CHAPITRE 03 **:** Problèmes d'accord

SOMMAIRE

CHAPITRE 03 **:** Problèmes d'accord

1. Introduction:

La mise en œuvre et la gestion d'un système réparti sont devenues une tâche difficile et compliquée. Cette difficulté résulte de plusieurs problèmes, parmi ceux-ci : la synchronisation, l'ordre des événements, l'allocation de ressources, la reconfiguration, etc. Beaucoup de taches se fondent sur l'accord des entités distribuées, le consensus est une forme abstraite des problèmes d'accord qui est sujet à des recherches intensives.

Les protocoles d'accord sont au cœur de nombreux systèmes distribués et applications, ils sont utilisés comme un moyen pour assurer la coordination des entités distribuées. La coordination est essentielle pour construire un état cohérent et pour assurer une évolution cohérente du système. Les protocoles d'accord peuvent être utilisé à différents niveaux du système et pour une variété d'objectifs. Ceux-ci peuvent s'étendre sur des services de base d'un système tels que l'établissement de l'ordre total de l'exécution des primitives de communication, la gestion de groupes, la diffusion atomique, la validation atomique d'une transaction, l'élection d'un leader, et la synchronisation de l'horloge.

Dans ce chapitre, nous présentons quelques problèmes d'accord. Par la suite, nous donnons quelques résultats de recherches existants sur la solvabilité du consensus. Nous présentons également la notion d'un détecteur de défaillances et ses caractéristiques. Nous présentons aussi quelques autres classes de détecteurs de défaillances.

2. Des problèmes d'accord :

Nous décrivons dans cette section quatre exemples de problèmes d'accord : la diffusion atomique, la validation atomique, l'élection d'un leader et la gestion de groupes. La résolution de ce type de problème nécessite un accord d'un ensemble de n processus $\Pi = \{p_1, p_2, \ldots, p_n\}$.

2.1 La diffusion atomique :

La diffusion atomique est un paradigme très important dans les systèmes distribués tolérant aux fautes. Informellement, la diffusion atomique exige que tous les processus corrects délivrent le même ensemble de messages dans le même ordre. Formellement, La diffusion atomique est une diffusion fiable qui satisfait :

- Ordre total : si deux processus corrects p et q délivrent deux messages m et m', alors p délivre m avant m' si et seulement si q délivre m avant m'.

La diffusion atomique peut être également définie par les cinq propriétés suivantes :

- **Terminaison :** Si un processus correct diffuse un message m, alors tous les processus corrects délivrent ce message.

- **Validité :** Si un processus délivre un message *m*, alors m a été diffusé par au moins un processus.
- **Intégrité :** un processus remis un message *m* au plus une fois ;
- **Accord :** Si un processus correct délivre un message *m* alors tous les processus corrects délivrent *m* ;
- **Ordre :** Si un processus correct délivre un message *m* avant un message *m',* alors tous les processus corrects délivrent *m* avant *m'*.

Chandra et Toueg [1996] ont montré que le consensus et la diffusion atomique sont deux problèmes équivalents. En d'autres termes une solution à l'un implique automatiquement une solution à l'autre. Informellement une utilisation du consensus pour résoudre la diffusion atomique consiste à effectuer une décision sur les messages à délivrer, et sur l'ordre de ces messages. Par ailleurs, pour résoudre le consensus nous pouvons utiliser la diffusion atomique de la façon suivante : pour décider d'une valeur, un processus la diffuse de façon atomique. Pour décider d'une valeur un processus prend la valeur du premier message qu'il a délivré. Par la propriété d'ordre total de la diffusion atomique tous les processus corrects vont délivrer le même message.

2.2 La validation atomique :

Dans un système réparti où les processus coopèrent par transactions, un problème d'accord doit être résolu entre les processus. Ce problème, appelé validation atomique (AC pour Atomic Commitment), amène les processus à se mettre d'accord sur le résultat d'une transaction : *commit* (validation) ou *abort* (annulation). Plus formellement, la validation atomique est définie par les cinq propriétés suivantes :

- **Unanimité :** deux processus ne peuvent décider différemment ;
- **Validité :** la transaction n'est valide que si tous les processus votent *oui* ;
- **Terminaison :** si toutes les défaillance sont réparées et aucune défaillance ne survient, tous les processus doivent décider ;
- **Non-Trivialité :** les processus doivent décider de valider si tous les votes sont oui et si aucun processus n'est défaillant ou suspecté d'être défaillant ;
- **Non-Blocage :** tout processus correct décide et si certains processus sont défaillants, les autres doivent décider de la terminaison.

Lorsqu'il est demandé que tous les processus corrects finissent par décider en dépit de défaillances d'autres processus le problème est indiqué sous le nom de validation atomique non bloquante (NBAC pour Non-Blocking Atomic Commitment) .

Deux protocoles sont utilisés, le 2PC (Two-Phase Commit) et le 3PC (Three-Phase Commit). Le premier pour résoudre AC et le deuxième pour résoudre NBAC dans les systèmes synchrones.

Guerraoui [1995] a montré que NBWAC (Non-Blocking Weak Atomic Commitment) est réductible au consensus uniforme (qui est réductible au consensus).

Le NBAC implique que la décision est *commit* si tous les processus votent oui et s'il n'y a pas de processus défaillants. Le NBWAC implique que la décision est *commit* si tous les processus votent oui et si aucun processus n'est suspecté.

2.3 L'élection d'un leader :

Dans le problème d'élection d'un leader, à tout moment, au plus un processus se considère comme le leader et un nouveau leader doit être élu si le leader tombe en panne. Pour déterminer plus précisément la notion de coordonnateur (leadership), nous supposons que chaque processus a une copie locale d'une variable distribuée, dénotée par *leader*. La copie de *leader* pour un processus p_i est dénotée par $leader_{pi}$, et pour n'importe quel processus p_i $leader_{pi} \in \{vrai, faux\}$.

Nous disons qu'un processus p_i est le *leader* à l'instant *t*, si p_i n'est pas défaillant à l'instant *t* et $leader_{pi} = vrai$. Formellement, nous définissons le problème d'élection d'un leader par les deux propriétés suivantes :

- **Accord** : à l'instant t il existe un seul processus leader (deux processus ne peuvent pas être leader en même temps).
- **Terminaison :** à tout moment, il existe finalement un leader.

Sabel et Marzullo [1995] ont prouvé que le problème d'élection d'un leader est réductible au problème de consensus. Informellement, une utilisation du consensus pour résoudre l'élection d'un leader consiste à effectuer une décision sur le leader à élire.

3. Une brique de base : le consensus

Le consensus constitue une base pour la résolution des problèmes d'accord. Une résolution pour le consensus implique un élément essentiel pour la résolution des problèmes d'accord. Nous décrivons dans cette section le problèmes de consensus, puis certaines variantes de ce problème : le consensus probabiliste, le consensus uniforme, et le consensus ensemblistes.

CHAPITRE 03 **:** Problèmes d'accord

3.1 Le consensus :

Dans le problème de consensus, chaque processus correct propose une valeur v_i, et tous les processus corrects doivent atteindre une décision unanime et irrévocable sur une valeur *v*. Cette valeur doit avoir été proposée par au moins un processus participant au consensus. Nous implémentons le consensus par l'usage de deux primitives, *propose(v)* et *décide(v)*. Quand un processus exécute *propose(v)*, on dit qu'il propose *v* ; similairement, quand un processus exécute *décide(v)*, on dit qu'il décide sur *v*.

Formellement, le problème de consensus est défini par les trois propriétés suivantes :

- **Terminaison** : tout processus doit finir par décider ;
- **Validité :** si un processus décide une valeur *v*, alors *v* a été proposée par au moins un processus ;
- **Accord :** deux processus ne peuvent décider différemment.

3.2 Le consensus probabiliste :

Pour affaiblir la propriété de terminaison qui ne peut pas être garantie dans un système réparti, Ben-Or [1983] a proposé le consensus probabiliste. Le consensus probabiliste diffère du consensus par la propriété de terminaison.

Formellement, la propriété de terminaison dans le consensus probabiliste, appelée R-terminaison (Random-termination), est définit comme suit :

- **R-terminaison :** tous les processus corrects décident avec une probabilité égale à 1.

Le consensus probabiliste ne garantit pas la terminaison d'un processus correct, mais la probabilité de terminaison converge vers le 1 après une infinité de ronde.

3.3 Le consensus uniforme :

Dans la définition du consensus, les processus défaillants peuvent décidé différemment des processus corrects, cette définition ne convient pas certaines applications qui demandent un niveau de sûreté très élevé. Pour interdire ce défaut, le consensus uniforme a été défini et se diffère du consensus de base par la propriété d'accord :

- **Accord uniforme :** Si un processus décide une valeur *v*, alors tous les processus décident *v*.

Le problème de consensus uniforme est plus difficile à résoudre que le consensus, et il n'a aucun sens dans les systèmes sujet à fautes arbitraires.

3.4 Le consensus ensembliste (le k-consensus) :

Pour une tolérance aux défaillances fortes, Chadauri [1990] a introduit le k-consensus pour affaiblir la propriété d'accord. Le k-consensus autorise la décision sur au plus *k* valeurs . Dans le consensus ensembliste la propriété d'accord est remplacée par :

- **K-accord :** il existe au plus k valeurs différentes décidées par les processus.

Ce problème est plus faible que le consensus standard (plus que k est grand plus que le problème est faible). Une solution pour le k-consensus est une solution pour le (k+1)-consensus. Par conséquent, une solution au consensus (le 1-consensus) est aussi une solution pour le k-consensus. La difficulté de ce problème dépend de la valeur de k :

- pour $k \leq f$: où f est le nombre maximal de processus défaillants. La difficulté de problème dépend de système.
- pour $k > f$: l'algorithme de résolution de problème est simple, quand un processus reçoit une valeur diffusée par k processus, il décide cette valeur.

4. Le résultat de FLP :

La spécification de problème du consensus, présentée dans la section précédente, peut être implémentée dans les deux modèles de systèmes principaux : les systèmes synchrones et les systèmes asynchrones. Cependant, dans un système asynchrone, d'une part le consensus est facile à concevoir, et d'autre part il est difficile à résoudre en présence de défaillance.

Un résultat d'impossibilité de consensus dans un système réparti asynchrone est défini par Fisher, Lynch et Paterson [1985]. En effet, ce résultat montre qu'un consensus n'est pas réalisable de façon déterministe dans un système asynchrone soumis à des crashes de processus, même si le système n'est soumis qu'à une seule défaillance et que les canaux sont fiables. De manière intuitive, ce résultat d'impossibilité est justifié par le fait qu'il est impossible de distinguer un processus lent d'un processus en panne, lorsque l'on ignore les vitesses d'exécution des processus et les délais de transmission des messages.

Ainsi, une des principales difficultés rencontrées dans le domaine de l'algorithmique distribuée tolérante aux fautes concerne ce résultat d'impossibilité dans un environnement asynchrone puisqu'il empêche la résolution de nombreux problèmes d'accord dans de tels environnements. Cependant, différentes alternatives permettent de contourner cette impossibilité.

5. Contourner le résultat d'impossibilité de FLP :

Pour contourner le résultat d'impossibilité de FLP , nombreuses recherches ont été réalisées dans le domaine de l'algorithmique distribuée, afin de permettre la résolution du problème du consensus dans un système asynchrone dans lequel les processus peuvent être soumis à des défaillances franches.

Diverses approches ont ainsi été considérées pour palier à ce problème fondamental :

- La définition de problèmes affaiblis et leurs solutions. Par exemple, le cas avec le problème du K-consensus [1990] qui a été défini pour affaiblir la propriété d'accord et pour faciliter la résolution du consensus.
- La résolution du consensus est également possible dans un environnement partiellement synchrone, tel que le système défini par Dwork et al [1988].
- Une approche élégante a été développée par Chandra et Toueg [1996], à savoir d'enrichir le système par un mécanisme de détection de défaillances non fiable, appelé *détecteurs de défaillances non fiables*. Plusieurs travaux ont été menés autour de cette approche de résolution du consensus.
- Une autre approche récente a été proposée par Mostéfaoui et al [2003], appelée *approche contrainte par conditions*. Elle est basée sur l'identification des conditions, relatives aux valeurs proposables, autorisant une décision directe des processus.
- etc.

6. Les détecteurs de défaillances non fiables :

Une alternative élégante, pour contourner la non résolution du consensus dans les systèmes répartis asynchrones, a été proposée par Chandra et Toueg [1996]. Cette alternative basée sur l'idée d'augmentation du modèle de calculs répartis asynchrone avec un modèle d'un mécanisme externe de détection de défaillances qui peut faire des erreurs. En particulier, ce mécanisme spécifie le concept de détecteur de défaillance non fiable pour un système réparti asynchrone avec des défaillances par arrêt définitif.

Informellement, un détecteur de défaillances non fiable est ou FD-Oracle est un ensemble de modules distribués qui fournissent (possiblement incorrecte) aux processus du système une liste de processus suspectés d'être défaillants. Chaque processus a l'accès à un module local de détecteur de défaillances qui contrôle les autres processus dans le système, et maintient la liste des processus suspectés d'être défaillants. Cette liste peut être différente d'un processus à l'autre. Un module de détecteur de défaillances peut faire des erreurs par la

non suspicion d'un processus crashé ou par l'ajout erroné d'un processus correct à la liste des processus suspectés, c.-à-d., il peut suspecter qu'un processus p est crashé bien que p fonctionne encore. S'il estime plus tard que p est suspecté incorrectement, il peut enlever p de sa liste de processus suspectés. Ainsi, chaque module peut ajouter ou enlever à maintes reprises des processus de sa liste des processus suspectés.

Formellement, un détecteur de défaillances est caractérisé par deux propriétés abstraites : *la complétud*e et *la précision*. En général, *la complétude* exige qu'un détecteur de défaillances doit suspecter tous les processus défaillants, tandis que *la précision* restreint les erreurs (suspections erronées) qu'un détecteur de défaillances peut faire.

Chandra et Toueg ont défini deux propriétés de complétude et quatre de précision, lesquelles sont combinées pour engendrer huit classes de détecteurs de défaillances. Ils ont montré aussi que le problème du consensus peut être résolu avec n'importe quelle classe de ceux-ci.

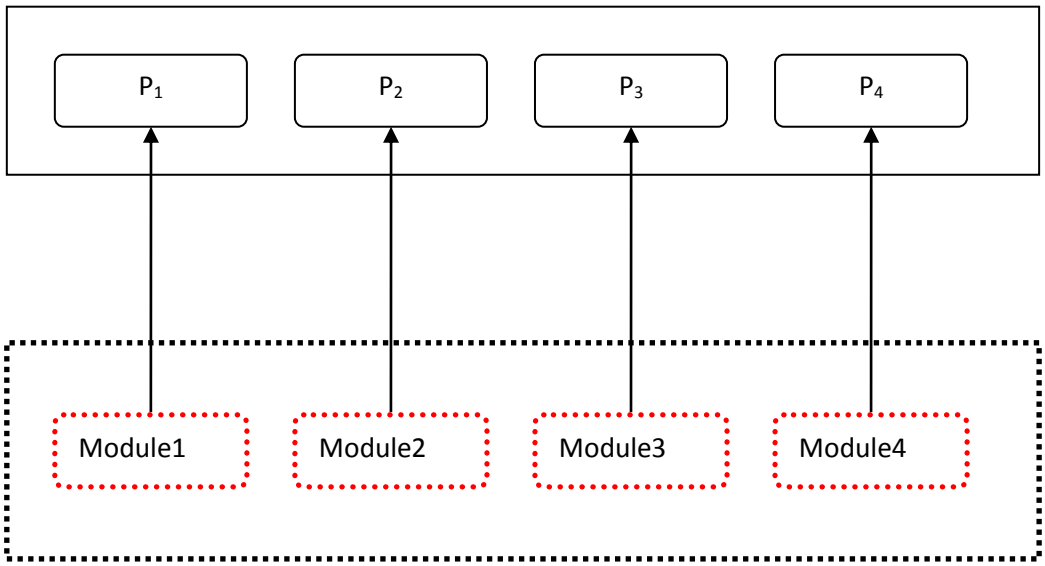

Figure 1. Système enrichi d'un détecteur dé défaillances

6.1 Propriétés :

Chandra et Toueg ont spécifié un détecteur de défaillances en termes de deux propriétés abstraites qui doivent être satisfaites : *la complétude* et *la précision*. Ces deux propriétés sont utilisées comme une base pour la conception des implémentations pour les détecteurs de défaillances et pour prouver la correction de ces implémentations.

- La complétude :

Nous considérons deux propriétés de complétude suivantes :

- **Complétude forte :** tous les processus défaillants finiront par être suspectés définitivement par tous les processus corrects. Formellement un détecteur de défaillances *D* satisfait la complétude forte si :

$$\forall F, \forall H \in D(F), \exists t \in T, \forall p \in crashed(F), \forall q \in correct(F), \forall t' \geq t : p \in H(q,t')$$

- **Complétude faible :** tous les processus défaillants finiront par être suspectés par au moins un processus correct. Formellement un détecteur de défaillances *D* satisfait la complétude faible si :

$$\forall F, \forall H \in D(F), \exists t \in T, \forall p \in crashed(F), \exists q \in correct(F), \forall t' \geq t : p \in H(q,t')$$

Tel que :

F : dénote une fonction de T sur 2^{Π}, où *F(t)* dénote l'ensemble des processus sujet de défaillances à l'instant *t ;*

H : dénote l'historique d'un détecteur de défaillances, elle représente une fonction de $\prod \times T$ sur 2^{Π} , où *H(p,t)* est la valeur du module de détecteur de défaillance du processus *p* à l'instant *t*

T : dénote le rang d'une horloge globale discrète, et prend des valeurs naturelles.

- La précision :

Nous considérons les quatre propriétés de précision suivantes :

- **Précision forte :** aucun processus correct n'est suspecté. Formellement, *D* satisfait la précision forte si :

$$\forall F, \forall H \in D(F), \forall t \in T, \forall p, q \in \Pi - F(t) : p \notin H(q,t)$$

- **Précision faible :** il existe au moins un processus correct qui n'est jamais suspecté. Formellement, D satisfait la précision faible si :

$$\forall F, \forall H \in D(F), \exists p \in correct(F), \forall t \in T, \forall q \in \Pi - F(t) : p \notin H(q,t)$$

- **Précision inéluctablement forte :** il existe un instant *t* à partir duquel aucun processus correct n'est suspecté. Formellement, D satisfait la précision inéluctablement forte si :

$$\forall F, \forall H \in D(F), \exists t \in T, \forall t' \geq t, \forall p, q \in correct(F) : p \notin H(q,t')$$

- **Précision inéluctablement faible :** il existe un instant *t* à partir duquel au moins un processus correct n'est jamais suspecté. Formellement, D satisfait la précision inéluctablement faible si :

$$\forall F, \forall H \in D(F), \exists t \in T, \exists p \in correct(F), \forall t' \geq t, \forall q \in correct(F) : p \notin H(q,t')$$

6.2 Classification :

En combinant les deux propriétés de complétude et les quatre propriétés de précision, Chandra et Toueg [1996] définissent huit différentes classes de détecteurs de défaillances (Figure 3.2). Hors de ceux-ci, Chandra, Hadzilacos et Toueg ont montré dans [1996] que ◊W est la classe la plus faible de détecteurs de défaillance exigée pour résoudre le consensus. Ils ont aussi montré qu'un détecteur de défaillances de la classe ◊*S* et un autre de la classe ◊W sont équivalents, c.-à-d., l'un peut être transformé à l'autre et vice versa.

Complétude	**Précision**			
	Forte	Faible	Inéluctablement Forte	Inéluctablement Faible
Forte	Parfait *P*	Fort *S*	Inéluctablement Parfait ◊*P*	Inéluctablement Fort ◊*S*
Faible	Quasi-parfait *Q*	Faible W	Inéluctablement Quasi-parfait ◊*Q*	Inéluctablement Faible ◊W

Figure 2. L es classes de détecteurs de défaillances

La transformation de la classe ◊*S* à ◊W est triviale, puisque ◊*S* est une sous-classe de ◊W. Pour transformer ◊W à ◊*S*, Chandra et Toueg ont proposé dans un algorithme distribué qui transforme la complétude faible à une complétude forte, en préservant la précision. Il en résulte donc des équivalences entre les classes ($P \equiv Q$, $S \equiv$ W, ◊$P \equiv$ ◊Q, ◊$S \equiv$ ◊W).

Cependant, un détecteur de défaillance est plus robuste s'il se base sur un algorithme plus simple. Par exemple, un algorithme qui résout le consensus est beaucoup plus simple si nous prenons un détecteur de défaillance de la classe ◊*S* que si on prend un détecteur de défaillances de la classe ◊W. clairement, le deuxième algorithme doit faire face au manque de complétude de ◊W en ce qui concerne ◊*S*.

7. Résolution du consensus en utilisant des détecteurs non fiables

Chandra et Toueg [1996] ont proposé un algorithme de consensus basé sur un détecteur de défaillances de la classe ◊*S*. La résolution du problème du consensus grâce à un détecteur de défaillances appartenant à cette classe ne sera réalisable que sous une condition essentielle.

Ainsi, l'hypothèse à laquelle le système doit répondre est que plus de la moitié des processus du système doivent être corrects, c'est à dire qu'ils ne doivent subir aucune défaillance.

Le principe de cet algorithme repose sur le paradigme du coordonnateur rotatif, à savoir que l'algorithme se décompose en diverses rondes asynchrones chacune étant gérée par un processus différent, le processus coordonnateur de la ronde. Ce processus coordonnateur est le processus c = (r mod N) + 1 où r représente la ronde courante et N est le nombre de processus du système. Ainsi, à une ronde donnée, tous les messages en transit sont soit à destination du processus coordonnateur soit en sa provenance.

Chaque ronde peut être décomposée en 4 phases asynchrones. Durant la première phase, chaque processus du système envoie au coordonnateur son estimation de la valeur de décision affranchie avec le numéro de l'étape durant laquelle il a choisi cette estimation.

La phase 2 consiste en la réception par le processus coordonnateur de (N +1)/2 messages envoyés durant la phase 1. Après avoir rassemblé ces diverses estimations, le processus coordonnateur sélectionne celle ayant le plus grand affranchissement et envoie cette nouvelle estimation à tous les processus du système.

Chaque processus doit désormais recevoir cette nouvelle estimation. Cette réception ainsi que son traitement représentent la troisième phase qui offre deux possibilités distinctes. Soit le processus qui reçoit cette nouvelle estimation envoie un acquittement au coordonnateur pour signifier qu'il adopte cette estimation. Soit après avoir consulté son module de détection de défaillances, le processus qui reçoit cette nouvelle estimation (ou ne la reçoit pas) suspecte la défaillance du coordonnateur, et envoie un "*nack*" à ce même coordonnateur.

Enfin, pour la quatrième phase, le coordonnateur attend (N + 1)/2 réponses ("*ack*" ou "*nack*"). Si toutes les réponses sont des acquittements, alors le coordonnateur sait que la majorité des processus a retenu cette nouvelle estimation, il envoie donc à tous les processus la décision sur cette estimation. Sinon, si le coordonnateur a reçu un "*nack*", il n'envoie pas de message de décision et passe directement à la ronde suivante.

Une bonne caractéristique des algorithmes basés sur la classe ◊*S* est qu'ils sont sûrs, c.-à-d., ils préservent la sûreté même si ni la complétude forte ni la précision inéluctablement faible est satisfaite, en particulier, même s'il y a une défaillance qui n'est jamais détecté par

aucun processus et que tous les processus corrects sont à plusieurs reprises (et pour toujours) faussement suspectés.

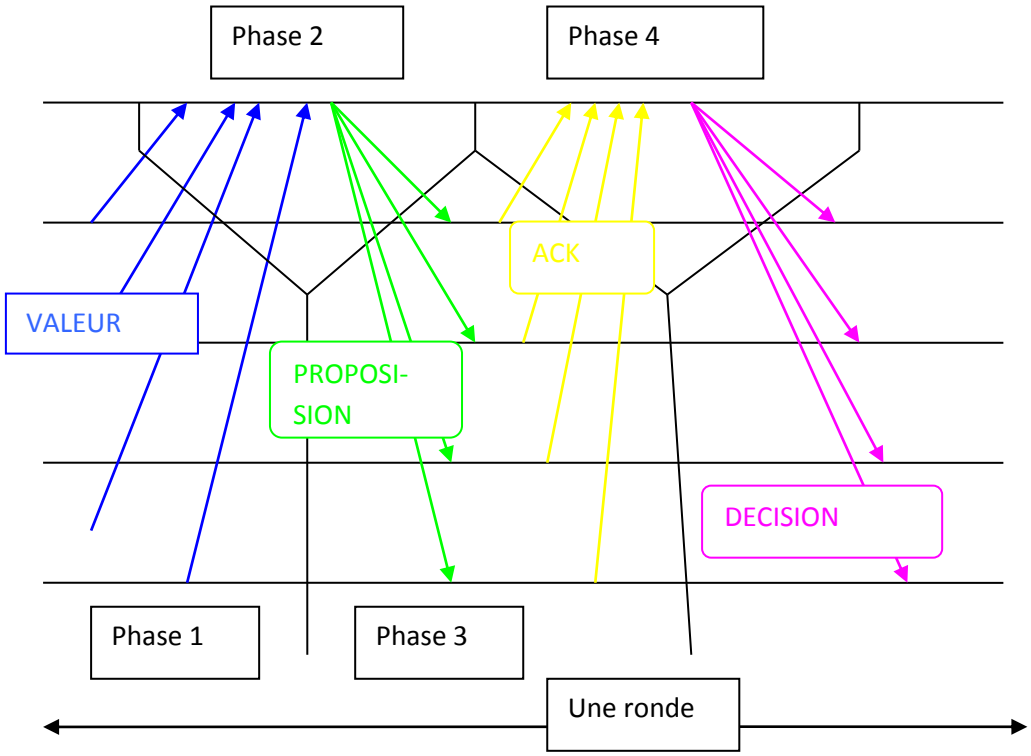

Figure 3. Une ronde de l'algorithme de Chandra et Toueg