].infoMessage = "monPére") alors
39 :          test ← faux;
40 :          Pour (j = 0; j < noeudsFils.taille; j++) faire
41 :            Si (noeudsFils[j] = messageReçu[i].idNoeud) alors
42 :              test ← vrai;
43 :            fin Si
44 :          fin Pour
45 :          Si (test) alors
46 :            noeudsFilsActif.ajouter(messageReçu[i].idNoeud);
47 :          fin Si
48 :        fin Si
49 :      sinon Si (messageRecu[i].type = DRef) alors
50 :        memebre ← faux;
51 :        Pour (j = 0; j < noeudsFils.taille; j++) faire
52 :          Si (noeudsFils[j] = messageReçu[i].idNoeud) alors
53 :            memebre ← vrai;
54 :          fin Si
55 :        fin Pour
56 :        Si (membre) alors
57 :          demarerReferendum ← vrai;
58 :          envoyerMessage(Ref, ToutLesLeaders, "Referendum");
59 :        fin Si
60 :      sinon Si (messageReçu[i].type = Ref) alors
61 :        messageAEnvoyer.ajouter(messageReçu[i]);
62 :        envoyerMessage(N-ACK, messageReçu[i].idNoeud, "Referendum");
63 :      sinon Si (messageReçu[i].type = NBVote) alors
64 :        Si (SessionVote) alors
65 :          traitementVoteNoeud(messageReçu[i]);
66 :        fin Si
67 :      sinon Si (messageReçu[i].type = LBVote) alors
68 :        Si (SessionVote) alors
69 :          traitementVoteLeader(messageReçu[i]);
70 :        fin Si
```

d) Noeud Membre ou membre intermédiaire

Si le noeud est un membre ou membre intermédiaire d'un groupe, il attend les sessions de vote, et réémet les messages qui sont destinés à son leader ou venant de celui-ci, s'il lui sont pas directement adressé.

Algorithme 3.6 - Traitement ou retransmission des messages selon les cas

```
1 : Si ((monStatut = NOEUD_MEMBRE) ou (monStatut = NOEUD_MEMBRE_INTERMEDIAIRE)) alors
2 :   Si (messageReçu[i].type = DAdd)) alors
3 :     Si (((messageReçu[i].idNoeud = monLeader.id) ou (messageReçu[i].idNoeudCible = monLeader.id)) et (messageReçu[i].idNoeud = monId)) alors
4 :       Si ((modeParainageForce) ou (monNiveauProfondeur < limitProfondeur) et (!monLeader.groupePlein)) alors
5 :         envoyerMessage(messageReçu[i]);
6 :       fin Si
7 :     sinon Si (messageReçu[i].type = PNSol) alors
8 :       Si ((modeParainageForce) ou ((niveauProfondeur < limitniveauProfondeur) et (!monLeader.groupePlein))) alors
9 :         envoyerMessage(messageReçu[i]);
10 :      fin Si
11 :    sinon Si (messageReçu[i].type = SYN) alors
12 :      Si (messageReçu[i].idNoeud = monLeader.id) alors
13 :        Si (messageReçu[i].infoMessage = "monFils") alors
14 :          envoyerMessage(SYN, messageReçu[i].idNoeud, "monPére");
15 :        fin Si
16 :      sinon Si (messageReçu[i].type = ACK) alors
17 :        Si (((messageRecu[i].idNoeud = monLeader.id) et (messageRecu[i].idNoeudCible != monId)) ou (messageRecu[i].idNoeudCible = monLeader.id)) alors
18 :          envoyerMessage(messageReçu[i]);
19 :        sinon Si ((messageRecu[i].idNoeudCible = monId) et (messageRecu[i].infoMessage = "ACK->DAdd") et !messageRecu[i].etat) alors
20 :          monStatut ← SOLITAIRE;
21 :        fin Si
22 :      sinon Si (messageReçu[i].type = LOCK) alors
23 :        Si (messageRecu[i].idNoeud = monLeader.id) alors
24 :          envoyerMessage(messageReçu[i]);
25 :        fin Si
26 :      fin Si
```

3.5) Organigramme

Selon l'état d'un noeud (Leader, noeud fils d'un leader, noeud chercheur ou solitaire) la réactivité et le comportement peuvent être différentes.

Le premier organigramme (figure 3.3) montre les réaction d'un noeud pour chaque situation possible, aux variations dynamique et aléatoire de réseau qui peuvent l'affecter directement.

Le deuxième organigramme (figure 3.4) montre les réaction d'un noeud aux différents messages qui peuvent lui être envoyé.

Le troisième organigramme (figure 3.5) montre les réaction d'un leader aux différents messages qui peuvent lui être envoyé, sauf ceux concernant le référendum qui seront décrit plus en détails dans le scénario d'execution.

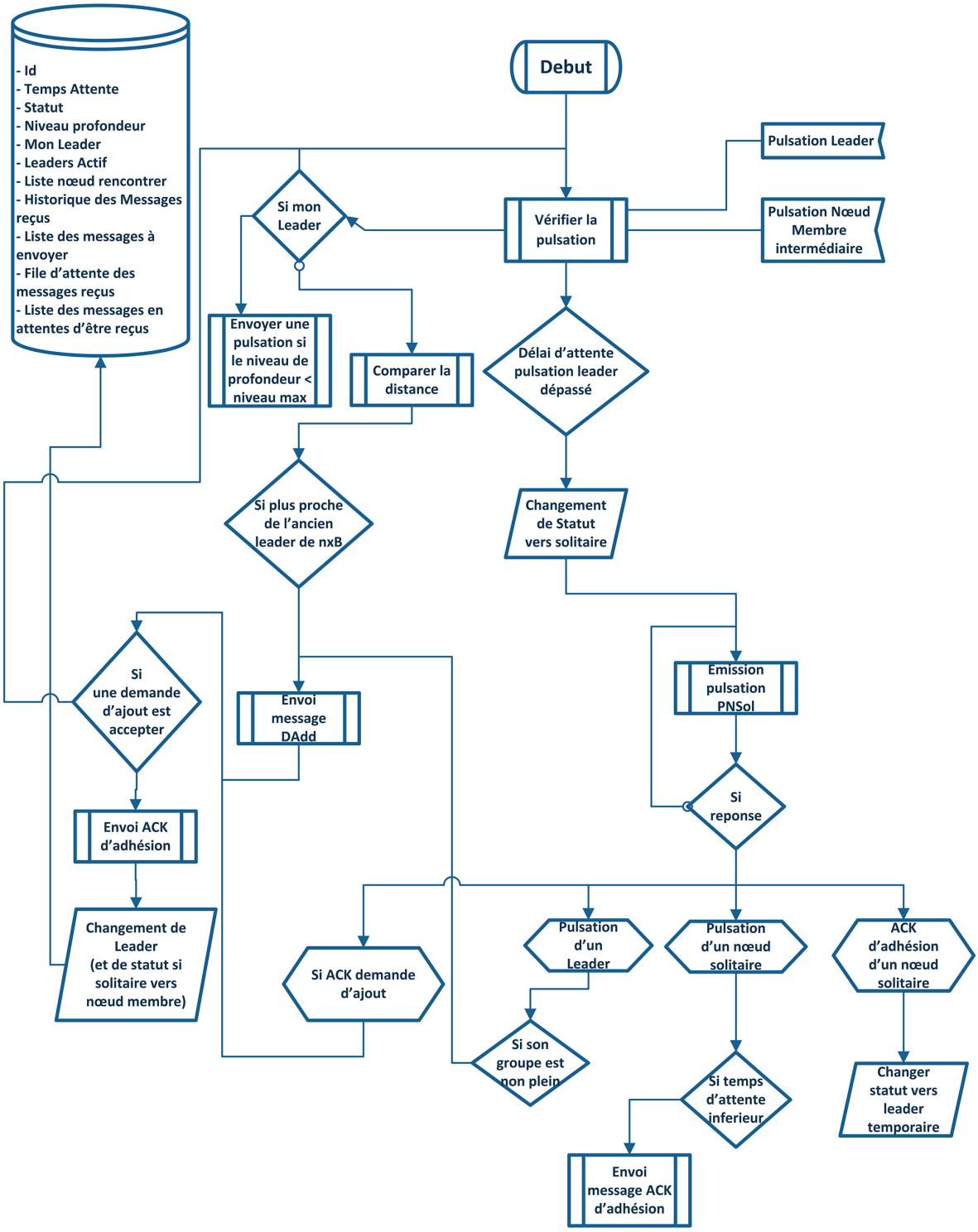


FIGURE 3.3 – Organigramme des changement d'état d'un noeud.

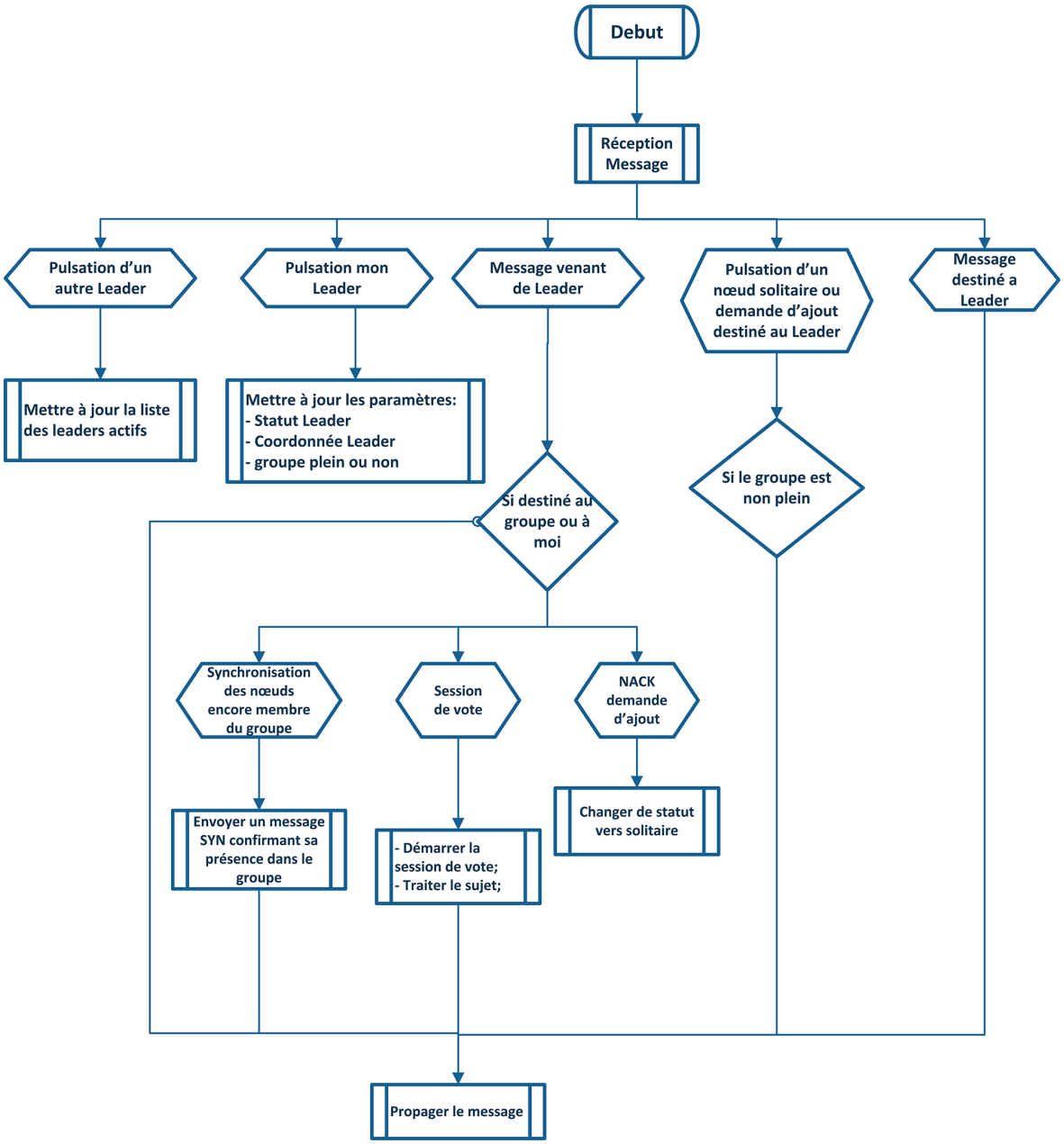


FIGURE 3.4 – Organigramme des réactions d'un noeud aux différent message reçu.

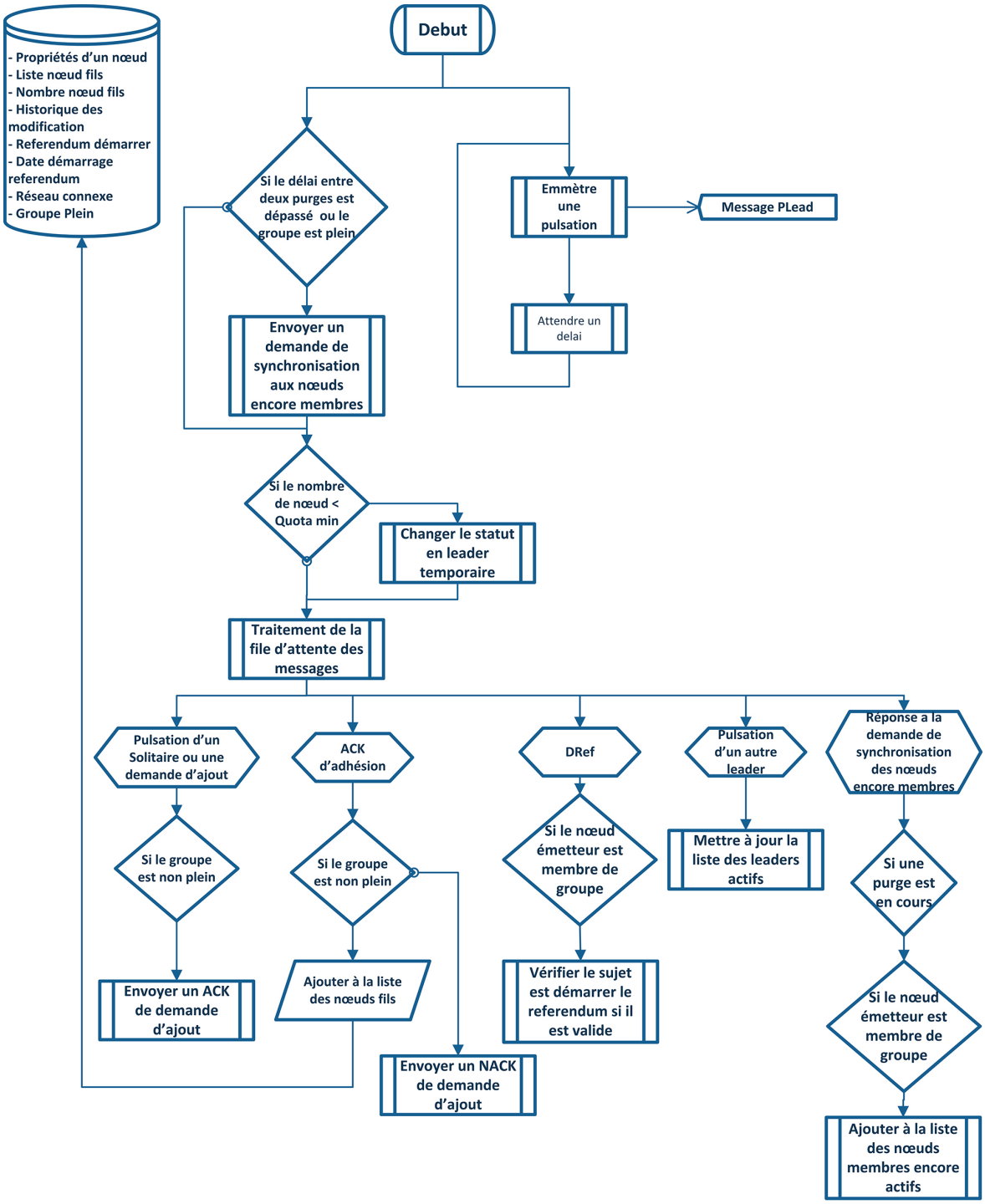


FIGURE 3.5 – Organigramme des réaction d'un leader aux différent message reçu.

3.6) Scénario d'execution

Pour concrétiser notre proposition, voici un scénario global d'exécution illustré par des diagrammes de séquence du langage UML.

Le premier diagramme illustre le démarrage et l'execution d'un consensus simple. Lancé par un noeud correct, dans un système où il y a deux leaders actifs. Le deuxième diagramme illustre la phase des échanges des bulletins de vote entre trois noeuds et leurs leaders, suivit d'une description.

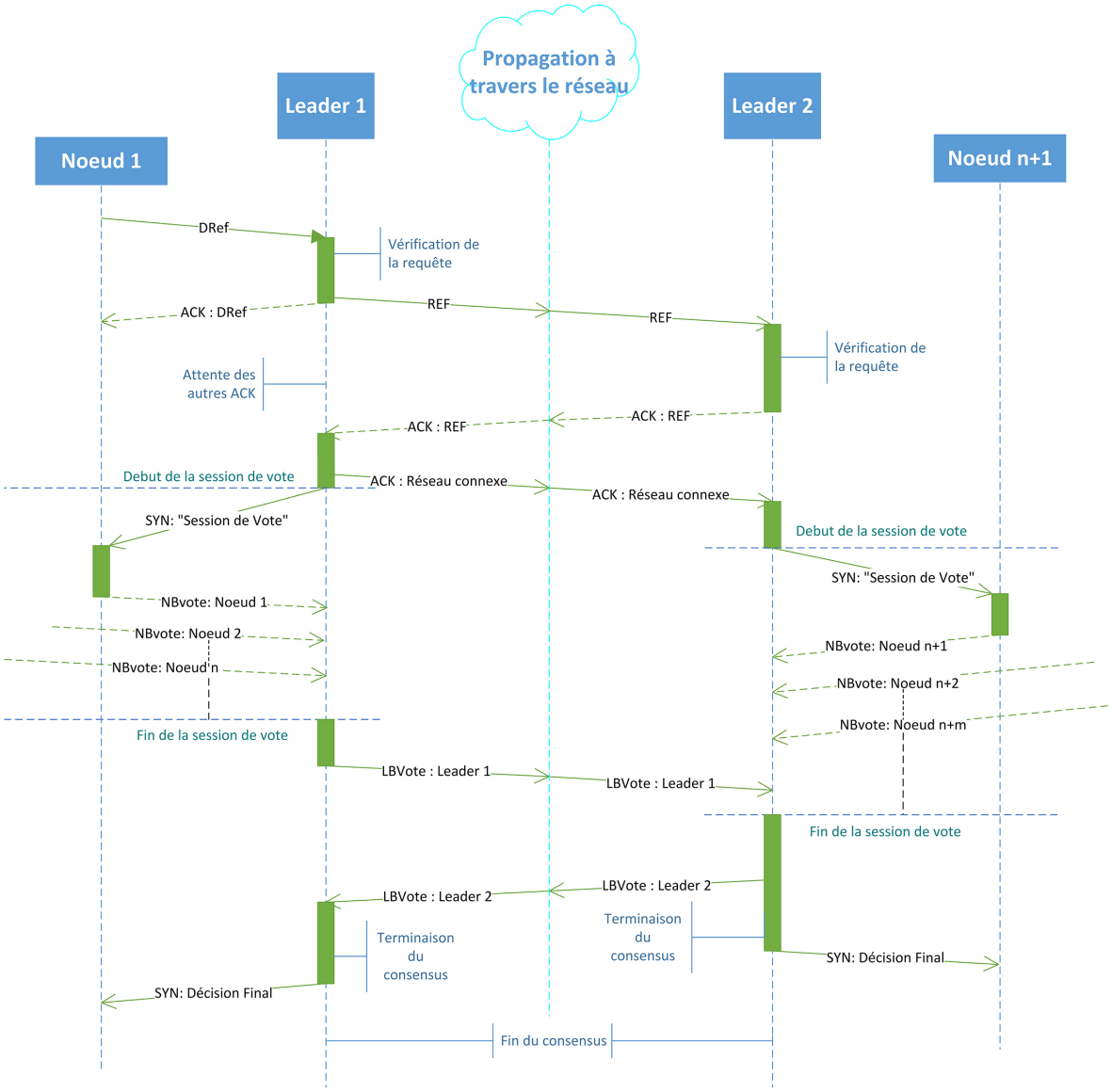


FIGURE 3.6 – Diagramme de séquence de démarrage et de l'exécution d'un consensus.

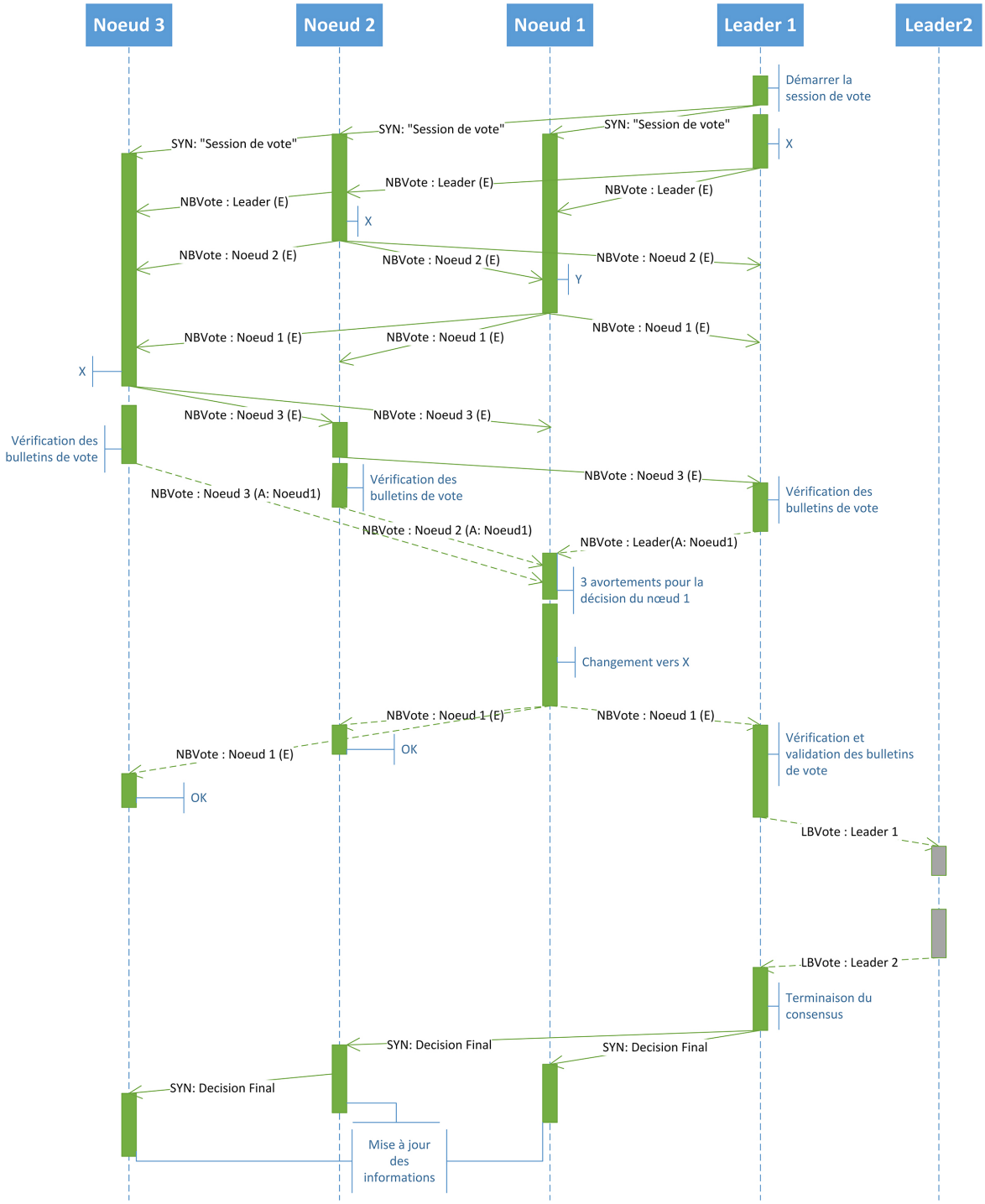


FIGURE 3.7 – Diagramme de séquence des échanges des bulletins de vote entre trois noeuds et leurs leader (E : engager, A : avorter).

Description :

- ***Vote Initial :***

  Disons que le noeud1 souhaite engager le chiffre Y, mais doit rester ouverte dans le cas où une grande partie du groupe vote pour l'une des autres options.

  Le vote est préliminaire et ne se produit qu'où niveau du noeud. Dans la première étape du processus de vote, le noeud1 Votes pour rester ouvert à la possibilité d'ac-

cepter Y en affirmant que X est valable et promettant qu'il n'a jamais et ne votera jamais individuellement pour toute option contredisant Y. Il peut cependant finir par accepter quelque chose d'autre que X si suffisamment de noeuds votent autrement (pression des pairs).

- ***Acceptation :*** Grâce à l'intersection du quorum (figure 3.2B), les tranches s'influencent les unes les autres. Si un autre chemin dans lequel le noeud2, le noeud3 et le Leader initialement vote pour X. Et si le noeud2, le noeud3 et le Leader sont en quorum avec le noeud1, ils peuvent bloquer la progression de l'acceptation de Y. Un ensemble de noeuds S contient au moins un noeud de chacune des tranches de noeud1 et peut bloquer l'action dans tous les quorums qui contiennent le noeud1, obligeant ainsi le noeud1 à accepter X à la place.
  Le noeud1 accepte X si :

  a) Il n'a jamais accepté une déclaration en contradiction avec X (par exemple, Z) ;

  b) Chaque membre de l'ensemble S prétend accepter X, ou chaque membre d'un quorum comprenant le noeud1, vote pour ou prétend accepter X.

- ***Ratification :***
  Lorsque chaque membre d'un quorum vote pour X, nous disons que le quorum ratifie X. Un noeud n'a pas besoin de ratifier une déclaration de première main.
  Par exemple, le noeud3 s'appuie souvent sur le noeud2 et le noeud1 pour décider quoi voter. Ils sont dans son quorum. Si tous les trois votent pour X, le quorum a ratifié X.
  Un noeud peut voter pour une option et plus tard accepter une contradiction.

- ***Messages de confirmation :***
  La confirmation est la dernière étape du processus de vote et implique un accord à l'échelle du système.
  Pour assurer de l'accord, les noeuds échangent des messages de confirmation. Le leader accepte une déclaration si, une fois que les messages sont livrés et traités, et peu importe les événements ultérieurs, chaque noeud réactif et précis acceptera l'énoncé.
  Le noeud2, par exemple, peut affirmer qu'il a accepté X en envoyant le message BVote, "Vote (X)".
  Lorsque le noeud2 envoie ce message, le noeud1 et le noeud3, diffusent "vote (X)".
  Ces messages peuvent convaincre des processus (ou agents) supplémentaires d'accepter X. Ces processus supplémentaires convaincront autant que possible, en diffusant "vote (X)" jusqu'à ce que tous acceptent X.
  Un message de synchronisation lancer ultérieurement par le leader permettra aux noeuds de confirmer X.

4) Cas d'étude

Nous adapterons notre système à un cas d'étude concret afin de mieux voir son fonctionnement. Nous avons opté pour le problème simple d'un partage équitable d'un espace entre des noeuds mobile.
Les problèmes liés à la coordination multi-agents, où des agents du réseau utilisent des technologies de capteur sans fil pour communiquer, par exemple un groupe de drones qui se déplace en parallèle devrait s'accorder sur la direction du mouvement et la vitesse pour éviter les collisions. Le Problème du contrôle de la formation, avec une approche sans tête, où les drones ne communiquent qu'avec leurs voisins les plus proches pour accomplir collectivement une tâche globale (comme l'évitement d'un obstacle ou de la trajectoire suivante, tout en maintenant la connectivité).
La principale difficulté pour les protocoles de consensus dans cette catégorie ne réside pas dans le grand nombre d'agents mobile, mais plutôt dans la topologie de commutation et les problèmes de connectivité.

Chapitre 4

Simulation de la solution apportée

« *En programmation, tout ce que nous faisons est un cas particulier de quelque chose de plus général - et souvent nous nous en apercevons trop vite.* »

Epigrams on Programming

Sommaire

1) Présentation de la simulation 61

1.1) Organisation des noeuds 63

1.2) Consensus 64

1.3) Diagramme de classes 66

2) Vérification des propriétés 68

3) Vérification des Objectifs 73

4) Synthese 74

5) Conclusion 75

Cette partie du mémoire est consacrée à la phase de test et de validation de la solution présentée précédemment. Pour cela nous commencerons avec la présentation de simulateur conçu avec l'API graphique OpenGL [35], suivie d'un scénario d'exécution sur le cas décrit au chapitre précédant, puis nous passons à la vérification des propriétés de vivacité, d'exactitude et de terminaison. Nous calculerons également les temps de réponses et le nombre de messages générés.

1) Présentation de la simulation

Nous avons exploité l'environnement de développement Microsoft Visual Studio 2017, on utilisant le langage de programmation C++ compléter avec Lua [37] qui est un langage de scripting très rapide et très connu pour son utilisation en intelligence artificielle.
Le script lua nous servira à modifier les paramètres de l'algorithme sans devoir recompiler ce dernier.

Nous avons choisi la bibliothèque OpenGL (voir Annexe), pour l'implémentation de la scène où se déroule la simulation. OpenGL est une API Graphique complète pour la création d'animations 2D ou 3D, très utilisé dans l'industrie graphique, ainsi que dans la conception de beaucoup de logiciel CAO (Conception Assisté par Ordinateur), de simulateurs et de jeux vidéo.

Nous avons également eu recours à la bibliothèque QT [36] pour l'éditeur de script et pour la surveillance et l'affichage du journal d'événements.

L'application se décomposera en deux fichiers exécutables distincts et communicants entre eux par l'intermédiaire d'une zone mémoire partagée.

La première interface nous servira à manipuler le scripte et à afficher les différentes informations générées par l'application figure 4.1.

La deuxième fenêtre affichera une scène en 3D, qui contiendra des drones en mouvements disposés aléatoirement dans l'espace. La fenêtre contient aussi une mini-carte en 2D indiquant la position de chaque noeud dans l'espace figure 4.1.

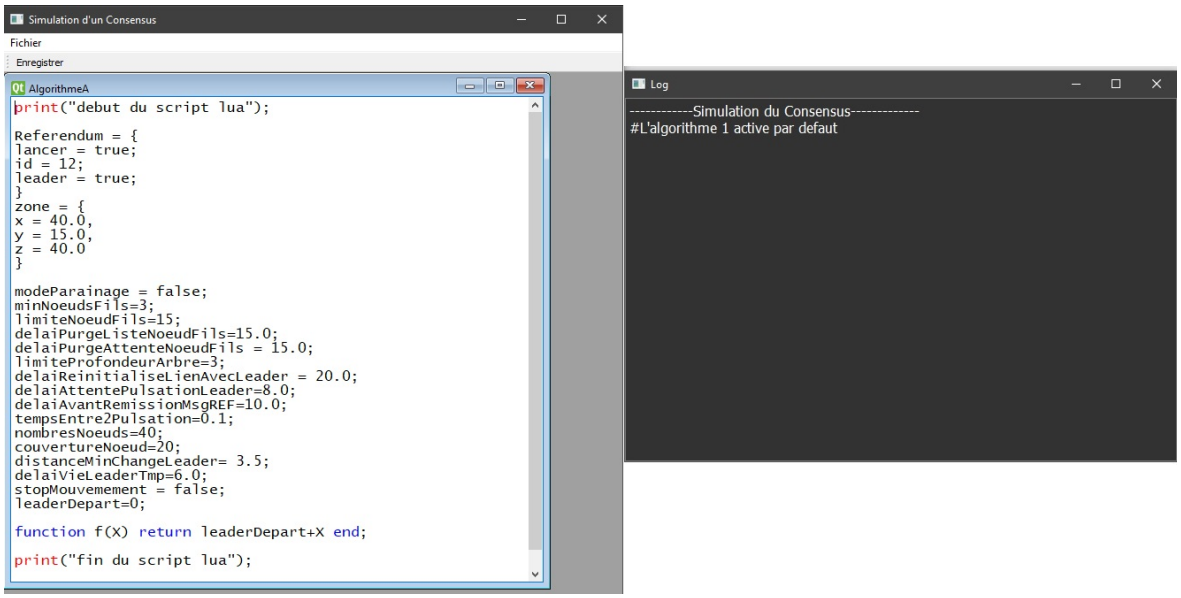


FIGURE 4.1 – Fenêtre de manipulation de script Lua et d'affichage de Log.

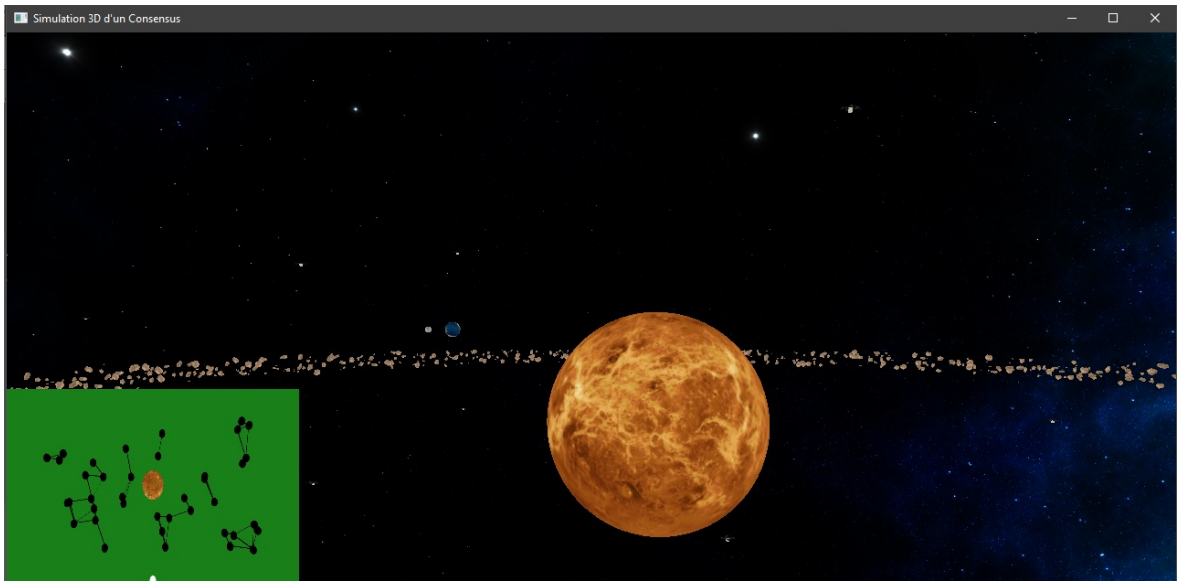
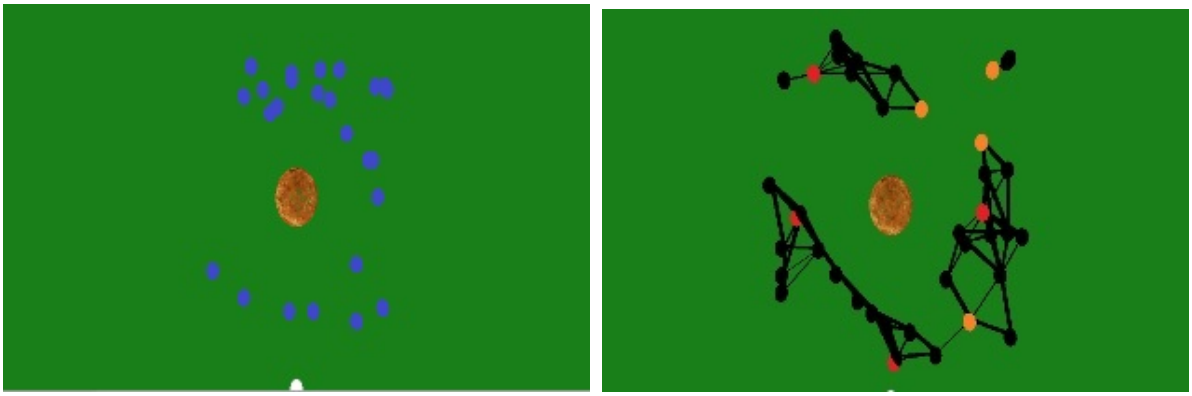


FIGURE 4.2 – Scène principale de la simulation.

Les différentes informations générées au cours de l'exécution seront stockées dans des fichiers externes, ainsi que les informations relatives à l'historique de génération d'arc reliant les noeuds (pour le graphe variant dans temps).

1.1) Organisation des noeuds

Chaque noeud dans un état donné est représenté par une couleur. Bleu pour un noeud Solitaire. Rouge pour un leader. Jaune pour un leader temporaire. Gris pour un noeud membre. Noire pour un noeud membre intermédiaire. Violet pour noeud souhaitant lancé un référendum.

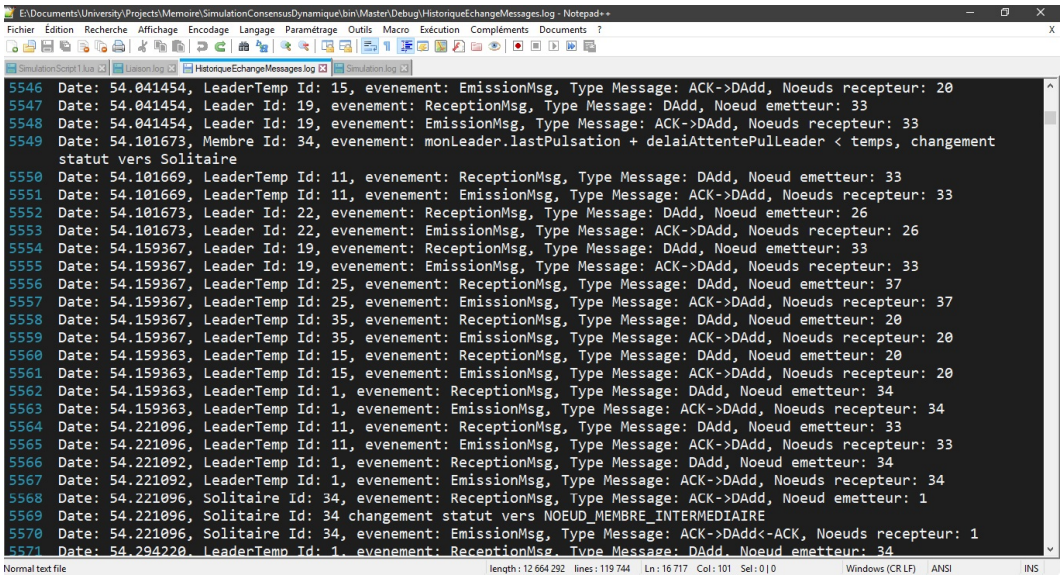


(a) État des noeuds au début de l'execution.

(b) État des noeuds à un instant t.

FIGURE 4.3 – Référendum

Au début de l'exécution de la simulation, tous les noeuds sont en mode Solitaire figure 4.3a, des échanges de messages s'ensuivent figure 4.4, et les premiers leaders temporaire apparaissent figure 4.3b, certain deviennent directement Leader à part entière, dus à une proximité d'un trop grand nombre de noeuds avec un temps d'attente inférieur, causant une saturation de la file d'attente avec des demandes d'ajouts.



```
5546  Date: 54.041454, LeaderTemp Id: 15, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 20
5547  Date: 54.041454, Leader Id: 19, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 33
5548  Date: 54.041454, Leader Id: 19, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 33
5549  Date: 54.101673, Membre Id: 34, evenement: monLeader.lastPulsation + delaiAttentePulLeader < temps, changement
      statut vers Solitaire
5550  Date: 54.101669, LeaderTemp Id: 11, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 33
5551  Date: 54.101669, LeaderTemp Id: 11, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 33
5552  Date: 54.101673, Leader Id: 22, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 26
5553  Date: 54.101673, Leader Id: 22, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 26
5554  Date: 54.159367, Leader Id: 19, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 33
5555  Date: 54.159367, Leader Id: 19, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 33
5556  Date: 54.159367, LeaderTemp Id: 25, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 37
5557  Date: 54.159367, LeaderTemp Id: 25, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 37
5558  Date: 54.159367, LeaderTemp Id: 35, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 20
5559  Date: 54.159367, LeaderTemp Id: 35, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 20
5560  Date: 54.159363, LeaderTemp Id: 15, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 20
5561  Date: 54.159363, LeaderTemp Id: 15, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 20
5562  Date: 54.159363, LeaderTemp Id: 1, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 34
5563  Date: 54.159363, LeaderTemp Id: 1, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 34
5564  Date: 54.221096, LeaderTemp Id: 11, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 33
5565  Date: 54.221096, LeaderTemp Id: 11, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 33
5566  Date: 54.221092, LeaderTemp Id: 1, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 34
5567  Date: 54.221092, LeaderTemp Id: 1, evenement: EmissionMsg, Type Message: ACK->DAdd, Noeuds recepteur: 34
5568  Date: 54.221096, Solitaire Id: 34, evenement: ReceptionMsg, Type Message: ACK->DAdd, Noeud emetteur: 1
5569  Date: 54.221096, Solitaire Id: 34 changement statut vers NOEUD_MEMBRE_INTERMEDIAIRE
5570  Date: 54.221096, Solitaire Id: 34, evenement: EmissionMsg, Type Message: ACK->DAdd<-ACK, Noeuds recepteur: 1
5571  Date: 54.294220, LeaderTemp Id: 1, evenement: ReceptionMsg, Type Message: DAdd, Noeud emetteur: 34
```

FIGURE 4.4 – HistoriqueEchangeMessages.log

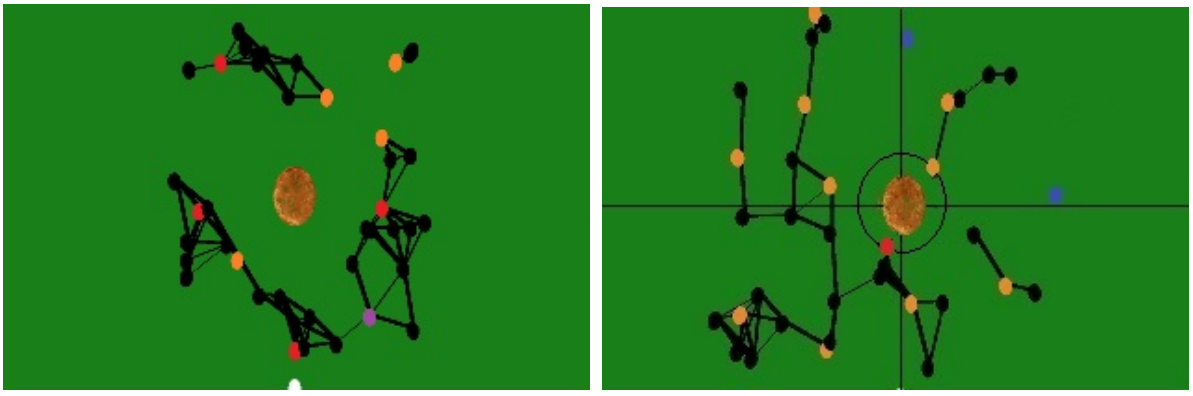
1.2) Consensus

Notre exemple de consensus est simple. Les noeuds disposés aléatoirement sur la scène figure 4.3 doivent se mettre d'accord sur un partage équitable de l'espace.

Le référendum est lancé (grâce au script) par une demande d'un noeud membre pour son leader, ou lancer directement par ce dernier.
Le leader qui a lancé le référendum est représenté en violet figure 4.5a.
Chaque leader qui reçoit la demande, répond par ACK et demande à son groupe de ralentir leurs mouvements aux maximum.
Quand le leader qui a lancé le référendum a reçu les ACKs aux nombre de sa liste de leader actif. Il réémet un ACK confirmant un réseau connexe, et les sessions de vote commence pour chaque leader qui le reçoit.

Chaque noeud propose à son leader une des cinq positions possibles (les quatre points cardinaux, et l'orbite autour de la planète) pour notre simulation, selon qu'il est proche d'une d'elle.

Une fois que la session de vote entre noeuds membre terminé, et une majorité pour un choix proposé a été atteinte, le leader envoie son résultat aux autres leaders via un message LBVote qui contient la décisions du groupe, un poids qui représentera le nombre de noeuds membres, ainsi que ses coordonnés, les calculs se font localement. Un consensus est terminé pour un leader quand il reçoit tous les bulletins de vote des leaders connus figure 4.5b. Les différentes phases et étapes de consensus sont décrites dans le fichier Simulation.log figure 4.6.



(a) État des noeuds au début de référendum.

(b) État des noeuds après un référendum.

FIGURE 4.5 – États des noeuds durant un référendum

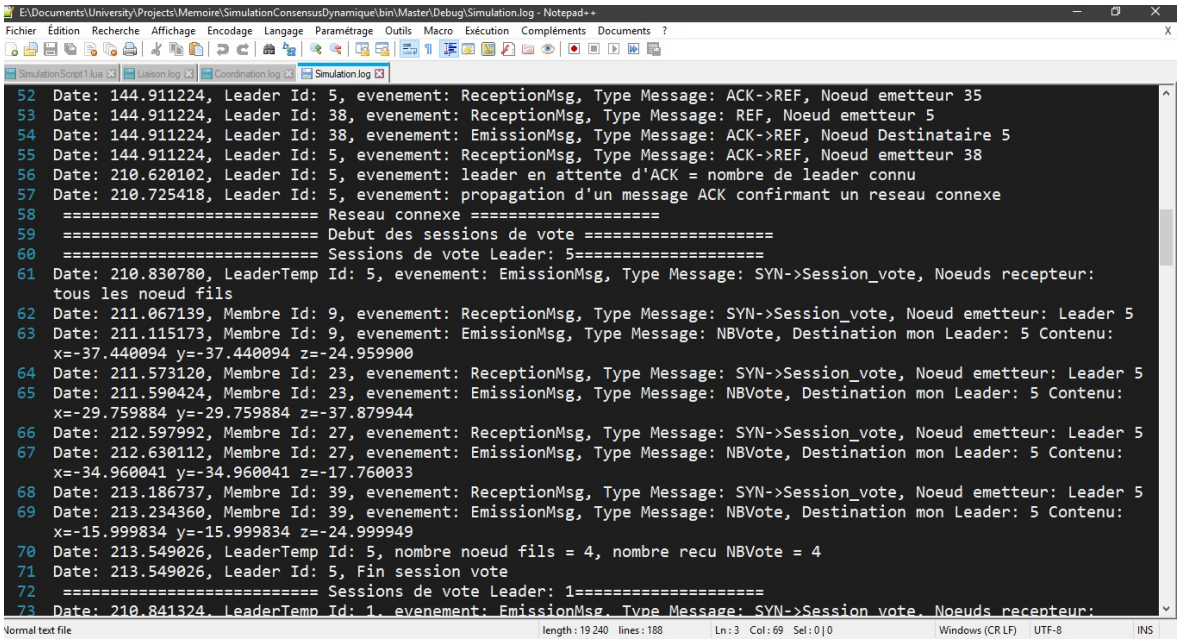


FIGURE 4.6 – Simulation.log

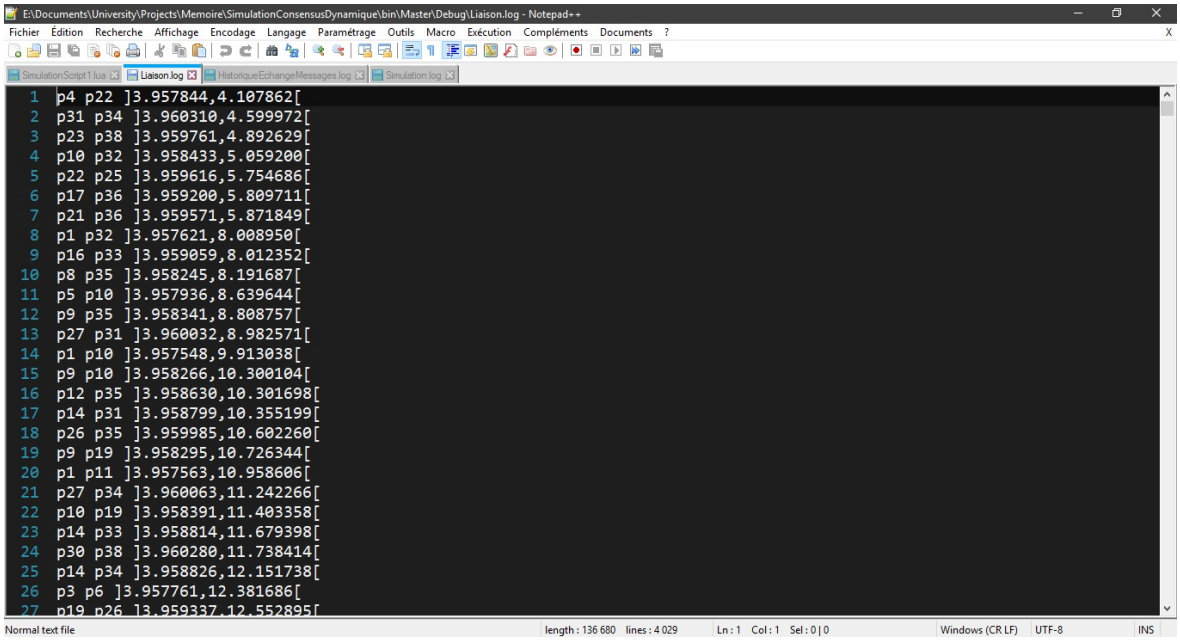


FIGURE 4.7 – Durées des liaisons de communications entre les différents noeuds

1.3) Diagramme de classes

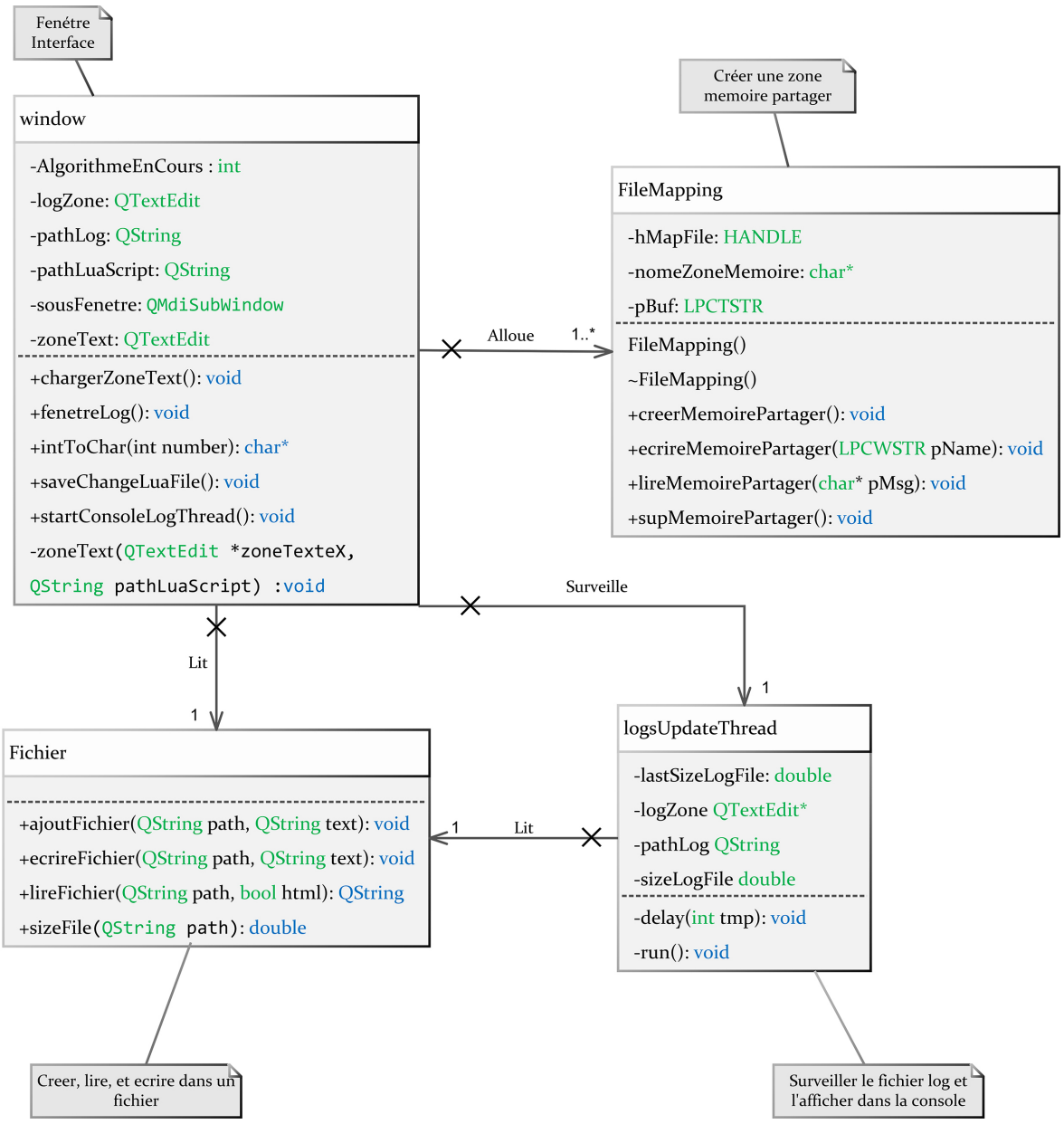


FIGURE 4.8 – Diagramme de class de l'interface.

Chaque noeud est un thread. Comme pour la programmation en C de certain Micro-contrôleur, la classe Noeud possède deux fonctions principales. La fonction setup() pour l'initialisation de paramètre, et la fonction loop() qui est la fonction principale ou tous les messages et événements seront traités.

La fonction draw() de la class principale scene3D, crée et gère les noeuds, et fera office d'intermédiaire pour les transitions des messages d'un noeud à un autre.

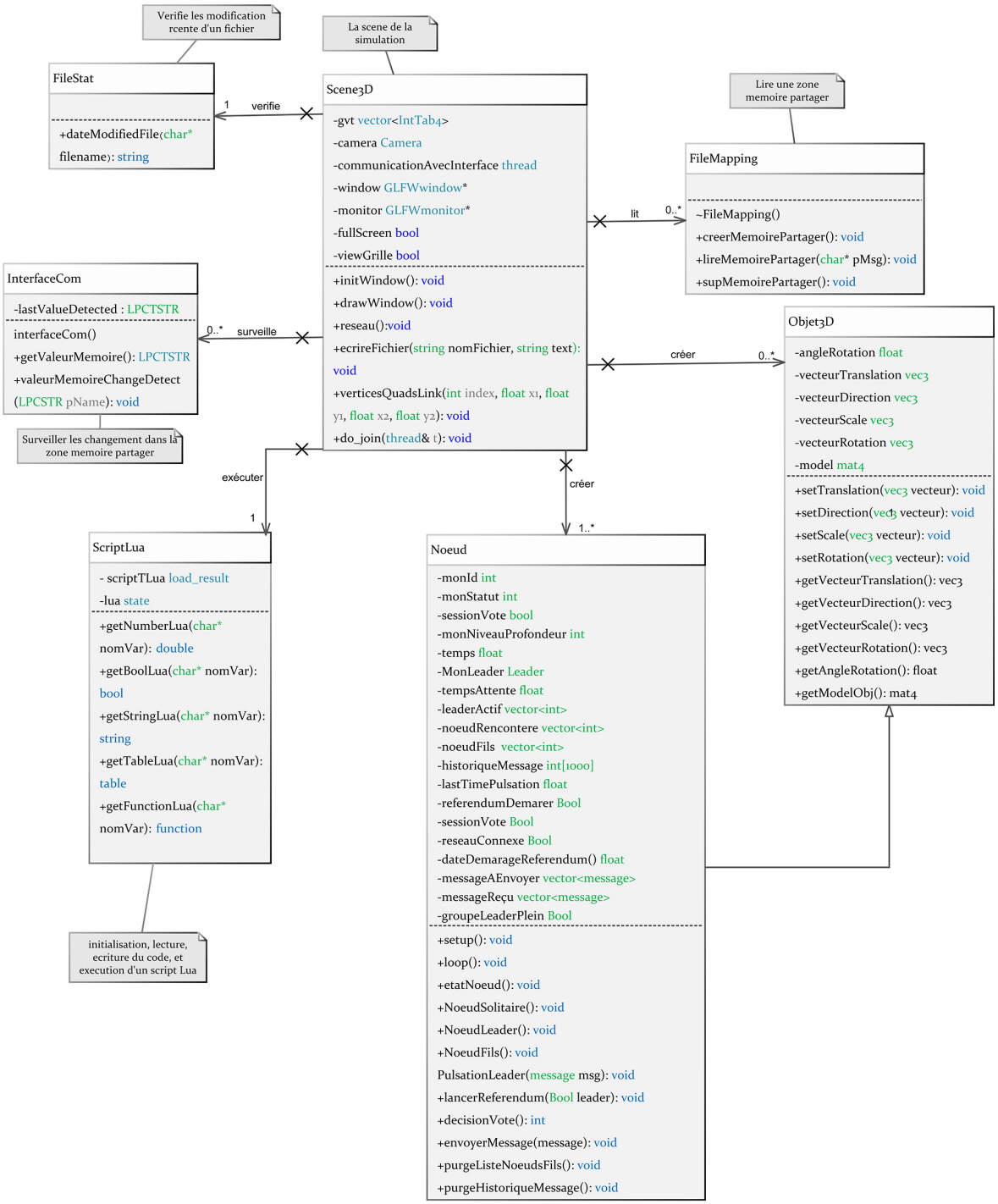


FIGURE 4.9 – Diagramme de class de la scène.

2) Vérification des propriétés

Afin de valider notre solution elle doit essentiellement assurer les propriétés suivantes :

La terminaison : La terminaison est assurée pour un événement, il n'y a pas de messages bloquants, ni d'état conduisant à un inter-blocage, il y a toujours une séquence d'événements à travers laquelle les noeuds s'entendent et s'engagent sur une valeur.
La fin d'une communication est assurée, quel que soit l'état de départ et du scénario d'exécution car :

- Un noeud conserve le hash des messages reçus pour que les mêmes ne soient pas traité indéfiniment. Exemple de problème qui aurait peut-être rencontré avec les noeuds membres 1, 2 et 3 de la figure 4.10.

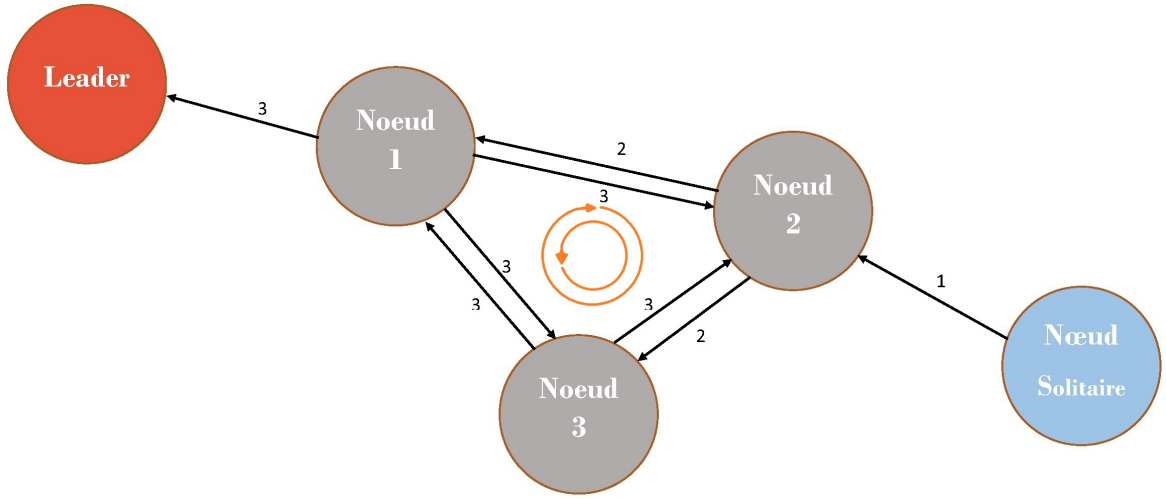


FIGURE 4.10 – Exemple d'une boucle de noeuds dont un message peut tourner indéfiniment.

- Un leader souhaitant communiquer avec un autre leader peut passer par l'intermédiaire des noeuds membres s'il est hors de portée.
- Les messages d'un noeud à un leader, arrivent directement si ce dernier est à sa portée, ou par l'intermédiaire d'autres noeuds frères (qui partage le même leader) s'il est hors de son rayon de couverture.
- Un noeud qui reçoit un message d'un noeuds d'un autre groupe, l'ignore, sauf s'il est destiné à son leader.
- Lorsqu'un réseau connexe est confirmé et qu'un référendum est lancé, les noeuds freinent leurs mouvements jusqu'à ce que la terminaison du consensus soit confirmé par son leader (dans le cas de notre simulation).
- Des limites de temps sont assigné à des événements supposés se produire (pulsation leader, etc.), si elles sont dépassées le noeud réémet le message ou change son état, sinon elles seront réinitialisé.

L'exactitude : Le comportement prévu au départ est toujours assuré :

- Les requêtes de demande d'ajout des noeuds solitaires sont toujours satisfaites, sauf si le leader a atteint sa limite de noeud fils.
- Un leader décide qu'un consensus est achevé, si un choix a atteint une majorité, dont lui-même à calculer le pourcentage, selon les bulletins reçus.

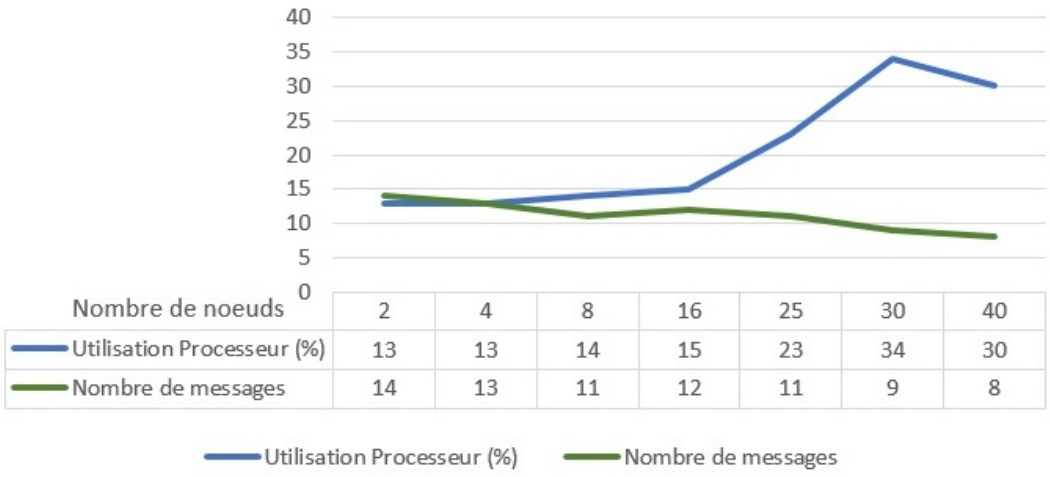
La vivacité : Même en cas de perte, les messages atteignent toujours leurs destinations.

- Si un ACK pour certains messages ni pas retourné après un délai, il est retransmis.
- À chaque changement d'état (leader temporaire vers solitaire, noeud membre vers solitaire, solitaire vers noeud membre, solitaire vers leader temporaire) le noeud supprime les files d'attentes d'envoie (non encore transmis) et de réception (non encore traités) de message pour qu'il n'y ait pas d'ambiguïté pour la suite.

Les performances : Pour la suite des mesures de performance, nous préciserons que notre simulation est compilée en 32bit, et tourne sur un ordinateur équipé d'un processeur Intel 6700HQ 2.6 GHz, et d'une carte graphique nVidia 950M avec une horloge graphique de 915 MHz, sous windows10-64Bit.
La mesure de performance se fera avec l'application Intel Graphics Performance Analyzers (voir Annexe), et les données prélevées seront représentées par des graphes (le bleu pour le pourcentage moyenne d'utilisation du CPU, le vert pour le nombre moyenne de message par seconde et par noeud), l'axe des x représente le nombre de noeuds.

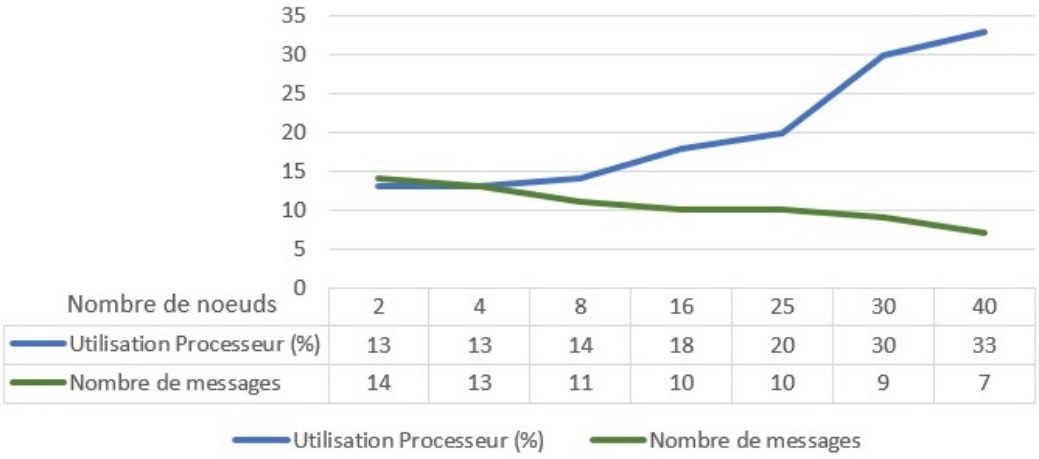
Les noeuds évoluent sur une scène de 160 unités de longueur, 100 unités de largeur et 40s unité de hauteur, avec une vitesse moyenne de 2.0 unités/s (mesures OpenGL).

Il y a neuf graphes. Chaque graphe représente des mesures différentes avec différents paramètres (limite nombre de noeuds fils pour un leader et délai d'attente d'une pulsation d'un leader pour un noeud membre).



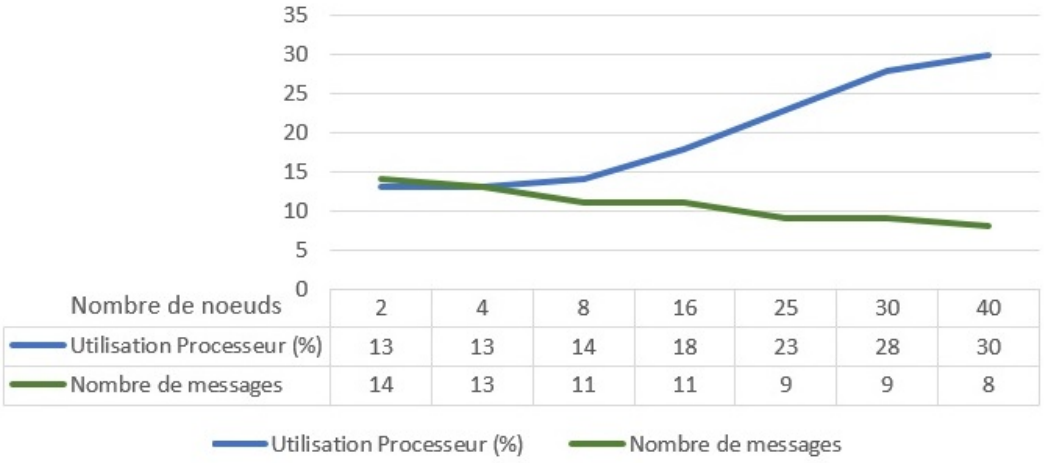
Nombre de noeuds	2	4	8	16	25	30	40
Utilisation Processeur (%)	13	13	14	15	23	34	30
Nombre de messages	14	13	11	12	11	9	8

FIGURE 4.11 – Graphe performance : Limite noeuds fils = 5, Délai d'attente pulsation leader = 2



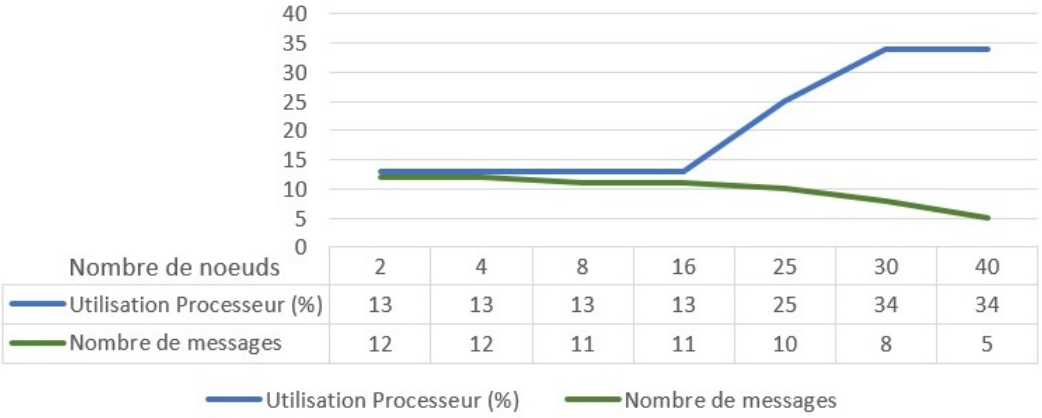
Nombre de noeuds	2	4	8	16	25	30	40
Utilisation Processeur (%)	13	13	14	18	20	30	33
Nombre de messages	14	13	11	10	10	9	7

FIGURE 4.12 – Graphe performance : Limite noeuds fils = 10, Délai d'attente pulsation leader = 2



Nombre de noeuds	2	4	8	16	25	30	40
Utilisation Processeur (%)	13	13	14	18	23	28	30
Nombre de messages	14	13	11	11	9	9	8

FIGURE 4.13 – Graphe performance : Limite noeuds fils = 15, Délai d'attente pulsation leader = 2



Nombre de noeuds	2	4	8	16	25	30	40
Utilisation Processeur (%)	13	13	13	13	25	34	34
Nombre de messages	12	12	11	11	10	8	5

FIGURE 4.14 – Graphe performance : Limite noeuds fils = 5, Délai d'attente pulsation leader = 5

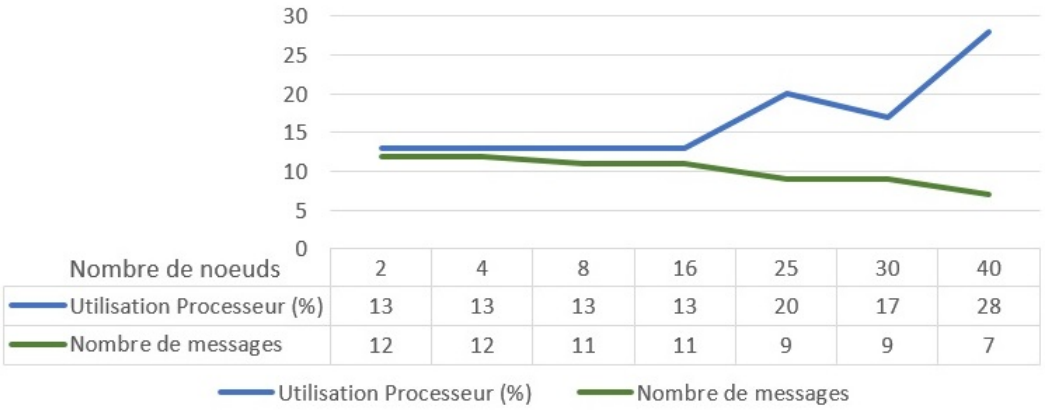


FIGURE 4.15 – Graphe performance : Limite noeuds fils = 10, Délai d'attente pulsation leader = 5

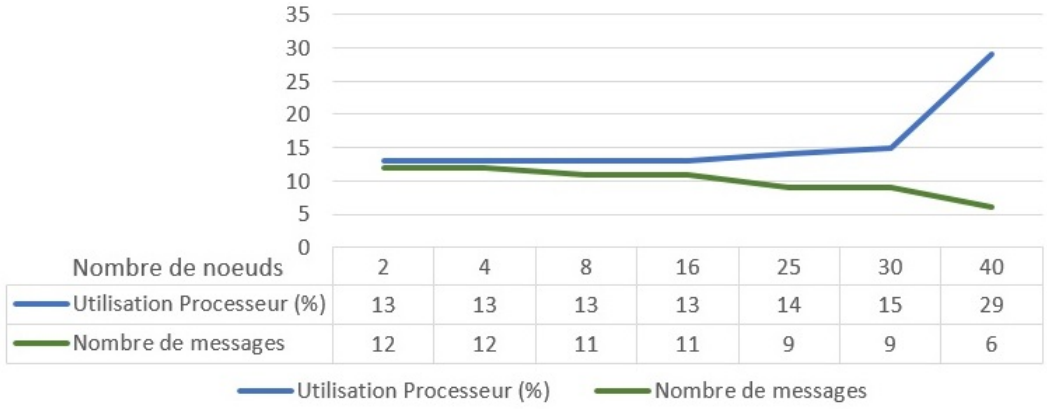


FIGURE 4.16 – Graphe performance : Limite noeuds fils = 15, Délai d'attente pulsation leader = 5

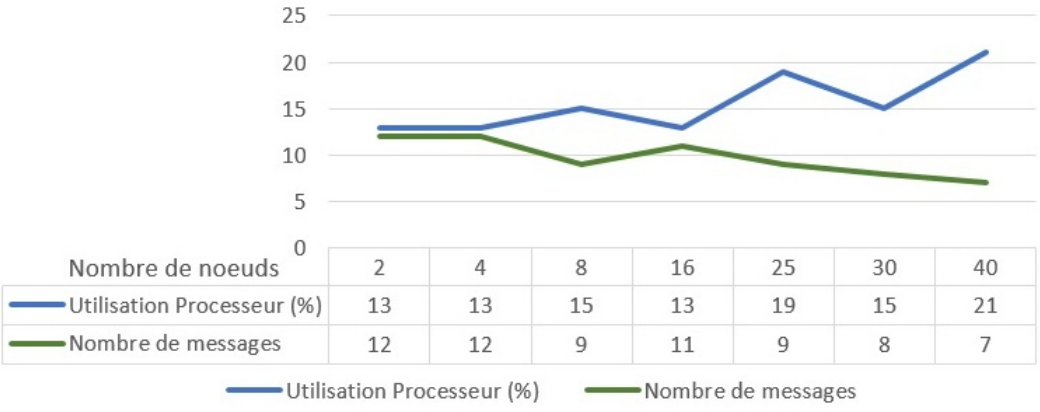
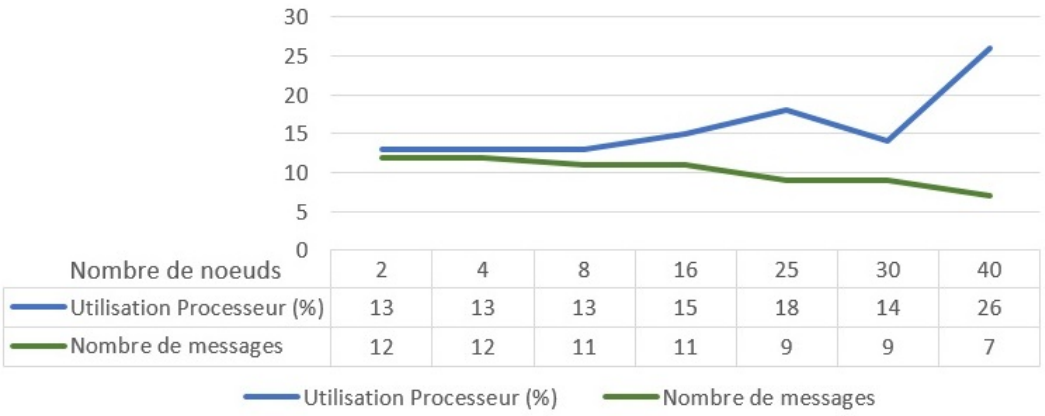
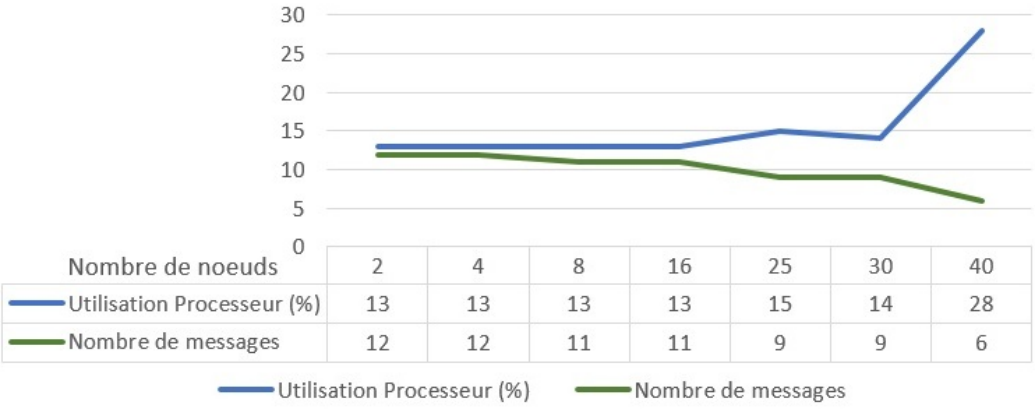


FIGURE 4.17 – Graphe performance : Limite noeuds fils = 5, Délai d'attente pulsation leader = 8



Nombre de noeuds	2	4	8	16	25	30	40
Utilisation Processeur (%)	13	13	13	15	18	14	26
Nombre de messages	12	12	11	11	9	9	7

FIGURE 4.18 – Graphe performance : Limite noeuds fils = 10, Délai d'attente pulsation leader = 8



Nombre de noeuds	2	4	8	16	25	30	40
Utilisation Processeur (%)	13	13	13	13	15	14	28
Nombre de messages	12	12	11	11	9	9	6

FIGURE 4.19 – Graphe performance : Limite noeuds fils = 15, Délai d'attente pulsation leader = 8

De par ces graphes nous remarquons que l'influence de ces paramètres (le délai d'attente de la pulsation de leader et la limite de noeuds fils) est flagrant.

- Plus le délai d'attente de la pulsation de leader est petit, plus les performance s'amenuise.
- Plus la limite de noeuds fils est graned, plus le nombre de message diminue.

On peut les résumer dans un tableau comme suit :

Paramètres / Nombre nœuds	Délai d'attente pulsation leader		Limite nœuds fils	
	Petit	Grand	Petit	Grand
Réduit	↑ CPU ↑ Nombre Messages	↓ CPU (Stable) Nombre Messages	(Stable) CPU ↑ Nombre Messages	(Stable) CPU ↓ Nombre Messages
Elevé	↑ CPU ↑ Nombre Messages	↓ CPU ↓ Nombre Messages	↑ CPU (Stable) Nombre Messages	↓ CPU ↓ Nombre Messages

↓ Diminue ↑ Augmente

TABLEAU 4.1–Tableau des performances selon les deux paramètres (Limite noeuds fils et Délai d'attente pulsation leader) proportionnellement aux nombre de noeuds

3) Vérification des Objectifs

D'après la problématique soulevée dans le chapitre précédent, nous avons dégagé les objectifs atteints qui tournent essentiellement au tour :

— D'une coordination en premier lieu des noeuds mobiles dans un système qui est dynamique : notre approche était d'y arriver grâce à un système de candidatures et d'élections de leaders, et en divisant les états que peut prendre un noeud en cinq catégories. En se basant sur le fait qu'un noeud peut à tout moment se retrouver seul ou éloigné de son leader, ou carrément indisponible, cela n'influencera aucunement les autres noeuds ;

— La collaboration entre les noeuds d'une manière équitable dans le but d'atteindre un consensus en un temps fini : nous nous somme basés pour cela sur le protocole stellar bien que son algorithme soit indisponible, nous avons basé notre approche sur les idées décrite dans son livre blanc [25], et nous l'avons adapté a un système de noeuds mobile. Son avantage comme nous l'avons décri dans le chapitre 3, est qu'il n'y a pas d'état bloqué pour un noeud. Chaque noeud possède sa propre liste de noeud de confiance, mit à jour selon des critères spécifiques. Le consensus utilise un système de référendum, lancer par un noeud membre ou un leader, suivi d'une session de votes entre les noeuds d'un groupe qui démarre après que le réseau soit confirmé comme étant connexe. Le leader prend la décision d'arrêter la session de vote après qu'une majorité sur un choix proposé auparavant a été atteinte. Le résultat est envoyé aux autres leaders qui feront les calculs localement et prendront une décision (accepter les résultats, changer localement le résultat d'un groupe selon son poids, refaire le référendum).

— S'assurer que le système est tolérant aux fautes : l'algorithme ne comprend aucune boucle tant-que, et il y a toujours une séquence d'événements à travers laquelle les noeuds peuvent s'entendre et continuer leurs exécutions.

— Utilisation d'un système de paramètres pour optimiser les performances de l'algorithme : Nous nous somme basés dans notre protocole sur l'utilisation des paramètres adaptables pour palier à certains problèmes, comme l'utilisation élevé du CPU et le nombre élevé de message échangé (tableau 4.1) ou bien d'un trop grand nombre de leaders en service.

4) Synthese

Les travaux existants dans la littérature qui traitent de consensus reposent essentiellement sur le fait q'un réseau est connexe et ou traitement des problèmes d'asynchronisme, certain comme [21] utilise un leader pour gérer la réplication des journaux.

Les auteurs de [22] ont conçus un modèle utilisant un système devisé en plusieurs phases, dans chacune est composée de 2 tours. À chaque fin des deux tours le processus dont l'ID correspond au numéro de phase actuelle est désignée comme roi de la phase (un leader temporaire).

L'approche utilisant les blockchains à était reconnu grâce aux travaux de [23], ces protocoles de consensus ne requiert aucune partie de confiance ou identité pré-supposée parmi les participants, une même base de données des transactions est présente sur l'ensembles des noeuds du réseau.

Toutes ces approches sont réalisables, elles ont prouvé leur efficacité en pratique, cependant la plupart sont couteuses et présentent les mêmes inconvénients dans un système dynamique, surtout en cas d'absence ou de panne d'un noeud.

C'est sur ce point que notre approche se distingue car elle est basée sur quatre protocoles différents, et une tout autre manière de décrire l'organisation, la coordination et la gestion d'envoi des messages entre les noeuds, elle ne fait partie d'aucune catégorie recensée dans le chapitre 2.

Notre solution exploite des leaders pour maintenir des clusters de noeuds, minimisant ainsi les problèmes qui peuvent être rencontré. Lorsqu'un leader souhaite envoyer un message à un autre leader, il peut utiliser des noeuds membres comme intermédiaires pour propager jusqu'à une destination lointaine. Lorsque un leader et son groupe ne répondent plus, un intervalle de temps est utilisé pour réémettre le message, c'est le groupe en question c'est disloquer, le noeud leader destitué répond par des NACK. Cette alternative offre une meilleure disponibilité des noeuds et des ressources, et à moindre cout.

Cependant en terme de sécurité et donc en grande partie au bon choix de Quorum-Slice

pour un noeud, celui-ci devrait être modulable selon l'environnement du système.

5) Conclusion

Nous avons présenté dans cette partie du document la validation de notre approche. D'abord par une simulation d'une organisation et d'une coordination entre noeuds, suivi d'un scénarios d'exécution pour un consensus simple, puis par la vérification des objectifs fixés au départ. Nous avons également fait une synthèse générale révélant les points forts et les points faible de notre solution.

Conclusion générale et perspectives

En raison de son impact pratique et pour des raisons théoriques, le problème de consensus est un problème des plus intéressant dans l'informatique tolérante aux pannes.
Rappelons que dans le problème d'un consensus basique chaque processus commence par une valeur initiale, et doit décider d'une valeur finale (terminaison). Cette valeur décidée doit être une valeur initiale de certains processus, qui l'on à un moment donné proposer (validité), et que tous ces processus doivent décider d'une même valeur finale (accord).
La complexité des implémentations de consensus est devenue l'un des sujets les plus importants dans la théorie de l'informatique distribuée.

On sait que le consensus ne peut pas être résolu dans un système de mémoire partagé asynchrone en lecture-écriture. De manière déterministe et tolérante aux pannes.
Une façon de contourner cette impossibilité est de ne garantir que la progression dans les exécutions répondants à certaines conditions, par exemple, en l'absence de contestation.

En raison de leurs généralités, les protocoles de consensus souvent proposés, peuvent s'adapter à toute les situations avec de légère adaptation à l'environnement dans lequel ils migrent.
Le problème récurrent dans leurs approche et l'utilisation de systèmes dont la topologie est connue (le nombre et la liste des participants est préalablement connue), les hypothèses classiques d'une connaissance préalable de la composition du système, ou même seulement du nombre de participants, ne sont plus possibles dans les systèmes dynamiques.

Les systèmes dynamiques présentent des variations au cours de temps, au niveau de leurs topologies, ainsi que du nombre de noeuds présents et de leurs positions à un instant t qui sont méconnus.
La coordination entre les actions des agents est un défi non trivial, les procédures de communication séquentielle ne sont pas garanties. Dans notre approche aucune dépendance réelle n'existe entre les noeuds. Chacun d'eux garde une certaine dépendance. Si une communication importante n'arrive pas, comme la pulsation de leader, le noeud peut changer son état ainsi que sa façon de traiter les messages.

Chacune des propriétés accordées à un noeud dans le chapitre 3 est importante, si une seule des conditions n'est pas présente ou n'est pas à sa place, l'algorithme se retrou-

vera incomplet. La présence de ces propriétés permet de pallier aux caractères aléatoires des noeuds.
On retrouve aussi souvent l'utilisation d'une diversité d'états pour un noeud dans les protocoles étudiés, le plus souvent deux ou trois états. Dans notre cas nous avons utilisé cinq états possibles, gagnons ainsi une certaine performance et une diminution de nombre de messages.

Tout au long de ce semestre, nous avons tenté d'apporter notre contribution aux problèmes de consensus dans des systèmes variant dans le temps avec des noeuds mobiles, et à moindre coût. La principale motivation qui a régi ce travail est la proposition d'une approche qui conserve les avantages de la dynamicité et un certain degré d'indépendance d'un noeud, qui n'engendre pas ou n'est pas influencé par des problèmes externes.
Notre solution permet d'une part, d'organiser et de coordonner les noeuds, et d'exploiter une continuité de service en cas d'indisponibilité d'un noeud voire même plusieurs, d'autre part réaliser un consensus en un temps fini. Même si la structure d'arbre avec un seul père n'est pas tolérante aux fautes, néanmoins c'est la structure la plus pratique dans notre cas.

Bien que le scénario d'execution est très basique, l'utilisation de moteur graphique OpenGL nous a permis de concevoir notre simulation à la main en partant de zéro, ce qui nous a permet de comprendre encore mieux les problèmes qui peuvent être liées à un réseau dynamique, et ainsi d'améliorer pas à pas et le plus finement possible notre algorithme, et de voir les problèmes liés à chaque étape de conception. L'utilisation d'un script externe était une bonne idée, cela nous a permis d'éviter des recompilations inutiles à cause d'une simple variable et de gagner beaucoup de temps.

Sur la base du travail réalisé nous dresserons d'autres perspectives à savoir :

a) Améliorer l'algorithme et l'intelligence des noeuds.
b) Augmenter la tolérance aux fautes, ainsi que la sécurité.
c) Simuler et évaluer la proposition pour atteindre de meilleures performances, en testant différents paramètres avec différents nombres de noeuds.
d) Tester notre approche sur un environnement réel.
e) Étendre notre solution aux réseaux dynamiques plus denses en terme de noeuds et de transactions.

Bibliographie

[1] R. Miller and L. Boxer. *Algorithms Sequential Parallel.* Prentice hall, 2000. 4, 10

[2] M. Niazi and A. Hussain. *Agent-based computing from multi-agent systems to agent-based models : a visual survey.* Springer, 2011. 5

[3] Mohamed Hamza KAAOUACHI. *Une approche distribuée pour les problémes de couverture dans les systèmes hautement dynamiques.* PhD thesis, Université Pierre er Marie Curie - Paris 6, 2016. 4, 16, 17, 18

[4] Mart Bakhoff. Consensus algorithms for distributed systems. http://ds.cs.ut.ee/courses/course-files/MartBakhoff-consensus.pdf, 2017. 6

[5] Thibault CARRON. *Des Systèmes Multi-Agents temporels pour des systèmes industriels dynamiques.* PhD thesis, école Nationale Supérieure des Mines de Saint-Etienne et de l'université Jean Monnet, 2001. 5

[6] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 1985. 1, 6, 13

[7] Claire Capdevielle. *Etude de la complexité des implémentations dobjets concurrents, sans attente, abandonnables et/ou solo-rapides.* PhD thesis, Université de Bordeaux, 2016. 10, 11, 12

[8] Stefan Poledna. *Fault-Tolerant Real-Time Systems : The Problem of Replica Determinism.* Springer; Softcover reprint of the original 1st ed, 1996. 14, 15

[9] Aguilera M. K. Stumbling over consensus research : Misunderstandings and issues. *Lecture Notes in Computer Science*, pages 59–72, 2010. 12, 13, 14

[10] Frank Harary and Gopal Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 1997. 16

[11] Y. Afek E. Gafni S. Rajsbaum M. Raynal C. Travers. *The k-simultaneous consensus problem*, nř1920. IRISA, 2009. 12

[12] Seyed Mehran Dibaji and Hideaki Ishii. *Consensus of second-order multi-agent systems in the presence of locally bounded faults*, volume 79 of Systems Control Letters. Springer Berlin Heidelberg, pages 23–29, 2015. 6, 12

[13] P. Burke and P. Prosseri. *A distributed asynchronous system for predictive and reactive scheduling*, volume 6, issue 3 of Artificial Intelligence in Engineering. Elsevier, pages 106–124, 1991. 14

[14] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *In Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, 1981.

[15] Colin Cooper and Alan Frieze. Crawling on simple models of web graphs. *Internet Mathematics*, 2004.

[16] Chen Avin, Michal Koucky, and Zvi Lotker. *Automata, Languages and Programming*. Springer Berlin Heidelberg, 2008.

[17] A. Ferreira. Building a reference combinatorial model for manets. *IEEE Network*, 2004.

[18] Leslie Lamport and Robert Shostak and Marshall Pease. *The Byzantine Generals Problem*, volume 4, nř3. ACM Transactions on Programming Languages and Systems, 1982. 15, 16

[19] Arnaud Casteigts, Paola Flocchini, Bernard Mans, and Nicola Santoro. *Deterministic computations in time-varying graphs : Broadcasting under unstructured mobility*, volume 323 of IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2010.

[20] L. Lamport. Paxos made simple. *ACM SIGACT News*, 2001. 19, 20

[21] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. https://raft.github.io/raft.pdf, 2013. 20, 21, 22, 23, 33, 74

[22] Berman Piotr and Juan A. Garay. Cloture votes : n/4-resilient distributed consensus in t + 1 rounds. *Theory of Computing Systems*, 2011. 23, 74

[23] J. Bonneau, A. Miller, J. Clark, A Narayanan, and EW Felten JA Kroll. Sok : Research perspectives and challenges for bitcoin and cryptocurrencies. eprint.iacr.org/2015/261.pdf, 2015. 23, 74

[24] Ripple Labs. The ripple protocol consensus algorithm. ripple.com/files/ripple_consensus_whitepaper.pdf, 2014. 24

[25] The Stellar Foundation. Le stellar consensus protocol (scp). https ://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2016. 25, 33, 40, 73

[26] Intel Corporation. Sawtooth lake consensus. intelledger.github.io, 2015. 26, 27, 30, 33

[27] Leemon Baird. *The Swirlds Hashgraph Consensus Algorithm : Fair, Fast, Byzantine Fault Tolerance*. Swirlds Tech Report Swirlds, 2016. 29

[28] Nathaniel E. Baughman and Brian Neil Levine. Cheat-Proof Playout for Centralized and Distributed Online Games. *IEEE Xplore*, 2002. 28, 34

[29] Nathaniel E. Baughman and Brian Neil Levine and Marc Liberatore. *Cheat-Proof Playout for Centralized and Peer-to-Peer Gaming*, volume 15, issue 1 of IEEE/ACM Transactions on Networking . IEEE Press, pages 1–13, 2007. 28, 34

[30] Game Networking Techniques, Explained with Pong. http://drewblaisdell.com/writing/game-networking-techniques-explained-with-pong/, 2014. 28

[31] M Burrows. The chubby lock service for loosely-coupled distributed systems. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006. 29

[32] Tushar D. Chandra and Robert Griesemer and Joshua Redstone. Paxos Made Live An Engineering Perspective. *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407, 2007. 29

[33] Google. PageRank, technologie du moteur de recherche Google. https ://www.pagerank.net, 2017. 29

[34] Ivan Palomares, Luis Martinez, and Francisco Herrera. A consensus model to detect and manage noncooperative behaviors in large-scale group decision making. *IEEE Transactions on Fuzzy Systems*, pages 516 – 530, 2013. 30

[35] Open Graphics Library. https://www.opengl.org, 2017. 61, 81

[36] Qt Development Frameworks. https://www.qt.io, 2017. 61

[37] The Programming Language Lua. https://www.lua.org, 2017. 61

[38] Intel Graphics Performance Analyzers. https://software.intel.com/en-us/gpa, 2017. 82

[39] AMGHAR Abdeslam. http://www.nysiris.com, 2017.

Annexes

1) Quorum slices

Dans un système distribué, le quorum est un ensemble de noeuds suffisants pour parvenir à un accord. L'accord byzantin fédéré introduit le concept d'une tranche de quorum, le sous-ensemble d'un quorum qui peut convaincre un noeud d'un accord particulier.
Il existe plusieurs différences importantes entre l'accord byzantin non fédéré traditionnel et l'accord byzantin fédéré. L'accord byzantin garantit un consensus réparti malgré l'échec byzantin des participants. Cependant, il faut un accord unanime sur l'appartenance au système par tous les participants. Chaque nud du réseau doit être connu et vérifié à l'avance.

L'accord byzantin fédéré apporte une adhésion ouverte et un contrôle décentralisé à l'accord byzantin. La principale différence est que chaque noeud choisit ses propres tranches de quorum. Les quorums du système résultent des décisions des nuds individuels.

Dans un accord byzantin fédéré, il n'y a pas de contrôleur et aucune autorité centralisée, de sorte que les nuds individuels décident de quels autres participants ils ont confiance. Les noeuds peuvent avoir plusieurs tranches, et ces choix de noeud individuel de confiance peuvent être basés sur des critères extrinsèques.

Chaque nud est responsable de s'assurer que son choix de tranche de quorum ne viole pas l'intersection du quorum. Faire un choix responsable c'est généralement de s'assurer que les tranches sont assez grandes et que les noeuds qu'elles contiennent sont suffisamment importants pour ne pas risquer leurs réputation en nourrissant des informations différentes pour différentes personnes.

2) OpenGL

Open Graphics Library (OpenGL) [35] est une interface de programmation multiplateforme (API) pour le rendu des graphiques vectoriels 2D et 3D. L'API est généralement utilisée pour interagir avec une unité de traitement graphique (GPU), pour obtenir un rendu accéléré par le matériel.

La spécification OpenGL décrit une API abstraite qui permet de dessiner des graphiques 2D et 3D, qui peut être entièrement implémenté dans un logiciel.
L'API est définie comme un ensemble de fonctions qui peuvent être appelées par le programme client, elle regroupe environ 250 fonctions différentes qui peuvent être utilisées pour afficher des scènes tridimensionnelles complexes à partir de simples primitives géométriques. Bien que les définitions des fonctions soient superficiellement similaires à celles du langage de programmation C, elles sont indépendantes du langage.

3) Intel Graphics Performance Analyzers

Intel GPA [38] fournit des outils danalyse de graphiques et doptimisation, pour concevoir des jeux et des applications graphiques exigeantes. L'outils prend en charge les plates-formes basées sur les dernières générations d'Intel Core et la famille de processeurs Intel Atom, pour des applications développées pour les système dexploitation Windows, Android ou linux .

Intel GPA fournit une interface utilisateur commune et intégrée pour la collecte des données de performances. Il est utilisé pour l'échantillonnage d'événements matériels qui utilise l'unité de surveillance de performance sur puce (nécessite un processeur Intel pour cela).

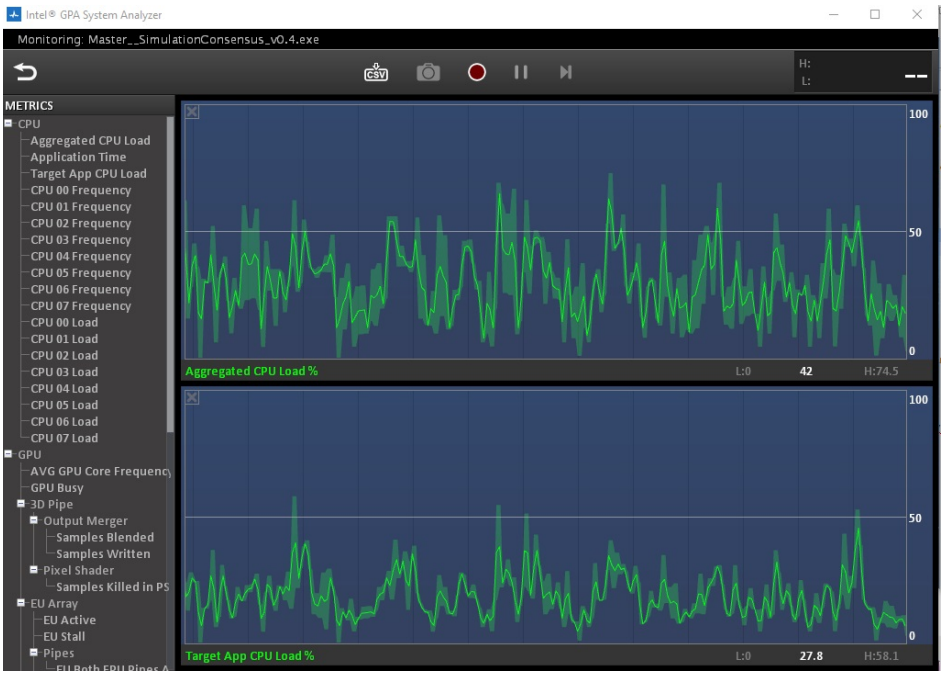


FIGURE 20 – Analyseurs de performances graphiques Intel

Résumé : Un problème fondamental dans les systèmes distribués et les systèmes multi-agents est d'obtenir une fiabilité globale du système en présence d'un certain nombre de processus défectueux. Cela nécessite souvent que les processus acceptent une certaine valeur de données qui est nécessaire pendant le calcul. Un protocole de consensus consiste à faire en sorte que tous les processus d'un groupe acceptent une certaine valeur spécifique en fonction des votes de chaque processus. Tous les processus doivent être d'accord sur la même valeur et celle-ci doit être une valeur soumise par au moins l'un des processus. Le consensus entre les processus est facile à réaliser dans un monde où tout est parfait. Cependant la réalité nous met face à des défis complexes et délicats, et le plus grand d'entre eux est l'asynchronisme. Mais quand le système devient dynamique et donc varie au cours de temps (en terme de topologie, de nombre de nœuds et de leurs positions, etc.), ce gros problème devient une habitude à laquelle on doit s'adapter. Dans ce mémoire nous essayerons de définir le consensus tournant dans un système dynamique, et nous verrons certains problèmes liés à sa conception. Nous présenterons également une solution pour la gestion de la coordination et de l'organisation des agents dans une topologie qui est dynamique et d'un protocole de consensus. Et nous terminerons avec la simulation de la solution apportée.

Summary: A fundamental problem in distributed systems and multi-agent systems is to obtain overall system reliability in the presence of a number of faulty processes. This often requires that the processes accept a certain data value that is needed during the calculation. A consensus protocol is to ensure that all processes in a group accept a certain specific value depending on the votes of each process. All processes must agree on the same value and it must be a value submitted by at least one of the processes. The consensus between processes is easy to achieve in a world where everything is perfect. However, reality presents us with complex and delicate challenges, and the biggest of them is asynchronism. But when the system becomes dynamic and therefore varies over time (in terms of topology, number of nodes and their positions, etc.), this big problem becomes a habit to which one must adapt. In this paper we will try to define the consensus in a dynamic system, and we will see some problems related to its conception. We will also present a solution for managing the coordination and organization of agents in a topology that is dynamic and a consensus protocol. And we will finish with the simulation of the solution provided.

Mots Clés: consensus, protocole, système, distribué, multi-agents, agent, nœud, coordination, organisation, processus, graphe, dynamique, réseau, topologie, asynchrone, raft, poet, stellar, lockstep, opengl, simulation.

Keywords : consensus, protocol, system, distributed, node, organization, process, graph, dynamic, network, topology, asynchronous.