Chapitre 4
Algorithmique répartie

1. Introduction

L'algorithmique distribuée est la branche de l'algorithmique s'intéressant à la conception d'algorithmes dédiés à tout ensemble d'unités de calcul (par exemple des processeurs) indépendants, exécutant chacun leur propre code, et dont l'objectif est la résolution commune d'un problème, ou la réalisation commune d'une tâche. La frontière entre algorithmique parallèle et algorithmique distribuée est parfois floue. Il convient néanmoins d'insister sur le caractère indépendant des processeurs impliqués dans l'exécution d'un algorithme distribué. Un tel algorithme ne doit pas supposer d'unité de contrôle centrale : le contrôle est lui-même distribué. Par ailleurs, ou en conséquence de la nature distribuée du contrôle, un algorithme distribué ne doit pas se baser sur des hypothèses trop fortes relatives au degré de connaissance dont les processeurs disposent de leur environnement. L'algorithmique distribuée cherche donc moins la performance comme dans le parallélisme que savoir gérer l'incertitude liée à l'environnement. Cette incertitude est générée par de multiples causes, dont la distance pouvant séparer les processeurs dans un réseau, la présence potentielle de pannes, le degré d'asynchronisme lié à des vitesses d'exécution variables, etc.

2. Définition

L'algorithmique répartie (distribuée) est la discipline visant à développer des solutions ou des applications dédiées aux systèmes distribués et prenant en compte les spécificités de ces systèmes :

- Répartition géographique
- Communication, Concurrence, parallélisme et coopération
- Absence de mémoire et horloge commune

Le problème posé est la recherche d'algorithmes répartis performants. La performance se mesure par:

- Le nombre de messages échangés
- La charge induite sur les voies de communication
- Le temps d'attente dans les processus

Un algorithme réparti décrit le fonctionnement d'un système informatique composé de plusieurs unités de calculs reliés par un réseau de communication. Il fait intervenir plusieurs sites à l'inverse d'un algorithme centralisé. Chaque site produit des résultats et échange des données avec d'autres sites.

- L'algorithme d'un site est appelé Algorithme Local : il correspond à un algorithme séquentiel classique
- L'ensemble des algorithmes locaux constituent un algorithme réparti appelé Protocol
- Lorsque les algorithmes locaux sont identiques, l'algorithme réparti est dit Uniforme

3. Problèmes et objectifs

L'algorithme réparti cherche à résoudre les problèmes lies aux systèmes et aux applications réparties. Parmi les problèmes typiques aux systèmes répartis, on trouve :

- Recherche d'un état global consistant (cohérent): L'intérêt est de pouvoir définir des points de reprise en cas de défaillances ou encore pour prévenir des situations d' interblocage.
- Détection de terminaison d'un algorithme réparti: non¬ évident car un processus passif peut être réactivé par réception de messages
- Problème de consensus (Accord) : trouver un Protocol pour permettre à tous les processus d'être d'accord sur une valeur donnée (problème difficile surtout en présence de pannes)
- Transactions réparties : conception d'algorithmes accédant et/ou modifiant des données persistantes partageables par un ensemble de sites : une transaction est soit appliquée par tous ou annulée

Dans ce chapitre, nous présentons le problème de la coordination et l'accord (Agreement) et plus précisément nous concentrons sur le problème du consensus.
La coordination et l'accord entre processus représente une famille de problème dans l'algorithmique répartie. Cette famille de problème contient les catégories suivantes :

- Accord sur l'accès à une ressource commune : Exclusion mutuelle
- Accord sur un processus jouant un rôle particulier: Election
- Accord sur une valeur commune : Consensus
- Accord sur une action à effectuer par tous ou personne : Transaction
- Accord sur l'ordre d'envoi de messages à tous : Diffusion Atomique

4. Problème du consensus

4.1 Principe général

Dans un environnement distribue, chaque processus fait un calcul ou une mesure locale et propose son résultat à tous les autres processus qui participent à l'exécution de la même application. Les processus doivent décider d'une valeur unique commune à partir des valeurs proposées. Un processus initie la phase d'accord. La phase d'accord est lancée à des instants prédéterminés.

4.2 Conditions à valider

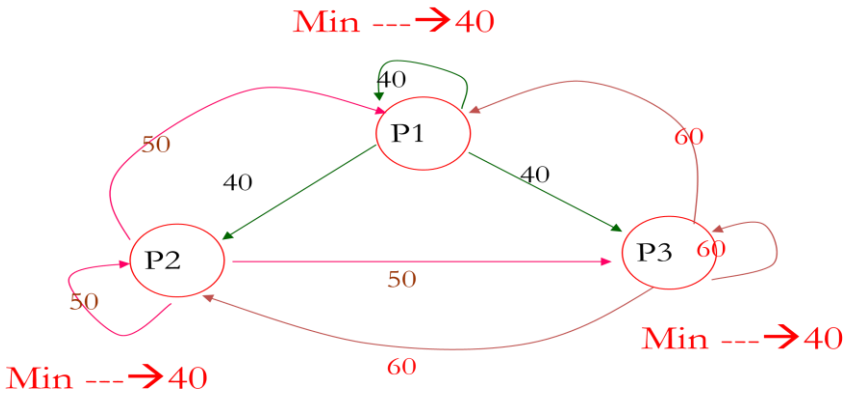
Le processus de consensus doit vérifier les conditions suivantes :

1. **Accord**: Une valeur décidée est la même pour tous les processus corrects. Deux processus corrects ne décident pas de façon différente.
2. **Intégrité**: Il n'y a pas de processus qui décide plusieurs fois. Tout processus décide au plus une fois.
3. **Validité**: la valeur décidée est une des valeurs proposées.
4. **Terminaison**: Tout processus correct décide d'une valeur au bout d'un temps fini.

5. Consensus dans différents environnements

5.1 Consensus dans un contexte sans fautes

Un processus (Pi) envoie sa valeur à tous les autres processus via une diffusion fiable. À la réception de toutes les valeurs, un processus applique une fonction donnée (ou algorithme) sur l'ensemble des valeurs. La fonction est la même pour tous les processus (fonction Min dans la figure suivante). Chaque processus est donc certain d'avoir récupéré la valeur commune qui sera la même pour tous.



Une autre solution possible permet d'utiliser un processus coordinateur qui centralise la réception des valeurs proposées et fait la décision puis la diffuse.

Ces solutions fonctionnent pour les systèmes distribués **synchrones ou asynchrones.** La différence est le temps de terminaison qui est borné ou non.

5.2 Consensus dans un contexte avec fautes

Il est existe plusieurs types de défaillance d'un processus. Un processus peut être déclaré défaillant dans cas suivants :

- Arrêt (crash failure ou panne franche) : le processus fonctionne correctement jusqu'a un point ou il cesse définitivement d'agir.
- Omission: Fautes Temporaires omission en émission : le processus omet certaines émissions qu'il aurait dû faire, ou cesse définitivement.
- Omission en réception : le processus ignore certains messages en réception, ou cesse définitivement.
- Arbitraire (byzantine failure) : le processus ment (par omission ou par contenu arbitraire des messages envoyés, fautes malicieuses volontaires).

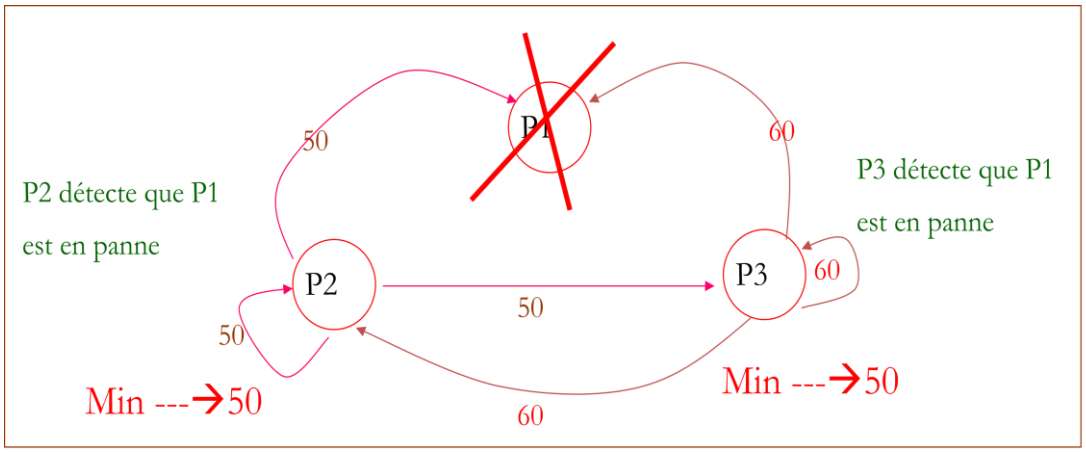
Le contexte avec fautes, nous considérons les deux cas suivants :

a. Système synchrone et pannes franches

Supposant que la communication est **fiable c.à.d** pas de perte ni de duplication de messages.

Une panne peut être déterminée si un processus n'a pas répondu, c'est-à-dire s'il est planté ou …. .

Selon la figure suivantes, deux peuvent se présentes :



- **1er cas :** P1 plante avant l'envoi de sa valeur à P2 et P3.
- **2ème cas :** P1 plante pendant l'envoi de sa valeur, un seul processus reçoit la valeur de P1

Dans les deux cas, tous les processus recevront exactement les mêmes valeurs et pourront appliquer la même fonction de choix qui aboutira forcément au même choix.

b. Système asynchrone et pannes franches

En 1983, Fischer, Lynch & Paterson (FLP) ont montré que dans un système asynchrone, même avec un seul processus fautif, il est impossible d'assurer que l'on atteindra le consensus (ne se termine pas toujours).
FLP précise qu'on n'atteindra pas toujours le consensus, mais pas qu'on ne l'atteindra jamais.

En théorie, le consensus n'est pas atteignable systématiquement en asynchrone. Mais, malgré ce résultat, il est possible en pratique d'atteindre des résultats satisfaisants en se basant sur les systèmes partiellement asynchrones et en utilisant des détecteurs de fautes.

Paxos et Lamport ont proposé (Lamport, 89) un algorithme non parfait de consensus en asynchrone, contexte de pannes franches. Son principe se résume comme suit :

- Un processus coordinateur tente d'obtenir les valeurs des processus et choisi la valeur majoritaire.
- Si le processus coordinateur se plante, d'autres processus peuvent reprendre son rôle.

5.3 Consensus dans un contexte avec fautes et pannes byzantines

a. Système synchrone

En synchrone, contrairement aux pannes franches, il n'est pas possible d'assurer que le consensus sera toujours atteint dans un contexte de fautes byzantines.
En 1982, Lamport, Shostak et Pease ont montré que « si f est le nombre de processus fautifs, il faut au minimum 3f+1 processus pour que le consensus soit assuré. En dessous il n'est pas assuré. » Le consensus ne peut être résolu que s'il y a plus de deux tiers de processus corrects.

Exemple : avec 2 processus fautifs, il faut au minimum 7 processus au total (5 corrects) pour terminer le consensus.
Les algorithmes à mettre en œuvre dans ce contexte sont relativement lourds en nombre de messages

b. Système asynchrone

En théorie, le consensus n'est pas toujours atteint (très difficile à atteindre). En pratique, quelques algorithmes spécifiques existent mais ils ne sont pas parfaits.

6. Problème des généraux byzantins

Le problème des généraux byzantins est un problème d'informatique distribuée qui a été formalisé par Leslie Lamport, Robert Shostak et Marshall Pease en 1982. Il s'agit d'une métaphore faisant intervenir des généraux qui assiègent une ville ennemie et désirent l'attaquer avec leur armée.

Dans la métaphore, l'armée est celle de l'Empire Byzantin, aussi connu comme l'Empire romain d'Orient, qui a perduré jusqu'en 1453 après sa séparation avec l'Empire d'Occident en 395, et qui s'étendait aux alentours de la Grèce et de la Turquie. Le nom de cet empire fait référence à « Byzance », le nom antique de sa capitale, renommée Constantinople par Constantin I^{er}, et qui est aujourd'hui appelée Istanbul.

D'après Leslie Lamport, les généraux devaient originellement être albanais mais il en a été décidé autrement. Dans le problème, l'armée comporte des traîtres. Il fallait donc choisir une nationalité qui ne risque pas d'offenser le sentiment patriotique du lecteur. L'Albanie, qui était un pays très fermé à l'époque, constituait un choix acceptable. Cependant, l'option byzantine était bien meilleure, car « byzantin » constituait une appellation *a posteriori* à laquelle personne ne s'identifiait profondément.

Le problème des généraux byzantins est plutôt simple à énoncer. Des généraux de l'armée byzantine campent autour d'une cité ennemie avec leurs unités et souhaitent l'attaquer. Ils ne peuvent communiquer qu'à l'aide de messagers oraux et doivent établir un plan de bataille commun. L'idée est de coordonner une attaque à un moment précis, disons à l'aube. Les généraux partagent ce qu'ils vont faire en envoyant le message « attaque » pour confirmer l'assaut, et « retraite » pour l'annuler.

Cependant, un certain nombre de ces généraux peuvent s'avérer être des traîtres qui essaient de semer la confusion au sein de l'armée. Ainsi, ils envoient le message « retraite », pour convaincre certains généraux loyaux de battre en retraite au moment de l'assaut et pour causer une défaite certaine.

Le problème est de trouver une stratégie (c'est-à-dire un algorithme) pour s'assurer que tous les généraux loyaux arrivent à se mettre d'accord sur un plan de bataille**.** Les traîtres

trahiront tout de même en battant en retraite, mais puisque leur nombre est supposé être restreint, l'attaque sera un succès.
Même en désignant des commandants auxquels des généraux subordonnés obéiront, la situation fait qu'il est très difficile de parvenir à un consensus car le commandant peut également être un traître.

La situation très imagée des généraux byzantins peut se rencontrer dans des systèmes informatiques divers qui nécessitent une certaine synchronisation. Elle est notamment problématique dans des domaines critiques, comme l'aéronautique : l'infrastructure informatique du Boeing 777 prend par exemple en compte les pannes byzantines.
Mais ce problème se rencontre surtout au sein des réseaux pair-à-pair de nœuds souhaitant se mettant d'accord sur le contenu d'un registre. Bitcoin par exemple se base sur un réseau de participants qui entretiennent chacun le registre des transactions réalisées : la fameuse chaîne de blocs**.**

Ces systèmes sont en effet soumis aux pannes byzantines (*byzantine faults*), représentées par les messages des traîtres dans le problème des généraux byzantins. Le but est donc de trouver un algorithme qui permettrait à tous les nœuds honnêtes de se mettre d'accord en présence de nœuds traîtres (ou « byzantins »), c'est-à-dire parvenir à un consensus. La propriété possédée par ce type d'algorithme s'appelle la tolérance aux pannes byzantines, ou *byzantine fault tolerance* en anglais, qui est souvent abrégée en **BFT**.

Dans le cas de Bitcoin qui est un système de transfert de valeur, l'enjeu est de se mettre d'accord sur qui possède quoi de manière distribuée, sans reposer sur une autorité centrale comme cela est fait dans le système bancaire, et sans que des nœuds malveillants ne puissent altérer le consensus.

Il a été montré que le problème des généraux byzantins peut être résolu de manière absolue si (et seulement si) les généraux loyaux représentent strictement plus des deux tiers de l'ensemble des généraux. Autrement dit, il ne peut pas y avoir plus d'un tiers de traîtres au sein de l'armée.

Avant Bitcoin, le problème était ainsi résolu par des algorithmes dits « traditionnels » qui étaient soumis à des contraintes fortes, notamment sur le nombre de nœuds pouvant interagir. Le plus connu est sans doute l'algorithme de consensus **PBFT** (pour *Practical Byzantine Fault Tolerance*), qui a été mis au point par Miguel Castro et Barbara Liskov en 1999. Il permet à un nombre donné de participants de se mettre d'accord en gérant des milliers de requêtes par seconde avec une latence de moins d'une milliseconde. Des variantes de PBFT sont encore utilisées aujourd'hui au sein de protocoles gérant des registres distribués comme **Hyperledger Sawtooth** (Sawtooth PBFT) et **Neo** (dBFT). Il existe également bien d'autres algorithmes traditionnels qui se basent sur les mêmes idées et ont les mêmes propriétés.

7. Autres algorithmes de consensus

Un algorithme de consensus ou mécanisme de consensus, abrégé parfois simplement en consensus, est un procédé par lequel les nœuds d'un réseau pair-à-pair se mettent d'accord sur un ensemble d'informations.
Dans le contexte des crypto-monnaies, un tel algorithme permet aux nœuds d'être en consensus sur le registre des transactions

7.1 Preuve de Travail, ou Proof of Work (PoW)

Le système de validation par preuve de travail a été le premier consensus existant dans le monde des crypto-monnaies, et pour cause : il est à la base de Bitcoin. La plupart des premiers crypto-actifs utilisent cette méthode car c'était un système révolutionnaire lorsque Satoshi Nakamoto a présenté ce concept en 2008 et l'a appliqué au Bitcoin. Pour résumer assez simplement, les récompenses sont distribuées aux personnes qui ont le plus contribué à la validation des transactions. *Si vous voulez plus d'informations au sujet de la preuve de travail, veuillez consulter le lien : https://cryptoast.fr/qu-est-ce-que-le-pow-proof-of-work/*

Avantages :

- Puisque la participation est ouverte, ce modèle est très robuste.
- Il est très onéreux de passer outre la sécurité de ce système lorsque le réseau est suffisamment développé.
- Tout le registre est objectivement vérifiable.

Inconvénients :

- Le système de vérification des transactions est lent.
- C'est un processus qui consomme énormément d'énergie, ce qui est un problème économique et écologique.
- Les chaînes les moins développées sont très sensibles aux attaques des 51 %.

7.2 Preuve de capacité, ou Proof of Capacity (PoC)

La preuve de capacité, aussi appelée **preuve d'espace** (*proof of space*) ou preuve de stockage (*proof of storage*), est une alternative à la preuve de travail qui se base, non pas sur la dépense énergétique des machines validatrices, mais sur leur capacité à garder en mémoire des données.

Il s'agit ni plus ni moins d'une autre forme de travail, dans le sens où, dans un tel système, les validateurs dépensent des ressources pour obtenir une récompense.

Avantages :

- La preuve de capacité possède en théorie les mêmes avantages que la preuve de travail.
- Selon ses promoteurs, cette forme de sélection serait moins énergivore que la preuve de travail.

Inconvénients :

- Tout comme la preuve de travail, la preuve de capacité ne saurait pas échapper aux attaques des 51 % en cas d'une faible participation au réseau.

7.3 Preuve d'Enjeu, ou Proof of Stake (PoS)

La preuve d'enjeu, ou *proof of stake* (PoS), est la deuxième méthode pour sélectionner les personnes participant au consensus après la preuve de travail. Le but est de réduire considérablement les dépenses énergétiques par rapport à la preuve de travail, tout en assurant une sécurité acceptable.

La preuve d'enjeu équivaut souvent dans le langage courant à la preuve de possession, c'est-à-dire à la possession du jeton natif de la blockchain considérée. Il existe néanmoins de multiples variantes dont on parlera dans la suite.

Toutefois, le système PoS possède une vulnérabilité importante : le *"nothing at stake" problem*, ou problème du « rien en jeu » (ou de l'absence d'enjeu). En effet, dans un système PoW, la validation des transactions nécessite une dépense d'énergie (et donc d'argent). Or, le système de *staking* de la preuve d'enjeu ne coûte pratiquement rien à une personne pour valider les transactions. De ce fait, dans un système naïf utilisant la preuve d'enjeu, lorsque plusieurs blockchains concurrentes sont en compétition (en cas de fork), les personnes en charge d'approuver les transactions peuvent continuer à les valider sur ces deux chaînes sans que cela n'impacte significativement leurs finances. Il en résulte alors la possibilité que deux blockchains parallèles puissent co-exister et se parasiter mutuellement.

Cela pourrait permettre, entre autres, à un forgeur de créer une chaîne parallèle dans laquelle il s'attribue des jetons, et faire en sorte que cette chaîne survive puisque les autres forgeurs travailleraient également sur sa chaîne par simple souci de maximisation des profits.

Ce problème du « rien en jeu » rajoute de la complexité aux algorithmes utilisant la preuve d'enjeu. C'est notamment pour cette raison qu'Ethereum n'est pas encore passé à la preuve d'enjeu (prévue pour novembre 2020).

Pour plus d'informations à propos de la preuve d'enjeu, veuillez consulter ce lien *:*
https://cryptoast.fr/qu-est-ce-que-pos-proof-of-stake/

Avantages :

- La consommation d'énergie est maîtrisée, ce qui assure la bonne santé économique de la cryptomonnaie (moins de frais ou moins d'inflation).
- Plus écologique par rapport à la preuve de travail.
- Résistance forte aux attaques des 51 %.

Inconvénients :

- Le problème du « rien en jeu », qui rend les algorithmes de PoS plus complexes.
- La subjectivité de la chaîne.

7.4 Preuve de Conservation, ou Proof of Hold (PoH)

La preuve de conservation, ou *proof of hold* (PoH), est une variante de la preuve d'enjeu qui se base sur le montant des jetons qu'une personne possède multiplié par le temps où ces jetons n'ont pas bougé. Cette métrique s'appelle l'âge des pièces (*coin age*).

Le protocole Peercoin (anciennement PPCoin) implémente cette méthode de manière hybride, en la combinant avec de la preuve de travail.

Avantages :

- Permet une implémentation simple de la preuve d'enjeu.
- Incite à la conservation.

Inconvénients :

- Peut créer un vecteur d'attaque lié aux pièces qui n'ont pas bougé depuis le lancement de la crypto-monnaie.