Critères de Cohérence faible byzantine appliquée aux environements cloud

JOLY Amaury
Encadrants : GODARD Emmanuel, TRAVERS Corentin

Aix-Marseille Université, Scille

8 novembre 2023

Résumé

lorem ipsum dolor sit amet.

Table des matières

1 Introduction

1.1 Motivation

La plupart des applications colaborarifs sur le marché fonctionnent sous la forme d'une entité centralisé qui traite les données, les redistribuent aux différents client et résoud les problèmes de cohérences. De manière à favorisé l'intéractivité de l'applications et ainsi fournir l'experience utilisateur la plus optimal, on constate que certaines de ces application font appels à des algorithmes qui ne respectent pas ce qu'on attendrait d'une éxécution séquentielles, pouvant ainsi menés à des incohérences dans les éxécutions. Et infine avoir un impact sur l'experience utilisateur.

Il nous semble donc intéressant de se pencher sur l'état de la rechecrhe concernant les divers manières d'aborder ce compromis dans la gestion de la cohérence dans ce genre de systèmes. Et ainsi mettre en avant les différentes propriétés qui en résultents. L'intérét ici est de fournir une base permettant d'éclairer la prise de décision lors de l'implémentation d'un algorithme visant à satisfaire un problèmes distribué.

Ce document à était réalisé dans le cadre de mon stage de fin d'étude de Master. L'objectif est de fournir un état de l'art de la recherches autour des différents compromis réalisables dans la gestion de la cohérence dans le contexte d'applications distribués. Il à pour but de servir de base à un projet de thèse en CIFRE financé par l'entreprise Scille SAS, et encadré par le Laboratoire d'Informatique et Système.

1.2 Introduction

L'étude du comportement des systèmes distribués date des années 1970 avec l'arrivé des premiers processeurs multicoeur. Il semblait essentiel à l'époque de pouvoir répartir une tâche entre plusieurs acteurs de manières asynchrone afin d'accroitre les performances. Ainsi Lamport définit un model de cohérence qu'il nomme "séquentiel". C'est à dire que, une tâche pouvant être découper en un ensemble d'opération, chaque opération étant répartit entre plusieurs acteurs, il est possible de resequentialiser ces opérations de facon à ce qu'infine leurs remaniement séquentiel soit indicernable vis-à-vis de ce qui pourait être attendu de l'éxécution de la même tache dans un environement non distribués.

Le désaventage de cette approche est qu'elle nécéssite de garder une synchronicité forte entre les acteurs. La difficulté à préserver cette synchronicité croit considérablement avec le nombre d'acteur et la latence entre eux. L'usage d'une approche séquentiel sur des applications visant à faire travailler des acteurs distant à travers un réseau mondia tel qu'internet semble à ce titre loin d'être efficace. Et restreint ces dernières à des usages limités où une faible intéractivité doit être acceptée par les concepteurs.

Néanmoins là ou l'approche séquentiel est la seule fournissant une cohénce absolue dans la gestions des données du point de vue général et du point de vue de l'utilisateur. Il est possibe d'accepter une perte de cette cohérence partiel ou total afin de gagner en performances sur des applications a grande echelle. Ce compromis entre cohérence et performances est l'objet de ce document qui vise à faire un état du paysage des différentes approches et solutions existantes, en définissant les cas d'usages associés.

2 Cohérence Forte

La cohérence forte présente, dans un premier temps, l'approche la plus intuitive pour une application collaborative dans le cloud. Prenons l'approche de chercher à répliquer un processus d'édition local d'un document. Formalisons dans un premier temps ce qu'on attend d'une application d'édition de document local.

On voit sur la figure 1 que Bob émet une requête à son interface utilisateur. À partir de là le clavier partage l'information au système jusqu'à l'application d'édition. En fonction du traitement de cette information, si elle est acceptée par l'application d'édition, alors elle affiche un document modifié en accord avec la saisie de Bob à l'écran. Ce qui donne un feedback à Bob concernant son action initiale. Dans notre modélisation, nous séparons bien les actions de l'utilisateur sur ses interfaces (l'écran et le clavier) des interactions au sein du système.

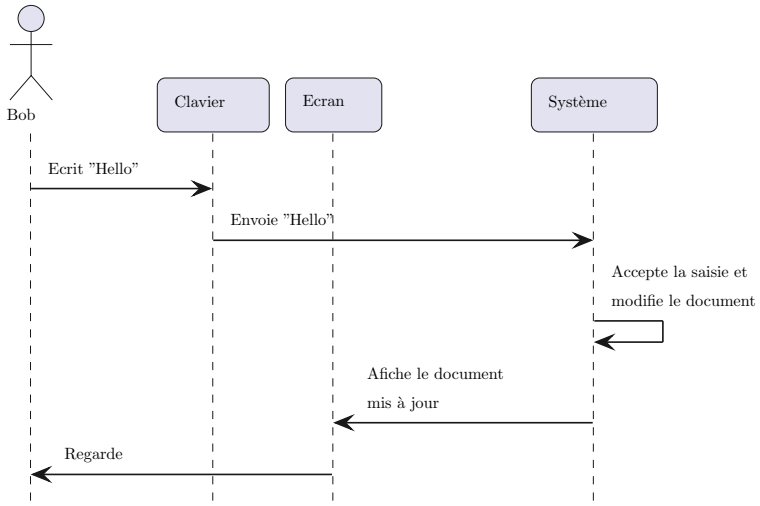

FIGURE 1 – Bob edit un document localement

Dans un cas d'usage courant, une application d'édition local ne présente pas de problèmes liés à la présence de plusieurs éditeurs en simultanés. Il y a en effet une seule interface homme-machine, c'est-à-dire de façon pragmatique un unique écran et clavier.

Imaginons ainsi un cas où Alice et Bob souhaitent éditer ensemble un même document. Si on voulait simplement appliquer le fonctionnement local que nous venons de décrire à cette nouvelle problématique. Il conviendrait de simplement rajouter un IHM à Alice. Ainsi il y a toujours une seule application d'édition collaborative sur lequel viennent se connecter cette fois-ci deux interfaces (comme visible dans la figure 2).

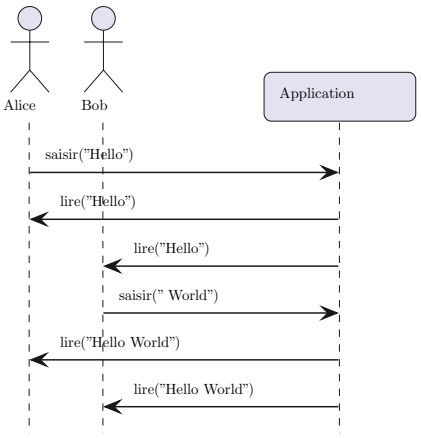

FIGURE 2 – Alice et Bob éditent un document localement

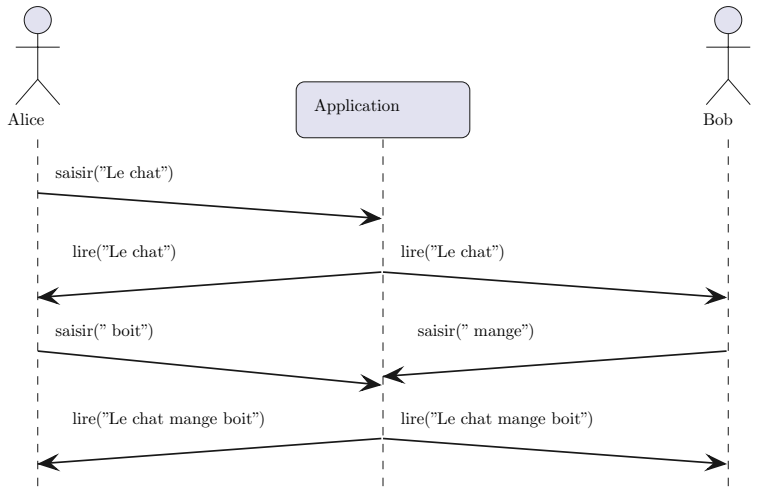

FIGURE 3 – Alice et Bob éditent un document localement de manière concurente

Nous avons à présents deux acteurs qui éditent et lise le document, et un autre qui s'occupe de stocker et de synchroniser le document entre ces deux acteurs.

La limite liée à cette approche réside dans le fait que les communications entre Alice et l'application et Bob et l'application ne peuvent pas être garentits comme équitables. En effet, Alice et Bob peuvent

très bien se retrouver déconnectés de manière non prévisible et asynchrone. Ou bien avoir des temps de transmissions de leurs informations qui diffèrent. Le caractère non-deterministe des ces erreurs peut résulter en des incohérences dans l'ordre d'arrivé des informations soumises par Alice et Bob. Ce qui peut a terme créer des incohérences dans le document final.

C'est néanmoins une problèmatique qui peut être assimilée à de la gestion de la concurrence entre processus. C'est en effet un domaine qui est étudié depuis l'apparition des premiers processeurs multi-coeur. Une approche que nous pouvons donc prendre à la programmation concurente serait de résoudre ce problème en venant utliser des techniques de synchronisation de processus. C'est-à-dire en utilisant des mécanismes de synchronisation tel que des verrous dans notre application. On peut par exemple utiliser des sémaphores pour gérer les accès concurrents à la mémoire partagée. Ce qui resulterais en l'execution suivante :

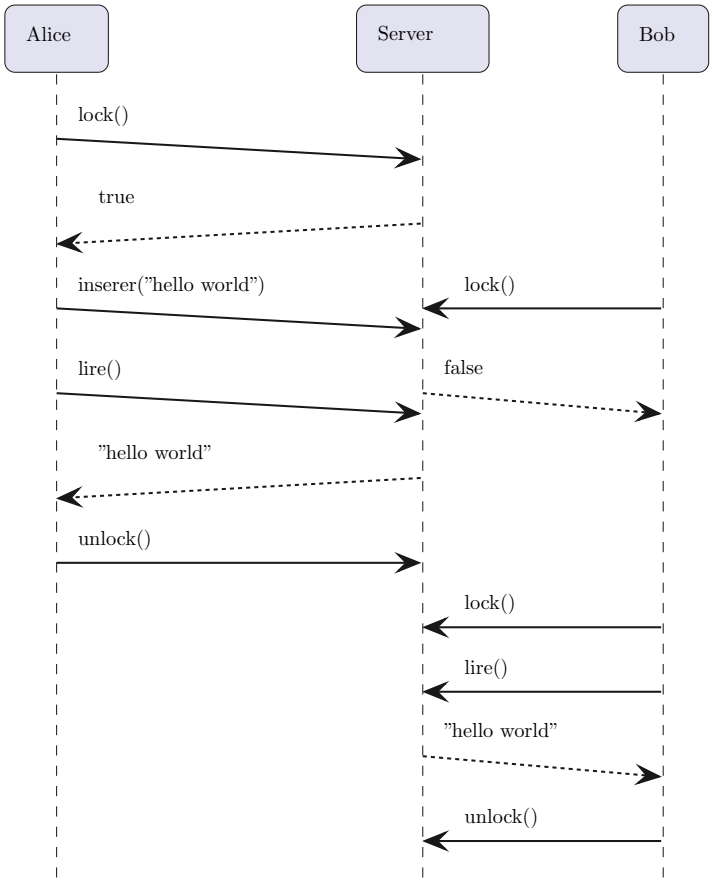

FIGURE 4 – Alice et Bob éditent un document localement avec des vérous

On voit sur la figure 4 que les actions d'Alice et Bob sont sérialisées. C'est-à-dire que les actions d'Alice et Bob sont exécutées l'une après l'autre. Ce qui permet d'éviter les problèmes de concurrence entre les deux acteurs. Cependant, cette approche est très limitée. En effet, elle ne permet pas de profiter des avantages de l'édition collaborative. En effet, cela signifie que l'un des deux acteurs doit attendre que l'autre ait fini d'éditer le document pour pouvoir à son tour l'éditer. Ce qui rabaisse les performance de l'application à la latence de l'acteur le plus lent.

En revenons dans notre contexte initial qui est une applicationn d'édition collaborative nous souhaitons pouvoir gérér un grand nombre d'utilisateur en simultanés. Ainsi la probabilité d'avoir une expérience dégradé pour l'ensemble des utilisateur croit avec le nombre d'utilisateur. Pour pallier à ce problème l'approche qui peut être adopté est d'exclure les utilisateurs possèdant des connections trop limités avec le serveur.

C'est néanmoins la seule approche qui nous permet d'obtenir une cohérence absolue autant dans la prise en compte des opérations d'écritures des acteurs que dans le traitement de leurs lectures.

Selon Perrin on se placerait dans une approche qui relèverait de la cohérence séquentielle.

2.1 Cohérence séquentielle

Validité Dans une application d'édition collaborative on s'attend à ce que toutes les entrées des utilisateurs soient prises en compte. Il nous parait en effet inacceptable que des saisies disparaisses de manière aléatoire durant l'exécution.

Ce principe nous permet d'introduire une première propriété utilisée par Perrin pour définir et classer les différents critères de cohérence qu'est la validité.

La validité peut ainsi être résumé comme la propriété qui permet d'assurer que toutes les saisies réalisées, et seulement elles, soient prises en compte dans l'exécution du système.

Ceci se traduit par l'affichage d'un retour quant à la saisie de bob sur son application. Lui signifiant que son interaction avec l'application à bien était prise en compte, et n'est donc pas passée à la trappe. Elle n'implique néanmoins aucune notion d'ordre dans les saisies, ce qui signifie que les saisies peuvent être réagencées durant l'exécution.

Convergence Une autre attente d'une application d'édition collaborative est en quelque sorte sont determinisme sur une durée infinie. On s'attend en effet, une fois toutes les éditions terminés, que le document soit identique pour tous les acteurs. C'est-à-dire que peut importe l'ordre dans lequel les saisies ont été réalisées, le document final doit être identique pour tous les noeuds.

C'est ce que Perrin appelle la convergence qu'il définit plus formellement comment étant la propriété qui assure que si un nombre finit d'ecriture et infini de lecture sont réaliséés, alors tous les noeuds finissent par avoir la même valeur.

Localité D'état La dernière propriétée du model de Perrin est la localité d'état.

Elle consiste en le fait que chaque acteur doivent garder une vision local cohérente du système. Par exemple, comme illustrée dans la Figure 5, au fur et à mesure des éditions de Bob, sa vue local va évoluer en conséquence, mais si il recoit à un moment donné une édition d'Alice qui est antérieur à une de ses éditions, alors il doit privilégier sa propre édition. C'est-à-dire que sa vue local doit être cohérente avec son historique d'édition quitte à diverger de la vue local d'Alice.

FIGURE 5 – Localité d'état

3 Les critères de cohérence

Nous venons de définir la cohérence séquentielle et de proposer une approche pour la réalisser. Néanmoins nous avons vue que cette approche est limitès puisqu'elle pousse le système à être nivelé par le bas en s'adaptant à l'acteur le plus lent. Nous allons donc explorer d'autres approches réalisant plus de compromis dans la gestion de la cohérence du système.

Ainsi là où la cohérence séquentielle (et par extension la cohérence forte) permettait de garantir nos 3 propriétées que sont la validité, la convergence et la localité d'état. Nous allons explorer des solution qui certaines de ces propriétées afin de gagner en performance et en résilience.

3.1 Adaptation à notre problème

Dans le cadre de notre problématique il convient dans un premier temps de spécifier quels sont les critères qui semblent être essentiels à la réalisation d'une application d'édition collaborative. Pour ca nous pouvons dèja imaginer ce qu'implique concretement ces différents critères sur le comportement de notre application.

3.1.1 Validité

Comme dit précèdemment, la validité permet d'assurer que les saisies de Bob soient toujours prise en compte en tout point du système. C'est-à-dire que si Bob saisit "Hello" puis "World" alors il s'attend à ce que ces deux saisies soient prises en compte. Ce qui se traduit par l'affichage de "Hello World" ou "World Hello sur son écran. Les états finaux incorects qui ne respecterais pas la validité sont alors "Hello", "World" ou "$\emptyset$".

De la même manière, les saisies des aitres acteurs du sytème peuvent donc très bien étre ignorés, intercalés entre les saises de Bob, et réagencés durant l'éxécution.

Ne pas prendre en compte le critère de validité dans notre application reviendrait donc à être capable d'ignorer des saisies soumises par l'utilisateur. C'est ce qui est formalisé par le critère de cohérence dit de Sérialization (selon Perrin).

L'application de la validité demande à notre système une approche d'acknowledgement. Puisque toutes saisies ce doit d'être considéré, le client doit s'assurer que sa saisie à bien était recu par l'ensemble des noeuds. Dans le cas d'une application centralisé, seul un intermédiaire doit être contacté. Mais dans un contexte plus distriué, le problème augmente en compléxité et en coût pour le réseau.

La Serialisation La sérialisation se propose donc comme étant l'exact inverse de la validité, c'est à dire un critère de cohérence fort auquel nous aurions retiré la validité. ($SER = \{C_\top\} - \{V\}$)

3.1.2 Convergence

La convergence permet d'assurer que le document final soit identique pour tous les acteurs. C'est-à-dire que peut importe l'ordre dans lequel les saisies ont été recu par chaque noeud, le document final doit être identique pour tous.

Il est important de noter que la convergence seule ne dit rien sur la source des données qui composent le document. Ainsi il est possible que le document final ne prenne pas en compte toutes les saisies réalisées sur le système. Ou bien même qu'il prenne en compte des saisies qui n'ont jamais été réalisées.

Appliqué à notre problème, la convergence assure que le document final soit identique pour tous les acteurs, ce qui peut sembler essentiel dans notre cas.

L'application de la convergence introduit dans notre système une notion de gestion des conflits. Si plusieurs acteurs souhaitent écrire au même moments, alors des mécanismes doivent être mise en place pour une gestion des conflits deterministe. C'est à dire, permettant de garantir une seule version pour tous les acteurs quelque soit sa position dans le système, et donc l'ordre potentiel de reception des saisies.

[Un exemple de gestion de conflits pas si évidente que ca]

Cette gestion des conflit peut être réalisé de manière distribué impliquant la mise en place d'un algorithme de consensus. Ce qui peut être très couteux en terme de performance et de complexité. Ou bien être délégué à un acteur centralisé qui aura le pouvoir de manipuler l'ordre final des saisies. Ce qui peut poser des problèmes en termes de gouvernance du système, donnant le pouvoir à un seul acteur.

Il est bon de noter que bien que la convergence est une propriétée qui semble importante dans le cadre d'une application d'édition collaborative classique. Il existe des manière de concevoir une application collaborative fournissant un document final identique, mais ne garantissant pas nécéssairement la convergence dans l'ordre où les saisies sont interprétés. Ce point sera discuté lorsque nous aborderons les CRDTs.

3.1.3 Localité d'état

La localité d'état permet d'assurer que les mises à jour d'affichages futures soient toujours cohérentes avec les mises à jour d'affichages passées. Indirectement, cela interdit le changement de l'ordre dans lequel les transactions passés sont interprétés une fois que le résultat est affiché à l'utilisateur. Mais empeche aussi l'insertion et la supression de transctions dans ce même passé.

L'objectif à haut niveau est de garder une expérience cohérente pour l'utilisateur, en interdisant à l'application de tolérer des retours en arrières de l'état courant.

Appliqué à notre problème, la localité d'état permet d'assurer que les saisies de Bob et celles qu'il a reçu soient toujours affichés dans l'ordre où elles ont étaient interprétés initialement. Cette propriétée permet par exemple d'empecher le cas où Alice traiterais la saisie de bob, bob de son coté suprime la saisie qu'il vient de réalisé, et Alice renvoit à Bob la saisie qu'elle vient de modifier. Ce cas est explicité dans la figure 6.

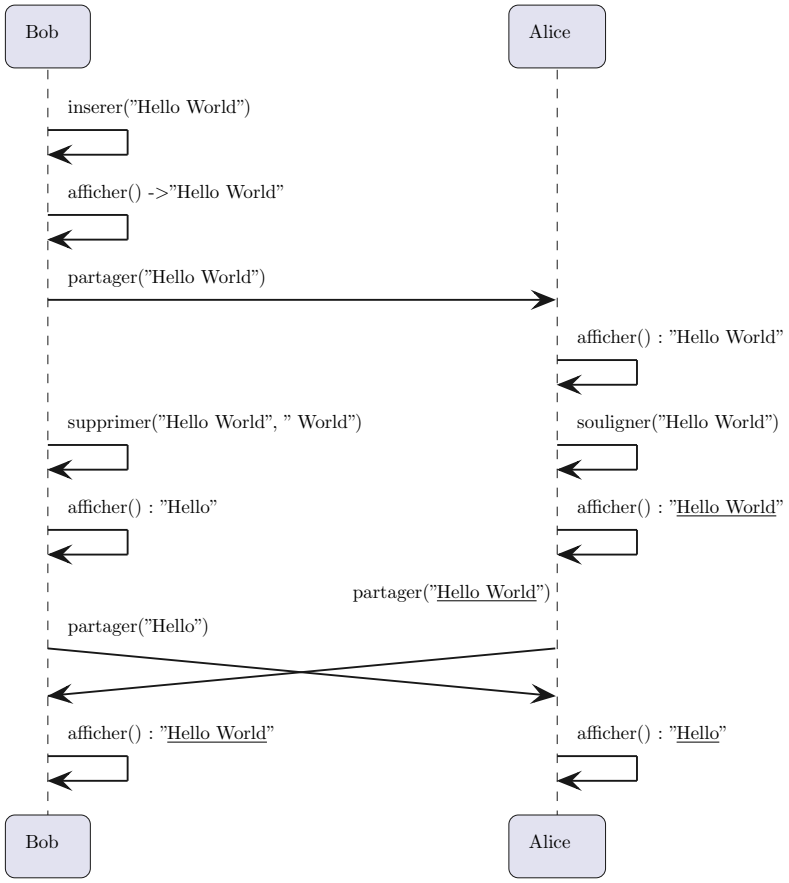

FIGURE 6 – Localité d'état non respectée

Du point de vue de Bob, il vient de revoir aparaitre une ligne qu'il vient de supprimer, ce qui rend son expérience incohérente.

En effet, premièrement Bob à vue s'afficher sur son écran la ligne qu'il venant de saisir $\delta = \{"HelloWorld"\}$ qui est le resultat de l'éxécution inserer("$HelloWorld$"). Puis il supprime le mot "World". Le passé des éxécutions de Bob devient donc inserer("$HelloWorld$") $\cdot$ supprimer("$World$") $= \{"Hello"\}$. Mettant ainsi à jour son état local. Enfin il recoit l'opération d'Alice qui vinet de réaliser une modification en se basant sur un état antérieur à l'état actuel de Bob. L'état qu'il recoit est donc le résultat des opérations inserer("$HelloWorld$") $\cdot$ inserer("$HelloWorld$") $= \{"HelloWorld"\}$. On omet ici une opération que Bob à soumise, qu'il à pu constater comme étant comptabilisé dans le calcul de son état local, mais qui à ensuite était ignoré par le système. La cohérence de l'état local de Bob n'a donc pas était respectée.

Références

[1] LAMPORT. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In : *IEEE Transactions on Computers* C-28.9 (sept. 1979). Conference Name : IEEE Transactions on Computers, p. 690-691. ISSN : 1557-9956. DOI : 10.1109/TC.1979.1675439.