Consensus et détecteurs de fautes

Élise Jeanneau
elise.jeanneau@univ-lyon1.fr
Adapté du cours de Pierre Sens

1 Algorithmes distribués tolérants aux fautes

Les précédents cours ont présenté des systèmes distribués dans lesquels les processus sont fiables. Cette hypothèse est peu réaliste : dans le monde réel, les pannes sont inévitables. Dans ce cours, nos modèles de systèmes distribués tiennent compte des défaillances des processus (on parle de faute, ou de panne d'un processus). Un algorithme distribué tolérant aux fautes doit accomplir son but malgré les fautes.

1.1 Modèles de fautes

Il existe différents modèles de fautes. Voici quelques uns des plus classiques.

- **Pannes franches** : certains processus tombent en panne une fois pour toutes. Une fois en panne, ils n'envoient plus aucun message jusqu'à la fin de l'exécution.
- **Pannes temporaires** : similaire au modèle à pannes franches, mais les processus peuvent se remettre de leurs pannes et reprendre un fonctionnement normal. Plusieurs variantes de ce modèle :
  - certains modèles considèrent que chaque processus qui tombent en panne récupère plus tard ;
  - d'autres considèrent qu'un processus en panne peut récupérer ou non. Le modèle à pannes franches est un cas particulier de ce modèle dans lequel aucun processus ne récupère.
- **Fautes d'omission** : certains messages sont perdus. Le modèle à pannes temporaires est un cas particulier de ce modèle : un processus dont tous les messages sont perdus pendant une période peut être vu comme « en panne » pendant cette période.
- **Fautes Byzantines** : certains processus malveillants n'exécutent pas l'algorithme et cherchent à mettre des bâtons dans les roues des autres. Par exemple en envoyant des messages contenant des informations erronées, ou en se faisant passer pour un autre processus. Les autres modèles ci-dessus sont des cas particuliers du modèle Byzantin : un processus malveillant peut faire semblant d'être en panne ou cesser d'envoyer des messages.

Le modèle Byzantin est donc le plus générique, et c'est dans celui-ci qu'il est le plus difficile de résoudre des problèmes distribués.

Dans ce cours, nous étudierons uniquement le modèle à pannes franches. Toutes les fautes sont donc des pannes, et nous utiliserons ces deux termes indifféremment.

Dans le modèle à pannes franches, chaque processus est soit un processus correct, soit un processus fautif.

- Un **processus correct** ne tombe jamais en panne au cours de l'exécution.
- Un **processus fautif** tombe en panne à partir d'un certain moment, et ne récupère jamais.

À noter qu'un processus qui tombe en panne est dit « fautif » pendant toute l'exécution, même avant de tomber en panne. Du point de vue des processus, il est impossible de distinguer les fautifs des corrects avant que les pannes n'aient eu lieu.

2 Exemple applicatif : bases de données réparties

Cette partie présente un exemple applicatif afin de démontrer l'importance du consensus.

Les sites web avec un fort trafic mondial s'appuient sur des bases de données à grande échelle. Il existe deux stratégies pour structurer ces bases de données.

Dans le **modèle centralisé**, une unique base de données sert le monde entier. Cette solution présente des limites importantes. Les utilisateurs éloignés souffrent d'une forte latence, la congestion réseau est très forte autour de l'unique noeud, et le site est vulnérable à la panne d'un seul noeud.

La solution consiste à utiliser le **modèle distribué**. Il s'agit alors de construire une base de données répartie. Plusieurs réplicas de la base de données sont alors répartis dans le monde. Chaque utilisateur s'adresse alors au réplica le plus proche de lui. Les mises à jour faites par les utilisateurs sont diffusées à tous les réplicas, afin de maintenir un état cohérent de la base de données. Cependant, maintenir cette cohérence des données n'est pas trivial.

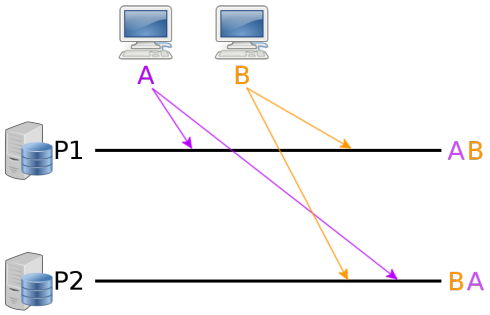

FIGURE 1 – Le problème de la cohérence des données.

Dans l'exemple de la Figure 1, deux réplicas P1 et P2 reçoivent des mises à jour envoyées par deux utilisateurs A et B. Cependant, les deux réplicas reçoivent ces mises à jours dans un ordre différent : P1 effectue les mises à jour dans l'ordre A puis B, alors que P2 effectue d'abord la mise à jour de B, puis celle de A. Les deux réplicas finissent donc dans des états différents.

La solution consiste à retarder l'exécution des mises à jour jusqu'à ce que P1 et P2 aient pu se mettre d'accord sur l'ordre dans lequel ces mises à jour seront appliquées. Pour cela, il est nécessaire de résoudre le problème du consensus.

3 Consensus

3.1 Définition du problème

Le consensus est l'un des problèmes classiques de l'algorithmique distribuée.

Le problème survient dans un système distribué dans lequel chaque processus propose initialement une valeur. Le but est que tous les processus se mettent d'accord sur une même valeur parmi les valeurs proposées. On dit que les processus « décident » une valeur.

Pour que le problème soit résolu, les propriétés suivantes doivent êtres vérifiées :

1. **Validité**: la valeur décidée doit être l'une des valeurs proposées.
2. **Terminaison**: tous les processus corrects décident en un temps fini.
3. **Cohérence**: deux processus corrects ne peuvent décider différemment.
4. **Intégrité**: un processus doit décider au plus une fois.

La Figure 2 présente un système distribué constitué de cinq processus (en noir) qui résolvent le problème du consensus. Les valeurs proposées sont indiquées en bleu, les valeurs décidées sont en rouge.

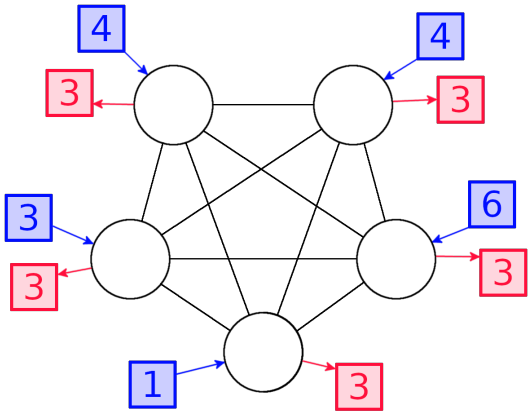

FIGURE 2 – Un système distribué résout le problème du consensus.

Les Figures 3, 4 et 5 présentent des contre-exemples dans lesquels le système échoue à résoudre le consensus.

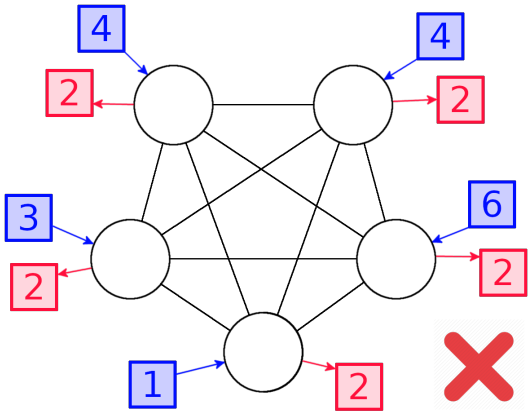

FIGURE 3 – Cet exemple viole la propriété de validité.

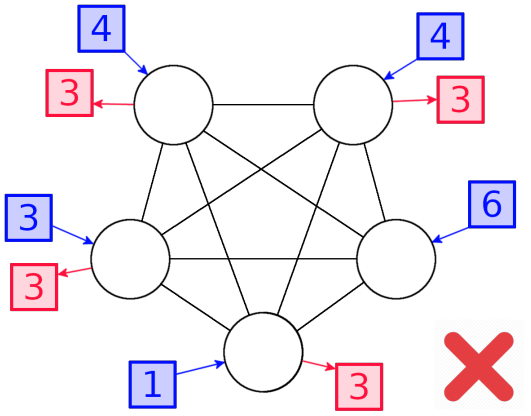

FIGURE 4 – Cet exemple viole la propriété de terminaison.

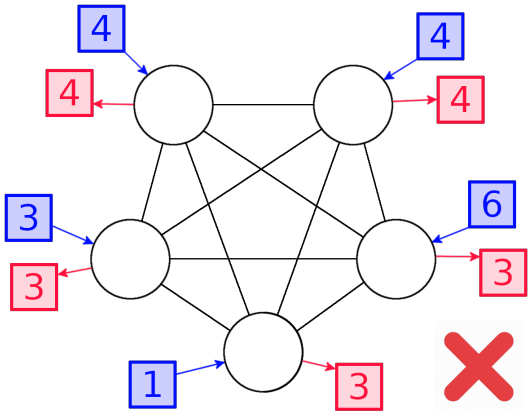

FIGURE 5 – Cet exemple viole la propriété de cohérence.

3.2 Consensus uniforme

L consensus uniforme est une version plus difficile du consensus dans laquelle la propriété d'accord s'applique à tous les processus, même fautifs.

1. **Validité**: la valeur décidée doit être l'une des valeurs proposées.
2. **Terminaison**: tous les processus corrects décident en un temps fini.
3. **Cohérence uniforme**: deux processus, **même fautifs**, ne peuvent décider différemment.
4. **Intégrité**: un processus doit décider au plus une fois.

Dans l'exemple de la Figure 6, un processus fautif (marqué par une croix) décide différemment des autres avant de tomber en panne. Cet exemple d'exécution est donc valide si l'on cherche à résoudre le consensus non-uniforme, mais invalide si l'on cherche à résoudre le consensus uniforme.

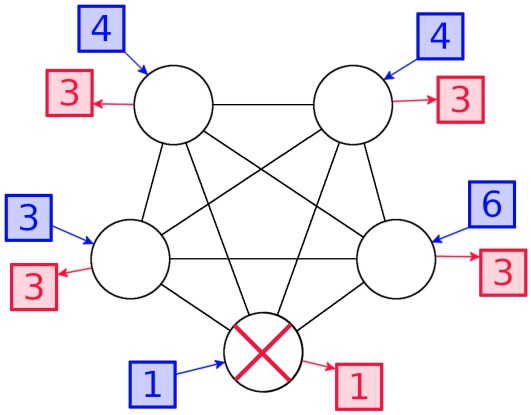

FIGURE 6 – Le système résout le consensus non-uniforme, mais pas le consensus uniforme.

4 Modèle de système distribué

Dans le reste de ce cours, nous utiliserons les hypothèses suivantes.

— Le graphe de communication est **complet**, c'est à dire que chaque processus est capable d'envoyer un message directement à chaque autre processus. Les solutions discutées ici peuvent être employées dans un graphe connexe, ce qui nécessiterait alors de faire appel à des mécanismes de retransmission qui ne sont pas le sujet de ce cours.
— Les liens de communications sont bidirectionnels et **fiables**, c'est à dire que les messages envoyés sont toujours reçus correctement.
— Le système est constitué de n noeuds statiques, c'est à dire que les noeuds sont présents dès le début de l'exécution et ne quittent pas le système, sauf s'ils tombent en panne. Les processus connaissent le nombre n.
— Chaque noeud a un **identifiant unique connu**.
— Certains processus sont sujets à des **pannes franches**.

5 Algorithme de consensus naïf

Voici un algorithme de consensus très simple.

1. Chaque processus envoie sa valeur proposée à tous les autres ;
2. chaque processus attend de recevoir les valeurs envoyées par tous les autres ;
3. chaque processus décide la valeur la plus faible parmi toutes les valeurs connues.

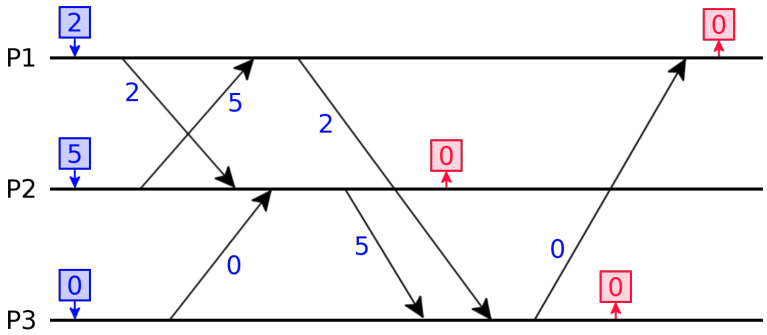

FIGURE 7 – Un exemple d'exécution de l'algorithme naïf.

La Figure 7 présente un exemple d'exécution de l'algorithme ci dessus. Les processus P1, P2 et P3 proposent les valeurs 2, 5 et 0, respectivement. Finalement, tous décident la valeur 0.

Cet algorithme très simple fonctionne **uniquement en l'absence de pannes**.

6 Modèles temporels

Les deux modèles temporels classiques sont le **modèle synchrone** et le **modèle asynchrone**.

6.1 Le modèle synchrone

Dans le modèle synchrone, il existe une borne, nommée Δ, sur le délai de propagation des messages (latence). On dit que **les communications sont synchrones**.

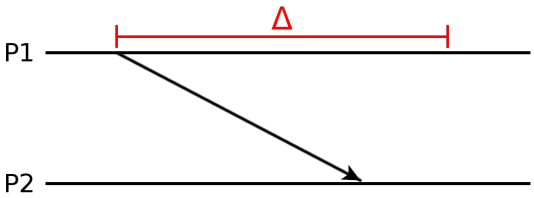

FIGURE 8 – Transmission d'un message dans le modèle synchrone.

D'autre part, il existe une borne, nommée Φ, sur la vitesse relative des processus. C'est à dire que si un processus prend x instants pour effectuer un calcul local, alors aucun autre processus ne prend plus de $x * \Phi$ instants pour effectuer ce même calcul. On dit que les **processus sont synchrones**.

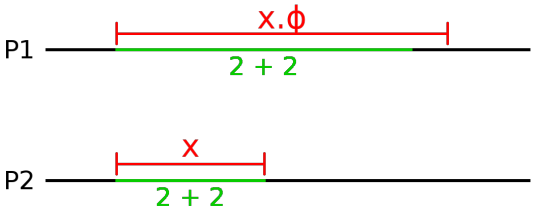

FIGURE 9 – Processus synchrones.

Dans l'exemple de la Figure 9, il faut x instants au processus P2 pour calculer le résultat de l'opération «2+2». Par conséquent, aucun autre processus (y compris P1) ne peut prendre plus de $x * \Phi$ instant pour calculer «2+2».

Intuitivement, la synchronie des processus signifie que tous les processus sont à peu près aussi rapides.

Dans le modèle synchrone, les processus connaissent les bornes Δ et Φ, ce qui permet une **détection parfaite des fautes**. En effet, si un processus envoie un message puis attend $2\Delta + x.\Phi$ instants (où x est le temps qu'il faut au processus pour envoyer une réponse à un message reçu), il est possible de déterminer à coup sûr si le destinataire est en panne ou non. Si aucune réponse n'est reçue après cet intervalle de temps, il y a nécessairement une panne.

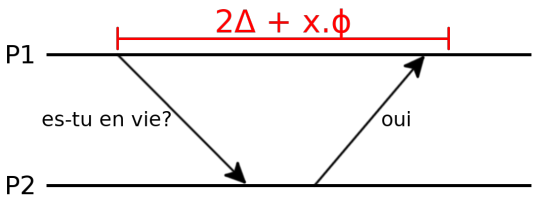

FIGURE 10 – P1 s'assure que P2 n'est pas en panne.

Bien que le modèle synchrone soit très utile, il est peu réaliste en pratique.

6.2 Le modèle asynchrone

Dans le modèle asynchrone, il n'y a ni borne sur le délai de propagation des messages, ni borne sur la vitesse relative des processus. Il est donc impossible de prévoir combien de temps un processus mettra à répondre à un message. Par conséquent, il est impossible de faire la différence entre un processus en panne, et un processus qui met simplement du temps à répondre.

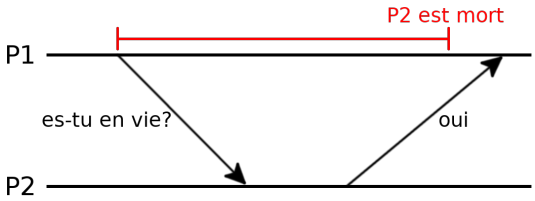

FIGURE 11 – P1 ne peut pas détecter de façon fiable l'état de P2.

Dans la Figure 11, quelque soit le temps (en rouge) que P1 choisit d'attendre avant de déclarer que P2 est en panne, il se peut que la réponse de P2 soit reçue après cette durée.

7 L'impossibilité de Fischer, Lynch et Paterson

7.1 Intuition du problème

Nous cherchons à résoudre un consensus dans un système asynchrone, en présence de pannes franches.

Considérons deux algorithmes :

- Dans l'**Algorithme A**, chaque processus attend un message de chaque autre processus avant de décider.
- Dans l'**Algorithme B**, chaque processus attend un certain temps, puis décide l'une des valeurs connues, même s'il n'a pas reçu d'information de tous les processus.

Soit un système distribué constitué de deux processus P1 et P2. Considérons également deux scénarios d'exécution :

- Dans le **Scénario 1**, P2 tombe en panne immédiatement au début de l'exécution.
- Dans le **Scénario 2**, personne ne tombe en panne, mais P2 est particulièrement lent à répondre (latence et/ou vitesse du processus).

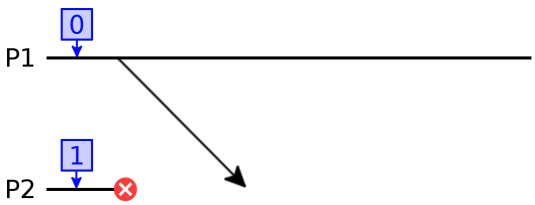

FIGURE 12 – Scénario 1, algorithme A : violation de la propriété de terminaison.

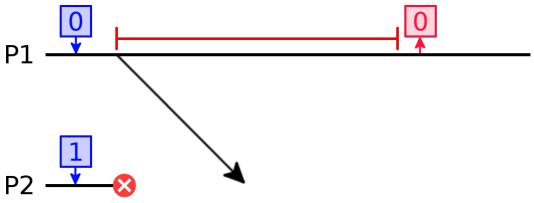

FIGURE 13 – Scénario 1, algorithme B : consensus résolu.

Dans le scénario 1, l'algorithme B permet de résoudre le consensus (Figure 13), mais l'algorithme A viole la propriété de terminaison puisque P1 attend perpétuellement le message de P2 et ne décide jamais (Figure 12).

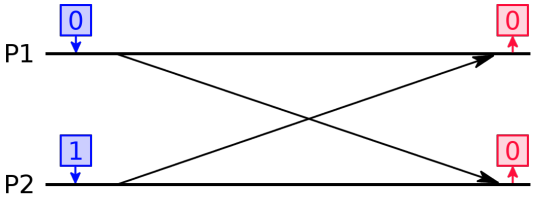

FIGURE 14 – Scénario 2, algorithme A : consensus résolu.

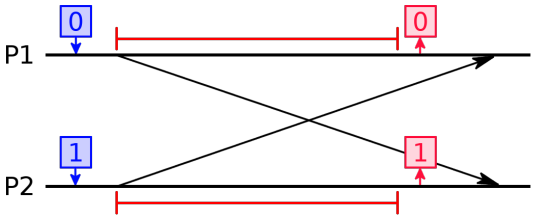

FIGURE 15 – Scénario 2, algorithme B : violation de la propriété de cohérence.

Dans le scénario 2, l'algorithme A permet de résoudre le consensus (Figure 14), mais l'algorithme B viole la propriété de cohérence puisque les deux processus décident séparément sans attendre de communiquer (Figure 15).

Rappelons que dans un système asynchrone, il est impossible pour P1 de faire la différence entre les scénarios 1 et 2 ! En effet, P1 ne peut jamais savoir avec certitude si P2 est en panne ou simplement lent. Il est donc impossible de choisir entre les algorithmes A et B. Il n'y a pas de solution déterministe.

7.2 Résultat d'impossibilité

Dans un article de recherche de 1985, Fischer, Lynch et Paterson [FLP85] ont démontré qu'**il est impossible de résoudre le consensus de façon déterministe dans un système asynchrone en présence de ne serait-ce qu'une seule panne**.

Depuis ce résultat, de nombreux chercheurs ont proposé des compromis permettant de contourner cette impossibilité. Beaucoup de ces compromis peuvent se classer parmi les catégories suivantes :

- **Changer le problème.** Puisque le consensus est trop difficile, on peut plutôt chercher à résoudre par exemple le k-accord, une famille de problèmes généralisant le consensus et qui peuvent être résolus en présence de partitions dans le système.
- **Changer le modèle.** Puisque le consensus ne peut pas être résolu dans un système asynchrone, on peut chercher à le résoudre dans d'autres modèles moins réalistes, tel que le modèle partiellement synchrone.
- **S'abstraire du modèle.** Grâce aux détecteurs de fautes, on peut faire abstraction de l'hypothèse de synchronie en s'intéressant uniquement à l'information sur les fautes fournie aux processus.
- **Résoudre un consensus «imparfait».** Bien qu'une solution déterministe soit impossible, certains algorithmes permettent de résoudre un consensus dans lequel la propriété de terminaison n'est assurée que de façon probabiliste. Certains processus peuvent alors ne pas terminer.

La suite de ce cours va présenter brièvement ces différentes méthodes, en s'attardant particulièrement sur l'approche des détecteurs de fautes.

8 Le problème du k-accord

Le problème du k-accord [Cha93] est une généralisation du consensus dans lequel plusieurs valeurs différentes peuvent être décidées. Le paramètre k détermine le nombre maximal de valeurs qui peuvent être décidées.

Plus formellement, les propriétés suivantes doivent être vérifiées. Seule la propriété de cohérence est différente du consensus.

1. **Validité**: la valeur décidée doit être l'une des valeurs proposées.
2. **Terminaison**: tous les processus corrects décident en un temps fini.
3. **Cohérence**: au plus k valeurs différentes peuvent être décidées.
4. **Intégrité**: un processus doit décider au plus une fois.

À noter que pour $k = 1$, le k-accord est identique au consensus. Plus la valeur de k augmente, plus le problème devient facile. Contrairement au consensus, le k-accord pour $k > 1$ peut être résolu même si le système est séparé en plusieurs partitions.

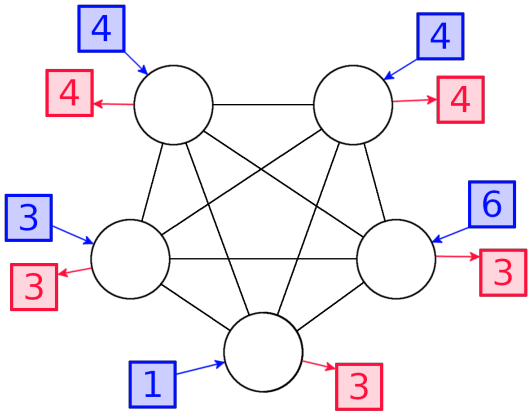

FIGURE 16 – Dans cet exemple, le 2-accord est résolu, mais pas le 1-accord.

9 Le modèle partiellement synchrone

Bien que le consensus soit impossible à résoudre dans un système asynchrone, il peut être résolu dans un système partiellement synchrone [DLS88]. Les systèmes partiellement synchrones sont une famille de modèles de systèmes qui sont plus réalistes que le modèle synchrone, tout en permettant la résolution du consensus.

Il existe plusieurs variantes de modèles partiellement synchrones, dont la plupart se retrouvent dans l'une de ces deux descriptions :

— dans certains modèles, les bornes Δ et Φ sont connues mais ne s'appliquent qu'après un certain temps ;
— dans d'autres modèles, les bornes Δ et Φ sont inconnues, mais s'appliquent dès le début de l'exécution.

Dans le second cas, les processus s'appuient sur une estimation de la valeur $2\Delta + x.\Phi$, qui est revue à la hausse à chaque fois qu'une réponse d'un processus est reçue «en retard». À terme, cette estimation finira par dépasser la valeur réelle et deviendra donc fiable.

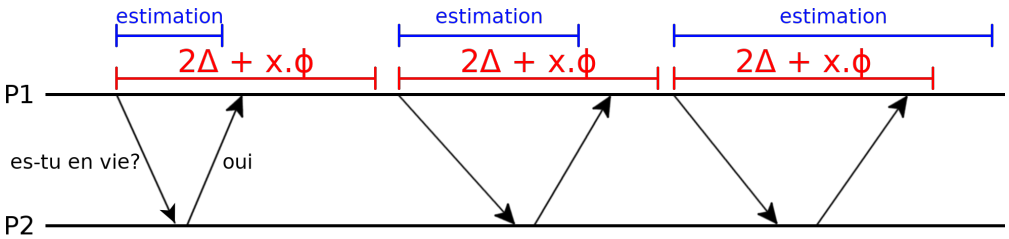

FIGURE 17 – L'estimation de $2\Delta + x.\Phi$ par P1 s'améliore au fil du temps.

Toutes les variantes de modèles partiellement synchrones reviennent globalement au même : au début de l'exécution les processus se trompent en détectant les fautes (période instable), mais à terme la détection des fautes devient aussi parfaite que dans un système synchrone (période stable).

10 Détecteurs de fautes

Les détecteurs de fautes [CT96] sont des outils qui fournissent à chaque processus des informations sur les fautes des autres processus dans le système. Cette information prend la forme d'une liste de processus «de confiance». Les processus sur cette liste sont présumés corrects, tandis que les processus absents de la liste sont présumés en panne.

On dit qu'un processus P1 **fait confiance** à un processus P2 si P2 est présent dans la liste de confiance fournie par le détecteur à P1.

Inversement, on dit que P1 **suspecte** P2 si P2 n'est pas présent dans la liste de confiance fournie par le détecteur à P1.

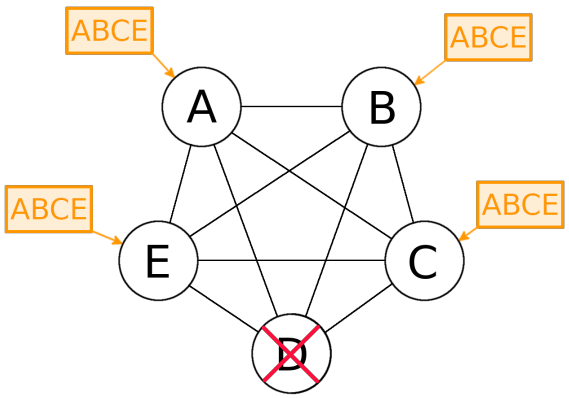

FIGURE 18 – A,B,C et E suspectent D, qui est en panne.

La Figure 18 présente un exemple dans lequel le détecteur de faute fournit à chaque processus correct une liste de confiance (en orange). Cette liste indique que A,B,C et E se font confiance, mais suspectent D.

Les informations fournies par le détecteur ne sont **pas fiables**. Un processus peut faire confiance à un processus au panne, ou bien suspecter un processus correct (on parle de **fausse suspicion**).

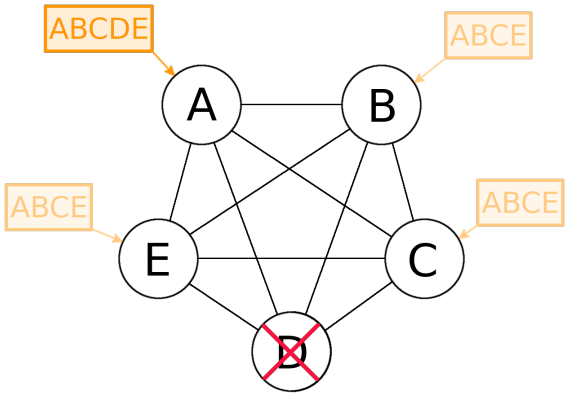

FIGURE 19 – A n'a pas encore détecté la panne de D.

Dans la Figure 19, D est en panne mais n'est pas suspecté par A. Ce cas de figure se produit inévitablement, puisqu'il est impossible pour A de détecter instantanément la panne d'un autre processus : il y a toujours un retard dans la détection des fautes.

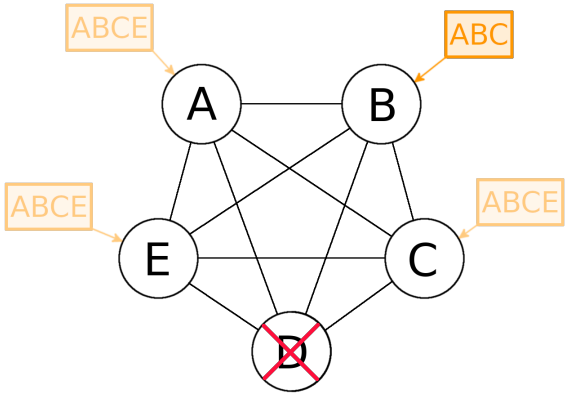

FIGURE 20 – B suspecte E, à tort.

Dans la Figure 20, E est correct mais est suspecté à tort par B.

Dans un système asynchrone, les fausses suspicions sont inévitables : comme vu dans la Section 6.2, il est impossible de faire la différence entre un processus en panne et un processus lent.

En revanche, dans un système synchrone, il est possible d'éviter complètement les fausses suspicions.

Dans un système partiellement synchrone, les fausses suspicions sont inévitables en période instable (au début de l'exécution) puis évitables une fois que la période stable est atteinte (le système se comporte alors comme un système synchrone).

Il existe différentes classes de détecteurs de fautes, catégorisées en fonction de la fiabilité des informations qu'ils fournissent. Les critères les plus importants pour les différencier sont les propriétés de complétude et justesse.

10.1 Propriétés de complétude

Intuitivement, la complétude exige que les corrects détectent les fautifs. On considère deux variantes de la propriété de complétude.

- **Complétude forte**: après un certain temps, tous les fautifs sont suspectés par tous les corrects.
- **Complétude faible**: après un certain temps, tous les fautifs sont suspectés par un correct.

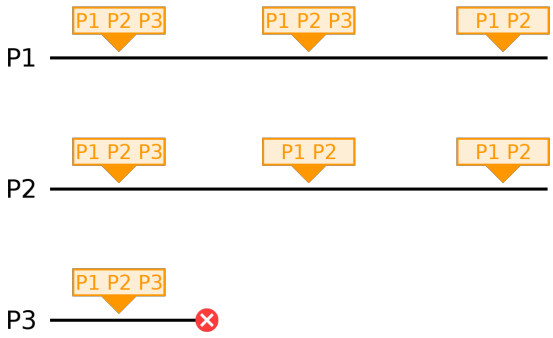

FIGURE 21 – Cette exécution vérifie la complétude forte.

La Figure 21 présente trois processus, dont l'un tombe en panne (P3). À terme, P1 et P2 suspectent tous les deux P3, la propriété de complétude forte est donc vérifiée.

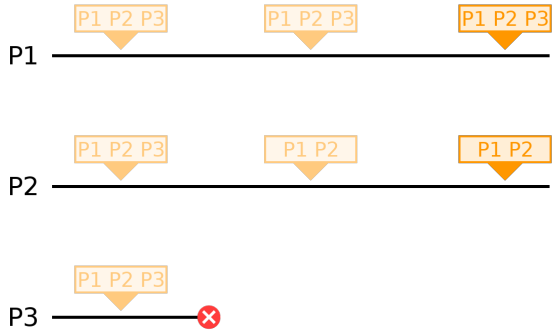

FIGURE 22 – Cette exécution vérifie la complétude faible, mais pas la complétude forte.

Dans la Figure 22, P1 ne suspecte cette fois-ci jamais P3 : la complétude forte n'est donc pas vérifiée. En revanche, P2 suspecte bien P3, et la complétude faible est donc bien vérifiée.

Il est en réalité toujours possible de construire la complétude forte, pour peu que la complétude faible soit vérifiée. En effet, il suffit que chaque processus informe les autres de ses suspicions pour que tous les corrects suspectent tous les fautifs.

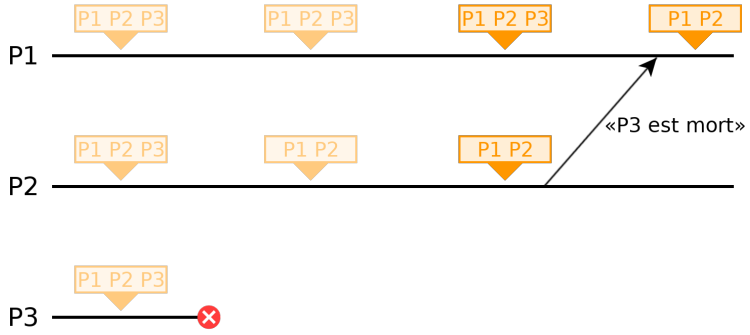

FIGURE 23 – D'un simple envoi de message, P2 transforme la complétude faible en complétude forte.

Les complétude forte et complétude faible sont donc équivalentes. Par soucis de simplicité, on ne considère en général que la complétude forte.

10.2 Propriétés de justesse

Intuitivement, la justesse interdit les fausses suspicions. On considère quatre variantes de la propriété de justesse.

- **Justesse forte**: les corrects ne sont jamais suspectés.
- **Justesse faible**: un correct n'est jamais suspecté.
- **Justesse finalement forte**: après un certain temps, les corrects ne sont jamais suspectés.
- **Justesse finalement faible**: après un certain temps, un correct n'est jamais suspecté.

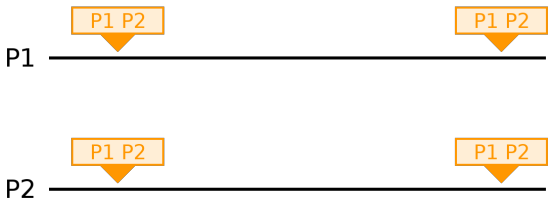

FIGURE 24 – Aucune fausse suspicion : la justesse forte est vérifiée.

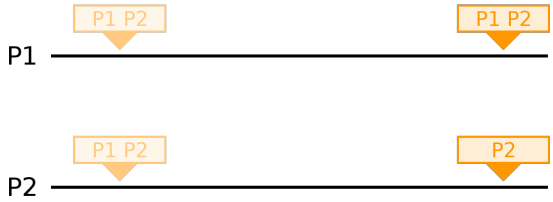

FIGURE 25 – P2 n'est jamais suspecté : la justesse faible est vérifiée, mais pas la justesse forte.

Les deux propriétés de justesse finalement forte et justesse finalement faible sont similaires aux deux premières, mais elles ne doivent pas nécessairement s'appliquer dès le début de l'exécution. Des fausses suspicions peuvent donc avoir lieu pendant une période d'instabilité.

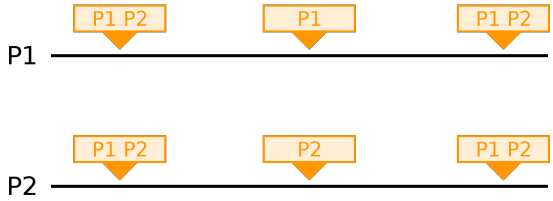

FIGURE 26 – P1 et P2 se suspectent temporairement, mais se font confiance à terme : la justesse finalement forte est vérifiée.

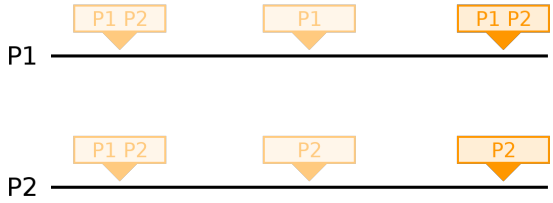

FIGURE 27 – P1 suspecte temporairement P2, mais lui fait confiance à terme : la justesse finalement faible est vérifiée.

10.3 Classes de détecteurs de fautes

En combinant les différentes variantes des propriétés de complétude et de justesse, on obtient différentes classes de détecteurs de fautes.

	Justesse			
	Forte	Faible	Finalement forte	Finalement faible
Complétude forte	P	S	$\diamond P$	$\diamond S$
Complétude faible	Q	W	$\diamond Q$	$\diamond W$

FIGURE 28 – Ce tableau présente les différentes classes de détecteurs, et les propriétés qui les définissent.

Étant donnée l'équivalence entre complétude forte et complétude faible, on ignore généralement la seconde ligne du tableau de la Figure 28.

Les deux détecteurs les plus importants sont :
- le détecteur P (aussi appelé le détecteur parfait), qui n'autorise aucune fausse suspicion ;
- le détecteur $\diamond S$ (prononcé «diamant S»), qui ne fournit de garanties qu'à terme.

Ces différents détecteurs de fautes peuvent être comparés entre eux.

10.4 Notion de force des détecteurs

On dit qu'un détecteur D1 est **plus fort** qu'un détecteur D2, si et seulement si il existe un algorithme qui permet de construire D2 en s'appuyant uniquement sur les informations fournies par D1. Note : cette relation est transitive.

Par exemple, le détecteur Q est plus fort que P. En effet, si l'on se munit d'un détecteur Q, il est possible d'implémenter un détecteur P (chaque processus informe les autres de ses suspicions, comme vu dans la Figure 23).

Cette notion de détecteur plus fort est importante, puisqu'elle nous informe que tout problème qui peut être résolu avec un détecteur P peut également être résolu avec un détecteur Q.

Si deux détecteurs sont mutuellement plus forts l'un que l'autre, on dit qu'ils sont **équivalents**.

C'est le cas de P et Q. P est trivialement plus fort que Q, puisque toute exécution qui vérifie la complétude forte vérifie également la complétude faible.

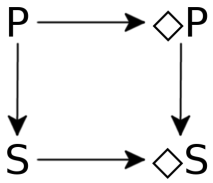

FIGURE 29 – Comparaison des différentes classes de détecteurs.

La Figure 29 présente une comparaison de la force des détecteurs de fautes $P, S, \diamond P$ et $\diamond S$. Une flèche signifie «est plus fort que». P est plus fort que les trois autres, et $\diamond S$ est plus faible que les trois autres. La force de S et $\diamond P$ n'est pas comparable.

Soit $\mathcal{P}$ un problème distribué. On dit que le détecteur D est **suffisant** pour résoudre $\mathcal{P}$ si et seulement si il existe un algorithme qui résout le problème $\mathcal{P}$ en s'appuyant uniquement sur l'information fournie par D.

On dit que D est le **détecteur de fautes minimal** pour résoudre $\mathcal{P}$ si et seulement si :
- D est suffisant pour résoudre $\mathcal{P}$
  et
- Tous les autres détecteurs suffisants pour résoudre $\mathcal{P}$ sont plus forts que D.

Il existe donc un algorithme qui construit D à partir de $\mathcal{P}$, et vice-versa. Tout système distribué dans lequel $\mathcal{P}$ peut être résolu peut également être muni d'un détecteur D, et vice versa. Intuitivement, on peut dire que D est nécessaire et suffisant pour résoudre $\mathcal{P}$.

Dans un système distribué comportant une majorité de processus corrects, $\diamond S$ **est le détecteur de fautes minimal pour résoudre le consensus**. Cela implique que P, S et $\diamond P$ sont suffisants pour résoudre le consensus.

Cela implique également qu'en vertu de l'impossibilité de Fischer, Lynch et Paterson, il est impossible d'implémenter $\diamond S, P, S$ ou $\diamond P$ dans un système asynchrone.

10.5 Le détecteur de leader Ω

Le détecteur Ω [CHT96] est un peu différent des détecteurs précédemment vus : il ne fournit pas une liste de processus de confiance, mais seulement l'identité d'un unique processus (le leader).

Ω garantit qu'à terme, le détecteur indique l'identité d'un même processus correct à tous les processus.

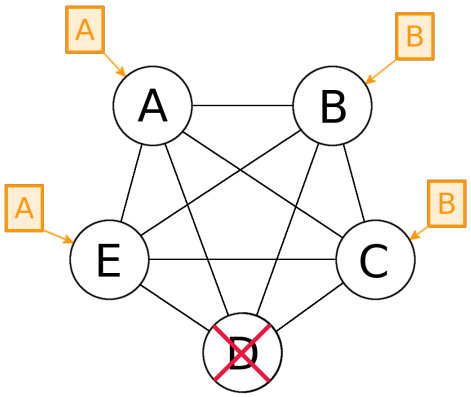

FIGURE 30 – Au début de l'exécution, l'information fournie par Ω est arbitraire.

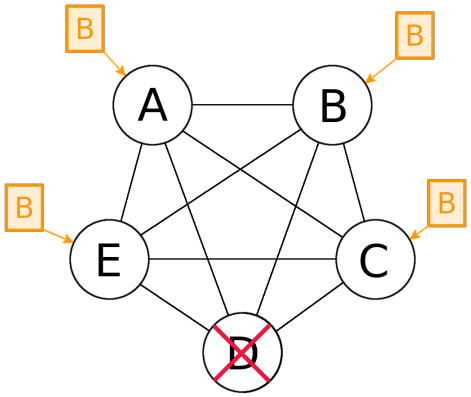

FIGURE 31 – À terme, Ω indique l'identité d'un même correct à tous les processus.

Attention : les processus ne sont pas informés du moment à partir duquel le leader ne change plus. Il ne savent donc jamais si leur leader actuel est fiable ou non.

Ω est équivalent à $\diamond S$, il s'agit donc du détecteur minimal pour résoudre le consensus avec une majorité de corrects. La plupart des travaux récents font référence à Ω plutôt qu'à $\diamond S$.

10.6 Le détecteur de quorums Σ

Le détecteur Σ [DFG10] fournit à chaque processus une liste de confiance, en respectant les propriétés suivantes.

— **Complétude forte**: après un certain temps, tous les fautifs sont suspectés par tous les corrects.
— **Intersection**: les listes fournies par Σ s'intersectent deux à deux.

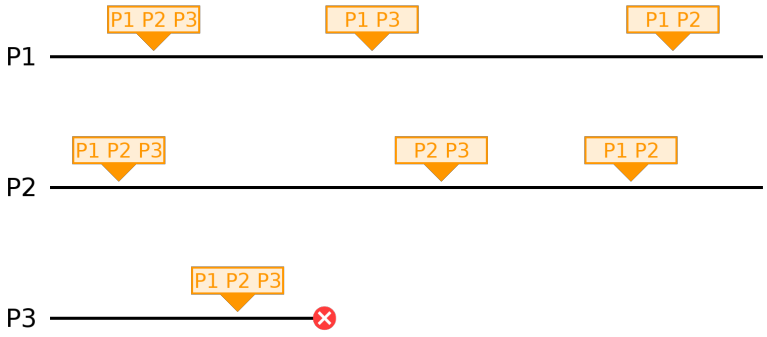

FIGURE 32 – Cette exécution vérifie les propriétés de Σ.

Attention, la propriété d'intersection s'applique à toutes les paires de listes émises par Σ. Cela vaut pour les listes émises pour deux processus différents, et aussi pour les listes émises à différents moments. Dans l'exemple de la Figure 32, tous les cadres oranges doivent s'intersecter deux à deux.

On appelle (Ω, Σ) le détecteur de fautes qui combine Ω et Σ. Ce détecteur fournit deux outputs, l'un vérifiant les propriétés de Ω, l'autre celles de Σ.

(Ω, Σ) est le détecteur minimal pour résoudre le consensus sans l'hypothèse d'une majorité de corrects.

10.7 Résumé : détecteurs de fautes minimaux

	Consensus	k-accord	Exclusion mutuelle
Majorité de corrects	Ω	$\overline{\Omega}_k$	$\mathcal{T}$
Jusqu'à $n-1$ fautes	(Ω, Σ)	?	$(\mathcal{T}, \Sigma^l)$

FIGURE 33 – Les détecteurs de fautes minimaux pour le consensus, le k-accord et l'exclusion mutuelle.

La Figure 33 présente les détecteurs de fautes minimaux pour quelques problèmes distribués.

La première ligne indique le détecteur minimal dans l'hypothèse d'une majorité de processus corrects. La seconde ligne indique le détecteur minimal avec un nombre de fautes arbitraire ($n-1$ étant le plus grand nombre de fautes pertinent, puisqu'il est évident qu'on ne peut rien faire si les n processus sont tous en panne).

Les détecteurs pour le k-accord et l'exclusion mutuelle sont donnés à titre indicatif, afin d'illustrer le fait que les détecteurs de fautes peuvent être utilisés pour d'autres problèmes que le consensus. Le détecteur minimal pour le k-accord sans hypothèse sur le nombre de corrects est actuellement inconnu.

10.8 Algorithme de consensus avec détecteur P en n rondes

Ce premier algorithme résout le consensus en s'appuyant sur un détecteur P (détecteur parfait).

Pour rappel, P interdit toute fausse suspicion. Les processus ne sont donc suspectés que lorsqu'ils sont réellement en panne.

L'algorithme fonctionne à l'aide de rondes asynchrones. À chaque ronde, un leader différent orchestre l'algorithme. Le leader est le processus dont l'identifiant est égal au numéro de ronde : chaque processus prend donc son tour comme leader. L'algorithme se termine une fois que tout le monde a été leader, au bout de n rondes.

Chaque processus dispose d'une variable locale v. Initialement, chaque processus affecte sa valeur proposée à v.

À chaque ronde, le leader décide sa valeur v, puis la diffuse à tous. Chaque autre processus attend :
- soit de recevoir la valeur envoyée par le leader (auquel cas on affecte à v la valeur reçue) ;
- soit de détecter une panne du leader, auquel cas on passe directement à la ronde suivante.

Puisque chaque leader décide et que chaque processus est leader tour à tour, tous les processus corrects décident une et une seule fois (terminaison et intégrité).

Les variables v ne prennent jamais que des valeurs qui ont été initialement proposées (validité).

Les processus affectent toujours la valeur décidée par les leaders précédents à v, à moins que le leader ne soit tombé en panne. Tous décident donc la même valeur (cohérence).

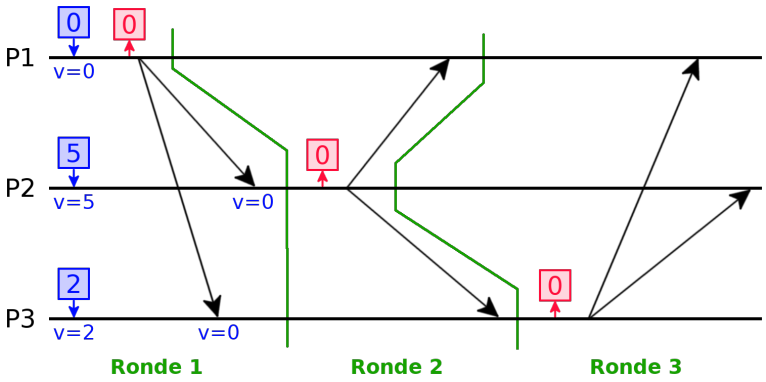

FIGURE 34 – Exemple d'exécution sans faute.

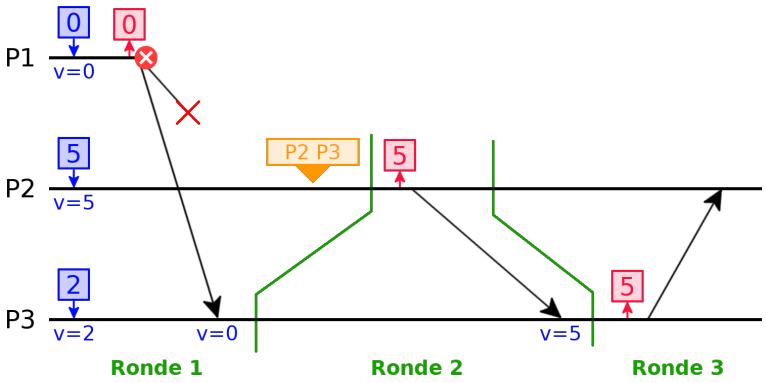

FIGURE 35 – Exemple d'exécution avec une panne de P1.

10.9 Algorithme de consensus avec détecteur P en $f+1$ rondes

Comme le précédent, cet algorithme s'appuie sur un détecteur P.

Soit f le nombre maximum de fautes possibles ($f \leq n-1$). f doit être connu des processus.

Au lieu de la variable v du précédent algorithme, chaque processus stocke en mémoire un vecteur contenant la valeur proposée par chaque processus. Au début, chacun ne connaît que sa propre valeur.

À chaque ronde, chaque processus diffuse son vecteur à tous les autres, puis attend de recevoir les vecteurs de tous les processus qu'il ne suspecte pas.

Après $f+1$ rondes, chaque processus décide la première valeur non vide de son vecteur.

Chaque correct décide donc une et une seule fois (terminaison et intégrité).

Les seules valeurs présentes dans le vecteur sont celles qui ont été proposées (validité).

Le pire scénario est le suivant : à la première ronde, P1 envoie son vecteur à P2 puis tombe en panne. À la seconde ronde, P2 envoie son vecteur à P3 puis tombe en panne, puis la situation se répète à chaque ronde. À chaque ronde, il y a donc une valeur (la valeur proposée par P1) qui n'est présente que dans un seul vecteur. Mais comme il n'y a que f pannes, à la ronde $f+1$ personne ne tombe en panne et l'unique processus qui connaît la valeur de P1 parvient finalement à la diffuser à tous.

Dans tous les cas, toutes les valeurs parviennent donc à tous les corrects, qui décident donc la même valeur (cohérence).

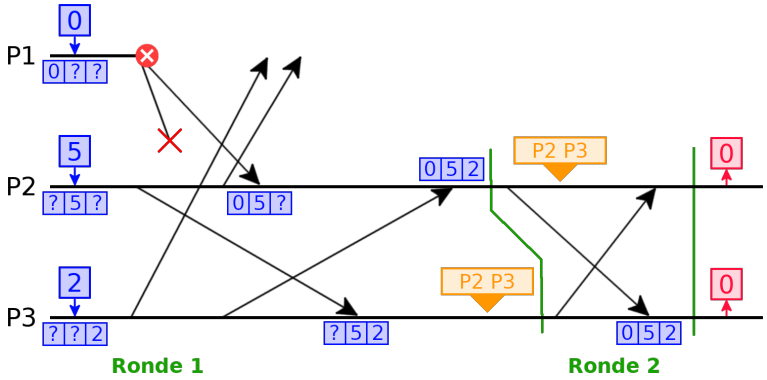

FIGURE 36 – Exemple d'exécution avec une panne de P1.

10.10 Algorithme de consensus avec détecteur $\diamond S$

Ce troisième algorithme s'appuie sur le détecteur $\diamond S$ et sur l'hypothèse d'une majorité de corrects. Contrairement aux deux précédents, cet algorithme doit donc gérer les fausses suspicions.

Il s'agit de l'**algorithme du coordinateur tournant** [CT96].

De façon similaire au premier algorithme, le coordinateur tournant fait usage de rondes asynchrones, et utilise un leader différent à chaque ronde (ici appelé coordinateur). À chaque ronde, le coordinateur est le processus dont l'identifiant est égal au numéro de ronde modulo n. Le modulo est important car il peut y avoir plus de n rondes.

À chaque ronde, le coordinateur tente d'imposer sa valeur, qui sera choisie si il n'est pas suspecté.

Chaque processus a deux variables locales : sa valeur courante v (initialisée à sa valeur proposée), et une variable ts (initialisée à 0) qui indique le numéro de la dernière ronde durant laquelle v a été mise à jour.

Chaque ronde est découpée en quatre phases.

- **Phase 1 : estimation**
  Chaque processus envoie au coordinateur sa valeur courante (ainsi que son ts).
- **Phase 2 : proposition**
  Le coordinateur attend de recevoir les valeurs d'une majorité de processus. Cela finira forcément par arriver puisqu'il y a une majorité de corrects.
  Il choisit ensuite l'une des valeurs les plus à jour (c'est à dire l'une de celles dont le ts est le plus élevé), puis la diffuse.
- **Phase 3 : ack**
  Les processus attendent :
  - Soit que le coordinateur soit suspecté, auquel cas on envoie nack au coordinateur puis on passe à la ronde suivante ;

— Soit de recevoir le message du coordinateur, auquel cas on répond ack, on met à jour v avec la valeur reçue puis on affecte le numéro de ronde actuel à ts.

— **Phase 4 : décision**
Le coordinateur attend de recevoir une réponse d'une majorité de processus.
Si il a reçu ack (et non pas nack) d'une majorité de processus, alors il décide sa valeur v puis diffuse à tous l'ordre de décider. Les autres processus décident à réception du message, à moins qu'ils n'aient déjà décidé.
Sinon, on passe à la ronde suivante.

Il ne faut pas oublier que des fausses suspicions peuvent se produire, en particulier en phase 3. À chaque ronde, il se peut que le coordinateur soit suspecté par une majorité de processus, ce qui empêche l'algorithme de progresser. On change alors de ronde (et de coordinateur) constamment, jusqu'à ce qu'un coordinateur parvienne à s'imposer.

$\diamond S$ garantit qu'à terme, l'un des corrects ne sera plus jamais suspecté par personne : lors du prochain tour de ce processus comme coordinateur, il parviendra donc à obtenir une majorité et à forcer une décision. Cela permet de garantir la terminaison.

La nécessité d'une majorité pour décider et l'utilisation de la variable ts garantit que si deux coordinateurs s'imposent tour à tour, ils proposeront la même valeur en phase 2. La cohérence est donc vérifiée.

Seules les valeurs proposées sont affectées à v, la validité est donc vérifiée. Les processus ne décident qu'une fois et l'intégrité est donc également vérifiée.

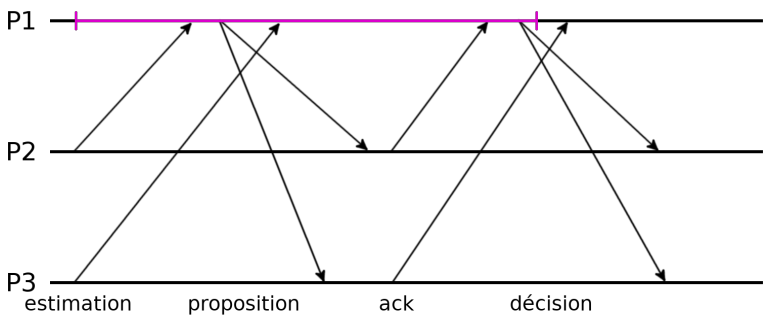

FIGURE 37 – Exemple d'exécution sans faute.

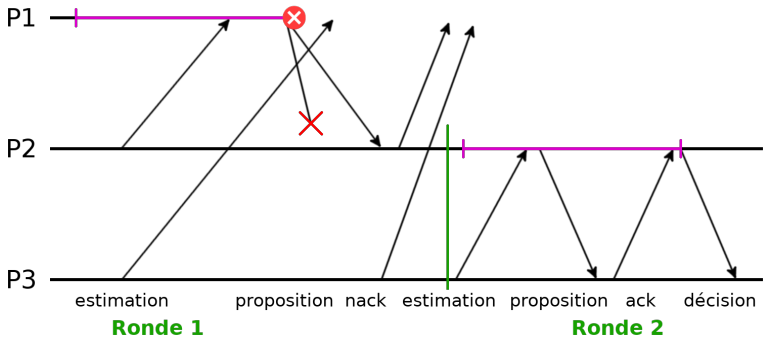

FIGURE 38 – Exemple d'exécution avec une panne de P1.

11 Consensus probabiliste

Puisque l'impossibilité de Fischer, Lynch et Paterson interdit toute solution déterministe au consensus asynchrone, un compromis possible est de chercher une solution probabiliste.

On ne peut pas compromettre les propriétés de validité, cohérence et intégrité du consensus : si l'une de ces propriétés est violée, on ne peut plus maintenir la cohérence des données (voir l'exemple applicatif en Section 2).

En revanche, il est tout à fait raisonnable de ne garantir la terminaison que de façon probabiliste. Certaints processus peuvent alors ne pas terminer. C'est l'approche utilisée par l'algorithme de Ben-Or [Ben83].

L'algorithme de consensus le plus utilisé est probablement l'algorithme de Paxos [Lam98]. Cet algorithme, inspiré du fonctionnement du parlement de l'île de Paxos dans l'antiquité, permet de résoudre un (presque) consensus dans un système asynchrone en s'appuyant sur des hypothèses très faibles. En effet, l'algorithme tolère les pannes temporaires de processus et les pertes de messages.

L'algorithme ne garantit pas la propriété de terminaison. Cependant, la probabilité de terminer tend vers 1 en fonction du temps.

Références

[Ben83] Michael Ben-Or. Another advantage of free choice : Completely asynchronous agreement protocols (extended abstract). In PODC 1983, pages 27–30. ACM, 1983.

[Cha93] Soma Chaudhuri. More Choices Allow More Faults : Set Consensus Problems in Totally Asynchronous Systems. Inf. Comput., 105(1) :132–158, 1993.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. JACM, 43(4) :685–722, 1996.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. JACM, 43(2) :225–267, 1996.

[DFG10] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. JACM, 57(4), 2010.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. J. ACM, 35(2) :288–323, 1988.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. JACM, 32(2) :374–382, 1985.

[Lam98] Leslie Lamport. The Part-Time Parliament. ACM Trans. Comput. Syst., 16(2) :133–169, 1998.