Accord de validation de transaction dans un système de base de données distribué : le problème du *commit*

Michel Chilowicz

21 février 2006

1 Introduction

Un des problèmes introduit par l'utilisation d'un système de base de données distribué réside dans la validation d'une transaction. Ce problème, appelé également problème du *commit*, consiste à trouver un accord entre plusieurs processus pour décider, soit de la validation (*commit*), soit de l'annulation (*rollback*) d'une transaction en cours. Chaque processus doit décider préalablement de sa volonté de validation ou d'annulation de la transaction ; tous les processus doivent ensuite communiquer entre-eux pour finalement décider :

- soit de la validation de la transaction ssi tous les processus ont cette volonté,
- soit de l'annulation de la transaction ssi il existe au moins un processus émettant la volonté d'annuler la transaction.

Nous présentons ici les résultats exposés par le chapitre 7.3 du livre *Distributed Algorithms* [1] sur le problème du *commit*. Dans un premier temps, nous définissons formellement le problème du *commit*, ensuite nous traitons d'une méthode en deux rondes pour résoudre le problème en mode bloquant (tous les processus décident s'il n'y a aucune défaillance). Nous abordons ensuite une méthode en trois rondes pour le problème en mode non-bloquant (tous les processus non-défaillants décident) et enfin nous discutons du nombre de messages générés par un algorithme résolvant le problème du *commit*.

2 Définition du problème

2.1 Pré-conditions

2.1.1 Communications fiables

On suppose que les communications sont possibles entre tous les processus (graphe complet de communication) et que l'acheminement des messages est fiable (aucune perte de message et acheminement en délai borné). Si les communications sont non-fiables, il est possible de rapporter le problème du *commit* à celui de l'attaque coordonnée, problème pour lequel il n'existe aucun algorithme déterministe : il est alors nécessaire d'utiliser une méthode probabiliste. Nous abordons ici le problème en mode synchrone.

2.1.2 Processus défaillants

On autorise la défaillance d'un nombre non-limité de processus. Ainsi le cas extrême de la totalité des processus défaillants peut être rencontré : dans un tel cas, aucune décision ne peut être prise.

2.2 Condition d'agrément

Aucun couple de processus ne peut décider de valeurs différentes.

En d'autres termes, tous les processus décisionnaires (et même les processus défaillants) prennent la même décision : soit la validation (1), soit l'annulation de la transaction (0). Cette condition d'agrément est nécessaire afin d'assurer la consistance de la base de données : en effet, un processus défaillant peut se rétablir ultérieurement, il est donc nécessaire que sa décision vis-à-vis de la transaction soit identique à celle des autres processus.

2.3 Condition de validité

1. Si au moins un des processus commence avec 0, alors 0 est l'unique décision possible.
2. Si tous les processus commencent avec 1 et qu'il n'y a pas de défaillance, alors 1 est la seule décision possible.

2.4 Condition de terminaison

2.4.1 Condition de terminaison faible – terminaison bloquante –

La condition de terminaison faible stipule que s'il n'y a aucune défaillance, alors tous les processus décident.

2.4.2 Condition de terminaison forte – terminaison non-bloquante –

Dans le cadre de la condition de terminaison forte, on autorise la défaillance de processus. On stipule alors que tous les processus corrects décident (les processus défaillants peuvent être non-décisionnaires).

2.5 Similarités avec d'autres problèmes

Le problème du *commit* présente quelques similarités avec d'autres problèmes classiques d'informatique distribuée. On pourrait en particulier le rapprocher du problème de l'attaque coordonnée : la différence réside cependant dans le synchronisme et la non-défaillance des liens de communication.

3 Commit bloquant en deux rondes

3.1 Algorithme

Nous souhaitons résoudre le problème du *commit* en utilisant la condition faible de terminaison : tous les processus décident s'il n'y a pas de défaillance. Une des solutions envisageables consiste à utiliser un algorithme en deux rondes avec un processus de référence (processus 1) centralisant le choix de tous les processus :

1. Lors de la première ronde, tous les processus d'état initial 0 décident 0. Tous les processus, excepté le processus 1, communiquent leur état initial au processus 1 : le processus 1 établit alors un vecteur des états initiaux des n processus. La valeur d'un élément du vecteur (état du processus) peut être soit 0, soit 1, soit ? si aucune réponse n'a été reçue du processus. Finalement, le processus 1 décide :
   - 1 si le vecteur ne comprend que des valeurs 1,
   - 0 sinon (ce qui implique que le vecteur comprenne au moins une valeur 0 ou au moins une valeur ?).
2. Le processus 1 communique sa décision (0 ou 1) à tous les autres processus : les processus qui n'avaient pas encore choisi à la ronde 1 (les processus d'état initial 1) adoptent le choix communiqué par le processus 1.

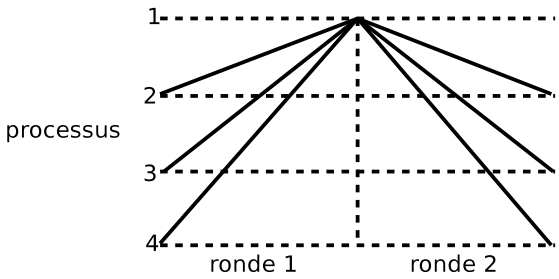

Figure 1: Schéma de communication de l'algorithme de commit en deux rondes

3.2 Complexité

Cet algorithme nécessite exactement 2 rondes : si aucune défaillance n'intervient, dans la première ronde, $n-1$ processus envoient un message notifiant leur état au processus 1 et dans la seconde ronde, le processus 1 notifie sa décision à ces $n-1$ processus. Ainsi, au plus (lorsqu'il n'y a pas de défaillance) $2n-2$ messages sont échangés (voir figure 7 pour le schéma de communication).

3.3 Preuve

Théorème 1. *L'algorithme proposé de* commit *en deux rondes satisfait les conditions d'agrément, de validité et de terminaison faible mais pas la condition de terminaison forte.*

Preuve. S'il n'y a aucune défaillance, tous les processus reçoivent la décision du processus 1 et décident celle-ci (si leur décision n'avait pas encore été faite) : la condition faible de terminaison est trivialement vérifiée.
Si au moins un processus est d'état initial 0 : soit le processus 1 reçoit sa valeur et décide 0, soit il ne la reçoit pas (défaillance du processus) – état initial ? – et décide 0. Ensuite le processus 1 diffuse sa décision 0 : si le processus 1 subit une défaillance, certains processus peuvent recevoir sa décision 0 et d'autres non. Les processus d'état initial 1 (qui n'ont donc pas décidé en ronde 1) qui n'ont pas reçu la décision du processus 1 ne peuvent décider. Tous les autres décident 0.
Si tous les processus sont d'état initial 1, aucun n'a encore décidé : le processus 1 collecte les valeurs des processus qui peuvent être ? ou 1. Si au moins une valeur ? (processus défaillant) est collectée, alors le processus 1 décide 0. Sinon il décide 1. Si le processus 1 n'est pas défaillant, tous les processus corrects reçoivent la décision du processus 1 et s'y conforment. Les processus qui n'ont pas reçu de message de 1 ne peuvent décider.
Les conditions d'agrément et de validité sont donc vérifiées.
Par contre la condition forte de terminaison n'est pas vérifiée : en effet, des processus corrects d'état initial 1 ne peuvent pas décider en cas de défaillance du processus 1 s'ils ne reçoivent pas sa décision. Dans une telle situation, on pourrait modifier l'algorithme afin que les processus corrects ayant reçu la décision 1 la diffuse aux autres processus. Cependant si le processus 1 faillit avant d'avoir initié la diffusion de sa décision et que tous les autres processus, corrects, sont d'état initial 1, alors ces processus corrects ne peuvent décider, car méconnaissant l'état initial du processus 1, ils ne peuvent satisfaire la condition de validité. □

4 Commit non-bloquant en trois rondes

4.1 Algorithme

Principe On propose maintenant un algorithme en trois rondes afin de garantir la terminaison forte, c'est-à-dire la décision de tous les processus corrects. Cet algorithme se base sur la version en deux rondes : une ronde est ajoutée afin de s'assurer que le processus 1 ne puisse décider 1 que si chaque autre processus correct est prêt à décider 1.

Description de l'algorithme en trois rondes

1. Lors de la première ronde, tous les processus envoient leur valeur initiale au processus 1. Les processus de valeur initiale 0 décident 0. Le processus 1 constitue un vecteur des valeurs initiales. Il entreprend alors les actions suivantes selon le vecteur des valeurs initiales :
   - si le vecteur ne contient que des 1, plutôt que de décider 1, le processus 1 entre dans l'état *prêt* ;
   - sinon, si le vecteur contient au moins un 0 ou ? (processus défaillant), le processus 1 décide 0.
2. Pour la seconde ronde, si le processus 1 a déjà décidé 0, alors il diffuse aux autres processus sa décision : les processus recevant ce message adoptent la décision 0. Sinon, s'il est en état *prêt*, il diffuse le message *prêt* : un processus qui reçoit ce message entre en état *prêt* ; le processus 1 décide ensuite 1.
3. La ronde 3 est exécutée si le processus 1 a décidé 1 : il diffuse sa décision aux autres processus et tous les processus qui la reçoivent l'adoptent en décidant 1.

États des processus On associe à chaque processus un automate à états que nous énumérons :

- *dec0* : le processus a décidé 0.
- *dec1* : le processus a décidé 1.
- *prêt* : le processus n'a pas décidé mais est prêt à décider 1.
- *incertain* : le processus n'a pas décidé et n'est pas *prêt* (il s'agit de l'état initial des processus de valeur initiale 1).

4.1.1 Propriétés

On propose le lemme suivant concernant l'algorithme en trois rondes :

Lemme 1. *À la fin des trois rondes de l'algorithme, les propriétés suivantes sont vraies :*

1. *Si un processus est dans l'état* prêt *ou* dec1*, alors les valeurs initiales de tous les processus sont 1.*
2. *Si un processus est dans l'état* dec0*, alors aucun processus est en* dec1*, et aucun est* prêt*.*
3. *Si un processus est dans l'état* dec1*, alors aucun processus est en* dec0*, et aucun processus correct est* incertain*.*

Preuve du lemme.

1. Si un processus est *prêt*, alors p1 a constitué un vecteur de valeurs initiales $(1, ..., 1)$. Si un processus est *dec1*, il a été auparavant *prêt*.
2. Si un processus est *dec0*, cela signifie soit que le processus a une valeur initiale 0, soit que p1 lui a transmis sa décision 0. Si ce processus a une valeur initiale 0, alors le vecteur d'états initiaux de $p1$ contient au moins un 0 ou ? ce qui implique pour p1 *dec0*. Les processus recevant la décision *dec0* de p1 deviennent *dec0*, ceux qui ne la reçoivent pas restent *dec0* ou *incertain* si leur choix initial est 1. Aucun processus est donc *dec1* et aucun processus est *prêt*.
3. À la fin de la ronde 2, après avoir diffusé le message *prêt* à tous les processus, p1 ne peut décider que 1. Ainsi à la fin de la ronde 2, les processus autres que p1 ont soit reçu *prêt* et sont entrés dans l'état *prêt*, soit ils sont défaillants et restent dans l'état *incertain*. À la fin de la ronde, p1 rentre dans l'état *dec1*.

□

4.1.2 Preuve

On propose le lemme suivant statuant sur le respect des conditions d'agrément, de validité et de terminaison de l'algorithme :

Lemme 2. *Après les trois rondes, les propositions suivantes sont vraies :*

1. *La condition d'agrément est vérifiée.*
2. *La condition de validité est vérifiée.*
3. *Si le processus 1 est correct, tous les processus corrects décident.*

Preuve du lemme. Le lemme 1 montre qu'il ne peut coexister de processus *dec0* et *dec1* à la fin des trois rondes, ce qui montre l'agrément. De plus ce lemme montre que si un processus a pour valeur initiale 0 (et donc décide 0), la seule décision possible est 0.
Si tous les processus débutent avec 1 et qu'il n'y a pas de défaillance, le vecteur de p1 est $(1, ..., 1)$: p1 devient *prêt* et envoie le message *prêt* à tous les autres processus qui deviennent *prêt* également. Puis p1 décide 1 et envoie sa décision aux autres processus qui choisissent tous *dec1* : cela montre le deuxième point de la condition de validité.
Si le processus 1 ne faillit pas, il décide et diffuse sa décision à tous les processus : parmi ceux-ci, les corrects adoptent sa décision. □

4.2 Algorithme complet : protocole de terminaison

L'algorithme en trois rondes tel que nous l'avons proposé ne résout toujours pas la condition forte de terminaison : en effet, si le processus 1 connaît une défaillance, certains processus corrects peuvent ne pas pouvoir décider et rester, s'ils étaient d'état initial 1, soit dans l'état *incertain*, soit dans l'état *prêt*. La décision de tous les processus corrects n'est garantie que si le processus 1 ne connaît pas de défaillance.

Afin de remédier à cette situation, on propose donc de mettre en place un protocole de terminaison. Son principe consiste, si le processus 1 connaît une défaillance, à utiliser le processus 2 comme nouveau processus référent. Tous les processus envoient leur état à p2 qui répète alors l'algorithme à trois rondes. En cas de défaillance de p2, on répète le même procédé avec le processus 3, ..., jusqu'à éventuellement le processus n.

4.2.1 Description du protocole de terminaison

- Ronde $3(k-1)+1$: Tous les processus envoient leur état au processus k, soit *dec0*, *dec1*, *prêt*, *incertain* ou *?* pour les processus défaillants (message non reçu par le processus k). Le processus k forme un vecteur des états des processus. Le processus k entreprend alors une action selon les valeurs du vecteur :
  - Si le vecteur contient au moins une valeur *dec0* et que le processus k n'a toujours pas décidé (état *incertain*), il décide 0.
  - Si le vecteur contient au moins une valeur *dec1* et que le processus k n'a toujours pas décidé, il décide 1.
  - Si le vecteur contient uniquement des valeurs *incertain* (et éventuellement ?), alors le processus k décide 0.
  - Si le vecteur contient au moins une valeur *prêt*, alors le processus k devient *prêt*.
- Ronde $3(k-1)+2$: Si le processus k a décidé, il diffuse sa décision, sinon il est dans l'état *prêt* et diffuse un message *prêt* aux autres processus : les processus corrects qui reçoivent ce message deviennent *prêt* et le processus k décide 1.

- Ronde $3(k+1)+3$: Cette ronde est nécessaire si le processus k vient de décider 1 : il diffuse sa décision aux autres processus. Les autres processus corrects adoptent alors la décision 1.

On continue alors le protocole de terminaison avec le processus $k+1$, $k+2$, ..., n.

4.2.2 Terminaison forte de l'algorithme

Théorème 2. *L'algorithme à trois rondes complet (avec le protocole de terminaison) est un algorithme de commit non-bloquant.*

Preuve du lemme 1 pour l'algorithme complet. Tout d'abord, nous montrons la validité du lemme 1 pour l'algorithme complet par induction. Le lemme est vérifié pour la première exécution des trois rondes. On le suppose vérifié pour l'exécution k (processus coordinateur k).
À l'issue de l'exécution $k+1$, si un processus est dans l'état *prêt* ou *dec1*, cela implique qu'à la ronde 1 de cette exécution, le vecteur constitué par $k+1$ contient au moins une valeur *prêt* ou *dec1* : tous les processus ont donc pour valeur initiale 1.
Si un processus est *dec0* à l'issue de l'exécution $k+1$, soit cette décision a été prise lors d'une exécution précédente ce qui signifie que le vecteur d'états du processus $k+1$ contient au moins une valeur *dec0* ou que des valeurs *?*, soit prise lors de la présente exécution *k+1* (précédent état *incertain*). Dans les deux cas, cela implique que le vecteur de valeurs initiales de $k+1$ contient au moins un *dec0* ou que des *?* ; aucun processus ne peut évoluer vers l'état *prêt* ou *incertain* : ainsi par induction aucun processus est *dec1*, ni *prêt*.
Si un processus est *dec1* à l'issue de l'exécution *k+1*, cela peut être le fait soit d'une décision antérieure, soit d'une décision lors de la troisième ronde de l'exécution *k+1*. S'il s'agit d'une décision antérieure, soit le processus est encore correct et il transmet un état *dec1* au vecteur, soit il est défaillant et il transmet *?*, mais d'après la propriété 3, tous les processus corrects sont alors *dec1* ou *prêt*. Si la décision est prise lors de l'exécution $k+1$, alors tous les processus reçoivent *prêt* puis *dec1* : un processus défaillant peut être *incertain* mais les processus corrects sont *dec1* ou *prêt* (en cas de défaillance de $k+1$) : aucun processus peut être *dec0*.
Le lemme est donc vérifié pour l'algorithme complet. □

Preuve du théorème. La condition de validité statuant que si un processus a pour état initial 0, 0 est l'unique décision possible est vérifiée par la propriété 2 du lemme étendu : *dec0* et *dec1* ne peuvent coexister et un état *dec0*. Si un processus commence avec 0, il décide immédiatement 0 et donc 0 ne peut être que l'unique décision car *dec1* est impossible.
La condition de validité imposant que dans le cas où tous les processus commencent avec 1 (et sans défaillance), 1 est l'unique décision possible est également vraie car, en l'absence de défaillance, tous les processus décident 1 lors de la première exécution.
La condition d'agrément est vérifiée par le lemme étendu. Concernant la condition de terminaison forte, si tous les processus échouent, elle est trivialement vérifiée. Dans le cas contraire, on note qu'il existe au moins une exécution k avec le processus k correct : tous les processus corrects décident alors. □

4.2.3 Complexité

L'algorithme en trois rondes complet nécessite $3n$ rondes. Il est néanmoins possible de réduire le nombre de rondes en détectant au terme de chaque exécution si les processus encore présents sont tous corrects et ont donc tous décidé. En effet, la seule situation empêchant la terminaison forte à l'issue d'une exécution se produit si la processus coordinateur faillit : cette situation peut être détectée en exigeant l'ajout d'une quatrième ronde où le coordinateur envoie un message de non-défaillance à tous les processus. Si ce message est envoyé à au moins un processus, cela signifie que tous les processus corrects ont décidé (car le coordinateur n'a pas été défaillant lors des trois rondes précédentes) : le processus sait qu'il n'a pas à devenir processus coordinateur. Cependant, le coordinateur peut faillir lors de cette quatrième ronde : cela peut induire des exécutions supplémentaires inutiles de la part des processus n'ayant pas reçu le message. Ce

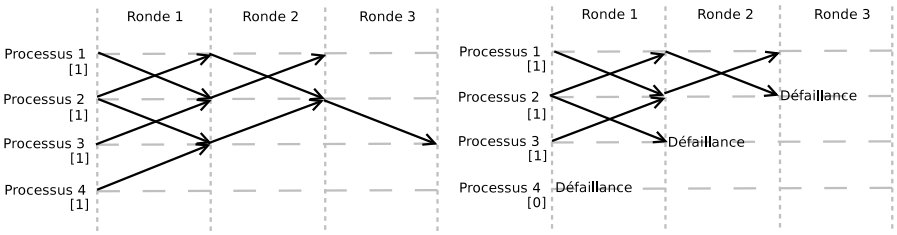

Figure 2: Exemples de schémas de communication dans lesquels 4 n'affecte pas 1 : on constate qu'en cas de défaillances, α_1 et α'_1 sont équivalents pour 1.

protocole permet néanmoins de limiter l'algorithme à $3(k+1)$ rondes, avec k le nombre de processus défaillants autorisés.

5 Borne inférieure du nombre de messages

Quel est le nombre minimum de messages nécessaires à l'exécution d'un algorithme de commit (satisfaisant les conditions d'agrément, de validité et au moins de terminaison faible) ? Nous montrons que cette borne inférieure est de $2n-2$ messages.

Théorème 3. *Un algorithme résolvant le problème du commit, même avec terminaison faible, nécessite au minimum $2n-2$ messages dans le cas d'un déroulement sans défaillance avec toutes les valeurs initiales à 1.*

Pour montrer ce théorème, nous nous intéressons tout d'abord au lemme suivant :

Lemme 3. *Soit A un algorithme de commit et α_1 le cas d'exécution sans défaillance de A avec toutes les valeurs initiales à 1, alors pour tous processus i et j, i affecte j dans $patt(\alpha_1)$.*

Ainsi en d'autres termes, dans un algorithme de commit, chaque processus doit recevoir une information de tout autre processus. Pour s'en convaincre, il est possible de prendre pour exemple un algorithme avec la présence de 4 processus mais sans que le processus 4 n'affecte le processus 1 (l'état initial de 4 n'affecte pas 1) : ainsi quel que soit la décision de 4, la décision de 1 n'est pas affectée et ne dépend uniquement que des valeurs initiales des trois premiers processus comme le montre la figure 5). On formalise cette situation par la preuve suivante :

Preuve. Le respect de la condition de validité et de terminaison (au moins faible), tous les processus décident 1 lors de l'exécution α_1 où toutes les valeurs initiales sont 1. Supposons le lemme faux et deux processus distincts i et j tels que i n'affecte pas j dans $patt(\alpha_1)$. Alors en considérant α'_1 avec la valeur initiale de i à 1 et la défaillance de tous les processus après avoir été affectés par le processus i, nous avons α_1 et α'_1 équivalents du point de vue de j : j décide alors 1 également dans α'_1 ce qui contredit la condition de validité. □

Nous introduisons maintenant le théorème utilisé pour montrer la condition de borne minimale :

Lemme 4. *Soit γ un schéma de communication quelconque. Si dans γ, chaque processus d'un ensemble de $m \leq 1$ processus affecte chacun des n processus du système, alors il y a au minimum $n+m-2$ messages utilisés dans γ (en particulier pour un ensemble $m=n$ processus, il y a au moins $2n-2$ messages).*

Preuve. On montre ce lemme par induction sur m :

- Initialisation pour $m=1$. Soit le processus i affectant tous les autres processus du système : tous les autres processus doivent recevoir une information de i, d'où la nécessité au minimum de $n-1$ messages.

- Induction : on suppose le lemme vrai pour m, on le montre pour $m+1$. Soit I un ensemble de $m+1$ processus qui affectent tous les n processus du système. Soit i un processus de I qui envoie un message à la ronde 1 (première ronde où un message est envoyé par un processus de I) de γ. Soit γ' un schéma de communication obtenu en enlevant ce message de la ronde 1 : ainsi tous les processus $I - i$ affectent les n processus dans γ'. Ainsi $n + m - 2$ messages sont nécessaires dans γ', d'où $n + m - 2 + 1 = n + (m + 1) - 2$ messages nécessaires dans γ.

□

Finalement, les lemmes 3 et 4 montrent le théorème de borne limite de message $2n - 2$ pour un algorithme de commit.

6 Conclusion

Nous avons présenté un algorithme de commit bloquant (en deux rondes) et un algorithme de commit non-bloquant (en trois rondes). Ensuite nous avons montré que la borne inférieure du nombre de messages nécessaires à un algorithme de commit dans le cas d'une valeur initiale de 1 pour tous les processus est $2n - 2$. Ces résultats peuvent être utilisés afin d'implanter un système de base de données distribué : il est cependant nécessaire dans une telle situation de considérer l'asynchronisme des communications ainsi que la possibilité de perte de messages.

References

[1] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.