Multi-Consensus distribué : agrégation et révocabilité

Célia Mahamdi

Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Ingénierie / Systèmes Informatiques

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

Multi-Consensus distribué : agrégation et révocabilité

Célia Mahamdi

Soutenue publiquement le : *02 Décembre 2024*

Devant un jury composé de :

Emmanuelle ANCEAUME, Directrice de Recherche, IRISA	*Rapporteuse*
Gil UTARD, Professeur, MIS Université de Picardie Jules Verne	*Rapporteur*
Frédéric Le MOUËL, Professeur, INSA Lyon	*Examinateur*
Jonathan LEJEUNE, Maître de Conférences, Sorbonne Université, LIP6	*Encadrant*
Pierre SENS, Professeur, Sorbonne Université, LIP6	*Encadrant*
Julien Sopena, Maître de Conférences, Sorbonne Université, LIP6	*Encadrant*
Mesaac MAKPANGOU, Chargé de Recherche, Sorbonne Université, LIP6, Inria	*Directeur de thèse*

À ma famille et ma grosse tête adorée

Remerciements

Initialement, l'idée de faire une thèse en informatique ne m'avait jamais traversé l'esprit. J'avais d'autres projets en tête, plus mathématiques, mais les choses ont changé lorsque j'ai suivi l'UE LU2IN015 - Introduction aux principes des systèmes d'exploitation, enseignée par **Swan Dubois**. C'est à ce moment précis que j'ai découvert le monde du Système. Un déclic ! Et pour cela, merci Swan. Jamais je n'aurais imaginé que, quelques années plus tard, j'aurais la chance d'enseigner cette matière à tes côtés. Parfois, il suffit d'un cours, d'un professeur ou d'une rencontre pour qu'une nouvelle voie se dessine devant nous. De nombreuses personnes ont contribué à rendre cela possible, alors permettez-moi de les remercier chaleureusement.

Je tiens tout d'abord à remercier les rapporteurs, **Emmanuelle Anceaume** et **Gil Utard**, pour le temps et l'effort qu'ils ont consacrés à la lecture approfondie et à l'évaluation de cette thèse. Merci également aux autres membres du jury **Frédéric Le Mouël** et **Mesaac Makpangou**.

Cette thèse aurait été impossible sans l'accompagnement de mes encadrants : **Jonathan Lejeune**, **Julien Sopena** et **Pierre Sens**. Jonathan, merci de m'avoir fait confiance et de t'être toujours rendu disponible. Julien, travailler avec toi a été un plaisir, même si, il faut l'avouer, certaines de tes idées ont parfois eu le don de nous perdre un peu ! Enfin, Pierre, ton expertise a été essentielle. Nos discussions sur Paxos et ses subtilités m'ont beaucoup aidé à approfondir ma compréhension de cet algorithme complexe. Merci à vous trois d'avoir pris le temps de répondre à mes nombreuses questions, d'avoir su alléger l'atmosphère et m'arracher des sourires même dans les moments les plus stressants. Merci également pour votre soutien et votre bienveillance, qui m'ont aidé à surmonter les incertitudes et à croire en la valeur de mon travail, même lorsque je doutais de sa qualité.

Aux anciens doctorants **Arnaud Favier**, **Guillaume Fraysse**, **Ilyas Toumlilt**, **Jonathan Sid-Otmane**, **Laurent Prosperi**, **Redha Gouicem** et **Saalik Hatia** : merci pour vos précieux conseils, nos discussions et vos nombreuses anecdotes qui

m'ont beaucoup fait rire. Vos expériences, combinées à votre bienveillance, ont été d'une grande aide.

Aux doctorants actuels, **Aymeric Agon-Rambosson**, **Etienne Le Louët** et **Baptiste Pires** : merci pour nos échanges et pour la camaraderie dont vous avez fait preuve. Je vous passe aujourd'hui le flambeau et vous souhaite sincèrement de réussir vos thèses avec brio. J'espère avoir l'honneur d'assister à vos soutenances et de célébrer vos réussites.

Un merci tout particulier à **Maxime Millet** pour nos pauses thé exceptionnelles (excepté la pause matcha au goût d'herbes). Je te remercie également pour les nombreux réels Instagram qui nous ont permis de rire de notre condition de doctorants. Nos discussions ont été source de réconfort et de motivation.

Un petit mot pour le sang **Yannis Karmim**, merci pour ton soutien et ton optimisme. Je te souhaite plein de succès pour ta soutenance à venir. Bientôt, on pourra dire, comme on l'a souvent imaginé : « On l'a fait ! »

Je voudrais remercier ma famille, qui a été mon pilier tout au long de cette aventure. **Omma**, **Appa**, **Marouan** et **Nawel**, merci pour votre soutien indéfectible. Les mots ne suffisent pas à exprimer toute la gratitude que j'éprouve envers vous. **Omma**, merci d'être ma meilleure amie et une mère d'une bienveillance inégalée. Je n'oublierai jamais tous ces soirs où, après une journée longue et épuisante, tu m'attendais avec un bon repas et des paroles réconfortantes. Je tiens également à remercier ma grand-mère **Ayo**, qui, tout au long de mes études, ne cessait de me répéter un dicton kabyle qui, en français, signifie « que ton travail soit révélé à la lumière ». Ces mots m'ont rappelé que la persévérance et la patience finissent toujours par porter leurs fruits. À mes adorables nièces **Inayah** et **Aleyah**, vous êtes encore petites, mais vous illuminez déjà tout autour de vous. Soyez curieuses, riez autant que possible et surtout ne cessez jamais de croire en vous. Votre tata sera toujours là pour vous guider, vous encourager et applaudir vos rêves, aussi grands soient-ils !

Enfin, un mot pour **ma grosse tête adorée, Nabil**. Merci pour ton amour

inconditionnel et ton soutien sans faille. Merci d'avoir été là à chaque instant, d'avoir séché mes larmes quand ça n'allait pas, et d'avoir trouvé le moyen de me faire rire entre deux sanglots. Merci aussi de m'avoir rappelée, avec douceur mais fermeté, de prendre des pauses quand je me perdais dans le travail. Je ne pourrais jamais résumer tout ce que tu représentes pour moi, et si je devais le faire, je pourrais probablement rédiger une nouvelle thèse (mais soyons honnêtes, personne n'a envie de retrouver une Célia Gremlins). Une chose est sûre : tu as été ma force et mon refuge, et sans toi, je n'y serais jamais arrivée.

Déjà plus de deux pages de remerciements, je vais donc m'arrêter ici, même si j'ai probablement oublié certaines personnes, et je m'en excuse.

Résumé

Cette thèse présente deux contributions dans le domaine des systèmes distribués : OMAHA et le Consensus f-Révocable.

OMAHA (Opportunistic Message Aggregation for pHase-based Algorithms) est un mécanisme d'agrégation de messages conçu pour les algorithmes à phases. Dans les environnements Cloud, où plusieurs applications partagent la même infrastructure, la bande passante est une ressource critique. Dans les datacenters, une partie importante du trafic est consacrée aux petits messages. Chaque message possède un en-tête, ainsi, l'accumulation de ces petits messages pose un véritable problème et entraîne une consommation non négligeable de bande passante. Plusieurs mécanismes ont été proposés pour adresser ce défi, mais peu d'entre eux prennent en compte les caractéristiques applicatives. Ils reposent principalement sur une agrégation des messages au niveau de la couche réseau. OMAHA exploite les spécificités des algorithmes à phases pour agréger intelligemment et de manière opportuniste les messages. Bien que gourmands en messages, de très nombreuses applications reposent sur des algorithmes à phases (Google Spanner, Zookeeper, etc.). Cependant, ils possèdent un avantage majeur : des communications prévisibles. En anticipant les futures communications, OMAHA retarde les messages en les regroupant avec d'autres destinés au même processus. Cela permet de réduire le nombre de messages envoyés sur le réseau et donc une économie de bande passante. Nos expériences montrent des économies de bande passante allant jusqu'à 30%, tout en limitant la dégradation de la latence à 5% pour le célèbre algorithme de Paxos.

Dans les systèmes distribués, atteindre un consensus sur une action ou une valeur est un défi complexe, surtout lorsque les processus sont soumis à des contraintes. De nombreux systèmes, tels que les systèmes multi-agents (véhicules autonomes, gestion d'agenda, robotique, etc.) ou encore les systèmes d'allocation de ressources, doivent respecter des contraintes tout en atteignant un objectif global commun. Cependant, les algorithmes de consensus traditionnels ne prennent pas en compte

ces contraintes, la décision se basant uniquement sur les valeurs proposées par les processus. Une solution triviale consisterait à recueillir toutes les contraintes, mais en raison de l'asynchronisme et des pannes, cela est impossible. Pour tolérer les pannes, certains algorithmes définissent un nombre maximal de fautes qu'ils peuvent supporter. Cela permet à l'algorithme de progresser sans attendre la réponse de tous les processus. En conséquence, la valeur décidée est souvent imposée par un sous-ensemble de processus, la majorité. Les contraintes de la minorité sont ainsi ignorées. Pour répondre à ce problème, nous avons introduit le Consensus f-Révocable. Ce consensus revisité permet de choisir une valeur qui respecte les contraintes des processus tout en offrant la possibilité de révoquer une décision prise par la majorité si celle-ci viole les contraintes d'un processus de la minorité. La convergence est assurée car le nombre de révocations est borné par le nombre de processus appartenant à la minorité. Nous avons développé deux adaptations de l'algorithme Paxos pour mettre en pratique ce nouveau consensus.

Abstract

This thesis presents two contributions to the field of distributed systems : OMAHA and the f-Revoke Consensus.

OMAHA (Opportunistic Message Aggregation for pHase-based Algorithms) is a message aggregation mechanism designed for phase-based algorithms. In cloud environments, multiple applications share the same infrastructure, making bandwidth a critical resource. A significant portion of traffic in data centers consists of small messages. Each message includes a header, leading to substantial bandwidth consumption. Several mechanisms have been proposed to address this issue, but few consider application-specific characteristics. Most rely on aggregation at the network layer. OMAHA leverages the features of phase-based algorithms to intelligently and opportunistically aggregate messages. Many applications, such as Google Spanner and Zookeeper, depend on phase-based algorithms. They are often message-intensive but offer a key advantage : predictable communications. By anticipating future communications, OMAHA delays messages and groups them with others intended for the same process. This approach reduces the number of messages sent over the network, resulting in bandwidth savings. Our experiments show bandwidth saving of up to 30%, while limiting latency degradation to 5% for the well-known Paxos algorithm.

In distributed systems, achieving consensus on an action or value is complex, especially when processes face constraints. Many systems, including multi-agent systems (like autonomous vehicles and robotics) and resource allocation systems, need to respect these constraints while working towards a common goal. Unfortunately, traditional consensus algorithms often overlook these constraints, focusing only on the values proposed by the processes. A straightforward solution would be to gather all constraints, but due to asynchrony and potential failures, this is impossible. To handle failures, some algorithms set a limit on the number of faults they can tolerate. This allows them to move forward without waiting for responses from every process. As a result, the final decision is made by a subset of processes known as the majority.

This leads to the exclusion of constraints from the minority. To tackle this problem, we introduced the f-Revoke Consensus. This new approach enables the selection of a value that considers processes' constraints. It also allows for the revocation of a majority decision if it violates the constraints of a minority process. Importantly, convergence is ensured because the number of revocations is limited by the size of the minority. We developed two adaptations of the Paxos algorithm to implement this new consensus.

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Chapitre

1

Introduction

Sommaire

Le consensus est un problème fondamental dans les systèmes distribués. Il permet à plusieurs processus de s'accorder sur une valeur commune malgré l'éventuelle présence de fautes. Le consensus est essentiel dans de nombreux environnements distribués, comme les bases de données répliquées ou encore les services cloud. Il permet de coordonner les accès concurrents à un objet partagé et d'assurer sa cohérence.

Cependant, les algorithmes de consensus actuels se concentrent surtout sur la réduc-

tion de la latence, souvent au détriment d'un grand nombre de messages échangés entre les processus. Cette complexité en messages devient problématique dans les environnements cloud où la bande passante est une ressource critique. L'envoi de nombreux messages, notamment dans des grandes infrastructures, peut entraîner une surcharge réseau et donc ralentir les performances globales.

De plus, dans les algorithmes de consensus classiques, la décision de la majorité prévaut et est irrévocable. Cependant, dans des environnements plus complexes, tels que les systèmes multi-agents ou les systèmes d'allocation de ressources, certains processus peuvent être soumis à des contraintes spécifiques. Ces contraintes peuvent inclure des exigences de sécurité, de temps ou encore des limites en ressources. Ainsi, il est difficile pour un processus d'accepter une décision imposée par la majorité qui viole ses contraintes.

1.1 Contributions

Les contributions principales de cette thèse sont les suivantes :

- **OMAHA** : un mécanisme destiné à réduire la complexité des algorithmes à phases (notamment ceux de consensus) en agrégeant les messages provenant de différentes applications.
- **Consensus f-Révocable** : une extension du consensus permettant de prendre en compte les contraintes des processus.

1.1.1 OMAHA : un mécanisme d'agrégation opportuniste

Dans les infrastructures cloud, les applications s'exécutent sur des systèmes distribués à grande échelle. La virtualisation permet à plusieurs d'entre elles de partager les mêmes ressources physiques. La ressource réseau étant partagée, la bande passante

devient alors un élément critique. Des études ont révélé que la bande passante est souvent un facteur limitant dans les datacenters [1, 2, 3, 4]. Par ailleurs, une proportion significative du trafic réseau est constituée de petits messages, ce qui conduit à une utilisation conséquente de la bande passante pour le transport des en-têtes [5, 6, 7]. Benson et al. [6] ont par exemple montré que près de 50% des paquets échangés mesuraient moins de 300 octets.

Plusieurs travaux ont tenté de réduire la bande passante consommée par les applications en proposant une répartition plus efficace des ressources réseau. Certains de ces travaux intègrent une meilleure compréhension des besoins des applications [8, 9]. En effet, certaines applications sont très sensibles à la bande passante, et toute réduction dégraderait considérablement leurs performances, tandis que d'autres ne seraient pas affectées. D'autres travaux se concentrent sur les couches réseau, notamment la couche MAC, en optimisant l'allocation de bande passante et en prévenant les collisions de données [10, 11]. Une autre approche repose sur l'agrégation des messages dans les couches basses de la pile réseau. Cependant, ces solutions ne tiennent pas compte des caractéristiques des applications. La couche réseau est incapable de distinguer les messages critiques, dont le retard pourrait nuire à l'efficacité de l'application, des messages moins urgents qui pourraient être retardés sans impact significatif sur les performances.

De nombreuses applications distribuées s'appuient sur des algorithmes à phases, notamment pour la gestion de bases de données [12, 13, 14, 15, 16]. Ces algorithmes, tels que le consensus (Paxos [17]) ou les protocoles de validation atomique (Zookeeper Atomic Broadcast [18], Two-phase commit [19]), impliquent de multiples échanges de messages entre les différents nœuds du système. La communication pendant les phases génère un volume élevé de messages, ce qui augmente la consommation de bande passante. Par exemple, l'algorithme Paxos [17], largement déployé dans les systèmes distribués [20], repose sur une diffusion de messages dont la complexité en termes de communication est quadratique en fonction du nombre de nœuds.

Dans ces algorithmes à phases, les étapes de communication entre les nœuds sont prévisibles. Ainsi, il est possible d'anticiper les échanges futurs et d'agréger certains messages pour économiser la bande passante allouée aux en-têtes. Fort de ce constat, nous avons conçu OMAHA (Opportunistic Message Aggregation for pHase-based Algorithms), un mécanisme générique, opportuniste et non intrusif d'agrégation de messages pour les algorithmes à phases. Ce mécanisme s'applique dans des environnements où plusieurs applications s'exécutent en parallèle sur une même infrastructure. OMAHA permet de réduire significativement la consommation de bande passante tout en limitant l'impact sur la latence. Il contribue ainsi à désengorger le réseau et à améliorer les performances des applications.

1.1.2 Consensus f-Révocable

Dans les systèmes distribués, parvenir à un accord sur une action ou une valeur peut s'avérer complexe. Au-delà du problème du consensus, la résolution de contraintes constitue un véritable défi dans des systèmes de plus en plus distribués. Les systèmes multi-agents, par exemple, imposent des contraintes spécifiques à chaque agent, rendant la gestion du consensus plus complexe. Chaque agent peut être soumis à des contraintes propres à sa mission, à sa localisation ou à sa capacité de calcul [21, 22, 23, 24]. Ces contraintes doivent être prises en compte pour garantir que les décisions respectent les besoins de chaque agent tout en atteignant un objectif global commun. De même, l'allocation de ressources dans un environnement cloud, telles que la bande passante, la mémoire ou le processeur, conduit à la résolution d'un système de contraintes [25]. Ce problème se manifeste, par exemple, dans le Software Defined Network (SDN) pour placer des fonctions réseau virtuelles. L'allocation de ressources dans un tel système doit non seulement respecter les contraintes spécifiques à chaque fonction, mais aussi prendre en compte le fait que chaque allocation peut engendrer de nouvelles contraintes pour les allocations à venir [26, 27].

Le consensus classique, tel qu'il est défini [28], ne gère pas les contraintes des processus participants. En effet, il permet à un ensemble de processus de s'accorder sur une

valeur proposée par l'un d'eux, sans imposer de conditions sur le choix de cette valeur. Choisir la valeur proposée par un des processus revient à ne considérer que les contraintes de ce seul processus, et non la combinaison des contraintes de tous. Une solution triviale serait d'échanger les contraintes entre les processus avant de parvenir à un consensus. Cependant, dans un système asynchrone sujet aux pannes, il est impossible de collecter l'ensemble des contraintes. En effet, un processus ne peut pas distinguer si un autre processus est en panne ou simplement lent.

Pour pallier ce problème, les algorithmes de consensus sont conçus pour tolérer un nombre maximal de fautes et ajustent leurs communications en conséquence. Ainsi, la valeur décidée est souvent déterminée par la majorité des processus, ceux considérés comme non fautifs (leur nombre étant déterminé en fonction du nombre de fautes tolérées par l'algorithme). Cela conduit à ignorer les valeurs proposées par la minorité. Une autre solution serait donc de relancer une instance de consensus si les contraintes d'un processus sont violées. Cependant, avec une telle approche, la convergence n'est pas assurée. En effet, pour que les contraintes de la minorité soient prises en compte, les processus de l'ancienne minorité doivent être suffisamment rapides pour faire partie de la majorité de la nouvelle instance. Dans le cas contraire, l'algorithme pourrait continuer à progresser en ignorant à nouveau les contraintes des processus de la minorité.

Il est donc essentiel non seulement de prendre en compte les contraintes des processus, mais aussi de permettre une réévaluation de la décision si elle s'avère incorrecte. Cela nécessite de repenser le modèle du consensus pour permettre une révocabilité des décisions, tout en respectant les obligations de chaque processus. Nous introduisons le Consensus f-Révocable, f étant le nombre de fautes tolérées. Ce consensus revisité permet de décider d'une valeur respectant les contraintes des processus, mais surtout de révoquer au plus f fois une décision prise par la majorité qui viole les contraintes d'un processus de la minorité. Outre la définition de ce nouveau type de consensus, nous proposons deux adaptations de l'algorithme Paxos pour réaliser un Consensus f-Révocable. Le premier algorithme utilise une séquence de consensus, où plusieurs instances de Paxos sont exécutées de manière itérative pour permettre aux processus

contraints de faire valoir leurs contraintes. Le second est une version permettant de détecter une incompatibilité des contraintes au sein même de l'instance en cours d'exécution afin d'améliorer les performances.

1.2 Publications

Les articles suivants ont été publiés au cours de la thèse :

Auteurs : Célia Mahamdi, Jonathan Lejeune, Julien Sopena, Pierre Sens et Mesaac Makpangou

- *Réduire le coût de communication des algorithmes à phases par l'agrégation de messages : application à Paxos*, COMPAS 2021 - Conférence francophone d'informatique en Parallélisme, Architecture et Système.
- *Agrégation opportuniste de messages pour les algorithmes à phases*, COMPAS 2023 - Conférence francophone d'informatique en Parallélisme, Architecture et Système.
- *OMAHA : Opportunistic Message Aggregation for pHase-based Algorithms*, PRDC 2023 - Pacific Rim International Symposium on Dependable Computing.
- *OMAHA : Opportunistic Message Aggregation for pHase-based Algorithms*, ACM Journal : Formal Aspects of Computing (FAC), Septembre 2024 (version étendue de PRDC 2023).

1.3 Organisation du manuscrit

La suite de la thèse est organisée comme suit :

Le **Chapitre 2** introduit les concepts et définitions essentiels à cette thèse. Nous y présentons les systèmes distribués, le consensus et ses extensions. Ce chapitre décrit

également certains algorithmes à phases ainsi que des mécanismes d'agrégation de messages.

Le **Chapitre 3** présente OMAHA, notre mécanisme d'agrégation opportuniste conçu pour les algorithmes à phases. OMAHA permet une agrégation intelligente des messages en tenant compte des spécificités des applications.

Le **Chapitre 4** introduit le Consensus f-Révocable, qui affaiblit la propriété d'irrévocabilité des décisions. Deux adaptations de l'algorithme Paxos sont proposées pour traiter ce nouveau problème.

Le **Chapitre 5** clôt le manuscrit en récapitulant les contributions et en présentant les perspectives futures.

Chapitre

2

Contexte et état de l'art

Sommaire

2.1 Introduction

Dans ce chapitre, nous introduisons le concept des systèmes distribués. Nous présentons ensuite le problème du consensus ainsi que ses limites dans un environnement asynchrone à travers le résultat d'impossibilité de Fischer, Lynch et Paterson. Nous détaillons ensuite les différentes manières de contourner ce résultat. Enfin, nous présentons des algorithmes de consensus à phases.

2.2 Modèles des systèmes répartis

Systèmes distribués Les systèmes distribués sont constitués de divers types de machines ou nœuds de calcul : ordinateurs portables, appareils mobiles, serveurs, etc. Les processus représentent une abstraction des machines qui composent un système distribué. Dans la suite de la thèse, nous considérerons comme interchangeables les termes processus et nœuds.

Les systèmes distribués sont généralement composés de nombreux processus. Chaque processus s'exécute indépendamment des autres. L'ensemble des processus du système

est noté $\Pi = \{p_1, \ldots, p_n\}$. Les processus ont généralement connaissance de Π ainsi que du nombre de processus du système (ce n'est pas toujours le cas, notamment pour les systèmes dynamiques [29] [30] ou encore très volumineux).

Asynchronie De nombreux travaux supposent que les processus sont asynchrones : chaque processus s'exécute sans faire d'hypothèses sur la vitesse relative des autres processus [31]. Ainsi, un système *asynchrone* ne possède aucune hypothèse de temps. Contrairement aux systèmes synchrones, il n'existe pas de bornes sur les délais de transmission et sur les vitesses relatives des processus.

Fautes Un processus est *correct* s'il s'exécute correctement et est exempt d'erreurs sur toute la durée de l'exécution [32]. Dans le cas contraire, il est fautif. Les fautes peuvent affecter un processus ou un canal de communication. Nous pouvons distinguer plusieurs types de fautes, qu'il est possible de hiérarchiser. Un type de faute est considéré comme plus faible qu'un autre s'il en est un sous-ensemble. Ainsi, les fautes introduites ci-dessous sont classées de la plus faible à la plus forte :

- **Faute franche** *(fail-silent)* : cette faute entraîne l'arrêt définitif du composant. Un processus cesse toute activité tandis qu'un canal de communication est coupé définitivement.
- **Omission** *(omission failure)* : cette faute entraîne une cessation momentanée de l'activité du composant. Dans le cas d'un processus, certains messages peuvent, par exemple, ne pas être envoyés. Le processus reprend ensuite le cours normal de son activité. Pour un canal de communication, cela correspond à des pertes ponctuelles de messages.
- **Faute temporelle** *(timing failure)* : cette faute suppose que les hypothèses temporelles réalisées sur la durée d'une tâche ou sur le délai de transmission d'un message ne sont pas respectées. Un résultat ou un message est délivré trop tôt ou trop tard.
- **Faute byzantine** *(byzantine failure)* : cette faute inclut tous les autres types de

fautes, y compris le fait de délivrer un résultat ou message erroné. Elle permet de modéliser l'intrusion d'une personne mal intentionnée dans le système.

Le nombre maximum de fautes qu'un algorithme peut supporter est noté f.

Canaux Il existe plusieurs types de canaux de communication : fiables, quasi-fiables et équitables. Ils possèdent certaines caractéristiques communes :

- Pas de *création* : Si un processus q reçoit un message m d'un processus p, alors le processus p a envoyé le message m au processus q.
- Pas de *duplication* : Si un processus p envoie un message m au processus q, alors le processus q recevra le message m du processus p au plus une fois.

En revanche, leur caractérisation en termes de perte de messages diffère [33, 34] :

- **Canaux non fiables** *(lossy channels)* : Tout message envoyé peut être perdu.
- **Canaux équitables** *(fair-lossy channels)* : Si un processus correct p envoie un message m une infinité de fois à un processus correct q, alors le processus q reçoit une infinité de fois le message m.
- **Canaux quasi-fiables** *(quasi-reliable channels)* : Si un processus correct p envoie un message m à un processus correct q, alors q reçoit finalement le message m.
- **Canaux fiables** *(reliable channels)* : Si un processus p envoie un message m à un processus correct q, alors q reçoit finalement le message m.

2.3 Consensus

Le consensus est un problème fondamental des systèmes distribués. Informellement, il peut être défini comme un accord sur une valeur ou une action auprès d'un ensemble de processus. Un grand nombre de variations de ce problème existent dans la littérature,

elles diffèrent de par leurs propriétés ou encore leurs hypothèses. En fonction de celles-ci, le consensus peut devenir insoluble en présence de fautes, comme le démontre le résultat d'impossibilité de Fischer, Lynch et Paterson.

Les algorithmes de consensus jouent un rôle crucial dans diverses applications, des bases de données répliquées aux blockchains. Ils sont essentiels pour construire des systèmes distribués, résilients et fiables, capables de supporter les défaillances tout en maintenant un certain degré de cohérence.

Dans ce contexte, implanter un algorithme de consensus nécessite de naviguer à travers un paysage complexe d'algorithmes, de stratégies de résilience et de protocoles de communication.

2.3.1 Définition du consensus

Le problème du consensus (C) doit assurer des propriétés de sûreté et de vivacité. Il définit deux primitives, *propose(v)* et *decide(v)*, permettant respectivement à un processus de proposer et de décider la valeur v. Il est généralement défini comme suit [35] :

- C-Cohérence : Deux processus corrects ne peuvent pas décider différemment.
- C-Validité : Si un processus décide d'une valeur v, alors v a été une valeur proposée par au moins un processus.
- C-Terminaison : Chaque processus correct doit à terme décider.
- C-Intégrité : Chaque processus décide au plus une fois.

2.3.2 Résultat d'impossibilité de Fischer, Lynch et Paterson (FLP)

Un résultat fondamental du problème du consensus a été démontré par Fischer, Lynch et Paterson [36]. Ce résultat prouve qu'il est impossible de concevoir un algorithme

de consensus déterministe dans un système distribué asynchrone si un processus peut subir une panne.

L'intuition derrière ce résultat d'impossibilité réside dans l'incapacité de différencier de manière fiable un processus en panne d'un processus très lent.

Ce résultat a incité les chercheurs à trouver des moyens de contourner FLP. Deux approches ont émergé :

- Identifier un ensemble d'hypothèses minimales qui, lorsqu'elles sont satisfaites par un système asynchrone, permettent de résoudre le problème du consensus.
- Affaiblir le problème.

2.3.3 Contourner FLP : hypothèses supplémentaires

Afin de contourner FLP, les chercheurs ont ajouté aux systèmes asynchrones une juste dose de synchronie permettant de résoudre le problème du consensus. Plusieurs approches ont émergé, parmi les plus connues figurent les systèmes partiellement synchrones et les détecteurs de fautes.

Systèmes partiellement synchrones

Dolev, Dwork et Stockmeyer [37] se sont intéressés à l'influence des différents types d'asynchronie sur le consensus. Ils sont arrivés à la conclusion qu'il existait 32 modèles partiellement synchrones. Ces travaux ont constitué une base pour leur définition.

Un modèle partiellement synchrone se caractérise par un fonctionnement asynchrone pendant une période donnée, suivi d'une stabilisation progressive vers un comportement plus synchrone. L'approche générale consiste à maintenir l'asynchronisme tout en établissant des hypothèses temporelles qui seront à terme satisfaites. Dwork, Lynch et Stockmeyer [38] ont introduit Δ et Φ qui sont respectivement des bornes sur les délais de transmission et sur la vitesse relative des processus. De ces bornes,

nous savons deux choses : elles existent et peuvent être inconnues ou sont connues, mais ne sont effectives qu'à partir d'un certain temps inconnu appelé GST (Global Stabilization Time).

Détecteurs de fautes

Chandra et Toueg ont proposé d'enrichir les modèles des systèmes asynchrones avec des détecteurs de défaillances non fiables [28]. Dans leur article, le système comprend un ensemble de n processus pouvant subir des pannes franches. Un détecteur de fautes est un oracle distribué qui fournit des indications, possiblement erronées, sur les processus en panne. Chaque processus a accès localement à une liste de processus suspectés d'être défaillants et consulte régulièrement le détecteur pour mettre à jour sa liste de suspects et résoudre le consensus.

Les détecteurs de fautes permettent d'obtenir des indices sur les fautes d'un processus. Nous parlons d'indice, car les informations peuvent être incorrectes (à cause de l'asynchronie et de FLP). Cependant, pour être utiles, ils doivent finir par fournir des informations correctes sur les processus défaillants. Ainsi, les détecteurs sont caractérisés par des propriétés de justesse et de complétude :

Complétude La complétude décrit la capacité à suspecter tous les processus fautifs. Il existe principalement deux niveaux de complétude :

- Forte : il existe un instant à partir duquel tout processus défaillant est suspecté par tous les processus corrects.
- Faible : il existe un instant à partir duquel tout processus défaillant est suspecté par un processus correct.

Justesse La justesse apporte des restrictions sur les suspicions erronées. Il existe principalement quatre niveaux de justesse :

- Forte : aucun processus correct n'est suspecté

- Faible : il existe au moins un processus correct qui n'est jamais suspecté
- Finalement forte : il existe un instant à partir duquel tout processus correct n'est plus suspecté par les processus corrects
- Finalement faible : il existe un instant à partir duquel au moins un processus correct n'est plus suspecté par les processus corrects

Chandra, Hadzilacos et Toueg [39] ont démontré qu'un détecteur de fautes avec une complétude faible et une justesse finalement faible est nécessaire et suffisant pour résoudre le consensus dans un système asynchrone avec une majorité de processus corrects. Leurs travaux ont ouvert la voie à de nombreuses recherches dans le domaine [40, 41, 42, 43, 44].

2.3.4 Contourner FLP : Affaiblissement du problème

Une autre manière de contourner FLP consiste à affaiblir le problème du consensus.

Consensus probabilistes

Une première approche consiste à sacrifier le déterminisme. L'idée est de substituer l'une des propriétés fondamentales du consensus par une autre propriété similaire, pouvant être satisfaite seulement avec une certaine probabilité. La plupart de ces algorithmes modifient la propriété de terminaison [45, 46]. Ainsi, bien que l'exécution de l'algorithme puisse potentiellement ne jamais se terminer, cette éventualité survient avec une probabilité de 0, évitant ainsi toute contradiction avec FLP. Il existe également des algorithmes qui modifient la propriété de cohérence [47]. L'accord est donc réalisé sur une valeur avec une certaine probabilité.

Les processus peuvent également effectuer des opérations qui renvoient des valeurs aléatoires. Le premier algorithme probabiliste a été proposé par Ben-Or [45]. Le principe est simple : chaque processus exécute des rondes successives jusqu'à ce que tous les processus parviennent à une décision. La première étape d'une ronde est

un vote où chaque processus envoie sa valeur aux autres. Si un processus reçoit une majorité de messages avec la même valeur, alors tous les processus décident de cette valeur pour la ronde suivante. Sinon, le processus choisit aléatoirement l'une des valeurs proposées et recommence.

Il est possible d'introduire un caractère probabiliste au niveau du modèle et/ou des processus. Pour le modèle, plutôt que d'exécuter toutes les opérations possibles à chaque étape, certaines opérations sont exécutées avec une probabilité donnée. Cette approche a été formalisée par Bracha et Toueg à travers les *schedulers* [48]. Un scheduler est un agent qui contrôle le système de messages et décide quels processus vont effectuer une nouvelle étape.

Affaiblissement de la terminaison

Plusieurs algorithmes de consensus affaiblissent la propriété de terminaison [17, 49, 50, 51]. Lorsqu'une majorité de processus n'est pas disponible, ou lorsque des scrutins sont lancés infiniment souvent (dans le cas des algorithmes à la Paxos), aucun de ces algorithmes ne progresse. Ce mécanisme assure que la propriété de cohérence ne soit jamais violée lorsque le système est asynchrone, au détriment de la propriété de terminaison.

K-set agreement

Le k-set agreement est une généralisation du problème du consensus. La propriété de cohérence est affaiblie : l'ensemble des valeurs décidées par les processus corrects a une taille d'au plus k. Contrairement au consensus traditionnel qui vise à s'accorder sur une seule valeur, le k-set agreement permet de converger vers au moins une, mais pas plus de k valeurs différentes. Le cas $k = 1$ correspond au consensus classique. Cette approche a été proposée pour la première fois par Chaudhuri en 1993 avec un algorithme permettant aux processus de se mettre d'accord sur un vecteur de taille k [52].

Gracefully Degrading Consensus

Une autre manière d'affaiblir le problème est de réaliser un Gracefully Degrading Consensus que l'on pourrait traduire en français par dégradation progressive du consensus. Ce type d'algorithme permet de dégrader le consensus en k-set agreement dans le cas où les conditions du réseau ne permettraient pas la réalisation d'un consensus classique. Il existe plusieurs variantes. Par exemple, Vaidya a proposé un algorithme permettant de réaliser un consensus qui supporte un certain nombre de fautes, mais qui au-delà dégrade le problème [53].

2.4 Extension du consensus

Cette section explore plusieurs variantes du consensus (ainsi que des problèmes connexes) liées à la révocation de décisions et à la prise de décisions dans des environnements où les processus sont soumis à des contraintes.

2.4.1 Consensus abandonnable et validation atomique

De nombreux systèmes tolérants aux pannes nécessitent un accord unanime. Nous pouvons citer comme exemple la réplication de machines à états [54]. Il existe néanmoins des cas où un léger désaccord est tolérable. En effet, sacrifier la progression au profit de l'accord n'est pas toujours optimal. Par exemple, le consensus peut être utilisé pour implémenter une diffusion atomique [28] , qui garantit l'accord sur l'ordre de livraison des messages. Cependant, un accord unanime sur cet ordre n'est pas forcément nécessaire. Felber et Pedone [55] ont étudié une version probabiliste de la diffusion atomique, où il suffit de garantir l'ordre des messages avec une probabilité élevée.

De manière générale, le consensus asynchrone et ses implémentations privilégient l'accord au détriment de la progression. Wei Chen propose une version plus flexible

du consensus asynchrone, appelée *abortable consensus* [56]. Ce type de consensus offre la possibilité de trouver un compromis entre la progression de l'algorithme et l'accord.

Intuitivement, lorsqu'un processus ne peut pas communiquer avec la majorité, il peut choisir d'abandonner le consensus plutôt que de décider d'une valeur. Cela lui évite de rester bloqué indéfiniment. Au lieu d'imposer à tous les processus corrects une décision unique, le consensus abandonnable offre la possibilité à certains de l'abandonner tandis que d'autres décident. Il est important de noter que les processus qui décident doivent obligatoirement s'accorder sur la même valeur. Chen introduit un seuil de probabilité d'abandon, noté α, permettant de choisir selon les besoins d'une application, la probabilité qu'un processus abandonne lors d'un consensus.

Guerraoui et al. [44] ont également défini la notion de *quittable consensus*. Ce type de consensus offre aux processus la possibilité de se mettre d'accord sur la valeur spéciale *quit* en cas de défaillances.

Ces variations du consensus sont corrélées à un autre problème des systèmes distribués : le *NonBlocking Atomic Commit* (NBAC) [57, 58]. Le NBAC est un problème bien connu qui se pose dans le traitement des transactions distribuées. Informellement, ce problème consiste pour un ensemble de processus à se mettre d'accord pour soit valider (*Commit*), soit annuler (*Abort*) une transaction. Initialement, chaque processus vote, dans le but d'aboutir à une décision commune. La décision de *Commit* n'est possible que si tous les processus votent en faveur. En l'absence de défaillances, un vote unanime pour le *Commit* doit conduire à cette décision.

Comme l'a observé Hadzilacos dans [59, 60], les spécifications du consensus binaire et du NBAC sont très similaires. Elles ne diffèrent que par leur propriété de validité. Dans le cas du consensus, les valeurs de décision possibles sont uniquement liées aux valeurs initiales des processus, tandis que dans le NBAC, elles dépendent à la fois des valeurs d'entrée et du modèle de défaillance. Une analyse plus approfondie montre que ces conditions de validité sont distinctes et ne peuvent être directement comparées. Ainsi, sur le plan strict des spécifications, le consensus et le NBAC sont

incomparables.

2.4.2 Cohérence et Intégrité à terme

Dans les systèmes de calcul distribué, il est souvent nécessaire de gérer un objet partagé (bases de données, systèmes de fichiers, verrous, etc.). Pour que ce partage soit efficace, deux propriétés sont recherchées [61] :

- Linéarisabilité : L'objet doit se comporter comme s'il était utilisé de manière séquentielle.
- Sans Attente : Chaque accès à cet objet doit se terminer en un nombre fini d'étapes.

Assurer ces deux propriétés nécessite de résoudre le problème du consensus afin de coordonner les accès concurrents des différents processus. Ainsi, lorsqu'ils accèdent fréquemment à un même objet, la rapidité de résolution devient cruciale pour les performances. Toutefois, le consensus tel qu'il est défini (cf 2.3.1), impose un ordre strict sur les accès ce qui peut affecter la latence. Plusieurs solutions ont été envisagées pour améliorer les performances parmi elles, la cohérence à terme (enventual consistency) ou encore le Consensus Généralisé (cf 2.4.3).

Cohérence à terme

Les techniques de réplication classiques sont pessimistes [62, 63, 64, 65]. Elles donnent aux utilisateurs l'illusion de disposer d'une seule copie des données [66, 67]. Les algorithmes synchronisent les réplicas pendant les accès et bloquent les autres utilisateurs lors d'une mise à jour. Cette approche impacte significativement les performances, notamment dans les bases de données géorépliquées où les latences entre réplicas peuvent être élevées. Pour pallier ce problème, une nouvelle approche a été proposée : la cohérence à terme.

Dans ce nouveau type de cohérence, les réplicas n'ont plus l'obligation d'être identiques à tout moment, mais doivent l'être à terme. Les algorithmes permettent ainsi de lire ou d'écrire des données sans nécessiter de synchronisation préalable, en s'appuyant sur l'hypothèse *optimiste* que les problèmes surviendront rarement, voire jamais [68]. Ainsi, la cohérence à terme autorise une divergence temporaire des réplicas, tant que cette période reste finie [69]. Plusieurs systèmes implémentent la cohérence à terme [70, 71, 72, 14].

Sur cette base, Dubois et al. [73] ont formalisé le problème du consensus à terme (eventual consensus). Le consensus à terme stipule qu'à partir d'un certain moment, deux processus ne peuvent plus décider de valeurs différentes.

Intégrité à terme

À notre connaissance, très peu de travaux remettent en question la propriété d'intégrité du consensus. Dans [73], la notion de consensus irrévocable à terme (eventual irrevocable consensus) est introduite. Cette variante du multi-consensus permet aux processus de décider plusieurs fois , mais seulement jusqu'à k d'instances du consensus, après k consensus ils ne peuvent plus revenir sur leur choix.

2.4.3 Consensus Généralisé

Dans la section 2.4.2, nous avons souligné que les accès concurrents et fréquents à un même objet peuvent dégrader les performances, principalement en raison du consensus qui impose un ordre strict sur chaque accès. Lamport propose une alternative visant à réduire la latence en ordonnant strictement que les accès non commutatifs.

Lamport illustre le problème de la manière suivante. Dans un système bancaire, chaque client possède un compte sur lequel il peut effectuer des dépôts ou des retraits d'argent. L'approche classique des machines à états utilise des algorithmes de consensus pour ordonner les demandes des clients. Dans Paxos (cf 2.5.1), les clients envoient leurs

demandes à un leader qui les ordonne avant de les soumettre aux autres processus. Fast Paxos (cf 2.5.2), une version optimisée, permet aux clients d'envoyer directement leurs demandes aux processus, ce qui réduit les délais de communication. Cependant, en cas de conflits (par exemple deux clients émettant des commandes simultanées), Fast Paxos peut nécessiter des étapes de communications supplémentaires, ce qui réduit son efficacité. Lorsque les commandes concurrentes sont commutatives (comme des opérations sur des comptes distincts, où l'ordre d'exécution importe peu), il n'est pas nécessaire d'avoir un ordre strict. Au lieu d'imposer un ordre global pour toutes les commandes, Lamport propose de se concentrer uniquement sur l'ordonnancement des commandes non commutatives, ce qui améliore l'efficacité (en réduisant le nombre de collisions). La définition de la commutativité des commandes doit être définie formellement par l'utilisateur.

Ce type de consensus est appelé *Consensus Généralisé*. Lamport a proposé une version de Paxos, appelée *Generalized Paxos*, qui résout le consensus généralisé en deux étapes de communication lorsque les accès concurrents sont commutatifs. Ce résultat est optimal [74]. Toutefois, lorsque des processus accèdent de manière concurrente et non commutative à l'objet, des conflits peuvent survenir. Dans ces situations, Generalized Paxos nécessite quatre étapes de communication supplémentaires, ce qui le rend moins performant que Fast Paxos.

Sutra et Shapiro [75] ont proposé *FGGC* (Fast Genuine Generalized Consensus) un algorithme permettant de résoudre une collision en une seule étape de communication sous certaines conditions. En outre, *FGGC* nécessite $2f + 1$ processus et fait usage seulement de $f + 1$ d'entre eux contre $3f + 1$ processus et un usage de $2f + 1$ pour le Generalized Paxos.

2.4.4 Consensus contraint

Les problèmes de contrôle coopératif distribué concernent des systèmes où plusieurs agents autonomes collaborent pour atteindre un objectif global commun. Ces agents

interagissent via des algorithmes distribués, ce qui leur permet de partager des informations et de coordonner leurs actions sans nécessiter de contrôle centralisé. Dans ce contexte, chaque agent effectue des décisions locales en se basant sur les données reçues de ses voisins, tout en contribuant à un objectif collectif. Ce type de contrôle est essentiel pour certaines applications (robotique, coordination de véhicules autonomes, etc.).

Malgré l'abondance des travaux dans ce domaine [77, 78, 76], la littérature existante aborde peu les situations où les valeurs des agents sont contraintes d'appartenir à des ensembles donnés. Ces contraintes sont pourtant cruciales dans de nombreuses applications, telles que la planification de mouvements ou les problèmes d'alignement, où la position de chaque agent est limitée à une certaine région ou plage.

Nedic et al. ont étudié le problème de contrôle coopératif où les valeurs des agents doivent respecter des contraintes appartenant à des ensembles convexes fermés [22]. Les informations concernant les contraintes sont elles-mêmes réparties entre les agents, c'est-à-dire que chaque agent ne connaît que son propre ensemble de contraintes. Ils proposent un algorithme de consensus sous contrainte qui fonctionne à partir des informations locales de chaque agent. Plus précisément, chaque agent combine linéairement la valeur qu'il a estimée avec celles reçues de ses voisins (agents adjacents) dans un réseau à connectivité variable, puis projette cette combinaison sur son propre ensemble de contraintes.

2.5 Algorithme à phases

Les algorithmes à phases sont des modèles d'algorithmes qui découpent la résolution d'un problème en différentes étapes distinctes. Ces algorithmes exécutent une séquence d'étapes où l'étape i+1 démarre après l'achèvement de tout ou d'une partie de l'étape i. L'exécution de ces phases se fait de manière séquentielle, avec des étapes interdépendantes les unes des autres.

Les phases peuvent avoir différents patterns de communications. Ces patterns sont illustrés dans la Figure 2.1.

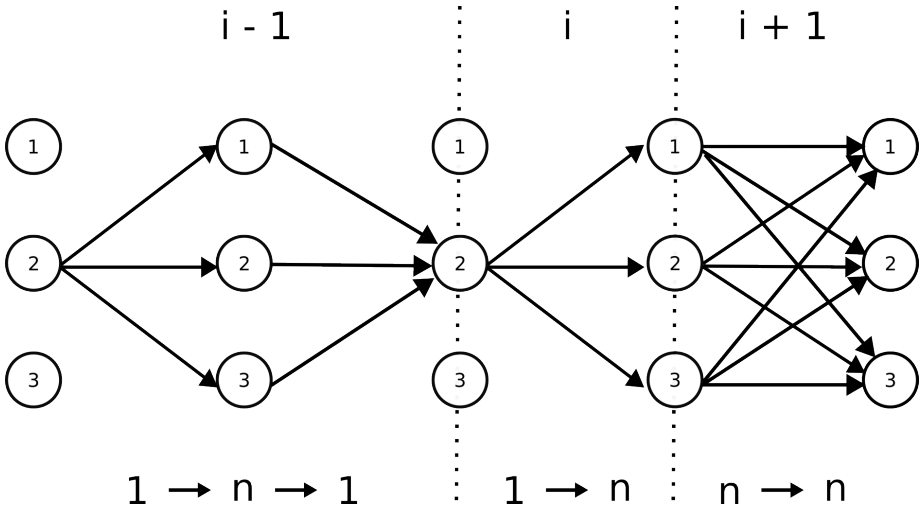


FIGURE 2.1 – Patterns des phases

Dans cette figure, nous distinguons trois phases, de $i-1$ à $i+1$, chacune avec un schéma de communication spécifique. La phase $i-1$ suit un modèle $1 \to n \to 1$, généralement représentatif d'une requête initiée par un nœud suivie de la réception des réponses. La phase i adopte un schéma $1 \to n$, tandis que la phase $i+1$ illustre un modèle $n \to n$ avec une série de diffusions (broadcasts). Ce type de schéma est couramment utilisé, c'est notamment le cas pour l'algorithme de Paxos, qui sera détaillé plus loin.

Les algorithmes à phases jouent un rôle crucial dans la résolution de problèmes fondamentaux en systèmes distribués.

Dans cette section, nous présentons quatre algorithmes à phases représentatifs que nous avons implantés pour évaluer nos contributions.

2.5.1 Paxos

Nous avons vu précédemment qu'affaiblir le problème du consensus était une manière de contourner FLP. En pratique, c'est ce qui est fait en industrie notamment par le biais d'algorithmes à phases et plus particulièrement d'algorithmes à la Paxos. Dans cette catégorie d'algorithmes, une exécution est divisée en une séquence (potentiellement illimitée) de scrutins numérotés. Un processus peut rejoindre un scrutin s (ou ballot) si et seulement s'il n'a pas rejoint un scrutin supérieur. Chaque scrutin s est associé à un coordinateur unique. Pendant le scrutin, le coordinateur de s essaie de convaincre les autres processus d'accepter une valeur proposée. Ce type d'algorithmes est très utilisé notamment pour la gestion de bases de données.

L'algorithme de Paxos [17], proposé par Lamport en 1998, est l'un des algorithmes de consensus les plus connus. Souvent perçu comme complexe, il demeure néanmoins largement utilisé dans l'industrie, notamment pour la gestion des réplicas de bases de données [79],[80].

Paxos est un algorithme de consensus robuste. Il est conçu pour tolérer la perte de messages et fonctionne avec un nombre fixe de nœuds capables de subir des fautes franches, tout en permettant la reprise en cas de défaillance (crash-recovery).

Les nœuds peuvent avoir différents rôles :

- **Proposer** : initie la proposition d'une valeur.
- **Acceptor** : accepte ou rejette une proposition émise par un proposer.
- **Learner** : surveille les réponses des acceptors.

Pour tolérer f *acceptors* défaillants, un total de $2f + 1$ *acceptors* est nécessaire et la taille du quorum s'élève à une majorité simple, soit $f + 1$ *acceptors*.

L'algorithme de Paxos est composé de trois phases : la phase de préparation (prepare), la phase d'acceptation (accept) et la phase de décision (commit). Voici un aperçu de chaque phase :

Phase de préparation : Le proposer p envoie un message de type *Prepare* contenant un numéro de *ballot* à tous les acceptors. Ce numéro de ballot doit être strictement supérieur à tous les ballots déjà envoyés par p. Dès réception du message *Prepare* venant de p, un acceptor a vérifie si le *ballot* est supérieur à tous les autres ballots auxquels il a participé. Si c'est le cas, alors a rejoint le *ballot* et informe p via un message de type *Ack* contenant la dernière valeur qu'il a acceptée et le numéro de ballot qui lui est associé. Sinon, le message est ignoré.

Phase d'acceptation : lorsque p détecte qu'une majorité d'acceptors ont rejoint son *ballot*, il doit choisir une valeur. Si parmi les messages *Ack*, des valeurs ont été acceptées, alors p choisit la valeur associée au plus grand ballot. Sinon, p est libre de choisir la valeur. Le proposer p envoie ensuite un message de type *Accept* à tous les acceptors contenant la valeur choisie et son numéro de *ballot*. Les acceptors, s'ils n'ont pas rejoint de ballot plus récent, vont retransmettre ce message aux learners via un message de type *Accepted*.

Phase de décision : Lorsqu'un learner l a reçu une majorité de messages *Accepted* pour un même numéro de ballot, il décide.

Pour garantir la progression de l'algorithme, un seul proposeur doit initier les propositions, ce proposeur est désigné comme le leader.

La Figure 2.2 illustre les différentes phases de Paxos. Le leader, le nœud 2, est représenté en gras.

2.5.2 Fast Paxos

Fast Paxos [81] est une amélioration de l'algorithme de Paxos, visant à réduire la complexité en messages. Conçu par Lamport en 2005, il permet d'accélérer le protocole de consensus en réduisant le nombre de messages.

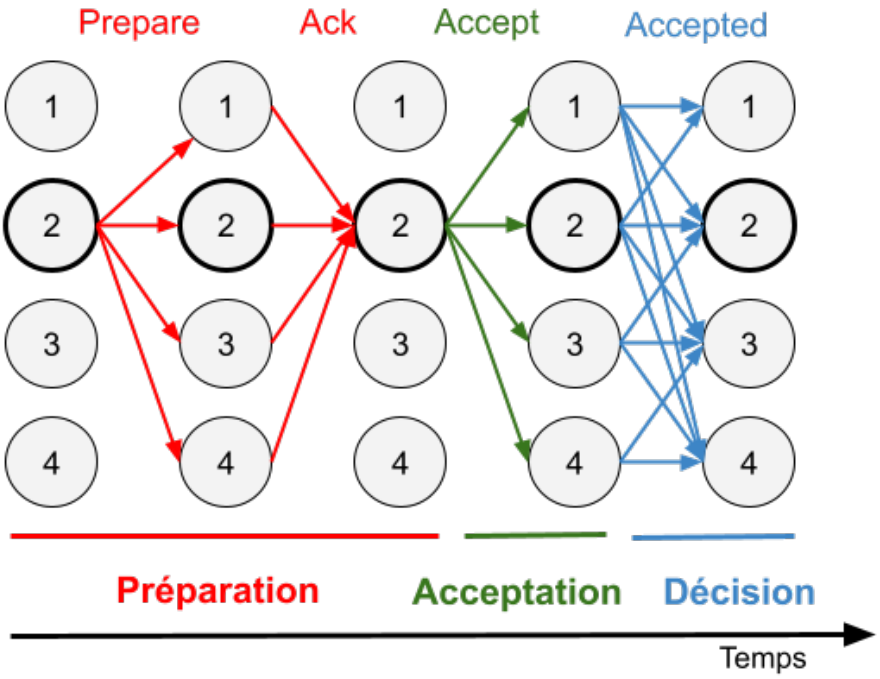


FIGURE 2.2 – Schéma des phases de Paxos

Fast Paxos offre la possibilité aux nœuds de proposer directement une valeur, contournant ainsi le besoin d'un leader pour initier chaque nouvelle proposition. Afin de tolérer f *acceptors* défaillants, un total de $3f + 1$ *acceptors* est nécessaire. La taille du quorum diffère et s'élève à $2f + 1$.

Ainsi, Fast Paxos peut se dérouler selon deux approches :

- Version classique : application de l'algorithme de Paxos en suivant les étapes de préparation, d'acceptation et de décision mentionnées précédemment.
- Version rapide : les nœuds peuvent proposer directement leur valeur sans passer par la phase de préparation. Cette valeur doit être acceptée par $2f + 1$ nœuds.

Le mode rapide nécessite l'intégration d'un mécanisme spécifique permettant de minimiser les collisions et donc de garantir la cohérence des décisions prises par les

nœuds du système. Lorsque plusieurs nœuds envoient des propositions simultanément, ces propositions peuvent être reçues dans un ordre différent. Les nœuds peuvent ainsi accepter différentes valeurs. Dès qu'une collision est détectée par le leader, celui-ci reprend la main et relance une nouvelle phase d'acceptation en choisissant une valeur parmi les différentes acceptées.

La Figure 2.3 illustre le déroulé d'une phase rapide. Le leader, le nœud 1, est représenté en gras.

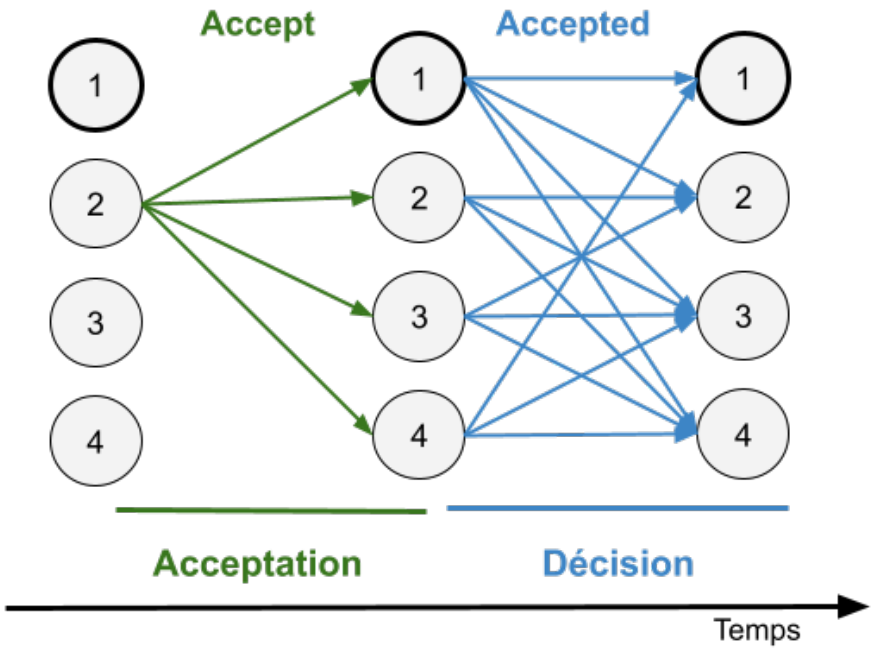


FIGURE 2.3 – Schéma des phases de Fast Paxos version rapide

En résumé, Fast Paxos vise à accélérer le processus de consensus en réduisant les étapes de communications entre les nœuds. Cependant, il peut être confronté à des collisions, ce qui peut nécessiter des étapes supplémentaires pour résoudre ces conflits.

2.5.3 Fast Byzantine Paxos

Fast Byzantine Paxos, présenté dans [82], est une extension de Paxos résolvant le consensus byzantin. FBP repose sur un leader. Il nécessite $5f + 1$ nœuds et, plus précisément :

- Un nombre a d'acceptors supérieur ou égal à $5f + 1$
- Un nombre p de proposers supérieur ou égal à $3f + 1$
- Un nombre l de learners supérieur ou égal à $3f + 1$

L'algorithme fonctionne de la manière suivante :

Phase d'acceptation : Cette phase est similaire à celle de Paxos. Le leader propose sa valeur et un numéro de ballot à tous les acceptors via un message de type *Propose*. Il continue de retransmettre ce message jusqu'à recevoir $(p + f + 1)/2$ messages de type *Satisfied*. Si les acceptors n'ont pas encore accepté de valeur, ils l'acceptent et retransmettent la valeur reçue via un message de type *Accepted* à tous les learners.

Phase de validation : Lorsqu'un learner reçoit $(a + 3f + 1)/2$ messages *Accepted*, la valeur est décidée. Les learners informent ensuite les proposers via un message de type *Learned*.

Phase de vérification : Lorsqu'un proposer reçoit $(l + f + 1)/2$ messages de type *Learned*, il envoie un message de type *Satisfied* à tous les proposers. Si un proposer n'est pas contacté par $(l + f + 1)/2$ learners, il commence à suspecter le leader. Si $(p + f + 1)/2$ proposers partagent cette suspicion, un nouveau leader est élu.

La Figure 2.5 illustre les différentes phases de FBP. Le leader, le nœud 2, est représenté en gras. Dû à la possible présence de nœuds byzantins, l'algorithme nécessite plus de nœuds que les algorithmes à phases évoqués précédemment.

En résumé, Fast Byzantine Paxos résout le problème du consensus en présence de

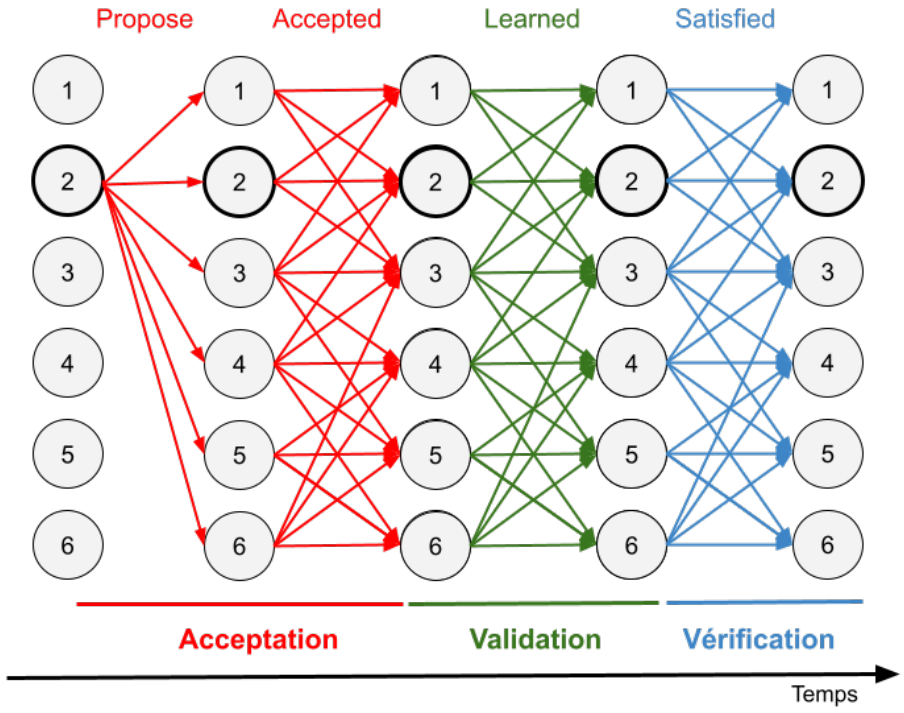


FIGURE 2.4 – Schéma des phases de FBP

fautes byzantines. Ils possèdent plusieurs étapes de communications parmi lesquelles figurent de nombreux broadcasts.

2.5.4 Zookeeper Atomic Broadcast - ZAB

ZAB (Zookeeper Atomic Broadcast) est un algorithme de diffusion atomique au cœur du service de coordination Zookeeper. Il comporte trois phases : découverte, synchronisation et diffusion.

Un processus ZAB peut avoir deux rôles : leader et/ou follower. Le leader joue le rôle principal en proposant des transactions, tandis que les followers les acceptent. Un leader est également un follower. Une *époque* est une unité de temps ou de séquence

utilisée pour identifier et ordonner divers événements ou transactions.

Phase de découverte Dans cette phase, les followers partagent avec leur futur leader les transactions qu'ils ont récemment acceptées. Cette étape vise à identifier la séquence la plus récente de transactions acceptées au sein d'un quorum Q, amorçant ainsi une nouvelle époque.

Phase de synchronisation Cette phase a pour objectif de synchroniser les réplicas en utilisant les transactions acceptées par le leader, appelées historique. Le leader propose aux followers, dans Q, des transactions extraites de son historique. S'ils n'ont pas encore rejoint une époque plus récente, les followers confirment la réception des propositions. Une fois que le leader a reçu un ensemble d'accusés de réception (acknowledgments) du quorum Q, il envoie un message de validation à tous les followers, marquant ainsi son établissement en tant que leader.

Phase de diffusion Enfin, vient la phase de diffusion. Le leader propose une nouvelle transaction aux followers du quorum Q. Une fois qu'il reçoit un quorum d'accusés de réception, le leader envoie un message de validation à tous les followers, confirmant ainsi la transaction. En l'absence d'incidents, les processus restent dans cette phase indéfiniment, diffusant des transactions chaque fois qu'un client Zookeeper lance une demande d'écriture.

La Figure 2.5 illustre les différentes phases de ZAB. Le leader, le nœud 2, est représenté en gras.

ZAB est un algorithme de diffusion atomique utilisé par Zookeeper. Il a été conçu pour assurer une transmission fiable et ordonnée des messages parmi un groupe de nœuds, garantissant ainsi la cohérence et la tolérance aux pannes.

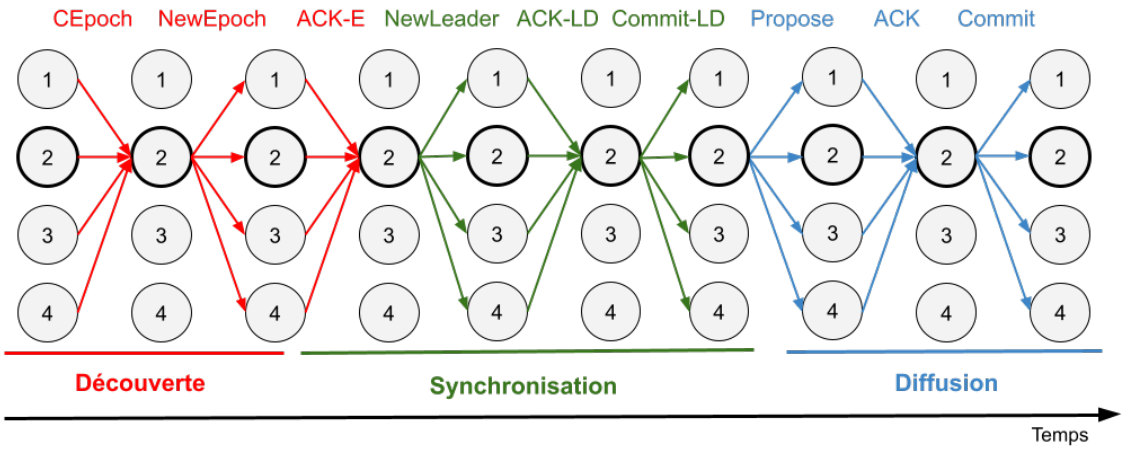


FIGURE 2.5 – Schéma des phases de ZAB

2.5.5 Comparaison des algorithmes

Dans la suite de cette thèse, pour Paxos et ses variantes, nous considérons que chaque nœud joue les rôles de proposer, acceptor et learner. Le tableau 2.1 résume les caractéristiques des quatre algorithmes.

	Classical Paxos[17]	Fast Paxos[81]	Fast Byzantine Paxos (FBP) [82]	ZAB [18]
Étapes	4	2	4	3
Nœuds	$2f+1$	$3f+1$	$5f+1$	$2f+1$
Taille des quorums	$f+1$	$2f+1$	$3f+1$ or $4f+1$ (selon les phases)	$f+1$

TABLE 2.1 – Tableau récapitulatif des protocoles pour un système supportant jusqu'à f nœuds défaillants

2.6 Mécanismes d'agrégation

Cette section présente les mécanismes d'agrégation existants à différents niveaux. À notre connaissance, nous n'avons pas trouvé de mécanismes d'agrégation similaires à OMAHA pouvant être activés de manière opportuniste et multiplexant les messages de n'importe quelle application en cours d'exécution. Cependant, plusieurs mécanismes permettant un certain niveau d'agrégation ont été proposés dans la littérature ou sont disponibles dans les couches basses du réseau.

2.6.1 Agrégation de messages au niveau des couches réseau

La transmission fiable de données sur un réseau non fiable (comme IP) nécessite l'utilisation de protocoles de bout en bout (comme TCP) qui reposent sur des mécanismes de feedback. Ces protocoles utilisent des messages de contrôle pour établir des connexions et accuser de la réception (totale ou partielle) des données. Cependant, cette approche entraîne une surcharge de communication, pouvant diminuer le débit global du réseau. Pour atténuer cette surcharge, il est possible d'agréger plusieurs messages de contrôle en un seul paquet, ce qui permet de réduire le nombre de transmissions. Cette méthode entraîne un délai supplémentaire. En effet, un message de contrôle peut être retardé jusqu'à l'arrivée d'autres messages de contrôle avant de les envoyer en un seul et même paquet [83].

Ces techniques d'agrégation réseau reposent sur le "piggybacking" de messages. Le protocole TCP offre un mécanisme d'agrégation basé sur l'algorithme de Nagle [84][85], activé par défaut dans la plupart des implémentations TCP. Étant donné que les paquets TCP/IP comportent un en-tête de 40 octets, l'envoi d'une multitude de petits messages peut entraîner une surcharge et une congestion du réseau. L'algorithme de Nagle consiste à stocker temporairement les données jusqu'à ce qu'un accusé de réception (ACK TCP) soit reçu ou que la mémoire tampon soit saturée.

L'application de telles techniques permet d'optimiser les communications point à point

en agrégeant certains messages adressés au même destinataire. Toutefois, l'agrégation peut également être mise en œuvre sur un graphe de communications plus complexe. Ceci est connu comme le problème du *joint replenishment problem* (JRP) [86] [87]. Pour l'illustrer, nous pouvons considérer un scénario où l'on cherche à optimiser les expéditions de marchandises d'un fournisseur vers des détaillants, en passant par un entrepôt partagé. Dans ce cadre, l'agrégation des expéditions s'opère à deux niveaux. D'une part, les articles destinés à différents détaillants peuvent être expédiés ensemble vers l'entrepôt. D'autre part, plusieurs articles destinés à un même détaillant peuvent être expédiés de l'entrepôt vers ce détaillant. Les demandes en attente génèrent des coûts de stockage jusqu'à ce qu'elles soient expédiées. L'objectif est de minimiser la somme totale des coûts d'expédition et des coûts de stockage.

Bienkowski et al. ont introduit la notion formelle de *Multiple-Level Agregation* (MLAP) [88]. Les différentes couches d'agrégation sont représentées à l'aide d'un arbre. Ainsi, le problème des ACK TCP et du JRP peuvent être vus comme des problèmes d'agrégation sur des arbres pondérés de profondeurs respectives 1 et 2. Dans le cas des ACK TCP, l'arbre est réduit à une simple arête reliant l'expéditeur au destinataire. En revanche, pour le JRP, l'arbre est structuré avec la racine représentant le fournisseur, un fils représentant l'entrepôt, et plusieurs petits-fils représentant les détaillants. Une expédition peut être modélisée comme un sous-arbre de cet arbre, les poids des arêtes correspondant aux coûts d'expédition. Cela s'étend naturellement aux arbres de profondeur quelconque, où l'agrégation est permise à chaque niveau.

Ainsi, l'agrégation de messages multi-niveaux a été étudiée dans divers contextes au sein des réseaux de communication. Badrinath et Sudame ont, par exemple, proposé un mécanisme d'agrégation appelé Gathercast [89]. Gathercast regroupe les petits paquets de différentes sources en un paquet plus grand vers une même destination, puis régénère les paquets d'origine à la destination. Ce processus réduit ainsi le nombre de paquets traités par les routeurs dans le réseau, contribuant à diminuer la congestion et à améliorer l'efficacité du réseau. Khanna et al. [90] ont, quant à eux, cherché à optimiser l'agrégation de données dans des arbres de multidiffusion (multicast). Un arbre de multidiffusion est donné avec un coût de communication

associé à chaque lien. Lorsqu'un paquet arrive à un nœud récepteur (feuille ou nœud interne), un message de contrôle (généralement un ACK) doit être transmis à la racine. Les nœuds peuvent retarder l'envoi de ces messages afin de les agréger, réduisant ainsi le coût de communication.

Cependant, ces mécanismes opèrent au niveau des couches réseau et ne tiennent pas compte des caractéristiques des applications. Cela peut affecter négativement les applications sensibles au temps telles que les chats ou encore le streaming, qui ne tolèrent aucun retard dans la transmission des messages.

Cependant, ces mécanismes opèrent au niveau des couches réseau et ne tiennent pas compte des caractéristiques des applications. Cela peut affecter négativement les applications sensibles au temps telles que les chats ou encore le streaming, qui ne tolèrent aucun retard dans la transmission des messages.

2.6.2 Agrégation de messages au niveau applicatif

Il est également possible d'agréger des messages en tenant compte du protocole applicatif. Par exemple, HotStuff [91], un protocole de réplication tolérant aux fautes byzantines, économise les messages en regroupant les phases des instances de consensus consécutives. Cela permet de réduire les communications nécessaires entre les nœuds du réseau. HotStuff est un protocole qui repose sur un leader et sur un système partiellement synchrone où un délai de transmission maximum est à terme respecté.

Le regroupement (batching) est une optimisation courante dans les systèmes de communication, offrant généralement des gains de performance significatifs [92]. Le batching est un exemple d'agrégation qui consiste à rassembler plusieurs requêtes dans une seule instance du protocole. Cela permet de réduire la surcharge (overhead) par requête et entraîne généralement un débit plus élevé. Santos et Schiper [93] ont étudié les performances de Paxos dans ce contexte. Le regroupement peut être appliqué à Paxos de manière à ce que le leader, au lieu de proposer une seule requête par instance, regroupe plusieurs requêtes dans une même instance. Une fois l'ordre de

l'ensemble des requêtes déterminé, l'ordre des requêtes individuelles est défini par une règle déterministe appliquée aux identifiants des requêtes. Santos et Schiper proposent également de coupler le batching au pipelining afin d'améliorer les performances. Ainsi, en plus du batching, Paxos peut être étendu pour permettre au leader d'exécuter plusieurs instances en parallèle [17]. Dans ce cas, lorsque le leader reçoit une nouvelle requête, il peut décider de lancer immédiatement une nouvelle instance, même si d'autres instances ne sont pas encore décidées.

Cependant, ces mécanismes d'agrégation de messages sont spécifiques à une seule instance d'application, ce qui signifie qu'ils sont appliqués à un seul processus ou flux de données de l'application.

2.7 Conclusion

Dans ce chapitre, nous avons présenté le problème du consensus dans les systèmes distribués. Le consensus joue un rôle fondamental dans la construction de systèmes robustes et fiables permettant de maintenir la cohérence. Nous avons décrit les propriétés du consensus ainsi que les limitations imposées aux environnements asynchrones par FLP. En réponse à ces défis, nous avons abordé plusieurs approches visant à contourner ce résultat d'impossibilité.

Parmi ces approches, l'affaiblissement du problème du consensus se distingue avec des algorithmes tels que Paxos et ses variantes largement utilisées dans le domaine du Cloud. Parmi les algorithmes existants, beaucoup se concentrent sur la réduction de la latence au prix d'un nombre élevé de messages. En effet, malgré leur efficacité, ils nécessitent d'envoyer beaucoup de messages via des séries de broadcasts. Cependant, selon le contexte, la bande passante est une ressource critique. Des recherches mettent en évidence la bande passante comme l'un des principaux points de congestion des infrastructures réseau des datacenters. Pour pallier ce problème, une solution consiste à utiliser des mécanismes d'agrégation dans les couches inférieures de la pile réseau. Le principe repose sur le multiplexage de plusieurs messages adressés

au même destinataire. Cependant, cette stratégie est agnostique aux applications, ce qui empêche une agrégation intelligente des messages. En effet, la couche réseau ne peut pas déterminer si un message est critique et/ou bloquant pour la vivacité de l'application. Nous nous sommes donc interrogés sur la possibilité de réduire ce coût. En réponse, nous avons proposé un mécanisme d'agrégation opportuniste qui s'applique aux algorithmes à phases. Il permet de diminuer la consommation de bande passante via une agrégation intelligente tout en limitant la dégradation de la latence.

Dans de nombreux systèmes distribués, les processus doivent coopérer pour atteindre un consensus. Cependant, dans certains environnements plus complexes, les processus peuvent être soumis à des contraintes internes (géographiques, algorithmiques ou matérielles). C'est notamment le cas des systèmes multi-agents (gestion d'agenda, coordination de véhicules autonomes, etc.) ou des mécanismes liés à l'allocation de ressources (cloud, Software Defined Network, etc.). Ces contraintes peuvent influencer le vote d'un processus. Le consensus traditionnel ne permet pas de prendre répondre aux contraintes des nœuds. Nous proposons donc une extension du consensus, appelée Consensus f-Révocable. Cette extension introduit la notion de processus contraints et remet en question l'irrévocabilité des décisions, permettant à la minorité de révoquer une décision qui viole ses contraintes. Nous proposons également deux algorithmes répondant à cette nouvelle définition du consensus.

Chapitre

3

OMAHA : un mécanisme d'agrégation opportuniste

Sommaire

3.1 Introduction

Dans les environnements cloud, les applications sont déployées sur une infrastructure distribuée et de grande échelle. Grâce à la virtualisation, plusieurs applications partagent simultanément les mêmes ressources physiques. La bande passante est une ressource particulièrement critique. Des travaux ont montré que la bande passante est l'un des principaux goulots d'étranglement dans les réseaux de datacenters [1, 2, 3, 4]. De plus, certaines études soulignent que les petits messages représentent une grande part du trafic réseau, ce qui entraîne une utilisation importante de la bande passante par les en-têtes de ces messages [5, 6, 7]. Par exemple, Benson et al. [6] ont constaté que 50% des paquets avaient une taille inférieure à 300 octets.

Dans les datacenters, plusieurs applications sont très consommatrices en bande passante, ce qui peut saturer le réseau et augmenter les délais de traitement. Des recherches récentes ont exploré des solutions pour mieux équilibrer l'utilisation de la bande passante, en s'appuyant soit sur la connaissance des besoins en bande passante des applications [8, 9], soit sur des optimisations au niveau de la couche MAC [10, 11]. Une autre approche repose sur des mécanismes d'agrégation des messages dans les couches inférieures du réseau, en regroupant plusieurs messages à destination d'un

même nœud. Toutefois, ces méthodes ne prennent pas en compte les besoins spécifiques des applications, empêchant ainsi une gestion plus fine et efficace de l'agrégation. Par exemple, la couche réseau ne peut pas identifier si un message est crucial pour le bon déroulement de l'application ou s'il peut être retardé sans impact négatif sur les performances.

De nombreuses applications distribuées s'appuient sur des algorithmes à phases, notamment pour la gestion des données [12, 13, 14, 15, 16]. Ces algorithmes, tels que le consensus (Paxos [17]), les protocoles de validation atomique (Zookeeper Atomic Broadcast [18], ou encore le two-phase commit [19]), nécessitent de nombreux échanges de messages entre les nœuds du système. Par exemple, l'algorithme Paxos [17], largement utilisé dans les systèmes distribués [20], génère une diffusion de messages dont la complexité est quadratique par rapport au nombre de nœuds.

Dans ces algorithmes à phases, les échanges de messages entre les nœuds peuvent être anticipés, ce qui permet de prédire quels nœuds communiqueront prochainement. Cela offre l'opportunité de retarder certains messages pour les regrouper et ainsi économiser la bande passante. Sur cette base, nous avons conçu OMAHA (Opportunistic Message Aggregation for pHase-based Algorithms), un mécanisme d'agrégation de messages générique, opportuniste et non intrusif. OMAHA réduit de manière significative la consommation de bande passante sans affecter fortement la latence, ce qui contribue à diminuer la congestion réseau et, par conséquent, à accélérer le traitement des applications. Ce mécanisme est applicable dans un contexte où plusieurs applications s'exécutent simultanément sur une même infrastructure et permet d'améliorer l'efficacité globale en réduisant la charge réseau et les délais de traitement. En surchargeant l'API de communication traditionnelle, il fournit une couche intermédiaire entre les applications et la pile réseau. Le cœur de cette nouvelle API réside dans la définition d'une nouvelle primitive "send", qui offre aux utilisateurs la possibilité de trouver un compromis entre l'économie de bande passante et la dégradation de la latence selon leurs besoins.

Dans ce chapitre, nous introduisons en détail notre mécanisme d'agrégation, en

abordant ses spécificités et son implantation. Nous poursuivons par une étude expérimentale pour tester et évaluer l'efficacité de notre mécanisme. Par la suite, nous expliquons le processus de dimensionnement des paramètres d'OMAHA, visant à déterminer la configuration permettant d'économiser le plus de bande passante tout en respectant un taux de dégradation de la latence imposé par l'utilisateur. Enfin, nous abordons la configuration des paramètres d'un point de vue théorique et pratique, offrant ainsi une vue d'ensemble sur le réglage des paramètres pour améliorer les performances de notre mécanisme.

3.2 Principe du mécanisme d'agrégation

Cette section introduit le corps d'OMAHA. Dans la suite, nous supposerons que chaque nœud et application disposent d'un identifiant unique. Chaque nœud maintient également un tampon de messages pour chaque destinataire.

Lors de la mise en place d'un mécanisme d'agrégation, deux questions essentielles se posent :

1. Doit-on mettre dans le tampon un nouveau message (et donc le retarder) ou l'envoyer immédiatement ?
2. Quand les messages mis dans le tampon doivent-ils être envoyés ?

Une manière simple de répondre à ces questions consiste à retarder systématiquement l'envoi des messages, et ce, pendant une durée fixe. Ce mécanisme agnostique à l'application sera utilisé comme référence dans notre étude expérimentale (Section 3.4).

Pour OMAHA, la réponse à la première question est conditionnée par la criticité du message pour l'algorithme. En effet, plus un message est critique, plus il risque de bloquer l'exécution de l'algorithme. Pour la seconde question, nous profitons des différentes phases de l'algorithme pour retarder l'envoi des messages sans impacter significativement les performances de l'application.

Dans un premier temps, nous introduirons le mécanisme d'agrégation agnostique à l'application et basé sur le temps. Ensuite, nous présenterons notre approche opportuniste appliquée au protocole Paxos. Enfin, nous détaillerons notre nouvelle API.

3.2.1 Approche agnostique basée sur le temps

L'approche temporelle agnostique agrège les messages sur une période spécifique ou jusqu'à ce que le tampon soit plein. Elle ne prend pas en compte les caractéristiques applicatives. Ainsi, pour répondre aux questions posées précédemment, il faut :

1. Mettre systématiquement les messages en tampon.
2. Choisir arbitrairement une durée de mise en tampon.

Cette méthode, facile à mettre en œuvre, améliore l'utilisation de la bande passante en réduisant considérablement le nombre de messages transmis.

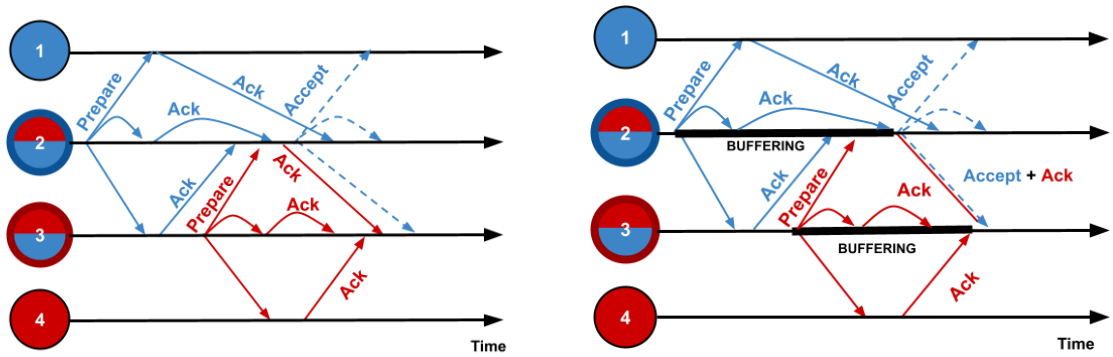
Néanmoins, cette approche indépendante de l'application retarde tous les messages émis. Par conséquent, elle ne distingue pas l'importance des messages, ce qui peut entraîner une dégradation significative de la latence. Il est ainsi possible de mettre en tampon un message à destination d'un nœud même si aucun autre message à destination de ce nœud ne doit être envoyé, ce qui se révèle inutile et coûteux. De plus, elle ne tient pas compte de la charge du système. Bien qu'elle soit efficace dans des situations de forte charge, l'approche agnostique devient moins performante lorsque la charge diminue ou fluctue.

3.2.2 Approche opportuniste

Notre approche s'applique aux applications reposant sur des algorithmes à phases. En exploitant la connaissance du schéma de communication de l'algorithme, il est possible

de réaliser une mise en tampon intelligente des messages. Ainsi, nous parvenons à proposer un compromis entre l'économie de la bande passante et la dégradation de la latence, quel que soit le niveau de charge.

La Figure 3.1 illustre le principe de notre approche appliqué au protocole Paxos [17] dans un contexte multi-applicatif. Chaque application opère de manière autonome par rapport aux autres et s'exécute sur son propre ensemble de nœuds physiques. Cependant, nous supposons que ces ensembles s'intersectent.



(a) Sans mécanisme d'agrégation (b) Avec OMAHA

FIGURE 3.1 – Début d'une instance de Paxos

Dans cet exemple, nous avons deux ensembles de nœuds, chacun représentant une application en cours d'exécution. L'ensemble bleu comprend les nœuds $1, 2, 3$, tandis que l'ensemble rouge est constitué des nœuds $2, 3, 4$. Chaque ensemble peut démarrer des instances de Paxos selon ses besoins. Dans cet exemple, les nœuds 2 et 3 agissent en tant que leaders respectivement pour les ensembles bleu et rouge.

La Figure 3.1(a) montre une exécution sans mécanisme d'agrégation. Initialement, le nœud 2 démarre une instance de Paxos pour l'ensemble bleu, puis le nœud 3 lance une instance pour l'ensemble rouge.

La première étape consiste à déterminer le moment propice pour retarder un message. Nous observons qu'une fois que le nœud 2 envoie un message `prepare` à l'ensemble

bleu (début de la phase 1), il diffusera un message `accept` dès qu'il recevra un quorum de messages `ack` (début de la phase 2). Le protocole garantit que tout nœud envoyant un message `prepare` contactera le même ensemble de destinataires à court terme (une fois le quorum atteint). Ainsi, il devient possible d'utiliser cette connaissance pour mettre en tampon des messages provenant d'une autre application et destinés aux nœuds du premier ensemble. Nous appelons ce mécanisme un **pledge** : un nœud peut mettre en tampon et retarder un message s'il s'engage à l'envoyer ultérieurement.

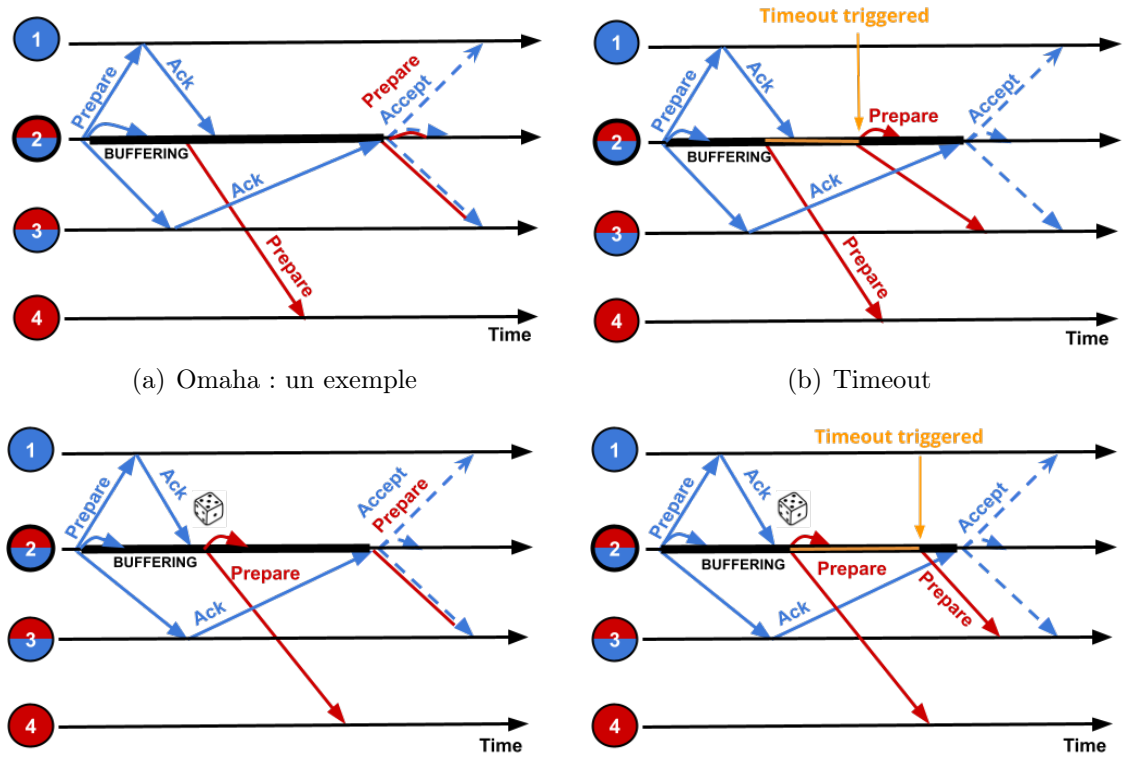
Dans la Figure 3.1(b), nous voyons que le nœud 2 appartient aux deux ensembles. Le mécanisme de pledge identifie ici la possibilité de mettre en tampon le message `ack` de l'ensemble rouge destiné au nœud 3 afin de l'agréger avec le message `accept` de l'ensemble bleu.

En résumé, OMAHA peut prédire quand un nœud A enverra un message à un nœud B en utilisant la connaissance des phases de l'algorithme. Pour lancer une nouvelle phase, l'attente d'un quorum de messages est nécessaire. Cette attente étant intrinsèque à l'algorithme, nous pouvons l'utiliser pour agréger les messages de A à B provenant d'autres applications.

3.2.3 Réduction de la latence

Pour limiter la dégradation de la latence, nous avons introduit deux mécanismes supplémentaires : le *timeout* et la *probaBuf*. Le mécanisme de *timeout* permet de borner le temps de mise en tampon d'un message. Le mécanisme *probaBuf*, quant à lui, associe à un message une probabilité d'insertion dans le tampon.

La Figure 3.2(a) illustre une exécution d'OMAHA sans ces deux nouveaux mécanismes. Le leader (le nœud 2 pour les deux ensembles) initie un Paxos bleu puis attend une majorité d'acquittements bleus. Il débute ensuite un Paxos rouge. Le nœud 4, n'étant pas impliqué dans le Paxos bleu, reçoit le message *prepare* directement. En revanche, pour les nœuds 2 et 3, nous pouvons retarder le *prepare* rouge pour l'agréger avec le *accept* bleu.



(a) Omaha : un exemple

(b) Timeout

(c) ProbaBuf

(d) ProbaBuf et timeout

FIGURE 3.2 – Mécanisme OMAHA : *timeout* et *probaBuf*

Timeout Dans la Figure 3.2(b), les *prepare* bleus à destination des nœuds 2 et 3 peuvent être retardés. Cependant, l'acquittement bleu du nœud 3 tarde à arriver. Pour limiter la dégradation de la latence et assurer la vivacité de l'application, chaque message mis en mémoire tampon arme un timeout. Si il est déclenché, alors les messages présents dans le tampon sont envoyés.

Probabuf Si un pledge peut s'appliquer à un message, une probabilité de mise en tampon lui est attribuée appelée *probaBuf*. Dans la Figure 3.2(c), le *prepare* à destination du nœud 2 est envoyé directement, tandis que celui du nœud 3 est retardé.

Ces deux paramètres peuvent être couplés (Figure 3.2) permettant ainsi à l'utilisateur de trouver un compromis entre le gain en bande passante et la dégradation de la latence. Le bon paramétrage d'OMAHA est donc essentiel pour ses performances. Ce point sera discuté dans la section 3.5.

3.3 Algorithme d'agrégation

Cette section décrit les algorithmes d'OMAHA, du mécanisme de *pledge*, ainsi qu'une application d'OMAHA à l'algorithme de Paxos.

3.3.1 Description de l'algorithme

Le pseudo-code du mécanisme de pledge est présenté dans l'Algorithme 1 et peut être appliqué à n'importe quel algorithme à phases. Afin de rester générique et peu intrusif, nous étendons l'API réseau en introduisant trois nouveaux paramètres à la primitive *send (msg, dests)* (voir ligne 21) :

- `probaBuf` : la probabilité de mise en tampon du message si un pledge est détecté. Zéro signifie que le message sera envoyé immédiatement (l'équivalent du *send* original), tandis que 1 (100%) indique que le message sera retardé. Opter pour une probabilité plutôt qu'un simple booléen permet de prendre en compte l'importance du message pour l'exécution de l'algorithme. Cette probabilité impacte directement la latence : une valeur plus élevée augmente la probabilité de mise en tampon et, par conséquent, accroît la latence.
- `timeout` (noté `t` dans le pseudo-code) : la durée maximale pendant laquelle le message peut être mis en tampon. Cela garantit qu'un message retardé à la suite d'un pledge sera finalement transmis, préservant ainsi les propriétés de sûreté et de vivacité de l'algorithme.
- `app` : l'identifiant de l'application.

Durant une période d'attente, notamment lors de l'attente d'un quorum, le nœud entre en "période de pledge". Deux primitives sont fournies à l'application pour indiquer le début et la fin d'une telle période dans son algorithme :

- `beginPledge(app, futureDests)` : marque le début d'une période de pledge pour l'application `app` (ligne 6). `futureDests` représente l'ensemble des nœuds qui seront contactés à la fin de cette période. Il devient alors possible de mettre en tampon tout message destiné à ces nœuds.
- `pledgedSend(msg,dests,probaBuf, timeout, app)` : marque la fin d'une période de pledge pour l'application `app` (ligne 16) et l'envoi du message `msg`. L'envoi de ce message suit le même protocole que celui des autres messages utilisant la primitive *send* surchargée.

Ces primitives définissent l'API d'une nouvelle couche, positionnée entre les couches réseau et applicative. Au sein de cette couche, chaque nœud maintient un tampon pour chaque destinataire (ligne 4). Les messages mis en tampon sont des messages applicatifs. Ainsi, l'envoi du tampon équivaut à un unique message réseau. Chaque tampon est associé à une échéance (indiqué par le paramètre *timeout*), déterminant le moment où les messages seront envoyés au plus tard. Cette échéance est définie par l'utilisateur et varie selon le type de message (lignes 32 et 33). Lors de l'appel à la primitive *send*, deux situations peuvent se présenter :

- si `probaBuf = 0`, le message est envoyé immédiatement sans passer par OMAHA. (ligne 23).
- si `probaBuf > 0`, la décision de mettre en tampon le message dépend à la fois de la valeur de `probaBuf` et de la période dans laquelle se trouve le nœud, en l'occurrence s'il est en période de pledge pour les destinataires (ligne 30). Ainsi :
  - en l'absence de période de pledge pour un destinataire r, le message doit être transmis à r (ligne 40) . Si le tampon associé aux destinataires n'est pas vide, le message est agrégé avec les autres messages présents (ligne 29). L'ensemble est ensuite envoyé immédiatement.
  - en période de pledge, avec une probabilité `probaBuf`, le message est ajouté

aux tampons de chaque destinataire, et leurs échéances sont mises à jour (lignes 29 à 34). Autrement dit, le message, ainsi que ceux déjà en attente, sont transmis immédiatement avec une probabilité de $1 - \texttt{probaBuf}$.

Enfin, dès qu'une échéance est atteinte, tous les messages sont envoyés ensemble en tant que message réseau unique.

3.3.2 Intégration du mécanisme : application à Paxos

Dans cette section, nous illustrons un exemple concret d'utilisation d'OMAHA à travers l'algorithme de Paxos. Les appels à notre API doivent être intégrés directement dans l'algorithme. Pour ce faire, il faut repérer le début et la fin d'une période de pledge en utilisant les fonctions `beginPledge` et `pledgedSend`, respectivement.

Dans l'Algorithme 2, nous nous concentrons sur la *phase de préparation* jusqu'au début de la *phase d'acceptation*. Nous supposons que l'identifiant de l'application est connu (ligne 6) et que chaque type de message est associé à un timeout (ligne 7) et à une probabilité de mise en tampon (ligne 8).

Quand le leader démarre une nouvelle instance de Paxos (ligne 9), il envoie un message de type *prepare* (ligne 12) à tous les nœuds de cette application. Pour passer à la phase suivante, le leader attend une majorité de messages *Ack*. OMAHA détecte cette attente via l'appel à la fonction `beginPledge` (ligne 13). La réception d'une majorité de *Ack* par le leader marque la fin de cette période d'attente (ligne 22). Le leader envoie alors un message *accept* via la fonction `pledgeSend`, signalant à OMAHA la fin de la période de pledge (ligne 29).

Algorithme 1 : L'algorithme du mécanisme de *pledge*

```
1  Local variables :
2  begin
3      curPledges : Map of (app : application id, nodes :Set of node ids)
       /* map associating an application id with a set of node ids for which
          we know they will be contacted in a near future                    */
4      bufs : Map of (nodeid,(msgs : Set of Message, deadline))
       /* map associating a recipient id with the list of buffered messages
          that are intended for it and the associated deadline of sending    */
5  end
6  Primitive beginPledge(app, futureDests) :
7  begin
8      put(app,futureDests) in curPledges
9  end
10 Primitive sendBuff(buff_dest, dest) :
11 begin
12     cancel any scheduling related to buff_dest
13     networkSend(buff_dest.msgs) to dest
14     clear buff_dest
15 end
16 Primitive pledgedSend(msg, dests, probaBuf, t, app) :
17 begin
18     removeEntry(app) in curPledges
19     send(msg, dests, probaBuf, t, app)
20 end
21 Primitive send(msg, dests, probaBuf, t, app) :
22 begin
23     if probaBuf == 0 then
24         networkSend(msg) to dests
25         return
26     end
27     for all d ∈ dests do
28         flushing ← true
29         add msg to bufs[d].msgs
30         if ∃(app′, nodes) ∈ curPledges where d ∈ nodes and app ≠ app′ then
31             if random() < probaBuf then
32                 if bufs[d].deadline does not exist or bufs[d].deadline > now + t then
33                     bufs[d].deadline ← now + t
34                     schedule sendBuff(bufs[d], d) at bufs[d].deadline
35                 end
36                 flushing ← false
37             end
38         end
39         if flushing == true then
40             sendBuff(bufs[d], d)
41         end
42     end
43 end
```

Algorithme 2 : Phase de préparation de l'algorithme Paxos pour un processus p_i

```
1  Original local variables to Paxos :
2  ballot : (numBallot, p_k) initially (0,⊥)
3  acceptBal initially ⊥
4  acceptVal initially ⊥
5  Dedicated local variables to OMAHA :
6  app_id
   /* unique id for each application                                    */
7  timeouts : Map of (message type, timeout)
   /* timeout value for each type of message                           */
8  probaBuf : Map of (message type, probability)
   /* probaBuf value for each type of message                          */

9  Upon Propose (new_val) :
10 begin
11 |  ballot ← Ballot(ballot.numBallot++, p_i )
12 |  send(<Prepare, ballot >, set_k, probaBuf[Prepare], timeout[Prepare], app_id)
13 |  beginPledge(app_id, all nodes of app_id)
14 end

15 Upon reception of message Prepare(bal) from p_j :
16 begin
17 |  if ballot ≤ bal then
18 |  |  ballot ← bal
19 |  |  send(< Ack, bal, acceptBal, acceptVal > , {p_j}, probaBuf[Ack], timeout[Ack],
   |  |    app_id)
20 |  end
21 end

22 Upon reception of message Ack (bal, acceptBal, acceptVal) from a majority of
     nodes :
23 begin
24 |  if all acceptVal = ⊥ then
25 |  |  val ← new_val
26 |  else
27 |  |  val ← the value associated with the biggest bal for all acceptBal
28 |  end
29 |  pledgedSend(< Accept, ballot, val > , all nodes of app_id, probaBuf[Accept],
   |    timeout[Accept], app_id)
30 end
```

3.3.3 *Implantation d'OMAHA*

OMAHA peut être implémenté sous forme de bibliothèque. Pour améliorer ses performances, il peut être déployé au niveau du noyau ou de l'hyperviseur. Ce choix permet d'éviter l'engorgement associé à une éventuelle couche middleware. L'implantation d'OMAHA a été conçue pour TCP/IP. Nous avons fait ce choix car son utilisation est très répandue notamment dans les applications utilisant des algorithmes à phases (Apache ZooKeeper, Google Spanner, Apache Kafka, etc.).

La Figure 3.3 présente l'implémentation d'un nœud sans OMAHA. Les messages applicatifs sont transmis d'un nœud à un autre via le port de l'application. Les messages applicatifs peuvent également être chiffrés, l'application 2 représentée en rouge illustre ce cas de figure.

La Figure 3.4 décrit l'intégration d'OMAHA dans un nœud. Chaque application conserve ses ports d'origine pour la transmission, indépendamment de l'utilisation d'OMAHA. Les applications utilisant OMAHA encapsulent sa bibliothèque afin d'utiliser les primitives *send* et *beginPledge*. Le serveur OMAHA maintient des tampons pour chaque nœud et possède son propre port, ce qui lui permet de multiplexer des messages destinés à plusieurs ports simultanément.

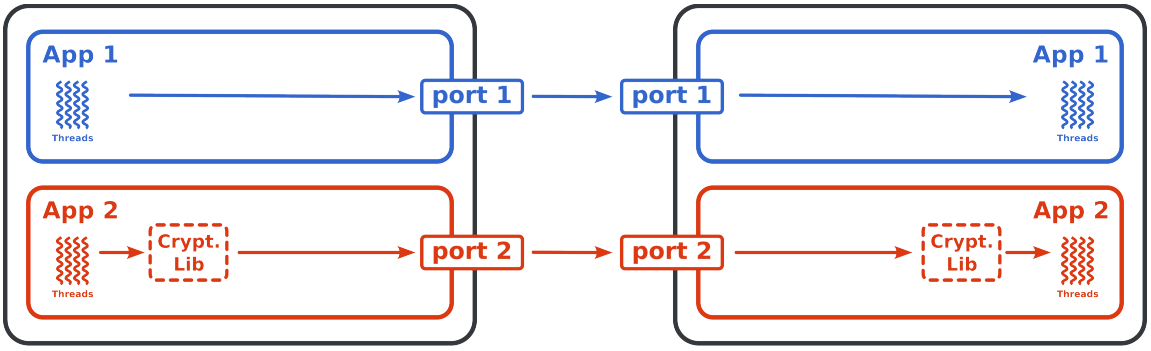


FIGURE 3.3 – Applications natives sans OMAHA

La Figure 3.4 illustre le flux des messages :

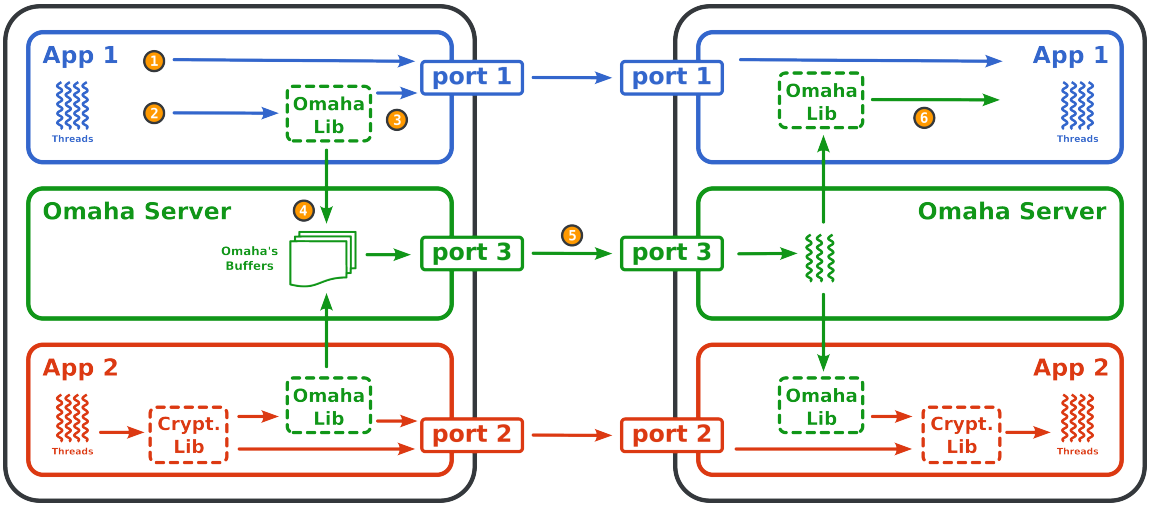


FIGURE 3.4 – Applications avec OMAHA

1. Les applications transmettent directement les messages qui ne peuvent pas être mis en mémoire tampon.
2. Les messages pouvant être mis en tampon passent par Omaha.
3. En fonction de la valeur de *probaBuf*, le message peut être envoyé directement.
4. Sinon, il est transmis au serveur OMAHA via une mémoire partagée pour être stocké dans le tampon du nœud de destination.
5. Lorsqu'une "période de pledge" se termine ou qu'un timeout expire, le contenu du tampon est envoyé en tant que message réseau unique.
6. À la réception de ce message, le serveur OMAHA se charge de distribuer les messages aux applications concernées.

3.4 Évaluation

Cette section présente les différentes expériences réalisées pour tester et évaluer OMAHA. Notre mécanisme a été évalué à la fois par simulation et dans un environ-

nement réel [94].

3.4.1 Environnement d'expérimentation et configuration

Cette section décrit l'environnement dans lequel nos expériences ont été réalisées. OMAHA peut être intégré à divers environnements, qu'ils soient simulés ou réels, grâce à une bibliothèque développée au sein de notre équipe (DELYS) [95].

Paramètres d'infrastructure

Afin d'analyser l'impact de différentes combinaisons de paramètres, nous avons initialement conduit nos expériences sur Peersim [96], un simulateur à événements discrets (Sections 3.4.2 à 3.4.5). Dans le but de valider nos résultats, nous avons ensuite testé OMAHA dans un environnement réel via la plateforme Grid'5000, également appelée g5k [94] (Section 3.4.6).

Dans les deux environnements, nous avons déployé 15 nœuds. Nous considérons un graphe de communication complet, permettant à chaque nœud d'échanger avec n'importe quel autre. Sur la plateforme g5k, chaque nœud est installé sur un hôte physique dédié [1]. La communication entre les nœuds s'effectue via des sockets TCP/IP.

La latence moyenne d'un RTT s'élève à 60 millisecondes, et suit une distribution normale avec un écart type de 10%. Sur la plateforme g5k, la latence réseau a été intégrée pour reproduire la même configuration.

Dans les sections 3.4.2, 3.4.3, 3.4.4, et 3.5, nous supposons un système fiable. Dans cette configuration, aucun nœud ne subit de défaillance, aucun message n'est perdu ni dupliqué, et les nœuds exécutent correctement leur protocole. Cette approche simplifie l'analyse des résultats en mettant l'accent sur l'impact du mécanisme d'agrégation.

1. Configuration d'un hôte physique g5k : 2 CPU Intel Xeon E5-2660, 8 cœurs par CPU, 64 Go de RAM, 1863 Go de stockage, 1 x 10 Gb Ethernet, fonctionnant sous Linux 5.10.0-16-amd64 avec Java 11

Enfin dans la section 3.4.5, nous abordons l'impact de la perte de messages. Notre mécanisme, reposant sur l'agrégation de plusieurs messages applicatifs en un seul message réseau, pourrait être particulièrement sensible à la perte.

Algorithmes à phases considérés

Nous avons étudié et implanté les quatre algorithmes à phases, présentés dans la Section 2.5 : trois variantes de l'algorithme Paxos (Paxos [17], Fast Paxos[81] et Fast Byzantine Paxos [82]) ainsi que l'algorithme de diffusion atomique Zookeeper (ZAB [18]).

Nous comparons les performances d'OMAHA avec :

- L'algorithme sans mécanisme d'agrégation.
- L'algorithme couplé à un mécanisme d'agrégation agnostique (cf. 3.2.1), qui stocke systématiquement les messages en mémoire tampon.

Workload

Les performances des mécanismes d'agrégation sont directement liées à la charge du système, et par conséquent, au nombre d'instances qui s'exécutent en simultanées. Dans les expériences à suivre, nous évaluons chaque mécanisme avec trois niveaux de charge : faible, moyen et élevé. Chaque modèle correspond respectivement à une moyenne de 2, 5 et 10 instances concurrentes du même algorithme.

Ces instances concurrentes ne sont pas synchronisées, ce qui signifie qu'une instance peut démarrer indépendamment de l'état des autres instances en cours d'exécution. Ainsi, elles ne se trouvent pas toutes dans la même phase simultanément. De plus, pour chaque instance, nous choisissons arbitrairement son nœud leader avant de lancer l'instance. Les 15 nœuds participent à chaque instance.

Métriques

Pour comparer l'efficacité de chaque mécanisme d'agrégation, nous définissons les deux métriques suivantes :

- La **latence moyenne**, qui correspond au temps écoulé entre le démarrage du protocole par un nœud et le moment où un quorum de nœuds parvient à une décision (que ce soit un consensus sur une valeur dans le cas des algorithmes de consensus Paxos ou la confirmation d'une transaction pour ZAB).
- La **consommation de bande passante**, qui correspond à la quantité totale de données générées par la couche réseau (IP) tout au long de l'expérience.

Chaque expérience se conclut à l'achèvement de la 3000ème instance de l'algorithme. Les 100 premières instances sont exclues des mesures afin de considérer une charge stable.

Il est important de noter que dans les tableaux et graphiques suivants, les valeurs ne sont pas absolues, mais relatives aux performances du protocole sans mécanisme d'agrégation. Ainsi, ces deux métriques sont exprimées respectivement en termes de dégradation de la latence (moins la valeur est élevée, meilleure est la performance) et d'économie de bande passante (plus la valeur est élevée, meilleure est la performance).

Nous avons étudié l'impact des deux paramètres suivants :

- Le paramètre *timeout*, qui définit la durée maximale qu'un message peut passer en mémoire tampon avant d'être envoyé. Il s'applique à l'approche agnostique et à OMAHA.
- Le paramètre *probaBuf*, qui indique la probabilité qu'un message soit mis en mémoire tampon par OMAHA si un *pledge* peut être appliqué.

Ces deux paramètres sont à définir par l'utilisateur pour chaque message en utilisant l'API d'OMAHA via la primitive *send* (Section 3.3). Dans les sections 3.4.2, 3.4.3, 3.4.4, et 3.4.5, nous considérons que ces paramètres restent constants, indépendamment

de la phase ou du type de message. Toutefois, dans la section 3.5, nous explorons différentes valeurs de *probaBuf* en fonction de la criticité du type de message.

3.4.2 Impact du paramètre probaBuf

Les tableaux 3.1 et 3.2 présentent les performances d'OMAHA sur des instances de Paxos. Pour cela, nous avons fait varier la charge et le paramètre *probaBuf*, tout en fixant le paramètre *timeout* à un RTT. Cette valeur a été choisie car elle correspond au temps moyen pour envoyer un message et obtenir une réponse.

Buffering probability Load	10	20	30	40	50	60	70	80	90	100
Low	0.6%	1.2%	1.8%	2.3%	2.9%	3.5%	4.0%	4.5%	5.1%	5.6%
Medium	1.6%	3.1%	4.6%	6.1%	7.6%	9.1%	10.5%	12.0%	13.4%	14.7%
High	3.4%	6.4%	9.5%	12.6%	15.9%	18.9%	22.2%	25.3%	28.1%	30.2%

TABLE 3.1 – Impact du paramètre *probaBuf* sur l'algorithme classique Paxos : gain en bande passante

Buffering probability Load	10	20	30	40	50	60	70	80	90	100
Low	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%	0.2%	0.3%	0.3%	0.3%
Medium	0.1%	0.3%	0.4%	0.5%	0.7%	0.8%	1.0%	1.2%	1.4%	1.6%
High	0.2%	0.4%	0.7%	1.0%	1.4%	1.8%	2.4%	3.2%	4.2%	5.4%

TABLE 3.2 – Impact du paramètre *probaBuf* sur l'algorithme classique Paxos : dégradation de la latence

Dans le tableau 3.1, nous observons que le gain en bande passante augmente de manière linéaire avec *probaBuf*, quel que soit le niveau de charge du système. Cette observation est cohérente car la valeur de *probaBuf* est la même pour tous les messages.

Le gain en bande passante varie selon la charge : il est plus important en cas de charge élevée. En effet, l'exécution simultanée de plusieurs instances de Paxos favorise la mise en place de *pledges*. À l'inverse, lorsque la charge est faible, peu d'instances de Paxos s'exécutent simultanément. Dans ce cas, OMAHA a plus de difficultés à

anticiper les communications futures, les gains en bande passante sont donc moins importants.

Le tableau 3.2 décrit l'effet d'OMAHA sur la latence de l'algorithme. Nous observons que l'impact de la probabilité diffère selon la charge. En situation de forte charge, la mise en place de nombreux *pledges* entraîne des retards supplémentaires dans la transmission des messages. Malgré cela, une économie de bande passante allant jusqu'à 30.2% peut être réalisée, avec une légère dégradation de la latence (5.4%).

3.4.3 Impact du paramètre timeout

Nous nous intéressons à présent à l'impact du paramètre *timeout* sur l'algorithme de Paxos. Nous présentons les résultats dans les tableaux 3.3 et 3.4. Cette fois ci, nous faisons varier la valeur du paramètre *timeout* entre *RTT/4* et *1.2 RTT* tout en fixant une probabilité de mise en tampon à 90%.

Timeout - Load	RTT/4	RTT/2	RTT	1.2RTT
Low	1.3%	3.7%	5.1%	5.2%
Medium	5.6%	9.9%	13.4%	13.8%
High	18.7%	24.9%	28.1%	29.2%

TABLE 3.3 – Impact du paramètre *timeout* sur l'algorithme classique Paxos : gain en bande passante

Timeout - Load	RTT/4	RTT/2	RTT	1.2RTT
Low	0.4%	0.3%	0.3%	0.2%
Medium	1.4%	1.3%	1.4%	1.3%
High	3.2%	3.8%	4.2%	4.1%

TABLE 3.4 – Impact du paramètre *timeout* sur l'algorithme classique Paxos : dégradation de la latence

Dans le tableau 3.3, le gain en bande passante suit globalement la même tendance pour toutes les charges. Initialement, entre $RTT/4$ et RTT, le gain augmente. Ensuite, il ralentit et se stabilise. Cette stabilisation s'explique par la durée moyenne d'une phase, qui est de un RTT. Ainsi, un timeout supérieur à RTT a une faible chance d'expirer, l'envoi de messages est principalement déclenché par le mécanisme de *pledge*, correspondant au début de la phase suivante de l'algorithme.

Dans le tableau 3.4, on remarque une légère dégradation de la latence, surtout lorsque la charge est faible car moins de messages peuvent être agrégés (jusqu'à 0,4%). En revanche, à des charges plus élevées, où il est possible de regrouper davantage de messages, la dégradation de la latence est plus prononcée (jusqu'à 4,2%).

3.4.4 Compromis entre le gain en bande passante et la dégradation de la latence

Dans cette section, nous étudions le compromis entre le gain en bande passante et la dégradation de la latence pour différents paramétrages de *probaBuf* et *timeout*. La Figure 3.5 présente les résultats pour une charge élevée. Les autres charges seront discutées ultérieurement. Pour une simulation donnée, la consommation totale de bande passante en valeur absolue pour Paxos, Fast Paxos, Fast Byzantine Paxos et ZAB sans mécanisme d'agrégation s'élève respectivement à 123,525 Mo, 105,12 Mo, 302,22 Mo et 22,04 Mo.

Sur l'axe des abscisses et l'axe des ordonnées se trouvent respectivement le gain en bande passante et la dégradation de la latence. La forme des points indique la valeur du paramètre *timeout*, tandis que la couleur représente la valeur de *probaBuf*. La couleur noire est réservée au mécanisme d'agrégation agnostique, qui sert de point de comparaison.

Tout d'abord, notons qu'il n'y a pas de corrélation linéaire entre les deux métriques. Nous observons une corrélation positive, avec des points très proches d'un front de Pareto. Nous remarquons également qu'il n'y a pas de valeurs aberrantes, ce qui

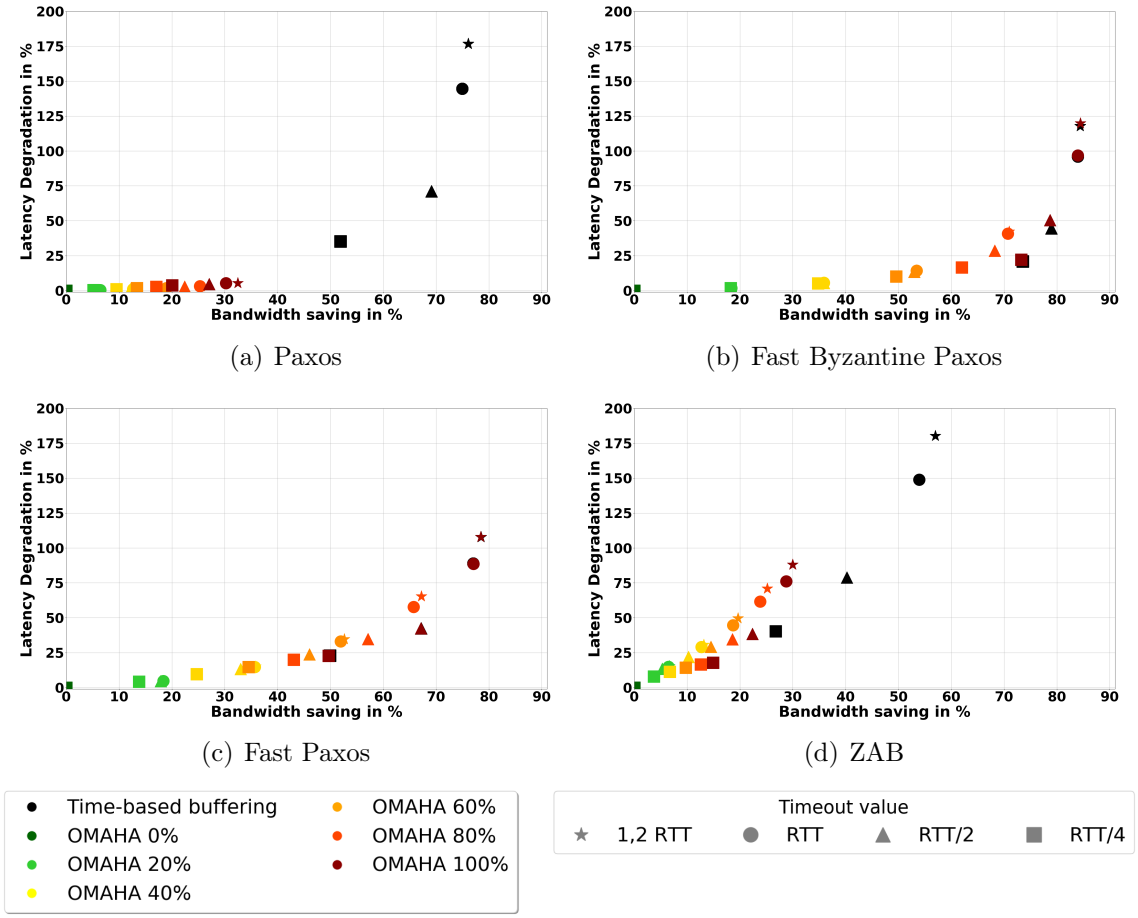


FIGURE 3.5 – Pareto à charge forte

s'explique par les faibles écarts-types. En moyenne, une instance de Paxos prend 120 ms pour converger, avec un écart type maximal de 3 ms. Cela souligne la stabilité des mécanismes d'agrégation malgré la gigue injectée dans l'expérience.

Deuxièmement, quelle que soit la variante de Paxos, la dégradation de la latence reste faible tant que le gain en bande passante reste en dessous de 50%. Nous pourrions supposer de prime abord, que ce phénomène est dû à la taille des quorums. En effet, si 50 % des réponses sont nécessaires pour faire progresser l'algorithme, il devient possible d'agréger l'autre moitié des messages sans affecter la latence. En suivant ce

raisonnement, une dégradation devrait apparaître plus tôt pour Fast Paxos et Fast Byzantine Paxos dont la taille des quorums est plus grande (voir Tableau 2.1).

Les quorums du Fast Byzantine Paxos (FBP) sont plus grands que ceux des autres algorithmes. De plus, FBP comporte un plus grand nombre de phases, entraînant ainsi plusieurs de quorums. Cela permet de réaliser davantage de *pledges* et donc une économie de bande passante plus importante. Ainsi, bien que la taille du quorum influence le mécanisme, l'efficacité d'OMAHA dépend surtout de la présence et de la réalisation des *pledges*. Les performances d'OMAHA restent, malgré tout, significatives pour l'algorithme de Paxos.

Il important de noter que chaque type de message peut être retardé, la *probaBuf* étant la même pour tous les messages. Cela signifie que les messages essentiels à la progression de l'algorithme seront potentiellement retardés, entraînant ainsi une dégradation de la latence.

L'impact du pattern de communication de l'algorithme est visible en comparant les Figures 3.5(a) et 3.5(b), où l'on constate que l'efficacité d'OMAHA est plus marquée avec FBP qu'avec Paxos. Cette différence peut s'expliquer par le caractère centralisé de l'algorithme de Paxos : la plupart des phases sont dirigées par le leader, qui concentre les *pledges*. En revanche, dans FBP, tous les nœuds peuvent effectuer des *pledges*. Le pouvoir d'agrégation est donc mieux réparti entre les nœuds.

ZAB est un algorithme centralisé, mais il utilise très peu de messages. À la différence de Paxos, le leader propose une transaction à un quorum de nœuds seulement. Après avoir reçu un accusé de réception de tous les nœuds du quorum initial, le leader envoie un message de confirmation à chaque nœud. Tout retard dans l'envoi d'un message à un nœud du quorum entraînerait une dégradation significative de la latence, comme le montre la Figure 3.5(d). Dans la Section 3.5, nous montrerons qu'il est possible d'améliorer les performances de ZAB.

Pour conclure, nous avons montré qu'OMAHA parvient à économiser de manière

significative la bande passante tout en maintenant une dégradation de la latence acceptable. Son efficacité repose sur le modèle des phases de l'algorithme et un bon paramétrage.

3.4.5 Impact de la perte de message

OMAHA agrège plusieurs messages applicatifs en un seul message réseau, ce qui peut susciter des interrogations sur l'impact de la perte de messages. Par conséquent, nous avons examiné le comportement d'OMAHA avec l'algorithme de Paxos dans différents scénarios de perte de messages.

À chaque envoi de message de type *prepare* ou *accept*, Paxos arme un timeout pour détecter d'éventuelles pertes de message. Lorsque ce délai expire, les messages sont renvoyés. Pour éviter les fausses détections dues à la mise en tampon, nous réglons le timeout de Paxos à une valeur supérieure au retard maximal pouvant être induit par OMAHA : $RTT + 2 * timeout$. Pour rappel, le *timeout* correspond au paramètre de la primitive *send* (cf Algorithm 1). Le *timeout* de Paxos est défini en tenant compte du pire cas. Il est basé sur le délai moyen d'une phase, soit un RTT, avec la possibilité que les deux messages de cette phase soient agrégés par OMAHA, ce qui peut entraîner un retard supplémentaire équivalent à un *timeout* à chaque fois.

Dans la Figure 3.6, nous présentons les fronts de Pareto obtenus en considérant différents taux de perte de messages. Nous remarquons que la perte de messages a un impact limité sur les performances d'OMAHA.

Pour un taux de perte de 1%, les performances d'OMAHA ne sont pas dégradées. À ce taux, la perte d'un message reste un phénomène ponctuel. De plus, étant donné que le message perdu est un agrégat de messages issus de différentes applications, son impact sur celles-ci est minime. En effet, chaque application concernée ne subit la perte que d'un seul message. Au-delà, nous observons une dégradation de la latence proportionnelle à l'économie de bande passante. En effet, un gain en bande passante élevé est dû à un nombre important de messages agrégés. La perte d'un message

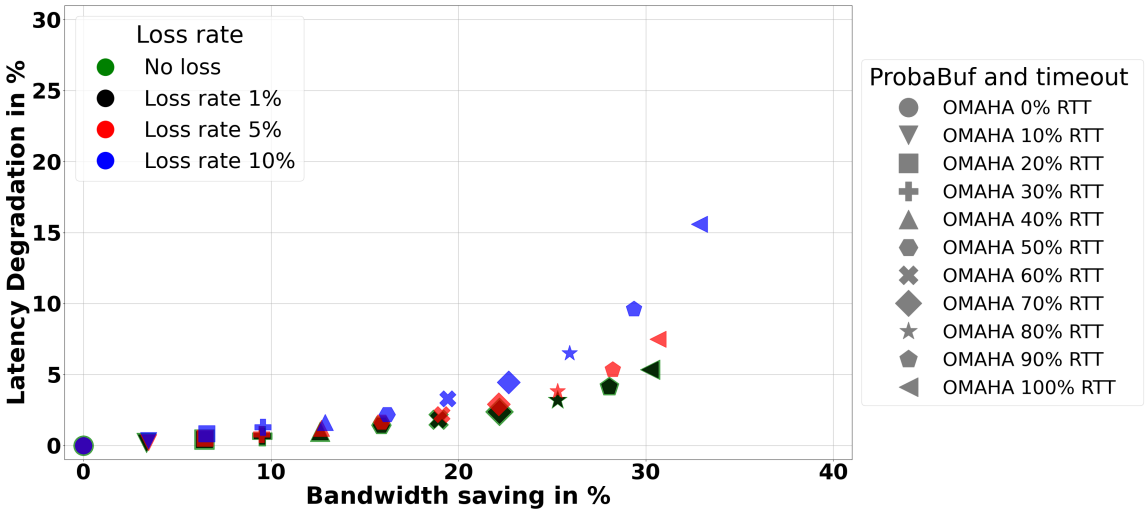


FIGURE 3.6 – Impact de la perte de message sur Paxos

réseau étant plus probable, cela peut finir par ralentir plusieurs applications.

3.4.6 Expérience sur la plateforme Grid'5000

Cette section présente les résultats des expériences menées sur la plateforme g5k (voir section 3.4.1 pour les paramètres de la plateforme).

Pour cette expérience, nous avons exécuté simultanément un nombre égal d'instances de Paxos, ZAB, Fast Paxos et Fast Byzantine Paxos. À chaque démarrage d'instance, le protocole est choisi de manière aléatoire selon une distribution uniforme.

La Figure 3.7 donne le front de Pareto de chaque protocole s'exécutant simultanément sur la même infrastructure.

Dans la Figure 3.5(d), nous avons observé qu'OMAHA n'était pas efficace pour ZAB. Comme évoqué dans la section 3.4.4, ZAB génère peu de *pledges*. Cependant, cette nouvelle expérience montre que ZAB est capable d'exploiter les périodes de *pledge* des autres protocoles pour réduire considérablement son coût en bande passante.

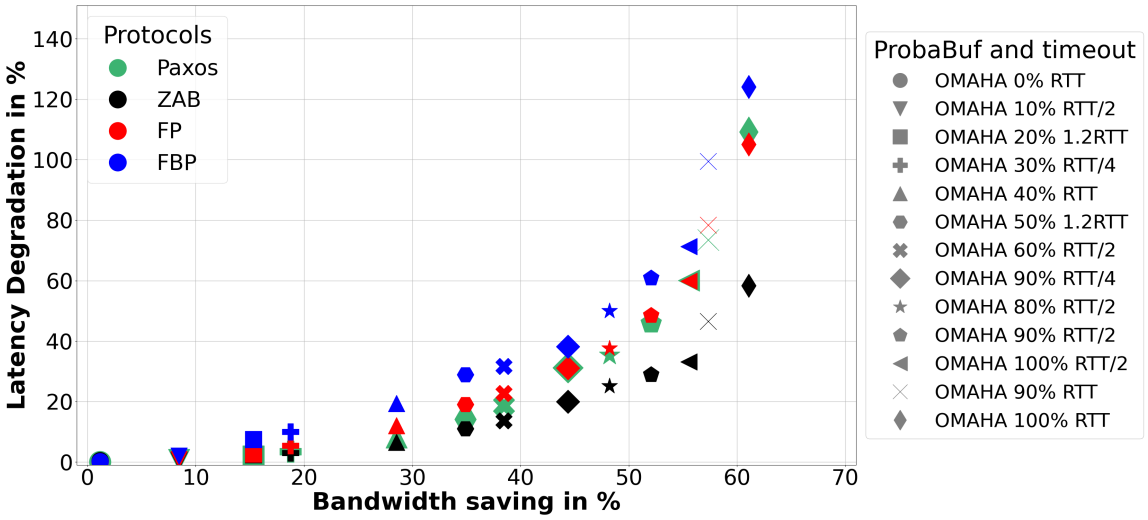


FIGURE 3.7 – Pareto obtenu en mixant différents protocoles concurrents sur la plateforme g5k

Parmi tous les protocoles, ZAB réalise le meilleur compromis entre économie de bande passante et dégradation de la latence.

Si nous comparons le comportement de ZAB dans les Figures 3.7 et 3.5(d), les résultats sont très différents. Pour la même configuration (probaBuf de 40%, un délai d'attente d'un RTT, triangle noir dans la Figure 3.7 et cercle jaune dans la Figure 3.5(d)), nous observons que l'économie de bande passante passe de 12.7% à 28.5%, tandis que la dégradation de la latence est réduite de 29.1% à 6.5%. Ce dernier résultat est encourageant et montre que OMAHA pourrait bénéficier à toutes les applications utilisant des algorithmes à phases en cours d'exécution dans le système.

3.4.7 Configuration d'OMAHA

Le paramétrage d'OMAHA est au cœur de son efficacité. Dans cette section, nous présentons une étude visant à identifier la meilleure configuration pour économiser de la bande passante tout en respectant un taux de dégradation de latence acceptable

pour l'utilisateur.

Nous avons considéré les quatre algorithmes précédemment mentionnés (Paxos, Fast Paxos, Fast Byzantine Paxos et ZAB) et évalué via un environnement simulé chaque mécanisme (OMAHA et approche agnostique) selon trois scénarios de charge :

- Faible : Un démarrage d'instance toutes les 100 ms.
- Moyen : Un démarrage d'instance toutes les 50 ms.
- Élevé : Un démarrage d'instance toutes les 20 ms.

Nous avons synthétisé les résultats dans le Tableau 3.5. Pour des taux de dégradation de la latence de 5%, 10%, 20% et 50%, nous indiquons les gains en bande passante réalisés par OMAHA et l'approche agnostique qui agrège systématiquement les messages. Les valeurs présentées correspondent au gain de bande passante le plus élevé tout en maintenant la dégradation de la latence égale ou inférieure aux différents taux considérés. Le taux maximal de dégradation n'est donc pas systématiquement atteint. Les cases étiquetées NR (Not Reachable) indiquent qu'il n'existe aucune configuration permettant d'économiser de la bande passante pour une telle dégradation de la latence. Les résultats sont marqués en vert lorsqu'OMAHA est la meilleure option, et en orange lorsque l'approche agnostique offre de meilleurs résultats.

Nous pouvons observer initialement qu'indépendamment de l'objectif, de l'algorithme ou de la charge (à l'exception d'un cas avec l'algorithme ZAB), OMAHA offre une configuration permettant d'économiser de la bande passante. Pour toutes les charges considérées et un taux maximal de dégradation de la latence de :

- 5% : les économies de bande passante vont de 1% à 29,2% pour une dégradation de la latence allant de 0,3% à 4,8%.
- 10% : les économies de bande passante vont de 1,6% à 44,7% pour une dégradation de la latence allant de 0,3% à 9,9%.
- 20% : les économies de bande passante vont de 1,6% à 62% pour une dégradation de la latence allant de 0,3% à 20,0%.

- 50% : les économies de bande passante vont de 1,6% à 74,5% pour une dégradation de la latence allant de 0,3% à 49,2%.

Pour des taux de dégradation de 5%, 10% et 20%, OMAHA l'emporte systématiquement. L'approche agnostique ne parvient jamais à proposer une configuration avec un taux de perte inférieur ou égal à ces valeurs.

Lorsque la charge est modérée, même avec une dégradation autorisée de la latence de 5%, des gains significatifs de bande passante sont observés. À 10% et 20%, ils deviennent importants, atteignant respectivement jusqu'à 27,2% et 45,8%. À un taux de dégradation élevé de 50%, OMAHA ne fournit aucun avantage par rapport à la stratégie agnostique qui, tout en compromettant significativement la latence, réalise des économies de bande passante plus importantes.

Nous avons vérifié la qualité de notre simulation en comparant nos prédictions avec des mesures réelles pour le Fast Byzantine Paxos, l'algorithme le plus efficace en termes de bande passante. Nous avons reproduit les mêmes expériences sur g5k. Pour chaque charge, nous utilisons la configuration d'OMAHA qui économise le plus de bande passante tout en limitant la dégradation de la latence à 10%. Les résultats sont synthétisés dans le Tableau 3.6.

	Prediction		Real measurements	
	Latency	Bandwidth	Latency	Bandwidth
Low	7,0%	9,7%	3,5%	10,1%
Medium	9,3%	27,3%	10,9%	27,1%
High	9,2%	44,7%	13,4%	43,4%

TABLE 3.6 – Simulation versus Grid5000

Nous constatons que les résultats obtenus correspondent aux prévisions faites hors ligne. Cependant, une différence de latence entre les prédictions et les mesures réelles à faible débit (7,0% vs 3,5%) est observée, attribuée à l'entropie. Pour réduire cet effet, il est nécessaire de répéter l'expérience un grand nombre de fois.

				5%	10%	20%	50%
PAXOS	Low	OMAHA	Latency Degradation	0,3%	0,3%	0,3%	0,3%
			Bandwidth Gain	5,7%	5,7%	5,7%	5,7%
		Timebased	Latency Degradation	NR	NR	NR	42,7%
			Bandwidth Gain	NR	NR	NR	8,2%
	Medium	OMAHA	Latency Degradation	1,5%	1,5%	1,5%	1,5%
			Bandwidth Gain	15,3%	15,3%	15,3%	15,3%
		Timebased	Latency Degradation	NR	NR	NR	38,6%
			Bandwidth Gain	NR	NR	NR	32,2%
	High	OMAHA	Latency Degradation	4,1%	5,3%	5,3%	5,3%
			Bandwidth Gain	29,2%	32,5%	32,5%	32,5%
		Timebased	Latency Degradation	NR	NR	NR	35,2%
			Bandwidth Gain	NR	NR	NR	52,0%
FAST PAXOS	Low	OMAHA	Latency Degradation	4,8%	6,1%	6,1%	43,4%
			Bandwidth Gain	1,4%	8,2%	8,2%	19,4%
		Timebased	Latency Degradation	NR	NR	NR	27,9%
			Bandwidth Gain	NR	NR	NR	4,1%
	Medium	OMAHA	Latency Degradation	3,0%	9,6%	19,3%	46,0%
			Bandwidth Gain	9,6%	18,2%	26,5%	53,3%
		Timebased	Latency Degradation	NR	NR	NR	46,0%
			Bandwidth Gain	NR	NR	NR	53,2%
	High	OMAHA	Latency Degradation	4,8%	9,0%	20,0%	42,5%
			Bandwidth Gain	18,4%	27,1%	43,1%	67,2%
		Timebased	Latency Degradation	NR	NR	NR	42,5%
			Bandwidth Gain	NR	NR	NR	67,2%
FAST BYZANTINE PAXOS	Low	OMAHA	Latency Degradation	4,2%	7,0%	16,8%	49,2%
			Bandwidth Gain	2,5%	9,7%	18,6%	41,6%
		Timebased	Latency Degradation	NR	NR	NR	27,3%
			Bandwidth Gain	NR	NR	NR	43,5%
	Medium	OMAHA	Latency Degradation	4,8%	9,4%	19,4%	47,3%
			Bandwidth Gain	16,0%	27,2%	45,8%	67,3%
		Timebased	Latency Degradation	NR	NR	NR	47,2%
			Bandwidth Gain	NR	NR	NR	73,2%
	High	OMAHA	Latency Degradation	3,3%	9,2%	16,7%	38,3%
			Bandwidth Gain	27,2%	44,7%	62,0%	74,5%
		Timebased	Latency Degradation	NR	NR	NR	44,7%
			Bandwidth Gain	NR	NR	NR	79,0%
ZAB	Low	OMAHA	Latency Degradation	3,2%	9,7%	9,7%	9,7%
			Bandwidth Gain	1%	1,6%	1,6%	1,6%
		Timebased	Latency Degradation	NR	NR	NR	43,2%
			Bandwidth Gain	NR	NR	NR	4,1%
	Medium	OMAHA	Latency Degradation	NR	9,6%	16,7%	36,9%
			Bandwidth Gain	NR	2,9%	6,8%	14,0%
		Timebased	Latency Degradation	NR	NR	NR	42,1%
			Bandwidth Gain	NR	NR	NR	12,7%
	High	OMAHA	Latency Degradation	4,7%	9,9%	17,9%	38,5%
			Bandwidth Gain	2,9%	5,3%	15,0%	22,4%
		Timebased	Latency Degradation	NR	NR	NR	40,3%
			Bandwidth Gain	NR	NR	NR	26,8%

TABLE 3.5 – Tableau récapitulatif des expérimentations

3.5 Analyse théorique du paramétrage

Pour utiliser notre mécanisme d'agrégation, l'utilisateur doit définir les paramètres de la primitive *send*. Choisir la bonne combinaison est crucial pour les performances d'OMAHA.

Dans un premier temps, nous proposons une analyse théorique visant à déterminer une valeur optimale du paramètre *probaBuf*. Notre objectif est de maximiser l'économie de bande passante tout en minimisant la dégradation de la latence. Pour cela, nous considérons que les messages ont tous la même importance (donc la même valeur de *probaBuf*) et peuvent être mis en mémoire tampon. Dans un second temps, nous étudions comment définir le paramètre *probaBuf* dans un cas concret où les messages ont une importance différente selon leur sémantique.

3.5.1 Probabilités uniformes

Le choix de la valeur de *probaBuf* est essentielle pour éviter de perturber l'exécution de l'algorithme. En effet, pour choisir la valeur de *probaBuf* d'un message, il est crucial de prendre en considération la probabilité des types de messages qui le précèdent dans le protocole.

La Figure 3.8 illustre un scénario impliquant quatre nœuds. Le nœud 2 envoie un message, m1, à tous les nœuds, avec une probabilité de 50% qu'il soit retardé. Après avoir reçu m1, les nœuds répondent via le message m2, qui a également une probabilité de 50% d'être retardé. Dans ce cas, la probabilité que le nœud 2 reçoive une réponse est de 25%, et non de 50%. La probabilité de recevoir le message m2 est calculée selon la formule suivante 3.1 :

$$P(receive\ m2) = P(send\ m1) \times P(send\ m2|\ receive\ m1) = 0,5 \times 0,5 = 0,25 \quad (3.1)$$

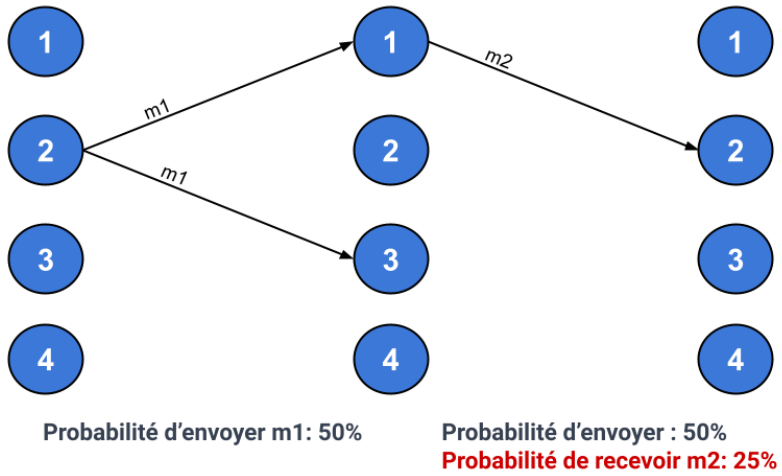


FIGURE 3.8 – Probabilités conditionnelles exemple

Cette probabilité étant inférieure à la valeur du quorum, le nœud ne recevra pas une majorité de réponses. Il est donc nécessaire de s'assurer qu'un quorum de messages soit toujours recevable. Pour cela, nous considérons le pire scénario où chaque message peut être mis en mémoire tampon. L'envoi des messages dépend alors de la probabilité qui leur est associée. Ainsi, pour passer d'une phase à l'autre, il est nécessaire de recevoir un quorum de messages.

De manière plus formelle, considérons un algorithme qui nécessite un quorum q. Le quorum indique le seuil minimum de réponses attendues par un nœud c pour prendre une décision. Nous voulons calculer la probabilité pour c de recevoir q messages. Soit $m_1, .., m_n$ l'ensemble des messages envoyés à c, chaque message étant associé à une probabilité d'être mis en mémoire tampon, notée $p_1, ..., p_n$. La probabilité que c reçoive les messages est exprimée par la formule 3.2 :

$$P(c\ receive\ \ messages) = \prod_{i=1}^{n} p_i \tag{3.2}$$

Pour s'assurer qu'un quorum de messages est reçu, cette probabilité doit être supérieure

au quorum. Nous avons alors l'équation suivante :

$$P(c\ receive\ \ messages) = q/n \Leftrightarrow \prod_{i=1}^{n} p_i = q/n \tag{3.3}$$

Maintenant, en supposant que les p_i sont uniformes avec p_1=p_2=...=p_n=p, nous pouvons calculer les valeurs de p, ce qui nous donne :

$$p^n = q/n \Leftrightarrow p = \sqrt[n]{q/n} \tag{3.4}$$

Cette valeur est théorique et s'applique uniquement si tous les p_i sont égaux, sans considérer l'importance des messages. Cependant, elle garantit la réception d'un quorum de messages pour chaque message, évitant ainsi de bloquer l'exécution de l'algorithme.

3.5.2 Probabilités non uniformes

Dans le calcul théorique, nous avons supposé que tous les messages étaient susceptibles d'être mis en mémoire tampon et que les probabilités étaient uniformes. Cependant, en pratique, ce scénario se produit rarement. Par conséquent, pour configurer OMAHA, nous avons décidé de choisir la valeur de *probaBuf* en fonction de l'importance des messages.

Certains messages sont essentiels à la progression de l'algorithme, et retarder leur envoi peut entraîner une dégradation significative de la latence. Dans le protocole Paxos, c'est le cas des messages *prepare* et *accept* envoyés par le leader. En revanche, certains messages peuvent être retardés sans affecter l'algorithme. C'est notamment le cas des messages permettant d'atteindre un quorum de réponses. Dans le protocole Paxos, c'est le cas des messages réponses *ack* et *accepted*. Tout message supplémentaire reçu après avoir réuni un quorum de réponses est inutile et peut être retardé sans

compromettre la vivacité de l'algorithme. Ainsi, prendre en compte l'importance des messages permet d'économiser de la bande passante tout en diminuant l'impact sur la latence.

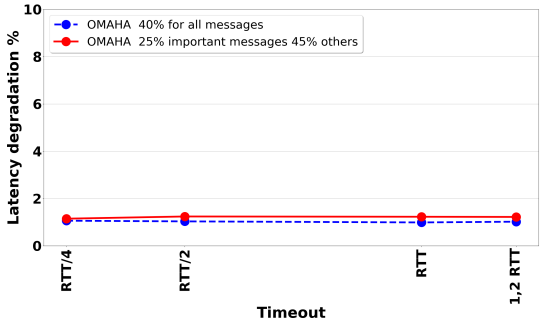
Les messages de réponses doivent avoir une probabilité de mise en tampon proche de $1 - \frac{taille(quorum)}{\#noeuds}$ pour garantir une réception suffisante des messages. Ainsi, pour un grand quorum, supposons 2/3 des nœuds, les messages auront une faible probabilité d'être retardés, de 33%.

Dans la Figure 3.9, nous avons ajusté la valeur de *probaBuf* pour les algorithmes Paxos, FBP et ZAB en considérant les 3 modèles de charge.

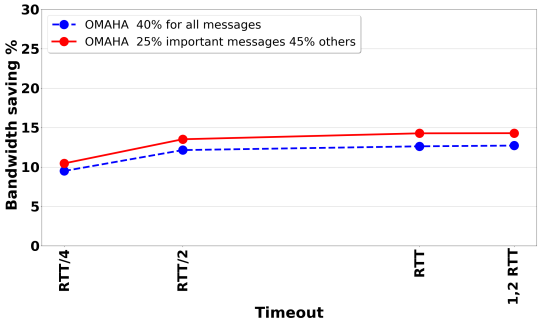
Dans Paxos (Figures 3.9(a), 3.9(b)), nous avons opté pour une probabilité de 45% pour les réponses. Cette valeur est légèrement inférieure à la taille du quorum (50% des nœuds). Pour les messages critiques, nous fixons une probabilité de 25%. Nous observons qu'adapter la probabilité de mise en mémoire tampon n'a peu d'impact significatif sur les performances. Néanmoins, on observe une légère amélioration de la bande passante avec une dégradation de latence presque équivalente.

Fast Byzantine Paxos nécessitant des quorums plus larges (60% et 80% selon les phases), nous choisissons une probabilité de 35% pour les réponses et de 15% pour les messages critiques. Ce réglage entraîne une baisse significative de la latence et un gain en bande passante, comme le montrent les Figures 3.9(c) et 3.9(d).

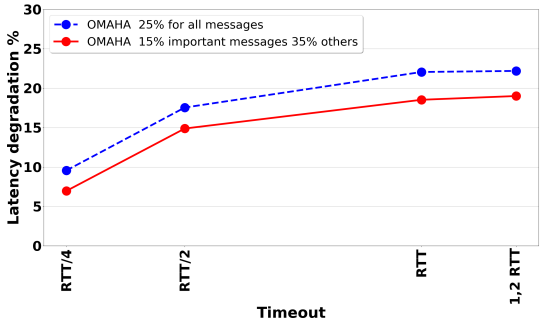
Concernant ZAB, où le paramétrage est crucial (cf explications Figure 3.5(d)), nous décidons de ne pas mettre en mémoire tampon les messages critiques. Les autres messages le sont avec une probabilité de 40%. Cette configuration conduit à une réduction importante de la dégradation de la latence, mais l'économie de bande passante est faible (Figures 3.9(e) et 3.9(f)). Toutefois, pour un même gain de bande passante s'élevant à 5,9%, les taux de dégradation de la latence varient considérablement. Ce gain correspond à un timeout de $RTT/2$ sur la courbe bleue et à $1.2RTT$ sur la courbe rouge. La dégradation de la latence associée est respectivement de 24,0% et 7,2%. Ainsi, pour le même gain de bande passante, nous observons une différence de



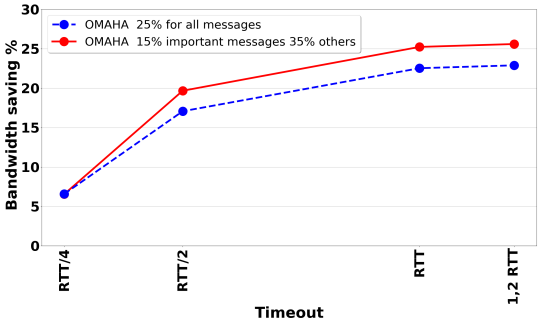
(a) Paxos débit élevé dégradation de la latence



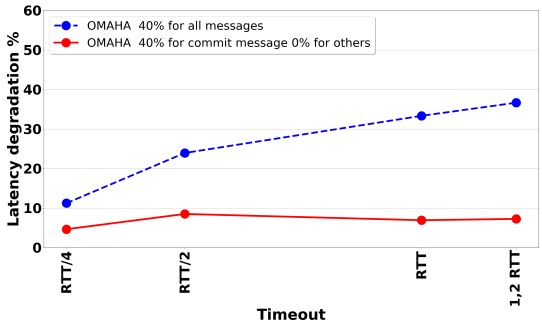
(b) Paxos débit élevé gain en bande passante



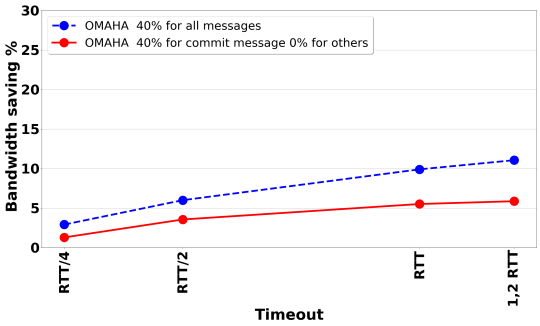
(c) FBP débit faible dégradation de la latence



(d) FBP débit faible gain en bande passante



(e) ZAB débit moyen dégradation de la latence



(f) ZAB débit moyen gain en bande passante

FIGURE 3.9 – Adaptation de la probabilité de mise en tampon

16,8 points. Nous avons donc une nette amélioration de la latence.

3.6 Conclusion

Ce chapitre a introduit OMAHA, un mécanisme d'agrégation de messages opportuniste permettant de trouver un compromis entre l'économie de bande passante et la dégradation de la latence. Il vise à des applications s'exécutant simultanément sur une même infrastructure physique et utilisant des algorithmes à phases.

Nous avons comparé les performances d'OMAHA avec une approche agnostique basée sur le temps, qui met en mémoire tampon tous les messages et les envoie périodiquement. Nos résultats montrent qu'il est possible d'économiser jusqu'à 30% de bande passante tout en limitant la dégradation de la latence à seulement 5% pour le protocole Paxos.

OMAHA offre une API permettant la mise en place d'une couche intermédiaire entre le réseau et les applications. Cette API a été conçue pour être peu intrusive et limiter les modifications des algorithmes considérés. Nous avons appliqué ce mécanisme d'agrégation à quatre algorithmes à phases largement utilisés : trois variantes de Paxos et l'algorithme de diffusion atomique de Zookeeper. En anticipant les futures communications, OMAHA réduit le nombre de messages échangés tout en limitant la dégradation de la latence. Son efficacité dépend des spécificités de chaque algorithme telles que le nombre de phases, la taille des quorums et le type des messages, qui doivent être clairement définies pour ajuster correctement les paramètres de l'API.

Chapitre

4

Consensus f-Révocable

Sommaire

4.1 Introduction

Dans les systèmes distribués, parvenir à un accord sur une action ou une valeur peut être particulièrement difficile. Au-delà du consensus traditionnel, la résolution de contraintes représente un défi croissant dans des environnements de plus en plus distribués. Par exemple, les systèmes multi-agents peuvent imposer des contraintes spécifiques à chaque agent, ce qui rend la gestion du consensus plus complexe. Chaque agent peut être soumis à des contraintes spécifiques à sa mission, sa localisation ou sa capacité de calcul [22, 97]. Très répandus, nombre de systèmes multi-agents nécessitent de gérer des contraintes : coordinations de véhicules autonomes [23, 24], gestion de réseaux de capteurs [98, 99] ou encore gestion d'agenda [21]. Il est donc crucial de prendre en compte ces contraintes pour s'assurer que les décisions respectent les besoins individuels tout en atteignant un objectif commun. De même, dans les environnements cloud, la gestion des ressources telles que la bande passante, la mémoire ou le processeur exige la résolution de systèmes de contraintes [25]. Ce problème se manifeste notamment dans les Software Defined Networks (SDN) pour le placement de fonctions réseau virtuelles. L'allocation des ressources dans les SDN doit respecter les contraintes propres à chaque fonction, tout en prenant en considération que chaque allocation peut introduire de nouvelles contraintes pour les suivantes [26, 27].

Le consensus, dans sa définition classique, ne prend pas en compte les contraintes des processus. En effet, il permet à un ensemble de processus de s'accorder sur une valeur proposée par l'un d'eux, sans imposer de conditions sur le choix de cette valeur. Si les valeurs proposées représentent des contraintes, cela implique que seules les contraintes d'un processus sont prises en compte, et non l'ensemble des contraintes. Une solution triviale consisterait à échanger les contraintes entre les processus avant d'atteindre un consensus. Cependant, dans un système asynchrone, il est impossible de collecter toutes les contraintes, car un processus ne peut pas toujours savoir si un autre est en panne ou simplement lent.

Pour résoudre ce problème, les algorithmes de consensus sont conçus pour tolérer un nombre maximal de pannes et ajustent leurs communications en conséquence. Ainsi, la décision est généralement déterminée par la majorité des processus, ceux considérés comme non fautifs. Cela conduit toutefois à ignorer les valeurs proposées par la minorité. Une alternative serait de relancer une instance de consensus si les contraintes d'un processus sont violées. Cependant, une telle approche ne garantit pas toujours la convergence. En effet, pour que les contraintes des processus minoritaires soient prises en compte, ces derniers doivent être suffisamment rapides pour rejoindre la majorité de la nouvelle instance. Sinon, l'algorithme pourrait continuer à progresser en ignorant de nouveau les processus de la minorité.

Il est donc important non seulement de prendre en compte les contraintes des processus, mais aussi de permettre la réévaluation d'une décision incorrecte. Cela conduit à repenser le modèle du consensus pour autoriser la révocabilité des décisions tout en prenant en compte les contraintes de chaque processus. Pour répondre à cette problématique, nous avons introduit le Consensus f-Révocable. Ce nouveau type de consensus permet non seulement de prendre une décision respectant les contraintes des processus, mais aussi de révoquer une décision qui violerait les contraintes d'un processus appartenant à la minorité.

Dans ce chapitre, nous introduisons formellement le concept de Consensus f-Révocable. De manière incrémentale, nous présenterons à travers l'algorithme de Paxos plusieurs

solutions permettant de répondre au Consensus f-Révocable. Nous détaillons ensuite le Paxos f-Révocable, une version améliorée des solutions présentées précédemment permettant de révoquer une instance en cours d'exécution. Par la suite, nous présentons des esquisses de preuves donnant l'intuition de la validité de notre algorithme. Enfin, nous présentons nos résultats expérimentaux à travers une étude des révocations et un cas d'étude concret.

4.2 Définition du Consensus f-Révocable

Dans cette section, nous présentons une définition révisée du consensus, intégrant la notion de contrainte ainsi que la possibilité de révoquer les décisions. Contrairement au consensus traditionnel, qui ne permet de choisir qu'une seule valeur parmi celles proposées par les processus, le Consensus f-Révocable (CR) permet à ces derniers de définir des contraintes sous forme d'ensembles de valeurs acceptables. Ainsi, chaque processus peut soumettre plusieurs valeurs, plutôt que de se limiter à une seule option.

Soit $\Pi = \{p_1, p_2, \ldots, p_n\}$ l'ensemble des processus participant au consensus, et $E = \{E_1, E_2, \ldots, E_n\}$ l'ensemble des contraintes, où E_i représente l'ensemble des valeurs que le processus p_i peut accepter. Le paramètre f du Consensus f-Révocable correspond au nombre maximal de processus pouvant être fautifs.

Le Consensus f-Révocable est défini par les propriétés suivantes :

- CR-Cohérence : Deux processus corrects ne peuvent, à terme, décider de valeurs différentes.
- CR-Validité : Si un processus correct p_i décide v, alors deux cas sont possibles :
  - $v \in E_i$ ou
  - v est égal à $\perp$ si l'intersection des ensembles reçus par au moins un processus correct est vide.
- CR-Terminaison : Chaque processus correct doit, à terme, décider.

- CR-Intégrité : Chaque processus peut décider au maximum f fois. Cependant, après avoir décidé $\perp$, un processus ne peut plus décider une autre valeur.

4.3 Modèle

Chaque nœud suit la spécification de l'algorithme, jusqu'à ce qu'il rencontre une éventuelle faute. Les nœuds peuvent subir des pannes franches. Un nœud est considéré comme correct s'il ne tombe jamais en panne. Dans le cas contraire, il est considéré comme fautif, jusqu'à ce qu'il revienne dans le système.

Les nœuds communiquent via des canaux de communications équitables (voir 2.2). Ainsi, des messages peuvent être perdus. Nous considérons un graphe de communications complet où chaque nœud est en mesure de contacter directement un autre nœud.

4.4 Application à Paxos

Cette section aborde différentes approches envisagées pour résoudre le Consensus f-Révocable en s'appuyant sur l'algorithme de Paxos.

Dans Paxos original, après avoir obtenu une majorité de `Ack`, le leader sélectionne une des valeurs proposées. Cependant, ce choix peut violer certaines contraintes au sein même de la majorité, entraînant potentiellement plus de f révocations. Par exemple, si deux nœuds de la majorité possèdent respectivement les contraintes *"nombre pair entre 1 et 10"* et *"nombre pair entre 4 et 10"*, proposer la valeur 2 violerait les contraintes du second nœud.

Pour remédier à cela, les nœuds incluent désormais l'ensemble de valeurs répondant à leurs contraintes dans le message de type `Ack` à la place de la valeur. Le consensus permet alors d'obtenir l'intersection des ensembles des nœuds. Une fois cette intersection

déterminée, une valeur est choisie arbitrairement, par exemple la plus petite. Grâce à cette approche, le leader peut également identifier d'éventuelles incompatibilités dans les contraintes des nœuds, facilitant ainsi la gestion des conflits.

Cependant, cette approche n'est pas suffisante. S'il est possible de satisfaire toutes les contraintes de la majorité, les contraintes de la minorité sont quant à elles ignorées.

4.4.1 Première approche : attente bornée

Une première approche consisterait à attendre pendant une durée fixe les potentielles réponses de la minorité. Cette stratégie pourrait permettre de prévenir une décision incompatible avec les contraintes de certains nœuds de la minorité. Cependant, elle présente un inconvénient majeur : une dégradation significative de la latence. En effet, l'algorithme doit s'ajuster au rythme du processus le plus lent. Étant donné que le pire cas est toujours pris en compte, ce retard peut se révéler inutile lorsqu'une décision acceptée par la majorité satisfait déjà tous les nœuds. De plus, le temps d'attente étant borné, il est possible de ne pas recevoir les contraintes d'un nœud lent.

Dans la Figure 4.1, nous considérons cinq nœuds, avec le nœud 2 comme leader. Tous les nœuds partagent les mêmes contraintes : l'ensemble des *nombres pairs*. La durée d'attente est fixée en fonction du nœud le plus lent (ici, le nœud 5). Ainsi, le leader prolonge intentionnellement son attente pour tenter de recueillir tous les ensembles. Cependant, ce retard s'avère superflu, car aucune des contraintes de la minorité ne contredit la décision que la majorité aurait prise.

4.4.2 Deuxième approche : séquence de consensus

Une autre approche consiste à lancer une nouvelle instance de Paxos chaque fois qu'un nœud détecte une décision incompatible avec ses contraintes. Cependant, relancer le consensus tel quel ne garantit pas nécessairement qu'une valeur satisfaisant tous les

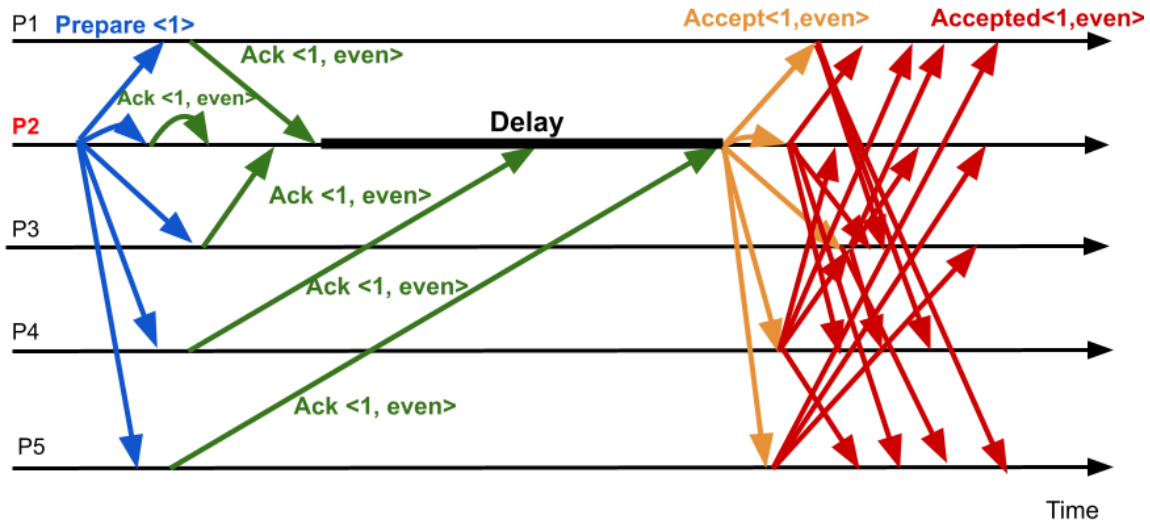


FIGURE 4.1 – Attente bornée

nœuds corrects soit décidée. Cette méthode pourrait même entraîner davantage de révocations (plus de f) et violer la propriété de terminaison.

En effet, la valeur à décider dépend des contraintes reçues après réception d'une majorité d'`Acks`. Ainsi, si le nœud ayant détecté l'incompatibilité n'est pas assez rapide pour faire partie de la majorité dans la nouvelle instance, ses contraintes ne seront de nouveau pas prises en compte.

Pour pallier ce problème, il est nécessaire de redémarrer le consensus en tenant compte à la fois de l'intersection des ensembles précédemment décidée par la majorité et de l'ensemble ayant provoqué la révocation. Nous introduisons ainsi un nouveau type de message le `Abort`.

Dans la Figure 4.2, les nœuds 1 à 4 partagent les mêmes contraintes : la valeur décidée doit être un nombre pair. Le nœud 5, quant à lui, doit choisir une valeur multiple de 4. Le leader, ici le nœud 2, initie une instance de Paxos en envoyant un message `Prepare`. À la réception de ce message, les nœuds répondent par des Acks,

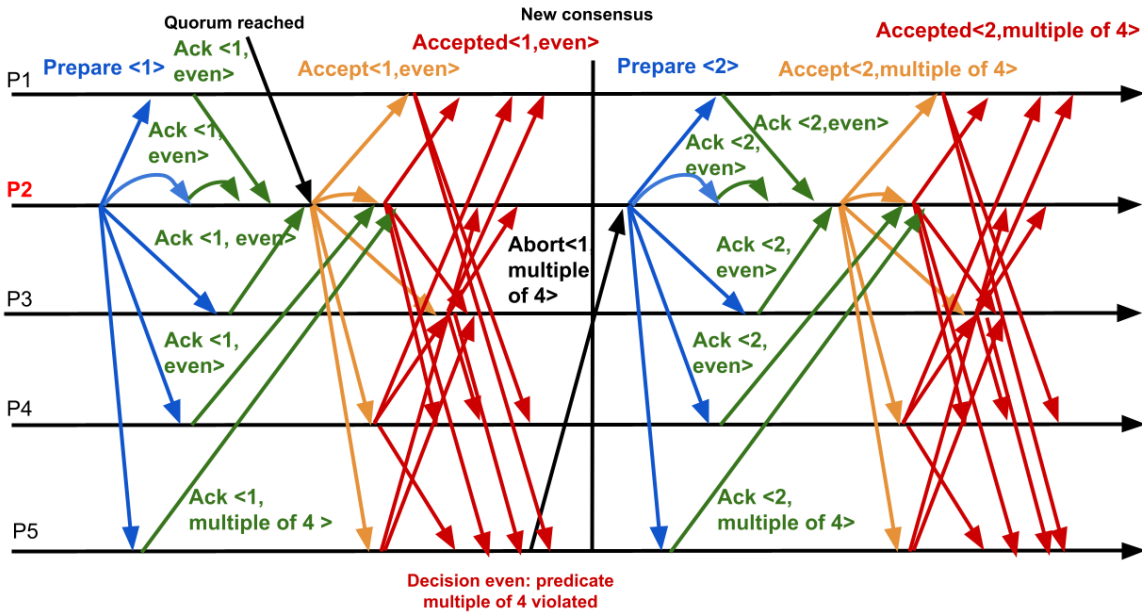


FIGURE 4.2 – Enchaînement de consensus

incluant leurs contraintes respectives. Une fois qu'une majorité d'Acks est reçue, le leader calcule l'intersection des ensembles. Ainsi, après avoir recueilli les réponses des nœuds 1, 3 et de lui-même, l'ensemble de valeurs retenu est celui des nombres pairs. Le leader envoie ensuite un message `Accept` (en orange), incluant l'intersection des ensembles reçus, qui est ensuite relayé par des messages `Accepted` (en rouge). Lorsqu'une majorité de messages d'acceptation est reçue, les nœuds doivent décider. À ce moment-là, les nœuds vérifient la compatibilité entre l'ensemble retenu et le leur. Autrement dit, ils vérifient l'inclusion de l'ensemble reçu dans leur propre ensemble. Dans notre exemple, le nœud 5 détecte que l'ensemble de valeurs qui a été accepté *"nombre pair"* n'est pas inclus dans son ensemble *"nombre multiple de 4"*. En cas d'incohérence, le nœud concerné envoie un message `Abort` au leader. Ce dernier initie alors un nouveau consensus en incrémentant le numéro de ballot pour ignorer les messages obsolètes, tout en prenant en compte à la fois de l'ensemble de valeurs précédemment accepté et celui ayant déclenché le message `Abort`.

Notons qu'il est possible d'associer une date de validité à une demande de révocation.

4.4.3 Dernière approche : révocation anticipée

L'approche précédente est fonctionnelle, mais peut être améliorée. En effet, nous pourrions tenter de stopper le consensus dès qu'un nœud détecte un conflit avec ses contraintes sans attendre la décision. Dans certains cas, cela pourrait même empêcher la prise d'une décision erronée. Dans le cas contraire, nous pouvons au moins omettre la phase de préparation lors d'un nouveau consensus. Les résultats dépendront de la rapidité des nœuds impliqués dans la révocation.

Détection d'incompatibilités avec le leader

Pour détecter les révocations le plus tôt possible, nous avons introduit un mécanisme permettant d'identifier proactivement les conflits entre les contraintes des nœuds et ceux du leader.

Dans Paxos original, le leader envoie un message `Prepare` contenant un numéro de ballot. Dans notre version, le leader inclut également son ensemble de valeurs, permettant ainsi aux nœuds de vérifier la compatibilité avec leur propre ensemble. En cas de détection d'un conflit, le consensus peut être interrompu rapidement et corrigé.

Dans la Figure 4.3, lorsque le leader (nœud 2) envoie un message `Prepare` indiquant la sélection d'une valeur paire, le nœud 4, qui doit choisir une valeur impaire, détecte l'incompatibilité. Il interrompt immédiatement l'instance en cours en envoyant un message `Abort` à tous les nœuds. Constatant l'impossibilité de satisfaire les contraintes des processus, les nœuds décident finalement d'opter pour la valeur spéciale $\perp$.

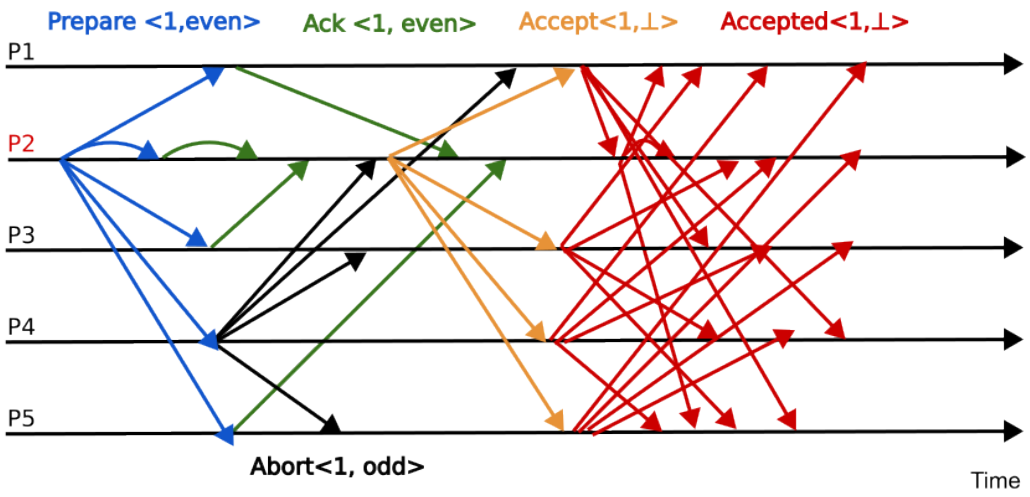


FIGURE 4.3 – Incompatibilité entre le leader et l'un des nœuds

Cas Général

Se limiter à la détection des incompatibilités avec le leader est insuffisant, car des conflits peuvent également survenir entre les nœuds autres que le leader. Contrairement à l'approche décrite dans la Section 4.4.2, les nœuds n'ont pas besoin d'attendre la décision finale pour identifier une violation des contraintes.

Dans la majorité des cas, le leader détectera une incompatibilité lors de la réception d'un `Ack` provenant de la minorité. Cependant, en cas de perte de messages ou de perturbations du réseau, les autres nœuds peuvent également tenter de stopper l'instance en cours en envoyant un message `Abort` via un broadcast dès la réception d'un message `Accept` ou `Accepted` contenant une incompatibilité.

Dans la Figure 4.4, dès réception du message `Accept` avec l'ensemble des *nombres pairs*, le nœud 5, qui doit décider d'un multiple de 4, envoie un `Abort` pour corriger l'ensemble accepté. À la réception de l'`Abort`, le leader inclus l'ensemble du nœud 5 et envoie un nouvel `Accept` avec un numéro de ballot plus élevé, contournant ainsi la phase de préparation.

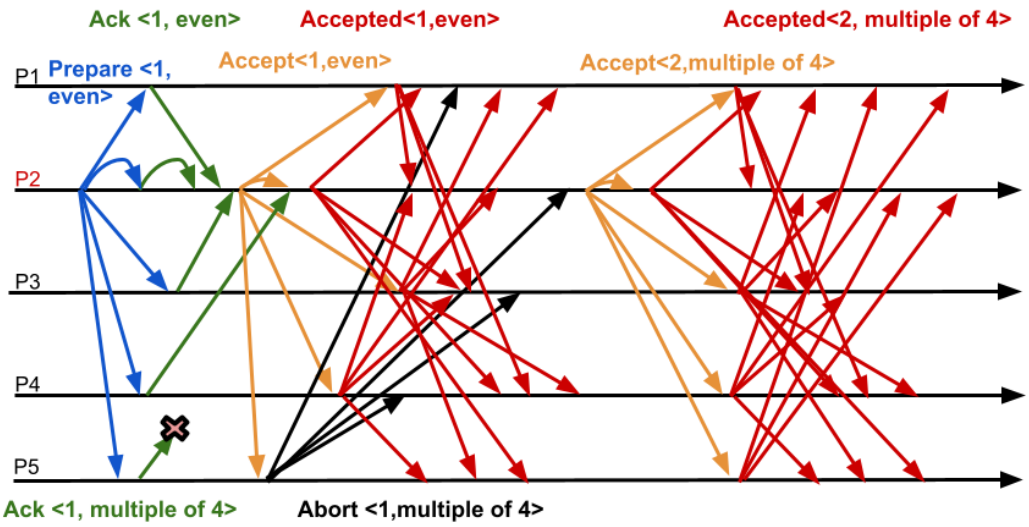


FIGURE 4.4 – Détection d'une incompatibilité à la réception d'un `Accept`

Dans Paxos original, lorsque le leader est stable, une optimisation courante consiste à omettre la phase de préparation. Cependant, dans notre scénario, cette phase reste nécessaire, car elle a été modifiée pour permettre aux nœuds de transmettre leurs contraintes. Toutefois, une fois les contraintes transmises et en cas de révocation, la phase de préparation devient inutile. Ainsi, en plus de la détection précoce des incompatibilités, nous pouvons éviter la phase de préparation après une révocation.

4.5 Paxos f-Révocable

Dans cette section, nous présentons le pseudo-code du Paxos f-Revocable. L'algorithme 3 détaille les variables locales utilisées par les processus.

Dans Paxos f-Révocable, toutes les variables sont représentées sous la forme de dictionnaires, associant un numéro d'instance k à une valeur (ballot, valeur décidée, etc.). Cette structure permet de distinguer clairement les différentes instances du

Algorithme 3 : Variables locales au processus i

1 **Variables propres à Paxos :**

2 val_i initially {(k1, none), (k2, none) ... }
`/* Map an instance number` k_j `with the last decided value */`

3 bal_i initially {(k1, -1), (k2, -1).... }
`/* Map the latest ballot known with an` k_j `instance number */`

4 $lastBalAccept$ initially {(k1, -1), (k2, -1) ... }
`/* Map an instance number` k_j `with the last ballot accepted */`

5 $lastIntersectionAccept$ initially {(k1, none), (k2, none) ... }
`/* Map an instance number` k_j `with the last intersection accepted */`

6 **Variables ajoutées pour la gestion des contraintes :**

7 set_i initially {(k1, s1), (k2, s2) ... }
`/* Map an instance number` k_j `to a set of constraints` s_j `*/`

8 $intersection_i$ initially {(k1, none), (k2, none) ... }
`/* Intersection of set known per instance */`

9 $abortBal_i$ initially $\emptyset$
`/* Ballots to ignore */`

protocole. Dans Paxos original, le numéro de ballot sert à identifier et à ordonner les propositions concurrentes des processus issues de différents leaders. Dans notre version, il gère également les révocations. Pour simplifier ce mécanisme, nous supposons un leader stable qui ordonne les propositions des nœuds. Ainsi, au lieu d'incrémenter le numéro de ballot, chaque nouvelle proposition (prepare) correspond à une nouvelle instance.

Variables propres à l'algorithme de Paxos

Le Paxos f-Révocable reprend les variables du Paxos original. Les variables sont décrites pour un numéro d'instance donné k_j :

- val_i (ligne 2) : la dernière valeur décidée par le processus p_i (pour un numéro

d'instance).

- bal_i (ligne 3) : le dernier numéro de ballot auquel p_i a participé.
- $lastBalAccept$ (ligne 4) : le dernier numéro de ballot accepté par p_i.
- $lastIntersectionAccept$ (ligne 5) : la dernière intersection d'ensembles acceptée par p_i.

Les variables $lastBalAccept$ et $lastIntersectionAccept$ permettent de limiter le nombre de révocations à f, même en présence de fautes. En cas de panne du leader, elles aident les nœuds à retrouver l'état où ils se sont arrêtés, évitant ainsi de repartir du début.

Variables permettant la gestion des contraintes

Cette section présente les variables ajoutées pour la gestion des contraintes des nœuds.

- set_i (ligne 7) : l'ensemble des valeurs respectant les contraintes du processus p_i.
- $intersection_i$ (ligne 8) : l'intersection des ensembles connus de p_i, représentant les contraintes que le nœud a réussi à rassembler.
- $abortBal_i$ (ligne 9) : l'ensemble des ballots qui ont été abandonnés.

4.5.1 Phase de préparation

L'algorithme 4 présente la nouvelle version de la phase de préparation.

Comme mentionné précédemment, le leader inclut désormais son ensemble de valeurs dans le message `Prepare` (ligne 1). À la réception de ce message, les nœuds réalisent l'intersection de leur ensemble avec celui reçu. Si l'intersection n'est pas vide, le nœud répond en envoyant un `Ack` au leader, contenant son ensemble ainsi que les valeurs des variables spécifiques à l'algorithme Paxos (derniers ballots et intersections acceptés) (ligne 6). Dans le cas contraire, un message `Abort` est immédiatement envoyé à tous les nœuds pour interrompre l'instance en cours (ligne 8).

Lorsqu'il reçoit un `Ack`, le leader met à jour sa connaissance de l'intersection des ensembles (ligne 14). Si l'intersection est vide, c'est-à-dire qu'aucune valeur ne peut satisfaire l'ensemble des nœuds, il est inutile d'attendre la réception d'une majorité d'`Ack`. Le leader envoie directement un message `Accept` (lignes 15 à 18).

Si une majorité d'`Ack` est obtenue, deux cas de figure se présentent. Si aucune intersection n'a été précédemment acceptée, l'intersection reste inchangée. Sinon, la dernière intersection acceptée associée au plus grand numéro de ballot est ajoutée à l'intersection (lignes 21 à 24). Le leader envoie alors un message `Accept` avec l'intersection retenue.

Enfin, si le leader détecte que l'intersection retenue par la majorité n'est pas incluse dans l'un des ensembles de la minorité, il déclenche une procédure d'`Abort` (lignes 28 à 30). Cela se traduit par l'envoi d'un nouvel `Accept` (Algorithme 6 ligne 15).

4.5.2 Phase d'acceptation

La phase d'acceptation, présentée dans l'algorithme 5, suit un schéma similaire à celui du Paxos original. Cependant, nous avons introduit une vérification des contraintes lors de la réception des messages `Accept` et `Accepted` (lignes 5 et 15) afin de détecter d'éventuelles incompatibilités entre les ensembles. Ainsi, si l'ensemble de valeurs reçues n'est pas inclus dans l'ensemble de valeurs du nœud lors de la réception d'un message `Accept` ou `Accepted`, une procédure d'abandon (*Abort*) est déclenchée (lignes 9 et 16).

Puisque l'élément accepté est un ensemble, une valeur doit être choisie lors de la décision. Deux scénarios sont possibles : si l'intersection des ensembles est vide, la valeur décidée est $\perp$; sinon, la valeur choisie est une valeur appartenant à l'ensemble, par exemple la plus petite (lignes 19 à 22).

Algorithme 4 : Phase de préparation pour le processus p_i

```
1  Upon reception of Prepare(k, bal, set_j) from p_j :
2  begin
3      if bal_i(k) ≤ bal then
4          bal_i(k) ← bal
5          if set_j ∩ set_i(k) ≠ ∅ then
6              send(Ack, k, bal, set_i(k), lastIntersectionAccept_i(k),
                lastBalAccept_i(k)) to p_j
7          else
8              send(Abort, k, bal, set_i(k)) to all
9          end
10     end
11 end

12 Upon reception of Ack (k, bal, set_j, lastIntersectionAccept_j,
     lastBalAccept_j) from p_j :
13 begin
14     intersection_i(k) ← intersection_i(k) ∩ set_j
15     if intersection_i(k) == ∅ and not already sent an Accept message for
        empty intersection then
16         lastIntersectionAccept_i(k) ← ∅
17         send(Accept, bal, ∅) to all
18         return
19     end
20     if p_i received Ack from a majority of participants for bal and
        bal ∉ abortBal(k) then
21         if p_i received an Ack with lastBalAccept≠ −1 then
22             highestLastIntersectioneAccept ← lastIntersectionAccept
                associated with the highest lastBalAccept received
23             intersection_i(k) ←
                intersection_i(k) ∩ highestLastIntersectionAccept
24         end
25         lastIntersectionAccept_i(k) ← intersection_i(k)
26         send(Accept, bal, intersection_i(k)) to all
27     end
28     if p_i received Ack from the minority and
        lastIntersectionAccept_i(k) ⊄ set_j then
29         abortAndSendNewAccept(k, bal, set_j)
30         return
31     end
32 end
```

Algorithme 5 : Phase d'acceptation pour le processus p_i

```
1  Upon reception of Accept(k, bal, intersectionAccept_j) from p_j :
2  begin
3      if bal_i(k) ≤ bal then
4          bal_i(k) ← bal
5          if intersectionAccept_j ⊂ set_i(k) and bal ∉ abortBal(k) then
6              lastIntersectionAccept_i(k) ← intersectionAccept_j
7              send(Accepted, k, bal, intersectionAccept_j ) to all
8          else
9              send(Abort, k, bal, set_i(k)) to all
10         end
11     end
12 end

13 Upon reception of Accepted  (k, bal, intersectionAccept_j) from p_j :
14 begin
15     if intersectionAccept_j ⊈ set_i(k) then
16         send(Abort, k, bal, set_i(k)) to all participants
17     end
18     if  p_i received Acccepted from a majority of participants for bal and
        bal ∉ abortBal(k) then
19         if intersectionAccept_j == ∅ then
20             val_i(k) ← ⊥
21         else
22             val_i(k) ← A value belonging to the intersection (e.g the smallest)
23         end
24     end
25 end
```

4.5.3 Abort

Lorsqu'un nœud détecte une incompatibilité, une procédure d'abort est initiée. Cette procédure est présentée dans l'algorithme 6. À la réception d'un message `Abort`, les nœuds se souviennent des numéros de ballots annulés afin d'ignorer les messages associés à ces ballots (ligne 3). Le leader, quant à lui, met à jour l'intersection des ensembles et le numéro de ballot, puis envoie un nouvel `Accept` à tous les nœuds (lignes 5 et 6).

Algorithme 6 : Abort pour le processus p_i

```
1  Upon reception of message Abort(k, bal, set_j) from p_j :
2  begin
3      abortBal(k).add(bal)
4      if p_i is the leader of bal then
5          if set_j ∩ set_i(k)==∅ or lastIntersectionAccept_i(k) ⊄ set_j then
6              abortAndSendNewAccept(k, bal, set_j)
7          end
8      end
9  end

10 Primitive : abortAndSendNewAccept(k, bal, set_j) :
11 begin
12     intersection_i(k) ← intersection_i(k) ∩ set_j
13     lastIntersectionAccept_i(k) ← intersection_i(k)
14     bal_i(k) ← max (bal, bal_i(k) )+1
15     send(Accept, bal_i(k), intersection_i(k)) to all
16 end
```

4.6 Esquisse de preuve

Cette section présente le principe de la preuve.

Pour garantir l'unicité de la décision finale, nous supposons des canaux équitables et

renvoyons périodiquement la valeur décidée.

4.6.1 Cohérence

Pour rappel, la propriété de cohérence est la suivante :

Deux processus corrects ne peuvent pas, à terme, décider de valeurs différentes.

Dans le protocole du Paxos original, avant qu'un *proposer* ne suggère une valeur, il vérifie auprès d'une majorité d'*acceptors* si une valeur a déjà été choisie. Si tel est le cas, le *proposer* propose cette valeur choisie aux autres *acceptors* afin d'assurer l'unicité de la valeur décidée et de maintenir l'intégrité.

En effet, si une majorité d'*acceptors* acceptent une valeur, celle-ci est considérée comme choisie et doit être décidée à terme. Dans le cas contraire, un autre *proposer* pourrait proposer une valeur différente à une autre majorité d'*acceptors*. Certains *acceptors* pourraient recevoir les deux propositions, car une majorité signifie que les deux ensembles ont au moins un *acceptor* en commun. Cela suggère qu'au moins un *acceptor* a changé d'avis sur la valeur finale, violant ainsi à la fois les propriétés de cohérence et d'intégrité.

Dans le Paxos f-Revocable, les nœuds peuvent révoquer des décisions et donc décider plusieurs fois. Il faut donc s'assurer qu'à terme, deux processus corrects ne puissent pas décider de valeurs différentes.

Pour garantir l'unicité de la décision finale, nous supposons que la dernière valeur décidée est périodiquement retransmise. La valeur associée au plus grand numéro de ballot sera ainsi décidée par tous les processus corrects.

Supposons par l'absurde que deux processus corrects p_1 et p_2 décident respectivement des valeurs différentes $v1$ et $v2$. Un numéro de ballot est incrémenté dès qu'une incompatibilité est détectée. Ce numéro de ballot est alors associé à une nouvelle

valeur. Comme $v1 \neq v2$, nous avons $b1 \neq b2$, et donc $b1 < b2$ ou $b1 > b2$. Le raisonnement étant symétrique, supposons donc $b1 < b2$. Dès qu'un message avec un numéro de ballot plus élevé est reçu, tous les messages associés à un numéro de ballot inférieur sont ignorés. Ainsi, si p_1 décide de $v1$, cela signifie qu'il n'a pas reçu de messages associés au ballot $b2$. Comme la décision est périodiquement retransmise, les canaux équitables et que p_1 est correct, cela n'est pas possible.

4.6.2 Validité

La propriété de validité stipule :

Si un processus correct p_i décide v, alors deux cas sont possibles :

- $v \in E_i$ **ou**
- v **est égal à $\perp$ si l'intersection des ensembles reçus par au moins un processus correct est vide.**

Soit $p_i \in \Pi$ un processus correct et $E_i \in E$ son ensemble de contraintes. Soit v la valeur décidée par p_i.

Supposons par l'absurde que $v \neq \perp$ et que $v \notin E_i$.

Puisque $v \neq \perp$, un processus de la minorité peut initier une révocation. Comme $v \notin E_i$, cela signifie que les contraintes de p_i n'ont pas été respectées. Étant donné que les décisions sont périodiquement retransmises et que p_i est correct, il finira par détecter que la valeur décidée ne fait pas partie de son ensemble de contraintes. À terme, sa demande de révocation sera prise en compte par au moins une majorité de processus corrects. Ainsi, l'intersection des ensembles connus intégrera les contraintes de p_i. Si cet ensemble n'est pas vide, alors v ne pourra pas être égal à $\perp$ et appartiendra nécessairement à E_i. Si cet ensemble est vide, v prendra la valeur $\perp$ et ne pourra donc pas appartenir à E_i. Dans les deux cas, il est impossible que v n'appartienne pas à E_i tout en étant différent de $\perp$.

4.6.3 Intégrité

La propriété d'intégrité est :

Chaque processus peut décider au maximum f fois. Cependant, après avoir décidé $\perp$, un processus ne peut plus décider une autre valeur.

Pour choisir une valeur, le leader doit recevoir les ensembles d'une majorité de processus. Une fois cette majorité atteinte, l'intersection des ensembles est acceptée et une valeur appartenant à cet ensemble est décidée. Par conséquent, les ensembles de la minorité ne sont pas pris en compte.

En conséquence, jusqu'à f processus (représentant la minorité) peuvent détecter une incompatibilité et déclencher une révocation en précisant leurs contraintes. Deux cas de figure se présentent alors :

- Si les contraintes de tous les processus corrects sont reçues par le leader après au plus f révocations, le leader choisit une valeur compatible qui sera acceptée.
- Sinon, le leader envoie $\perp$ en cas d'impossibilité et tous décident de cette valeur.

Dans les deux cas, aucune nouvelle révocation ne peut être initiée.

En cas de panne du leader, le dernier ensemble accepté est transmis via les messages `Ack` au nouveau leader. Cela permet aux nœuds de reprendre là où ils s'étaient arrêtés, évitant de repartir de zéro et de décider d'ensembles déjà jugés incompatibles.

Il est donc impossible de décider plus de f fois.

4.6.4 Terminaison

La propriété de terminaison impose que :

Chaque processus correct doit à terme décider.

Le protocole Paxos ne garantit pas la terminaison si des ballots sont démarrés infiniment souvent. Pour assurer la terminaison à terme, une proposition doit rester valide, c'est-à-dire avoir le numéro de ballot le plus élevé, suffisamment longtemps pour obtenir une majorité d'`Acks` et envoyer un `Accept` à une majorité de nœuds. Si au moins une majorité de nœuds coopèrent, l'algorithme peut se terminer.

Nous supposons donc la stabilité à terme du leader (détecteur Ω) ainsi qu'une retransmission des décisions. Le nombre de révocations étant borné, la terminaison est assurée.

4.7 Évaluation

Cette section présente les différentes expériences réalisées pour tester le Paxos f-Révocable.

4.7.1 Environnement d'expérimentation et configuration

Cette section décrit l'environnement dans lequel nous avons réalisé les expériences.

Paramètres d'infrastructure

Nos expériences ont été réalisées à l'aide du simulateur Peersim [96]. Nous avons déployé 15 nœuds dans un graphe de communication complet, où chaque nœud peut échanger avec tous les autres.

Selon les expériences, nous avons supposé trois localisations pour nos nœuds. Dans un premier scénario, nous avons considéré un environnement homogène, avec tous les nœuds situés à Paris. La latence moyenne d'un RTT (Round-Trip Time) y est de 8 ms, avec une distribution normale et un écart type de 10%. Dans un second scénario, nous avons réparti certains nœuds à Osaka et dans l'Oregon. La Figure 4.5

nous donne le RTT moyen entre chaque site [100].

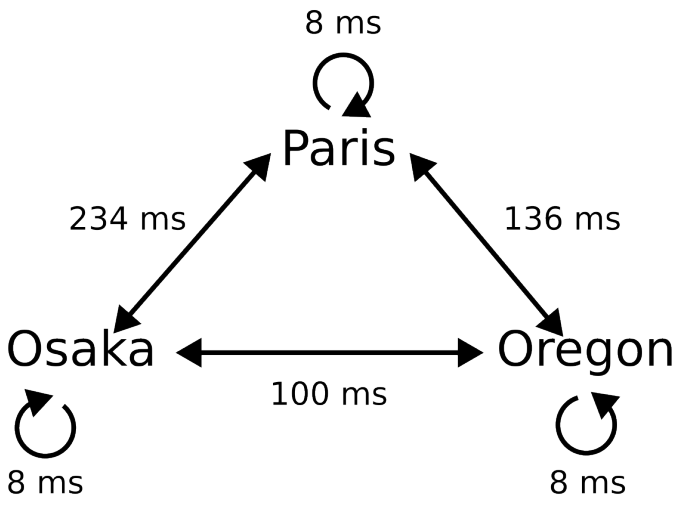


FIGURE 4.5 – RTT moyen entre les différents sites

Métriques

Nous comparons les performances de deux versions de Paxos implantant le problème du Consensus f-Révocable :

- L'enchaînement de consensus avec abort, présenté en 4.4.2. Cette version sera appelée `Paxos v1` dans la suite.
- L'algorithme Paxos f-Révocable, décrit en 4.5, que nous nommerons `Paxos v2`.

Nous avons étudié l'impact du nombre de révocations sur les deux métriques suivantes :

- La **latence moyenne**, qui correspond au temps écoulé entre le démarrage d'une instance de Paxos par un nœud et le moment où un quorum de nœuds parvient à une décision irrévocable.
- Le **nombre de messages**, qui correspond au nombre total de messages envoyés pour une instance de Paxos.

4.7.2 Étude des révocations

Dans un premier temps, nous avons étudié l'impact des révocations en nous abstrayant des ensembles de contraintes. Les révocations ont donc été déclenchées avec une certaine probabilité. Pour chaque probabilité considérée, nous avons exécuté 400 instances de Paxos, que nous avons ensuite regroupées en fonction du nombre de révocations. Pour l'algorithme Paxos v2, nous avons différencié les révocations selon le type de message.

En considérant 15 nœuds, la majorité est composée de 8 nœuds, ce qui limite à 7 le nombre maximal de révocations possibles. Tous les nœuds sont situés à Paris. Les courbes présentées ont été tracées avec des intervalles de confiance de 95%.

Les révocations sont déclenchées lors de la prise de décision pour Paxos v1, ou à la réception d'un message `Ack`, `Accept`, ou `Accepted` pour Paxos v2. Chaque type de révocation est représenté par une courbe distincte.

Les Figures 4.6 et 4.7 illustrent respectivement l'impact des révocations sur la latence et le nombre de messages échangés. Nous constatons que l'algorithme Paxos v2 offre de meilleures performances en termes de latence par rapport au Paxos v1. Plus la détection d'incompatibilité est précoce, moins la latence est affectée. Lors d'une révocation, la latence du Paxos v2 `Ack` est réduite de 55,6% par rapport à celle du Paxos v1. Même une détection tardive à la réception d'un `Accepted`, permet d'obtenir des gains significatifs, avec une latence inférieure de 17,1% par rapport au Paxos v1.

Pour Paxos v1, nous observons une augmentation significative de la latence dès la première révocation (Figure 4.6). C'est un résultat attendu puisque chaque révocation nécessite l'envoi d'un message `Abort` et le démarrage d'une nouvelle instance de Paxos. Toutefois, au-delà de la première révocation, la latence augmente assez peu. Nous pourrions penser de prime abord que chaque révocation déclencherait une nouvelle instance de Paxos dans son intégralité, entraînant une augmentation linéaire de la latence. Ce n'est pas le cas, et cette observation s'explique par l'intervalle de temps entre les révocations et le numéro de ballot.

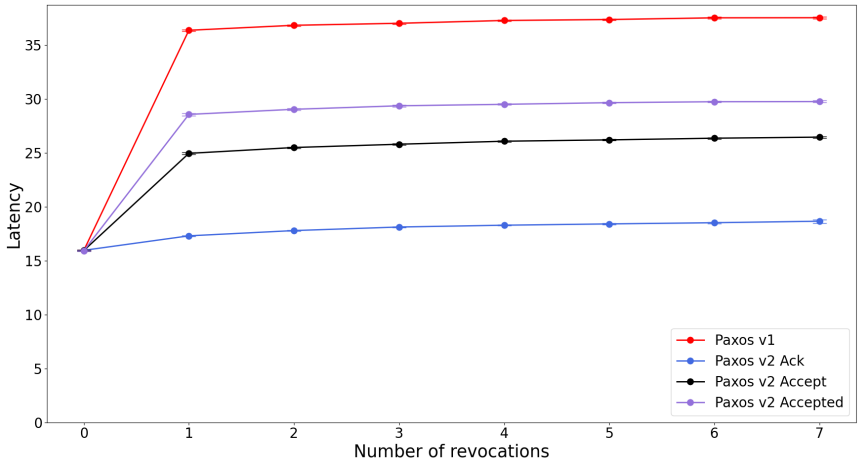


FIGURE 4.6 – Impact des révocations sur la latence moyenne

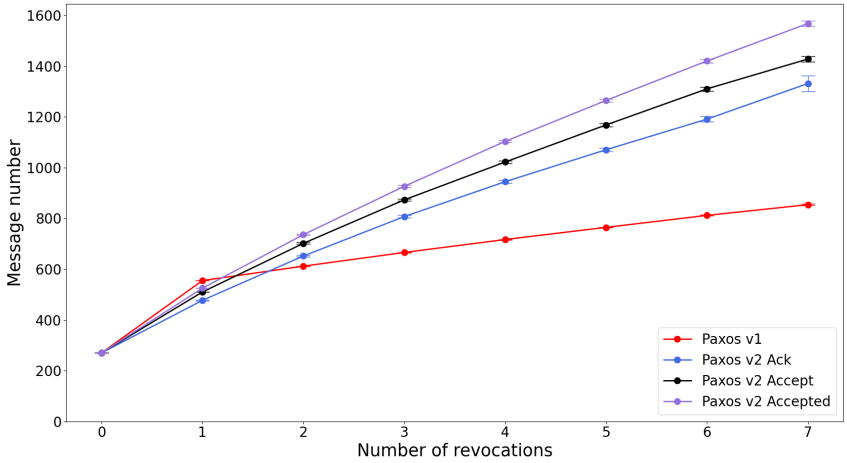


FIGURE 4.7 – Impact des révocations sur le nombre de messages

Les révocations sont lancées à la chaîne, ainsi lorsqu'un même message est reçu, les nœuds déclenchent une révocation avec une certaine probabilité. L'intervalle entre deux révocations est donc très court. La phase de préparation dans le Paxos v1 nécessite une vérification du numéro de ballot. Par conséquent, si deux révocations

sont lancées, la première ne pourra pas compléter son instance de Paxos avant que la deuxième instance ne la rende obsolète (son numéro de ballot étant plus élevé).

Intéressons-nous maintenant aux échanges de messages (voir Figure 4.7). Dans Paxos v2, pour réduire la latence liée à la première phase, le leader envoie directement un message `Accept` contenant la nouvelle intersection des ensembles de contraintes. Ce message est ensuite diffusé via un broadcast `Accepted` dès sa réception. Ainsi, à chaque nouvelle révocation, un `Accept` et des `Accepted` sont envoyés. Cela se traduit sur les courbes par une augmentation linéaire du nombre de messages en fonction du nombre de révocations.

Bien que des révocations aient été déclenchées pour chaque type de message, en pratique, nous nous attendons à observer une courbe similaire à celle des révocations initiées lors des `Ack`. En effet, en l'absence de fautes, le leader sera le premier à détecter et corriger les incompatibilités entre les contraintes. Par conséquent, l'envoi d'un `Abort` lors de la réception d'un message `Accepted` ne survient qu'en cas de perte de messages.

4.7.3 Étude d'un cas pratique

Nous avons évalué nos deux versions du Paxos f-révocable dans un scénario concret. Nous avons supposé que les nœuds étaient soumis à des contraintes sur deux variables : x et y. D'un point de vue pratique, ces variables pourraient correspondre à des ressources telles que la mémoire ou le CPU. Chaque nœud possède, pour chaque variable, un ensemble de valeurs respectant ses contraintes. L'objectif est de parvenir à un consensus permettant de déterminer les valeurs de x et y.

Parmi les 15 nœuds étudiés, 11 ne sont soumis à aucune contrainte, ce qui signifie qu'ils peuvent accepter n'importe quelle valeur. En revanche, les 4 nœuds restants établissent des contraintes :

- 1er nœud : impose que x soit supérieur ou égal à une valeur minimale, notée

$x_{\min}$.

- 2ème nœud : impose que x soit inférieur ou égal à une valeur maximale, notée $x_{\max}$.
- 3ème nœud : impose que y soit supérieur ou égal à une valeur minimale, notée $y_{\min}$.
- 4ème nœud : impose que y soit inférieur ou égal à une valeur maximale, notée $y_{\max}$.

Les quatre nœuds soumis à des contraintes sont choisis aléatoirement parmi les 15 nœuds, selon une distribution uniforme.

Probabilité des révocations attendues

Une révocation est déclenchée lorsqu'une incompatibilité est détectée avec une contrainte imposée par un nœud de la minorité. Autrement dit, si l'un des quatre nœuds contraints ne parvient pas à être suffisamment rapide pour rejoindre la majorité, une révocation se produira. La probabilité que x nœuds contraints fassent partie de la majorité est donnée par l'équation 4.1 :

$$P(\text{x contraintes dans } Q) = \frac{\binom{4}{x}\binom{11}{8-x}}{\binom{15}{8}} \tag{4.1}$$

Les équations 4.2, 4.3, 4.4, 4.5 et 4.6 déterminent les probabilités d'avoir de 0 à 4 nœuds dans le quorum (et donc 4 à 0 révocations). Nous notons Q comme étant le quorum.

$$P(4 \text{ contraintes dans } Q) = P(0 \text{ révocation}) = \frac{\binom{4}{4}\binom{11}{4}}{\binom{15}{8}} \simeq 5,13\% \tag{4.2}$$

$$P(3 \text{ contraintes dans } Q) = P(1 \text{ révocation}) = \frac{\binom{4}{3}\binom{11}{5}}{\binom{15}{8}} \simeq 28,72\% \tag{4.3}$$

$$P(\text{2 contraintes dans } Q) = P(\text{2 révocations}) = \frac{\binom{4}{2}\binom{11}{6}}{\binom{15}{8}} \simeq 43,08\% \tag{4.4}$$

$$P(\text{1 contrainte dans } Q) = P(\text{3 révocations}) = \frac{\binom{4}{1}\binom{11}{7}}{\binom{15}{8}} \simeq 20,51\% \tag{4.5}$$

$$P(\text{Aucune contrainte dans } Q) = P(\text{4 révocations}) = \frac{\binom{11}{8}}{\binom{15}{8}} \simeq 2,56\% \tag{4.6}$$

Résultats sur un réseau homogène

Nous avons étudié deux variantes de Paxos v2 : l'une avec envoi de broadcast `Abort` et l'autre sans. Lorsqu'un `Ack` de la minorité est reçu , si le leader détecte une incompatibilité avec la valeur précédemment acceptée, il ne diffuse pas de broadcast `Abort` pour prévenir les autres nœuds. À la place, il résout l'incompatibilité en envoyant directement un nouveau message `Accept`.

Par conséquent, certains nœuds de la minorité peuvent encore envoyer un message de broadcast `Abort` après avoir reçu le premier message `Accept`, ce qui entraîne un surcoût en termes de messages.

Nous souhaitons maintenant vérifier si les probabilités précédemment calculées se retrouvent dans les expériences. Pour cela, nous avons exécuté 10 000 instances de Paxos. Le Tableau 4.1 présente les proportions obtenues pour Paxos v1, v2 et sa version sans `abort` :

Ces valeurs concordent avec celles des équations 4.2, 4.3, 4.4, 4.5 et 4.6.

Intéressons-nous maintenant aux courbes représentées dans les Figures 4.8 et 4.9 sous forme de nuages de points. Cela permet de visualiser la proportion d'instances de Paxos en fonction du nombre de révocations. Nous observons d'abord que le nombre de révocations maximum s'élève à 4 ce qui est cohérent avec le scénario (4 nœuds

Nombre de révocations	Paxos v1	Paxos v2	Paxos v2 sans abort
0	5,82%	6,31%	5,97%
1	30,26%	29,33%	30,51%
2	41,73%	42,30%	42,13%
3	19,80%	19,70%	19,09%
4	2,39%	2,36%	2,30%

TABLE 4.1 – Proportion d'instances en fonction du nombre de révocations

avec contraintes et 11 nœuds sans).

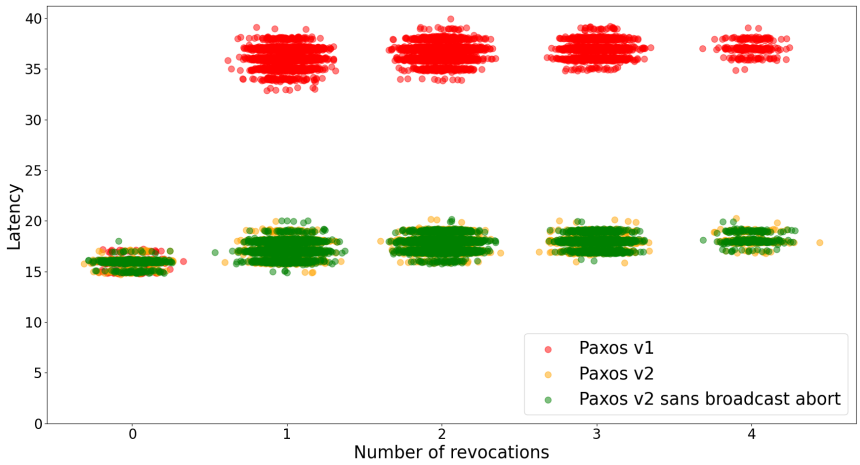


FIGURE 4.8 – Impact des révocations sur la latence

Dans la Figure 4.8, nous observons des résultats de latence similaires à ceux de la Figure 4.6. Pour Paxos v2, quelle que soit la variante, les valeurs correspondent à des révocations ayant eu lieu à la réception d'un `Ack`. Les résultats pour Paxos v1 sont également cohérents.

Dans la Figure 4.9, nous analysons l'impact des révocations sur le nombre de messages échangés. Les résultats sont également similaires à ceux présentés dans la Figure 4.7. Pour Paxos v2, bien que le `Abort` n'ait pas d'effet sur la latence, il est clair que son absence entraîne une diminution du nombre de messages. Ainsi, nous pouvons nous

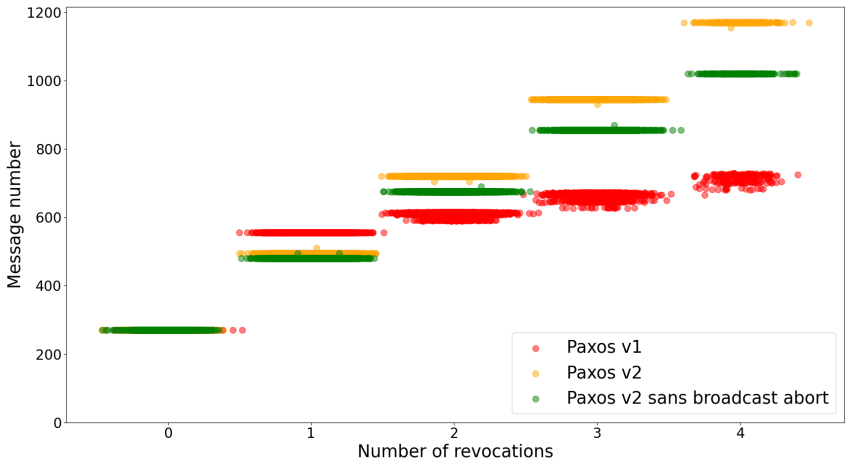


FIGURE 4.9 – Impact des révocations sur le nombre de messages

interroger sur l'utilité de ce `broadcast`.

Résultats dans un réseau hétérogène

Le broadcast `Abort` permet de pallier la perte de messages tolérée dans l'algorithme de Paxos. Si un `Ack` est perdu, le leader peut ne pas être informé des contraintes d'un nœud et donc d'une incompatibilité. Le broadcast `Abort` permet aux nœuds de stopper l'instance en cours le plus rapidement possible après la détection d'une incohérence. Cependant, comme nous l'avons observé dans la section précédente, ce broadcast entraîne un coût supplémentaire en messages. Mais ce n'est pas toujours le cas.

Nous considérons désormais que nos 15 nœuds sont répartis de manière égalitaire sur 3 sites : Paris, Osaka et Oregon (voir Figure 4.5).

Nous supposons que le leader se situe à Paris. De manière similaire à l'expérience précédente, les 4 nœuds possédant des contraintes sont choisis aléatoirement selon une loi uniforme, et 10 000 instances de Paxos sont lancées. Les Figures 4.10 et 4.11

présentent les résultats obtenus en termes de latence et de messages échangés dans ce contexte hétérogène.

Intéressons-nous dans un premier temps à la latence. Nous observons une fois de plus que les versions de Paxos f-Révocable, avec ou sans diffusion d'`Abort`, affichent des valeurs de latence identiques. Selon la localisation des nœuds ayant déclenché les révocations, les valeurs de latence varient. En effet, le leader se trouvant à Paris, une révocation initiée à Osaka prolonge davantage le temps de convergence qu'une révocation déclenchée dans l'Oregon. C'est pourquoi nous constatons deux regroupements distincts sur le nuage de points. Les points présentant les latences les plus élevées (environ 500 ms) indiquent qu'au moins une révocation a été initiée à Osaka. Les autres points (environ 300 ms) suggèrent que seules des révocations démarrant dans l'Oregon ont eu lieu. Cela explique également pourquoi le nombre de révocations issu de l'Oregon n'excède pas 2. En effet, le quorum est généralement constitué des nœuds les plus rapides par rapport à la localisation du leader, à savoir 5 nœuds situés à Paris et 3 dans l'Oregon.

En ce qui concerne le nombre de messages, nous pouvons observer une inversion de la tendance constatée en milieu homogène. Cette fois-ci, les diffusions de `Abort` permettent de réduire le nombre de messages. Cette diminution s'explique par la topologie du système.

L'idée derrière l'utilisation du `Abort` est, au-delà de la gestion de la perte éventuelle de messages, de stopper rapidement les nœuds. Cela permet d'empêcher leur progression dans l'algorithme et d'éviter ainsi le surcoût lié à l'envoi et à la réception de messages pour une valeur incompatible. Dans un réseau homogène où les nœuds sont proches les uns des autres, cette méthode n'est pas très efficace, car il est souvent déjà trop tard. En effet, lorsqu'une incompatibilité est détectée, un message `Accept` a déjà été envoyé. Sauf dans les cas où les messages sont reçus dans un ordre non FIFO, le message `Accept` sera reçu avant le `Abort` (ou un nouveau `Accept` dans le cadre d'une révocation au moment du `Ack`). La réception de cet `Accept` déclenche alors une série de diffusions `Accepted`, car les nœuds ne sont pas encore informés de l'incompatibilité,

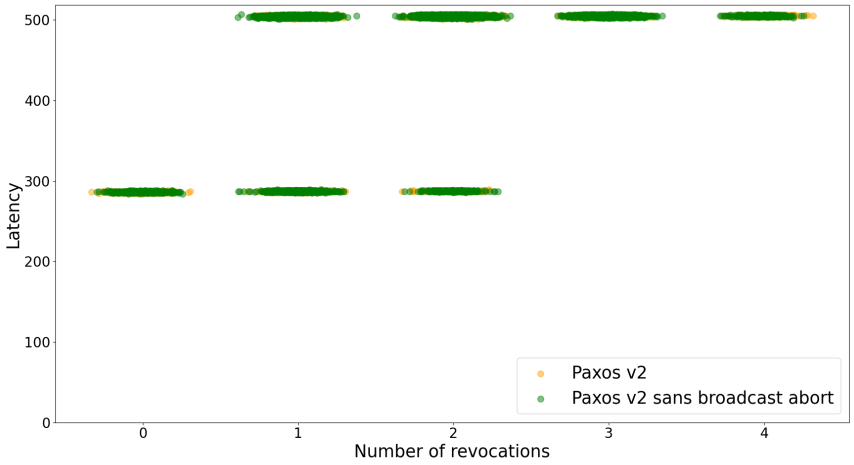


FIGURE 4.10 – Latence : Version avec ou sans abort dans un milieu hétérogène

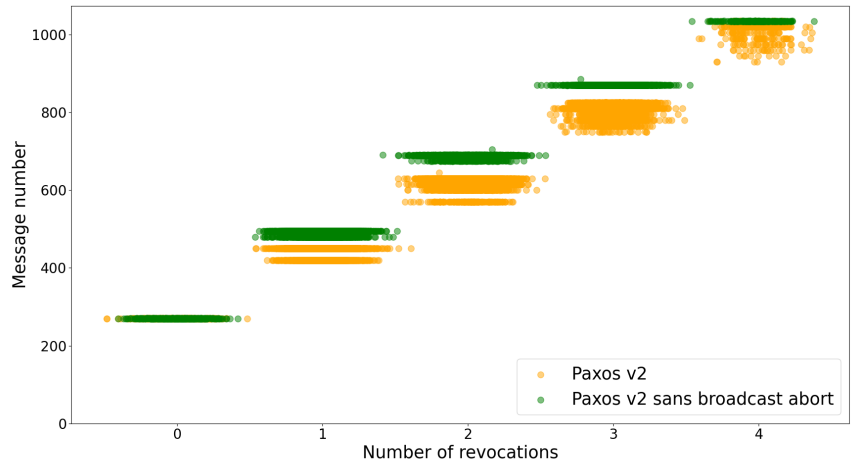


FIGURE 4.11 – Messages : Version avec ou sans abort dans un milieu hétérogène

sauf pour le nœud dont les contraintes sont violées et éventuellement le leader.

En revanche, avec les variations du réseau et dans cette configuration, un `Abort` peut empêcher la série de diffusions `Accepted` et ainsi limiter le coût en termes de messages. Dans la Figure 4.12, pour simplifier, nous considérons trois nœuds avec un

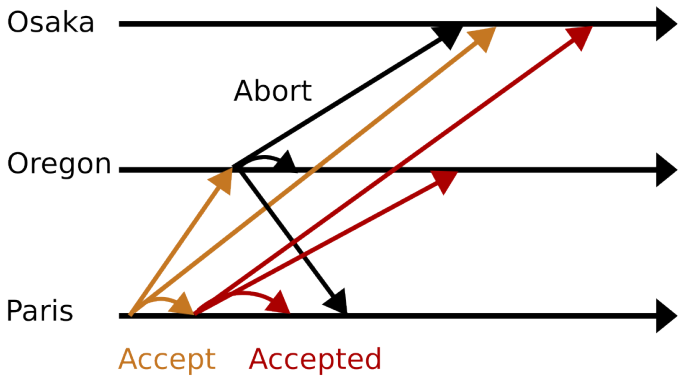


FIGURE 4.12 – Broadcast Accepted évité

leader situé à Paris. Lorsque le nœud en Oregon détecte une incompatibilité, il envoie immédiatement un broadcast `Abort`. En raison de la latence réseau entre les nœuds et de ses variations, le nœud à Osaka reçoit le `Abort` avant le `Accept` provenant de Paris, ce qui lui permet d'éviter la diffusion d'un `Accepted` à la réception de l'`Accept`. Ce phénomène peut se produire fréquemment, et, avec une expérience impliquant 15 nœuds, l'impact sur le nombre de messages est significatif.

4.8 Conclusion

Dans ce chapitre, nous avons présenté le concept de Consensus f-Révocable. Ce nouveau type de consensus affaiblit la propriété d'irrévocabilité des décisions. Dans le problème du consensus traditionnel, le choix de la majorité prévaut. Cependant, cela peut poser des problèmes dans des environnements complexes où les contraintes — qu'elles soient géographiques, algorithmiques ou matérielles — jouent un rôle crucial. Ainsi, n'importe quel nœud de la minorité peut révoquer une décision si celle-ci viole ses contraintes locales. Nous avons donc redéfini les quatre propriétés du Consensus f-Révocable : Cohérence, Validité, Intégrité et Terminaison.

Pour implanter ce consensus, nous avons proposé deux algorithmes adaptés de Paxos.

Nous avons d'abord présenté une solution fonctionnelle, puis nous avons cherché à l'améliorer afin d'augmenter ses performances. Cette nouvelle approche permet de réduire significativement le coût en latence : pour une révocation, la latence peut être diminuée de 55,6% par rapport à la première version. En revanche, le coût en messages est plus élevé, bien que celui-ci dépende de la configuration du réseau, comme nous l'avons observé lors des expériences réalisées.

Chapitre

5

Conclusion et Perspectives

Sommaire

Le consensus est un problème fondamental dans les systèmes distribués, permettant à plusieurs processus de s'accorder sur une valeur malgré l'éventuelle présence de fautes dans le système. Parmi les algorithmes de consensus largement utilisés, Paxos et ses variantes se distinguent. Ils présentent néanmoins des limitations concernant leur nombre de messages élevé et la consommation de bande passante qui en découle. La bande passante, en particulier dans les datacenters, est une ressource critique, souvent identifiée comme un goulot d'étranglement majeur. Une solution clas-

sique pour limiter la bande passante consiste à utiliser des mécanismes d'agrégation de messages. Cependant, ces mécanismes sont souvent agnostiques aux caractéristiques des applications, empêchant ainsi une agrégation intelligente des messages. En réponse à ces défis, nous avons proposé OMAHA, un mécanisme d'agrégation opportuniste applicable aux algorithmes à phases.

Dans des environnements complexes, tels que les systèmes multi-agents ou les systèmes d'allocation de ressources, les processus doivent composer avec des contraintes internes. Ces contraintes influencent le processus décisionnel, rendant les mécanismes de consensus traditionnels insuffisants pour satisfaire les obligations de chaque processus. Pour relever ce défi, nous avons proposé une extension du consensus : le Consensus f-Révocable. Cette approche introduit la prise en compte des contraintes propres à chaque processus et permet une réévaluation des décisions lorsque celles-ci entrent en conflit avec les contraintes d'un processus.

Les sections 5.1 et 5.2 résument ces deux contributions. La section 5.3 propose des perspectives pour des travaux futurs.

5.1 OMAHA : Agrégation opportuniste de messages pour algorithmes à phases

OMAHA est un mécanisme d'agrégation de messages opportuniste qui permet à un utilisateur de trouver un équilibre entre l'économie de bande passante et la dégradation de la latence. Il est conçu pour des applications fonctionnant simultanément sur une même infrastructure physique et utilisant des algorithmes à phases.

Dans les algorithmes à phases, les étapes de communication entre les nœuds sont prévisibles. OMAHA tire parti de cette connaissance pour anticiper les échanges à venir, ce qui lui permet d'agréger certains messages et, ainsi, d'économiser la bande passante allouée aux en-têtes.

OMAHA propose une API qui établit une couche intermédiaire entre le réseau et les applications. Cette API est conçue pour être peu intrusive, minimisant ainsi les modifications apportées aux algorithmes. Nous avons appliqué ce mécanisme d'agrégation à quatre algorithmes à phases largement utilisés : trois variantes de Paxos et l'algorithme de diffusion atomique de Zookeeper. L'efficacité d'OMAHA dépend des caractéristiques de chaque algorithme, telles que le nombre de phases, la taille des quorums et le type de message. Il est donc essentiel de définir clairement ces éléments pour ajuster correctement les paramètres de l'API.

Dans un premier temps, nous avons mené nos expériences dans un environnement simulé, puis nous avons validé nos résultats dans un environnement réel via la plateforme Grid'5000. Nous avons évalué les performances d'OMAHA par rapport à une méthode agnostique, qui met en mémoire tampon tous les messages pour les envoyer de manière périodique. Nos résultats indiquent qu'il est possible d'économiser jusqu'à 30% de bande passante tout en maintenant une dégradation de la latence limitée à seulement 5% pour le protocole Paxos.

5.2 Consensus f-Révocable

Nous avons défini le Consensus f-Révocable pour gérer l'accord entre les processus dans des environnements distribués soumis à des contraintes internes. Cela répond aux limitations des algorithmes de consensus classiques, où la décision de la majorité prévaut et peut donc entraîner la violation des contraintes de la minorité. Le Consensus f-Révocable permet à ces processus de contester et de révoquer des décisions qui ne respectent pas leurs contraintes internes tout en assurant la convergence.

Nous avons proposé deux algorithmes adaptés de l'algorithme de Paxos implantant un Consensus f-Révocable. Un premier algorithme, non optimisé, qui exécute plusieurs instances de consensus de manière itérative jusqu'à ce que les contraintes soient respectées. Nous avons ensuite modifié cette approche pour améliorer les performances. Le deuxième algorithme permet ainsi de détecter une incompatibilité entre

les contraintes au sein même d'une instance de consensus en cours d'exécution. Cette nouvelle version permet de réduire significativement la latence. En effet, celle-ci peut être diminuée jusqu'à 55,6% pour une révocation. De plus, nous avons fourni des esquisses de preuves qui offrent une intuition sur la validité de nos algorithmes.

5.3 Perspectives

Cette section présente quelques perspectives de recherche ouvertes par les travaux présentés dans cette thèse.

5.3.1 OMAHA

Lors de notre étude du mécanisme OMAHA, nous avons constaté que son paramétrage joue un rôle crucial pour ses performances. Actuellement, le choix des paramètres incombe à l'utilisateur. À terme, nous envisageons de rendre ce processus automatique en ajustant dynamiquement les paramètres en fonction de l'état du système (vitesse de communication, congestion, etc.).

Dans cette thèse, nous avons choisi d'appliquer OMAHA aux algorithmes à phases. Le fait de connaître les différentes phases permet d'anticiper les communications futures, facilitant ainsi une agrégation intelligente des messages. Toutefois, il serait intéressant d'étendre l'utilisation d'OMAHA à un plus grand nombre d'algorithmes, notamment en intégrant des techniques d'apprentissage. Cela permettrait d'identifier les schémas de communication récurrents, offrant ainsi la possibilité de prédire et d'agréger les messages, même pour des algorithmes qui ne sont pas à phases.

À ce jour, OMAHA est implanté sous forme de bibliothèque. Pour en améliorer les performances et la portabilité, une intégration au niveau de la couche OS pourrait être envisagée. Cette intégration pourrait prendre deux formes. La première, explicite, consisterait à ajouter une API dédiée à OMAHA directement dans le noyau. Elle

pourrait prendre la forme d'un patch de l'appel système générique *socketcall*, permettant à l'application d'indiquer quels sont les messages à mettre en mémoire tampon. La seconde, implicite, confierait au noyau la gestion de cette dernière. Il faudrait alors ajouter un mécanisme d'analyse du trafic réseau pour inférer l'activation du mécanisme.

5.3.2 Consensus f-Révocable

Dans nos travaux sur le Consensus f-Révocable, nous avons supposé que les contraintes des processus restaient inchangées. Bien que cela réponde aux besoins de nombreuses applications distribuées, il serait intéressant de pouvoir faire évoluer ces contraintes pendant la résolution du consensus. Dans le cas du problème d'allocations de ressources, par exemple, un nœud peut participer à plusieurs instances de consensus en parallèle. Une décision prise dans l'une de ces instances modifiera les ressources disponibles, et donc les contraintes de ce nœud pour les autres instances en cours d'exécution. Cependant, intégrer la dynamique des contraintes est complexe, car cela ne doit pas entraver la convergence de l'algorithme.

Bibliographie

[1] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell : a scalable and fault-tolerant network structure for data centers," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, 2008, pp. 75–86.

[2] Z. Chkirbene, S. Foufou, and R. Hamila, "Vaconet : Variable and connected architecture for data center networks," in *2016 IEEE Wireless Communications and Networking Conference*. IEEE, 2016, pp. 1–6.

[3] E. Baccour, S. Foufou, and R. Hamila, "Ptnet : A parameterizable data center network," in *2016 IEEE Wireless Communications and Networking Conference*. IEEE, 2016, pp. 1–6.

[4] Z. Chkirbene, R. Hadjidj, S. Foufou, and R. Hamila, "Lascada : A novel scalable topology for data center network," *IEEE/ACM Transactions on Networking*, vol. 28, no. 5, pp. 2051–2064, 2020.

[5] J. Alqahtani, S. Alanazi, and B. Hamdaoui, "Traffic behavior in cloud data centers : A survey," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, 2020, pp. 2106–2111.

[6] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010, pp. 267–280.

[7] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic : measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, 2009, pp. 202–208.

[8] A. Lester, Y. Tang, and T. Gyires, "Application-aware bandwidth scheduling for data center networks," *Journal on Advances in Networks and Services*, vol. 7, 2014.

[9] M. R. S. Katebzadeh, P. Costa, and B. Grot, "Saba : Rethinking datacenter network allocation from application's perspective," in *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, G. A. D. Luna, L. Querzoni, A. Fedorova, and D. Narayanan, Eds. ACM, 2023, pp. 623–638. [Online]. Available : https://doi.org/10.1145/3552326.3587450

[10] D. Georgantas and P. A. Baziana, "Traffic burstiness study of an efficient bandwidth allocation MAC scheme for WDM datacenter networks," in *IEEE International Mediterranean Conference on Communications and Networking, MeditCom 2023, Dubrovnik, Croatia, September 4-7, 2023*. IEEE, 2023, pp. 169–174. [Online]. Available : https://doi.org/10.1109/MeditCom58224.2023.10266597

[11] D. Georgantas and P. Baziana, "Po-mac : A software defined performance-optimized mac strategy for optical data center networks," in *2023 Annual Modeling and Simulation Conference (ANNSIM)*. IEEE, 2023, pp. 657–667.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "{ZooKeeper} : Wait-free coordination for internet-scale systems," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner : Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013. [Online]. Available : https://www.usenix.org/system/files/conference/osdi12/osdi12-final-16.pdf

[14] A. Lakshman and P. Malik, "Cassandra : a decentralized structured storage system," *ACM SIGOPS operating systems review*, vol. 44, no. 2, pp. 35–40, 2010.

[15] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb : The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 1493–1509.

[16] S. Zhou and S. Mu, "{Fault-Tolerant} replication with {Pull-Based} consensus in {MongoDB}," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 687–703.

[17] L. Lamport, "The part-time parliament, may 1998," 1998. [Online]. Available : https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf

[18] F. Junqueira, B. Reed, and M. Serafini, "Zab : High-performance broadcast for primary-backup systems," *2011 IEEE/IFIP 41st Int. Conference on Dependable Systems & Networks (DSN)*, pp. 245–256, 2011. [Online]. Available : https://marcoserafini.github.io/papers/zab.pdf

[19] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, "Low-latency multi-datacenter databases using replicated commit," *Proc. of the VLDB Endowment*, vol. 6, no. 9, pp. 661–672, 2013. [Online]. Available : http://www.vldb.org/pvldb/vol6/p661-mahmoud.pdf

[20] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proc. of the 7th symposium on Operating systems design and implementation*, 2006, pp. 335–350.

[21] E. Shakshuki, H.-H. Koo, D. Benoit, and D. Silver, "A distributed multi-agent meeting scheduler," *Journal of Computer and System Sciences*, vol. 74, no. 2, pp. 279–296, 2008.

[22] A. Nedic, A. Ozdaglar, and P. A. Parrilo, "Constrained consensus and optimization in multi-agent networks," *IEEE Transactions on Automatic Control*, vol. 55, no. 4, pp. 922–938, 2010.

[23] C. Feng, Z. Xu, X. Zhu, P. V. Klaine, and L. Zhang, "Wireless distributed consensus in vehicle to vehicle networks for autonomous driving," *IEEE Transactions on Vehicular Technology*, vol. 72, no. 6, pp. 8061–8073, 2023.

[24] Z. Li, L. Zhang, X. Zhang, and M. Imran, "Design and implementation of a raft based wireless consensus system for autonomous driving," in *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, 2022, pp. 3736–3741.

[25] B. Jennings and R. Stadler, "Resource management in clouds : Survey and research challenges," *Journal of Network and Systems Management*, vol. 23, pp. 567–619, 2015.

[26] G. Fraysse, J. Lejeune, J. Sopena, and P. Sens, "A resource usage efficient distributed allocation algorithm for 5g service function chains," in *Distributed Applications and Interoperable Systems : 20th IFIP WG 6.1 International Conference, DAIS 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, 2020, Proceedings 20*. Springer, 2020, pp. 169–185.

[27] S. K. Tayyaba and M. A. Shah, "Resource allocation in sdn based 5g cellular networks," *Peer-to-Peer Networking and Applications*, vol. 12, no. 2, pp. 514–538, 2019.

[28] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, p. 225–267, Mar. 1996. [Online]. Available : https://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/p225-chandra.pdf

[29] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Gossip algorithms : Design, analysis and applications," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 3. IEEE, 2005, pp. 1653–1664.

[30] G. Pandurangan, P. Raghavan, and E. Upfal, "Building low-diameter peer-to-peer networks," *IEEE Journal on selected areas in communications*, vol. 21, no. 6, pp. 995–1002, 2003.

[31] F. Cristian and C. Fetzer, "The timed asynchronous distributed system model," *IEEE Transactions on Parallel and Distributed systems*, vol. 10, no. 6, pp. 642–657, 1999.

[32] J.-C. Laprie, "Dependable computing and fault-tolerance," *Digest of Papers FTCS-15*, vol. 10, no. 2, p. 124, 1985.

[33] V. Hadzilacos and S. Toueg, *Fault-tolerant broadcasts and related problems*. USA : ACM Press/Addison-Wesley Publishing Co., 1993, p. 97–145.

[34] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic broadcast : From simple message diffusion to byzantine agreement," *Information and Computation*, vol. 118, no. 1, pp. 158–179, 1995.

[35] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[36] M. S. P. Michael J. Fischer, Nancy A. Lynch, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, 1985. [Online]. Available : https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf

[37] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *J. ACM*, vol. 34, no. 1, p. 77–97, Jan. 1987. [Online]. Available : https://groups.csail.mit.edu/tds/papers/Stockmeyer/DolevDS83-focs.pdf

[38] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, p. 288–323, Apr. 1988. [Online]. Available : https://groups.csail.mit.edu/tds/papers/Lynch/jacm88.pdf

[39] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, vol. 43, no. 4, p. 685–722, Jul. 1996. [Online]. Available : https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.2086&rep=rep1&type=pdf

[40] M. Kawazoe Aguilera, W. Chen, and S. Toueg, "Heartbeat : A timeout-free failure detector for quiescent reliable communication," in *Distributed Algorithms : 11th International Workshop, WDAG'97 Saarbrücken, Germany, September 24–26, 1997 Proceedings 11*. Springer, 1997, pp. 126–140.

[41] V. K. Garg and J. R. Mitchell, "Distributed predicate detection in a faulty environment," in *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No. 98CB36183)*. IEEE, 1998, pp. 416–423.

[42] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg, "The weakest failure detectors to solve certain fundamental

problems in distributed computing," in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, 2004, pp. 338–346.

[43] R. Guerraoui, "Non-blocking atomic commit in asynchronous distributed systems with failure detectors," *Distributed Computing*, vol. 15, no. 1, pp. 17–25, 2002.

[44] R. Guerraoui, V. Hadzilacos, P. Kuznetsov, and S. Toueg, "The weakest failure detectors to solve quittable consensus and nonblocking atomic commit," *SIAM Journal on Computing*, vol. 41, no. 6, pp. 1343–1379, 2012.

[45] M. Ben-Or, "Another advantage of free choice (extended abstract) : Completely asynchronous agreement protocols," in *PODC '83 : Proc. of the second annual ACM symposium on Principles of distributed computing*, 1983, p. Pages 27–30. [Online]. Available : http://www.cs.cornell.edu/courses/cs5414/2017fa/papers/p27-ben-or.pdf

[46] G. Bracha and S. Toueg, "Resilient consensus protocols," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983, pp. 12–26.

[47] S. Toueg, "Randomized byzantine agreements," in *Proc. of the Third Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '84. New York, NY, USA : Association for Computing Machinery, 1984, p. 163–178. [Online]. Available : https://doi.org/10.1145/800222.806744

[48] S. T. Gabriel Bracha, "Asynchronous consensus and broadcast protocols," *Journal of the ACM*, 1985. [Online]. Available : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.8693&rep=rep1&type=pdf

[49] B. M. Oki and B. H. Liskov, "Viewstamped replication : A new primary copy method to support highly-available distributed systems," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, 1988, pp. 8–17.

[50] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[51] L. Lamport, "Paxos made simple," in *Procedings of the 6th Int. Conference on Principles of Distributed Systems. OPODIS 2002*, 2001. [Online]. Available : https://lamport.azurewebsites.net/pubs/paxos-simple.pdf

[52] S. Chaudhuri, "More choices allow more faults : Set consensus problems in totally asynchronous systems," *Inf. Comput.*, vol. 105, pp. 132–158, 1993.

[53] N. H. Vaidya and D. K. Pradhan, "Degradable agreement in the presence of byzantine faults," in *[1993] Proceedings. The 13th International Conference on Distributed Computing Systems.* IEEE, 1993, pp. 237–244.

[54] F. B. Schneider, "Replication management using the state-machine approach," *Distributed systems*, vol. 2, pp. 169–198, 1993.

[55] P. Felber and F. Pedone, "Probabilistic atomic broadcast," in *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.* IEEE, 2002, pp. 170–179.

[56] W. Chen and B. S. Center, "Abortable consensus and its application to probabilistic atomic broadcast," *Microsoft Research Asia, Beijing, China, Tech. Rep. MSR-TR-2006-135*, 2007.

[57] O. Babaoglu and S. Toueg, "Understanding non-blocking atomic commitment," *Distributed systems*, pp. 147–168, 1993.

[58] B. Charron-Bost, "Comparing the atomic commitment and consensus problems," *Future Directions in Distributed Computing : Research and Position Papers*, pp. 29–34, 2003.

[59] B. Charron-Bost and F. Le Fessant, "Validity conditions in agreement problems and time complexity," in *SOFSEM 2004 : Theory and Practice of Computer Science : 30th Conference on Current Trends in Theory and Practice of Computer Science Měřín, Czech Republic, January 24-30, 2004 Proceedings 30*. Springer, 2004, pp. 196–207.

[60] V. Hadzilacos, "On the relationship between the atomic commitment and consensus problems," in *Fault-Tolerant Distributed Computing.* Springer, 1990, pp. 201–208.

[61] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[62] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray, "The design and architecture of the microsoft cluster service-a practical approach to high-availability and scalability," in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224)*. IEEE, 1998, pp. 422–431.

[63] N. P. Kronenberg, H. M. Levy, and W. D. Strecker, "Vaxcluster : A closely-coupled distributed system," *ACM Transactions on Computer Systems (TOCS)*, vol. 4, no. 2, pp. 130–146, 1986.

[64] P. A. Bernstein and E. Newcomer, *Principles of transaction processing*. Morgan Kaufmann, 2009.

[65] S. D. Gribble, E. A. Brewer, and J. M. Hellerstein, "Scalable, distributed data structures for internet service construction," in *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[66] P. A. Bernstein and N. Goodman, "The failure and recovery problem for replicated databases," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983, pp. 114–122.

[67] P. A. Bernstein, V. Hadzilacos, N. Goodman *et al.*, *Concurrency control and recovery in database systems*. Addison-wesley Reading, 1987, vol. 370.

[68] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, 2005.

[69] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[70] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo : Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[71] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable : A distributed storage

system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[72] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts : Yahoo !'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[73] S. Dubois, R. Guerraoui, P. Kuznetsov, F. Petit, and P. Sens, "The weakest failure detector for eventual consistency," in *Proc. of the 2015 ACM Symposium on Principles of Distributed Computing*, 2015, pp. 375–384. [Online]. Available : https://arxiv.org/pdf/1505.03469.pdf

[74] L. Lamport, "Lower bounds for asynchronous consensus," *Distributed Computing*, vol. 19, pp. 104–125, 2006.

[75] P. Sutra and M. Shapiro, "Fast genuine generalized consensus," in *2011 IEEE 30th International Symposium on Reliable Distributed Systems*. IEEE, 2011, pp. 255–264.

[76] R. Olfati-Saber, J. A. Fax, and R. M. Murray, "Consensus and cooperation in networked multi-agent systems," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 215–233, 2007.

[77] A. Amirkhani and A. H. Barshooi, "Consensus in multi-agent systems : a review," *Artificial Intelligence Review*, vol. 55, no. 5, pp. 3897–3935, 2022.

[78] Y. Li and C. Tan, "A survey of the consensus for multi-agent systems," *Systems Science & Control Engineering*, vol. 7, no. 1, pp. 468–482, 2019.

[79] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live - an engineering perspective," in *PODC '07 : Proc. of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2006. [Online]. Available : https://www.cs.utexas.edu/users/lorenzo/corsi/cs380d/papers/paper2-1.pdf

[80] J. Rao, E. J. Shekita, and S. Tata, "Using paxos to build a scalable, consistent, and highly available datastore," *CoRR*, vol. abs/1103.2408, 2011. [Online]. Available : https://arxiv.org/pdf/1103.2408.pdf

[81] L. Lamport, "Fast paxos," *Distributed Computing 19, pages79–103(2006)*, 2006. [Online]. Available : https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-112.pdf

[82] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006. [Online]. Available : https://www.cs.cornell.edu/lorenzo/papers/Martin06Fast.pdf

[83] D. R. Dooly, S. A. Goldman, and S. D. Scott, "On-line analysis of the tcp acknowledgment delay problem," *Journal of the ACM (JACM)*, vol. 48, no. 2, pp. 243–273, 2001.

[84] J. Nagle, "Congestion control in ip/tcp internetworks," *RFC*, vol. 896, pp. 1–9, 1984. [Online]. Available : https://doi.org/10.17487/RFC0896

[85] N. John, "Congestion control in ip/tcp internetworks," *ACM SIGCOMM Computer Communication Review*, vol. 14, no. 4, pp. 11–17, 1984.

[86] M. Bienkowski, J. Byrka, M. Chrobak, N. Dobbs, T. Nowicki, M. Sviridenko, G. Świrszcz, and N. E. Young, "Approximation algorithms for the joint replenishment problem with deadlines," *Journal of Scheduling*, vol. 18, no. 6, pp. 545–560, 2015.

[87] R. Levi, R. Roundy, D. Shmoys, and M. Sviridenko, "A constant approximation algorithm for the one-warehouse multiretailer problem," *Management Science*, vol. 54, no. 4, pp. 763–776, 2008.

[88] M. Bienkowski, M. Böhm, J. Byrka, M. Chrobak, C. Dürr, L. Folwarcznỳ, Ł. Jeż, J. Sgall, N. K. Thang, and P. Veselỳ, "Online algorithms for multi-level aggregation," *arXiv preprint arXiv :1507.02378*, 2015.

[89] B. Badrinath and P. Sudame, "Gathercast : the design and implementation of a programmable aggregation mechanism for the internet," in *Proc. Ninth Int. Conference on Computer Communications and Networks (Cat.No.00EX440)*, 2000, pp. 206–213.

[90] S. Khanna, J. S. Naor, and D. Raz, "Control message aggregation in group communication protocols," in *Automata, Languages and Programming : 29th*

International Colloquium, ICALP 2002 Málaga, Spain, July 8–13, 2002 Proceedings 29. Springer, 2002, pp. 135–146.

[91] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff : Bft consensus with linearity and responsiveness," in *Proc. of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19. New York, NY, USA : Association for Computing Machinery, 2019, p. 347–356. [Online]. Available : https://doi.org/10.1145/3293611.3331591

[92] R. Friedman and R. Van Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," in *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing (Cat. No. 97TB100183)*. IEEE, 1997, pp. 233–242.

[93] N. Santos and A. Schiper, "Optimizing paxos with batching and pipelining," *Theoretical Computer Science*, vol. 496, pp. 170–183, 2013.

[94] [Online]. Available : https://www.grid5000.fr

[95] [Online]. Available : https://gitlab.lip6.fr/jlejeune/diplab

[96] A. Montresor and M. Jelasity, "PeerSim : A scalable P2P simulator," in *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, Seattle, WA, Sep. 2009, pp. 99–100.

[97] D. Wang, Y. Yu, Y. Yin, and T. C. E. Cheng, "Multi-agent scheduling problems under multitasking," *International Journal of Production Research*, vol. 59, no. 12, pp. 3633–3663, 2021.

[98] Y.-Y. Zhang, W.-C. Yang, K.-B. Kim, M.-Y. Cui, M. Xue, and M.-S. Park, "An energy-efficient multi-agent based architecture in wireless sensor network," in *Progress in WWW Research and Development : 10th Asia-Pacific Web Conference, APWeb 2008, Shenyang, China, April 26-28, 2008. Proceedings 10*. Springer, 2008, pp. 124–129.

[99] K. Saleem, L. Wang, S. Bharany, K. Ouahada, A. U. Rehman, and H. Hamam, "Intelligent multi-agent model for energy-efficient communication in wireless sensor networks," *EURASIP Journal on Information Security*, vol. 2024, no. 1, p. 9, 2024.

[100] [Online]. Available : https://www.cloudping.co/grid/p_50/timeframe/1Y