Université Paris VII- Denis Diderot

Habilitation à diriger des recherches

Algorithmique distribuée et application à l'étude des modèles en rondes et des modèles avec détecteurs de défaillances

Soutenue le 18 décembre 2001

par

Hugues Fauconnier

Jury :	Robert CORI	Professeur (rapporteur)
	Paul GASTIN	Professeur (rapporteur)
	Jean-Michel HÈLARY	Professeur (rapporteur)
	Maurice NIVAT	Professeur
	Gérard LE LANN	Directeur de recherche
	Sam TOUEG	Professeur (rapporteur)

1 Introduction

L'essentiel de mes travaux porte sur l'algorithmique distribuée résistante aux pannes. On peut envisager l'algorithmique de plusieurs points de vue. Tout d'abord, bien entendu, il s'agit de résoudre, de façon si possible efficace, une certaine spécification. Mais, l'algorithmique peut aussi servir à des considérations moins pratiques, comme, par exemple, montrer que tel problème est réductible à tel autre, ou même qu'il n'existe pas d'algorithme pour le résoudre.

Plutôt que d'exposer les divers algorithmes correspondant à mes travaux, je vais les présenter dans un cadre plus général. Je vais essayer de montrer comment ces algorithmes, ainsi que d'autres, essaient de répondre (partiellement) à la question plus générale : quels sont les modèles les plus intéressants pour l'algorithmique distribuée ? Il ne s'agit pas uniquement d'une préoccupation théorique, et précisons tout de suite que le terme modèle peut aussi s'entendre dans un sens très pratique : les divers systèmes dits de *Middleware* sont finalement des modèles de calcul.

Il n'est plus nécessaire d'insister sur l'importance de l'algorithmique distribuée. Actuellement, il y a peu d'applications qui, d'une manière ou d'une autre, peuvent fonctionner sans coopération avec d'autres et le développement des réseaux est sans doute un des aspects principaux de l'évolution récente de l'informatique.

Cette distribution pose des problèmes en partie différents de ceux de l'algorithmique classique : il s'agit de permettre la coopération entre les processus/processeurs, répartir les tâches, répartir les données, communiquer etc...

Outre la complexité que cela entraîne, se pose le problème de la fiabilité. Maintenir la continuité du service est le plus souvent crucial pour des raisons de sécurité (avionique, nucléaire etc...) ou pour des raisons économiques (transactions boursières ou financières etc...), mais dans tous les cas cela est souhaitable. Si un service dépend de plusieurs composants, il devient difficile de pouvoir garantir que tous ces composants seront toujours sans défaillances. Il est d'autre part clair que si un service est assuré par un seul processeur, si ce processeur tombe en panne, il n'y pas d'espoir de pouvoir maintenir le service... Pour espérer la fiabilité on est donc amené naturellement à dupliquer les composants : c'est le principe de la redondance qui est à la base de l'algorithmique distribuée résistante aux défaillances.

De nouveaux problèmes se posent : par exemple, obtenir un accord entre processus, problème le plus élémentaire du distribué, sans lequel une coopération est difficilement envisageable, devient en présence de pannes difficile, voire impossible. Le *problème du consensus* (figure 1) est fondamental : en présence de plusieurs processeurs et/ou capteurs, en préliminaire à tout algorithme, il faut arriver à se mettre d'accord. Sans cet accord minimal, pas grand chose n'est possible : par exemple, pas de diffusion atomique (figure 2), pas de duplication active ou passive, pas de *commit* et, donc, pas de bases de données distribuées...

Dans les années 80, l'essentiel des recherches en algorithmique distribuée

résistante aux pannes concernait une variante du consensus : le problème des généraux byzantins [71]. Ces recherches supposaient un univers synchrone pour lequel les bornes sur les temps de calcul et sur les délais de communication sont connues. Pour ce problème, un général donne un ordre, il s'agit d'assurer que, si ce général n'est pas un traître, tous les autres généraux qui ne sont pas des traîtres s'accordent sur cet ordre et sinon qu'ils s'accordent au moins sur la même chose. Dans ce problème les traîtres ont tous les droits.

Dans un univers synchrone cet accord est possible si suffisamment de processus sont sans défaillances (même s'il peut, dans certains cas être coûteux), mais, dès que le système est asynchrone (pas de bornes sur les délais de communication), le résultat de Fischer, Lynch et Paterson [52] (1985) montre que le consensus même avec au plus un processus défaillant est impossible, et ce, même en restreignant le type de panne (un processus en panne s'arrête).

Et pourtant, il est la plupart du temps difficile voire impossible de considérer les systèmes existants comme synchrones : comment fixer des bornes sur les délais de communications ? Alors, il faut contourner ce résultat d'impossibilité. Pour cela, il existe plusieurs possibilités.

La première, que nous n'étudierons pas ici, consiste à modifier un peu la spécification du consensus et à assurer uniquement un consensus *probabiliste*.

La deuxième consiste à restreindre le caractère asynchrone du système. Par exemple, on peut supposer qu'il existe une borne sur les délais de communication qui est, disons, souvent assurée. Il faut, alors, définir correctement ce que signifie ce "souvent".

Enfin, une autre solution est de supposer que l'on dispose d'informations supplémentaires sur le système. Par exemple, on peut supposer que l'on dispose d'un mécanisme supplémentaire permettant de savoir (pas forcément de façon sûre) si un processus est en panne : on a un *détecteur de défaillances*. D'ailleurs, cela correspond souvent à la démarche du programmeur : il utilise généralement des *time-out*, et si le délai est dépassé il considère que le processus est mort. Reste à assurer que si le processus n'est pas vraiment mort l'algorithme fonctionne encore... L'intérêt de cette approche est qu'elle est suffisamment abstraite et permet de séparer l'algorithmique de la détection des pannes. On peut, en particulier, caractériser les propriétés de la détection de pannes qui permettront d'assurer le consensus.

Cependant, ces mécanismes de détection de pannes peuvent apparaître comme étant de trop bas niveau. Le programmeur ou le concepteur d'algorithmes distribués résistants aux pannes peut préférer soit d'autres mécanismes (pourquoi pas, par exemple, avoir un mécanisme qui fournirait directement le consensus ou la diffusion atomique) soit encore un système de *middleware* dans lequel figureraient encore d'autres mécanismes (par exemple la possibilité de créer et de gérer des groupes de processus).

Bien sûr, le problème qui reste est que toutes ces solutions ne peuvent pas être implémentées dans des systèmes asynchrones. Il faudra donc sortir, au moins partiellement, du caractère asynchrone du système et, d'une certaine manière, supposer des propriétés supplémentaires. Il faut qu'elles apparaissent *explicitement* là où elles sont utilisées et que l'on puisse les spécifier de manière précise.

d_p décision du processus p (si elle existe).
v_p valeur proposée par p.

Intégrité : d_p est une des valeurs proposées ($d_p \in \{v_1, \ldots, v_n\}$)

Accord : si p et q décident alors $d_p = d_q$,

Terminaison : tout processus correct p décide un jour,

FIG. 1 – Consensus (uniforme).

Un des inconvénients des systèmes de *middleware* existants est que, souvent, ils ne rendent pas explicites les conditions qu'ils supposent sur le système sous-jacent.

Dans ce qui suit, on va essayer de comparer ces diverses approches pour résoudre le consensus. Comme on vient de le voir, il y a trois catégories d'alternatives pour résoudre le consensus. La première consiste à considérer des *modèles de systèmes* partiellement synchrones pour lesquels le consensus est possible. On reste à un bas niveau en travaillant avec le temps comme en synchrone, mais les bornes ne seront pas connues ou, si elles le sont, ne seront pas respectées tout le temps (mais suffisamment souvent). La deuxième, garde le système asynchrone tel quel, mais rajoute un mécanisme externe en supposant l'existence *d'oracles* (ici des détecteurs de défaillances). On garde ainsi les avantages des systèmes asynchrones (ce qui ajouté est externe) mais, bien sûr, se posera naturellement la question de la réalisation de ces oracles. La troisième alternative définit des *modèles de calcul* et se place à un niveau plus abstrait : le modèle de calcul définit comment les processus s'exécutent. La réalisation du modèle de calcul lui-même est reléguée à plus tard. Les systèmes de *middleware* correspondent à cette approche.

On va étudier et comparer les divers modèles proposés dans la figure 3. Pour l'essentiel, les modèles de calcul considérés ici sont des modèles que l'on appellera modèles de calcul en rondes. Dans ces modèles les processus essaient d'avoir un comportement le plus synchrone possible basé sur la ronde (*round*) : à chaque ronde tout le monde envoie un message à tout le monde et attend d'avoir reçu les messages de la ronde avant de passer à la ronde suivante. Pour les systèmes asynchrones, on étudiera un modèle de calcul particulier, les *systèmes de phases synchronisées*, qui correspond à un *middleware* restreint développé dans le projet ATR.

Cette étude a pour but, d'une part, d'essayer de clarifier les choses et, d'autre part, de réunir les éléments qui permettent de comparer ces approches. Pour mener à bien cette comparaison pratique, il faudra d'abord définir, ce qui n'est pas si simple, un cadre formel suffisamment précis. Essentiellement, ces comparaisons correspondent à des réductions et des implémentations, c'est-à-dire des algorithmes. Bien entendu, chacune de ces approches présente son intérêt propre, mais il est utile aussi bien d'un point de vue pratique que d'un point de vue théorique de connaître la puissance de résolubilité et l'efficacité de chacune.

Cette première partie commencera par la définition des divers modèles considérés. On y mettra en place les éléments qui permettront cette étude, en particulier la notion de simulation et de traduction. On y montrera aussi les premiers résultats simples. Ensuite, on étudiera plus particulièrement l'utilité de l'usage des détecteurs de défaillances dans un cadre qui reste essentiellement synchrone : *a priori*, les détecteurs de défaillances ont surtout été définis pour les systèmes asynchrones et la question de leur utilité dans un cadre synchrone se pose. Perd-on quelque chose en les utilisant dans ce cadre ? Enfin, nous montrerons que les divers systèmes de *virtual synchrony* correspondent à des modèles de calcul pour l'asynchrone. Nous présenterons enfin divers avantages des systèmes de phases synchronisées qui correspondent à des modèles de calcul essayant d'être aussi synchrones que possible tout en restant dans un cadre asynchrone.

Dans un deuxième temps, je présenterai divers algorithmes que j'ai développés en insistant sur leurs particularités. On y retrouvera, présentés avec un autre point de vue, un certain nombre des algorithmes qui sont utilisés dans la première partie consacrée à la comparaison et l'étude des modèles.

On y verra l'algorithmique développée autour des systèmes de phases synchronisées : la préoccupation est, ici, de donner des conditions explicites sur le système sous-jacent de façon à assurer des propriétés temps réel de la diffusion atomique.

Ensuite, nous présenterons un algorithme qui permet, en utilisant des détecteurs de défaillances parfaits, de réaliser le consensus avec la *même* efficacité que dans un système synchrone. L'intérêt de cet algorithme est ainsi de montrer que l'approche des détecteurs de défaillances se justifie aussi dans ce cadre synchrone.

Nous verrons ensuite, un algorithme de multi-diffusion ordonnée. Cet algorithme montre qu'une approche centralisée peut souvent être efficace. Cet algorithme montre aussi comment on peut, sous certaines conditions, passer d'une algorithmique sans pannes à une algorithmique résistante aux pannes.

L'algorithme suivant réalise la diffusion générique : il s'agit de limiter l'usage de l'oracle à son strict nécessaire et d'éviter de faire plus que ce qui est demandé.

Enfin le dernier algorithme présenté est un algorithme d'élection ultime de leader : le but est d'obtenir un algorithme efficace dans un système partiellement synchrone qui ne change de leader que si cela s'avère nécessaire.

En l'absence d'un formalisme universellement reconnu, il est difficile d'être à la fois rigoureux et de faire comprendre les intuitions essentielles. Je vais essayer de présenter ces divers travaux en évitant d'être trop formel. La plupart du temps, je renverrai le lecteur aux articles correspondants. Cependant, la présentation de certaines sections ne correspondant pas exactement à des travaux actuellement publiés, je donnerai alors dans les introductions de ces sections une présentation plus informelle permettant de comprendre les principales intuitions tout en donnant plus de détails techniques dans la suite du texte.

La diffusion atomique est définie par deux primitives *A-diffuse* et *A-délivre* qui vérifient :

Validité : si un processus correct A-diffuse m, tous les processus corrects A-délivrent m,

Accord Uniforme : si un processus A-délivre m alors tous les processus corrects A-délivrent m,

Intégrité Uniforme : m est A-délivré au plus une fois par chaque processus uniquement si m a été A-diffusé

Ordre : si p et q A-délivrent m et m' alors ils A-délivrent m et m' dans le *même* ordre.

(sans la propriété d'ordre on a une *diffusion fiable*)

FIG. 2 – Diffusion atomique.

2 Les divers modèles

2.1 Introduction

Comme il s'agit de comparer diverses approches pour les systèmes distribués, la première chose à faire est de donner le cadre de cette comparaison : c'est le but de cette section.

Tout d'abord, il faut définir les modèles que l'on va considérer. Ensuite, il faut préciser comment peut se faire cette comparaison. On va, dans cette introduction, essayer de présenter les grandes lignes de cette approche de façon intuitive, en laissant les définitions et propriétés plus formelles aux sections suivantes. Les divers modèles considérés sont résumés dans la figure 3. Les détails des caractéristiques de ces divers modèles se trouveront dans les sections suivantes.

2.1.1 Systèmes, modèles de calcul, systèmes avec oracles

On peut regrouper les divers types de modèles que l'on va considérer en trois grandes catégories : les *modèles de systèmes*, les *modèles de calcul* et les *modèles de systèmes avec oracle*.

Cette distinction est un peu arbitraire, mais, essentiellement, un modèle de système décrit un système réel défini par ses caractéristiques (par exemple, temps de calcul, délai de transmission des messages, types de défaillances etc...) ; à ce système, on peut ajouter un mécanisme particulier, appelé oracle, défini formellement par sa spécification, que les processus peuvent consulter. L'oracle enrichit le système en lui permettant de résoudre certains problèmes (essentiellement les problèmes d'accord). Ajouter un oracle correspond à ajouter une nouvelle primitive au système en faisant abstraction de la façon dont elle peut être réalisée dans le système. Le choix de cet oracle doit donc être un compromis entre sa puissance et les conditions de sa réalisation dans un système. Enfin, un

Modèles de calcul		Modèles avec oracle		Modèle de systèmes	
RPS	calcul en rondes parfaitement synchronisées : tous les processus reçoivent les mêmes messages à chaque ronde			$Sync+$ RB	Systèmes synchrones avec diffusion fiable : messages reçus par tous ou par aucun, les messages respectent la borne δ
RS	calcul en rondes synchronisées : tous les processus reçoivent les mêmes messages *des corrects* à chaque ronde			$Sync$	Systèmes synchrones : les messages respectent δ
RAP	calcul en rondes contrôlées par un détecteur de défaillances $\mathcal{P}$	$\mathcal{P}$	détecteur de défaillances parfait : processus corrects jamais suspectés, processus incorrects finissent par être définitivement suspectés		
RAS	calcul en rondes contrôlées par un détecteur de défaillances $\mathcal{S}$	$\mathcal{S}$	détecteur de défaillances Fort : *un certain* processus correct n'est jamais suspecté, processus incorrects finissent par être définitivement suspectés	$Fort$	tous les messages issus d'un certain processus correct respectent δ
SPS	Systèmes de phases synchronisées	$\diamond\mathcal{P}$	détecteur de défaillances *éventuellement* parfait	$USync$	Ultimement synchrone : à partir d'un certain τ, tous les messages respectent δ
$Leader$	Système avec leader ultime	Ω	ultimement tous les corrects ont confiance dans le même processus correct		
		$\diamond\mathcal{S}$	détecteur de défaillances ultimement Fort	US	Ultimement fort : à partir d'un certain τ, tous les messages issus d'un certain processus correct respectent δ

FIG. 3 – Les modèles considérés.

modèle de calcul correspond à une certaine organisation des exécutions et à une abstraction par rapport au système. Un modèle de calcul est censé simplifier l'écriture des algorithmes. La question qui se pose est de savoir si un système donné peut *implémenter* un modèle de calcul et si cette implémentation est efficace.

Par exemple, un système synchrone au sens usuel correspond à un modèle de système défini par certaines caractéristiques de ses comportements comme l'existence d'une borne connue sur les délais de communication et d'une borne sur le temps des pas de calculs. Un modèle de calcul communément utilisé est celui des rondes synchronisées (*synchronized rounds*) : les exécutions sont organisées en ce que l'on appellera des rondes synchrones, à chaque ronde tous les processus émettent et reçoivent de tous les processus. Ce modèle en rondes est communément utilisé car il diminue l'ensemble des exécutions considérées et permet d'éliminer la référence au temps réel qui est remplacé par un temps virtuel correspondant aux rondes. D'autre part, comme il est assez clairement réalisable dans les systèmes synchrones, on peut transformer un algorithme fonctionnant dans un modèle de rondes synchronisées en un algorithme fonctionnant en synchrone. Donc, si on peut résoudre un problème dans ce modèle de calcul, on en déduit que l'on peut aussi le résoudre dans un système synchrone. Mais, réciproquement, pour une large classe de problèmes, on peut aussi montrer que s'il n'existe pas d'algorithmes dans le modèle de calcul en rondes synchronisées, il n'en existe pas non plus dans un système synchrone. C'est la raison pour laquelle on identifie parfois les systèmes synchrones avec le modèle de calcul des rondes synchronisées.

Rappelons que le consensus est impossible en asynchrone alors que les problèmes d'accord sont fondamentaux pour l'algorithmique tolérante aux pannes. Il faut donc contourner ce résultat d'impossibilité. Pour cela, on peut restreindre le caractère asynchrone du système et définir d'autres modèles de systèmes : les modèles partiellement synchrones. Mais, au lieu de changer de système sous-jacent, on peut préférer maintenir l'asynchronisme du système, et ajouter au système un *oracle* permettant de résoudre les problèmes de consensus. Par exemple, cet oracle peut être un *détecteur de défaillances* [21] donnant des informations (imparfaites) aux processus sur les pannes. On a donc alors des modèles de systèmes augmentés d'un oracle particulier. On peut alors développer une algorithmique en supposant l'existence de détecteurs de défaillances. Il s'agira alors de déterminer quelles doivent être les propriétés de ces détecteurs de défaillances pour assurer les propriétés des algorithmes. On pourra ensuite, déterminer des conditions sur le système nécessaires pour la réalisation de ces détecteurs de défaillances et envisager leur implémentation.

2.1.2 Modèles

Les modèles de systèmes que nous allons considérer sont les systèmes *synchrones*, les systèmes *asynchrones* et les systèmes *partiellement* synchrones. A un extrême, en synchrone, il existe des bornes connues sur le temps des pas de calcul et des délais de communication, tandis qu'à l'autre extrême, en asynchrone, il

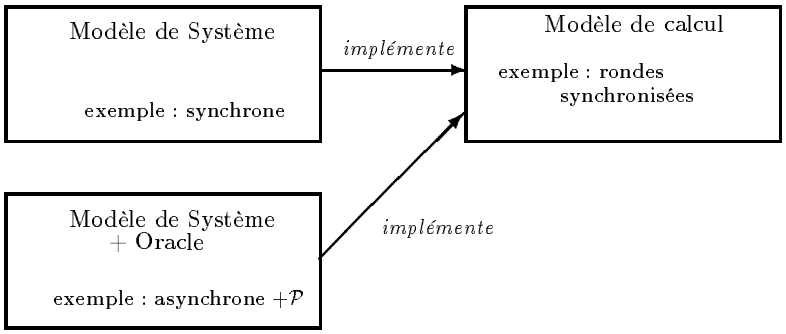


FIG. 4 – Schéma général : Systèmes, Modèles de calcul, Systèmes avec oracles.

n'existe pas de telles bornes (en tout cas pour les délais de communication). Les modèles de systèmes partiellement synchrones sont entre les deux : les bornes peuvent, par exemple, exister mais ne pas être connues. Précisons de plus que, sauf mention explicite du contraire, les seuls défaillances considérées ici sont les pannes franches des processus : un processus en panne s'arrête définitivement.

L'idée de base des modèles de calcul considérés ici est la *ronde* (*round*). Dans une ronde, les processus essaient de travailler de la façon la plus synchrone possible. En quelque sorte, les divers modèles en rondes essaient de capter le synchronisme des divers systèmes. A un extrême, les rondes seront parfaites : tous les processus font tout en même temps (dans la même ronde), à l'autre extrême, les rondes sont totalement désynchronisées et il peut y avoir, entre deux actions d'un processus un nombre arbitraire d'actions d'autres processus. En un certain sens, la démarche consistant à essayer d'avoir un modèle de calcul le plus synchrone possible, est proche de celle de la *virtual synchrony*[1]. Et, outre l'intérêt concernant l'étude de la puissance relative des divers systèmes, cette approche a un intérêt pratique : *a priori*, il est plus facile de développer une algorithmique dans un monde synchrone (ou ressemblant au synchrone) que dans un monde totalement asynchrone. Le projet ATR [12, 11, 24] est un exemple de cette approche.

Concernant les modèles de systèmes avec oracles, nous nous restreignons aux systèmes asynchrones utilisant des détecteurs de défaillances. Un des avantages de ces détecteurs de défaillances est qu'ils représentent une approche plus abstraite en séparant la part de ce qu'il faut ajouter au système pour obtenir le consensus du système lui-même qui reste asynchrone : l'oracle fournit ce qu'il faut en plus pour réaliser le consensus. De plus, Chandra, Hadzilacos et Toueg ont défini une hiérarchie de ces détecteurs et ont caractérisé les détecteurs de défaillances les plus faibles permettant de résoudre le consensus [19]. On pourrait aussi envisager d'utiliser d'autres types d'oracles. Par exemple, on pourrait

[1]Mais en éliminant beaucoup de propriétés, comme, par exemple, celles concernant la connaissance des groupes de processus, généralement supposée dans les divers systèmes de virtual synchrony !

comme dans [1] (discutée dans la section 6.5), supposer l'existence d'un oracle réalisant directement le consensus, ou d'un oracle réalisant la diffusion atomique. D'un point de vue pratique, le choix de l'oracle correspond à définir quelle est, en quelque sorte, la brique de base supplémentaire permettant de résoudre les problèmes d'accord.

2.1.3 Comparaisons

Une fois définis les modèles, il reste à définir comment les comparer. Le problème est réel : on verra plus loin que des résultats apparemment contradictoires sont parfois annoncés. Tant qu'on en reste au niveau informel, on peut affirmer une chose et son contraire. Mais, d'un autre côté, une approche formelle est nécessairement lourde. Dans la section 2.6 on donnera tous les éléments permettant de mettre en oeuvre cette approche formelle, mais en évitant d'entrer dans les détails. On en restera, pour l'instant, aux grandes lignes...

L'intérêt de comparer ces types de modèles n'est pas uniquement théorique. Supposons que l'on dispose d'un certain système, il s'agit, par exemple, de répondre à la question : quel est, suivant un certain critère, le meilleur modèle de calcul pour ce système suivant le genre d'applications que l'on veut développer ?

Comment comparer la puissance des modèles ? La première difficulté est que, par définition, ces modèles sont différents. Par exemple, la notion de temps joue un rôle *a priori* fondamental pour les systèmes synchrones et, par définition aussi, n'a pas réellement de sens dans des systèmes asynchrones. Pour comparer les divers modèles il faudra donc oublier certaines de leurs caractéristiques : en ce sens toutes les comparaisons se feront modulo des relations de *simulation*.

En ignorant le temps, il est clair qu'une exécution synchrone peut toujours être considérée comme une exécution asynchrone. De cette façon, les systèmes synchrones peuvent être considérés comme des cas particuliers des systèmes asynchrones. Ainsi, si un algorithme asynchrone résout une spécification, cet algorithme exécuté dans un système synchrone résoudra aussi cette spécification. Cependant, dans le cas général il faudra transformer l'algorithme et donc définir une notion de *traduction*.

Comme on le verra, les modèles de systèmes considérés ici correspondent assez directement à des modèles de calcul et, au lieu de comparer des modèles de systèmes, on va, plutôt, comparer les modèles de calcul. Il s'agit d'étudier dans quelle mesure on peut en simuler un par un autre. Ces simulations étant constructives, elles ont un intérêt pratique clair. Elles sont utilisées en particulier dans des techniques de masquage d'erreurs (par exemple, [14, 82]). Bien entendu, les performances et le coût de ces simulations deviennent un des critères les plus importants.

Enfin, pour évaluer les avantages des approches utilisant les oracles, il est particulièrement intéressant de déterminer quels sont les modèles de calcul correspondant aux diverses classes d'oracles. Et, on peut associer assez directement certaines classes de détecteurs de défaillances à des modèles de calcul. Une autre question importante est celle de l'implémentation des détecteurs de défaillances : dans quels types de modèles partiellement synchrones peuvent-ils être réalisés

et à quel prix ?

2.2 Généralités

Commençons par préciser quelques caractéristiques générales des systèmes considérés. $P = \{p_1, \ldots, p_n\}$ est l'ensemble des processus. On suppose toujours pour tous les systèmes considérés ici que :

- la communication se passe par l'intermédiaire de *canaux de communication* : quand p envoie un message m à q, il écrit m dans le canal de p à q ; recevoir m pour q consiste à extraire le message m du canal de p à q. Il s'agit donc d'une communication point à point par l'intermédiaire de canaux unidirectionnels. Sauf mention explicite du contraire, ces canaux sont FIFO et fiables [2].
- le graphe de communication est complet : il existe toujours un canal de communication de p vers q.
- on suppose aussi des propriétés d'équité : si p ne tombe pas en panne dans une exécution α, alors p exécute une infinité d'actions et si p et q sont deux processus corrects (c'est-à-dire ne tombant pas en panne), alors tout message écrit dans le canal de p à q sera consommé.

Pour simplifier, les seules pannes considérées ici sont des pannes définitives (*crash*) de processus. Il s'agit d'une restriction importante puisque l'on exclut ainsi, d'une part, la possibilité de reprise d'un processus après une panne, et, d'autre part, les pannes byzantines. Cependant, une extension à des types de pannes plus sévères pourrait être, dans certains cas envisagée.

Au temps τ, un processus peut être soit *vivant* soit *mort*. Dans une exécution, un processus mort au temps τ n'exécutera plus aucun pas de calcul après τ (la mort est définitive). Un processus vivant pour tout temps τ est dit *correct* et *incorrect* dans le cas contraire. On suppose toujours que les processus ne terminent pas. Aussi, avec les conditions d'équité du système, un processus incorrect n'effectue qu'un nombre fini de pas de calculs, alors qu'un processus correct effectue une infinité de pas de calculs. En d'autres termes, un processus est incorrect dans une exécution α si et seulement si il n'exécute qu'un nombre fini de pas de calculs pour α. Insistons sur le fait que, pour un processus p, être correct est une propriété *ultime* (pour tout τ, p est correct). Par contre, être vivant (ou mort) est une propriété définie à un instant τ.

Dans la suite, en général, n est le nombre (fixe) de processus du système, t représente le nombre de pannes tolérées par le système, f le nombre de processus en panne dans une exécution donnée : n et t sont des paramètres fixes du système considéré, alors que f est lui dynamique. On a toujours : $0 \leq f \leq t < n$.

[2] Contrairement à la deuxième, la première hypothèse n'est, généralement, pas nécessaire : on peut toujours, même en présence de pannes, assurer une communication FIFO.

2.3 Modèles de systèmes

2.3.1 Systèmes synchrones

En général, un système synchrone est un système pour lequel il existe un temps universel connu des processus, et tel qu'il existe une borne connue pour le temps d'un pas de calcul d'un processus et une borne connue pour les délais de communication. Ces trois éléments (horloge, borne sur les pas de calcul, borne sur les communications) caractérisent les systèmes synchrones. Pour éviter les aberrations, il faut aussi supposer qu'il existe une borne inférieure strictement positive pour les pas de calcul des processus.

2.3.2 Systèmes asynchrones

Dans un système asynchrone, il n'existe pas de bornes connues ni sur les temps de communication, ni sur le temps des pas de calcul. On suppose aussi généralement qu'il n'y a pas d'horloge globale accessible des processus. Des conditions d'équité permettent d'assurer que, dans toute exécution, les processus corrects font une infinité de pas de calcul. De même, les canaux étant supposés fiables, un message émis d'un processus correct vers un processus correct sera toujours reçu.

2.3.3 Systèmes partiellement synchrones

Le résultat d'impossibilité du consensus amène à restreindre l'asynchronisme des systèmes asynchrones. On peut définir de nombreux modèles partiellement synchrones dans lesquels le consensus peut être résolu (par exemple [43] ou [47]). Nous allons nous restreindre aux systèmes pour lesquels il existe une borne connue pour les pas de calculs des processus. On supposera aussi que les processus ont accès à une horloge parfaitement synchronisée [3]. Afin de pouvoir réaliser le consensus, on supposera l'existence d'une borne supposée δ sur les délais de communication.

Les différentes approches diffèrent par la façon dont le système se comporte par rapport à cette borne. Pour l'essentiel on peut s'intéresser soit au comportement ultime du système, soit aux périodes pendant lesquelles les bornes sont respectées. La première correspond à celle de [43] et de [47], la seconde plus pratique, à celle du *Timed Asynchronous Distributed System Model* de [29].

Propriétés ultimes

Suivant la façon dont cette borne est atteinte, on définit des systèmes partiellement synchrones :

Synchrone : (*Sync*) Il existe une borne (connue) δ sur les délais de toutes les communications entre processus corrects,

[3] En fait, une horloge ϵ-synchronisée suffit en général.

Ultimement synchrone : (Usync) Il existe une borne (connue) δ sur les délais de toutes les communications entre processus corrects mais qui n'est respectée que ultimement,

Fort : (S) Il existe une borne (connue) δ telle qu'il existe un processus correct tel que toutes les communications issues de ce processus respectent cette borne [4],

Ultimement fort : (US) Il existe une borne (connue) δ telle qu'il existe un processus correct tel que toutes les communications issues de ce processus respectent ultimement cette borne,

Dans ce qui précède la borne δ est connue des processus, on peut aussi supposer qu'elle est inconnue, dans ce cas on aura :

Ultimement synchrone avec borne inconnue : ($USync'$) Pour chaque exécution il existe une borne δ telle que toutes les communications entre processus corrects respectent cette borne,

Ultimement fort avec borne inconnue : (US') Pour toute exécution, il existe un processus correct, pour lequel il existe δ telle que toutes les communications issues de ce processus respectent cette borne.

On peut remarquer que ($Sync$) correspond exactement au modèle synchrone. D'ailleurs les modèles de système synchrone et de système asynchrone peuvent être présentés comme des cas particuliers des modèles partiellement synchrones [5]. Notons aussi que tous les modèles définis ci-dessus permettent de résoudre le consensus.

D'autre part, avoir une borne ultimement respectée ou une borne inconnue revient dans de nombreux cas essentiellement au même. En général, la borne sur les délais de communication permet de construire des mécanismes de *time-out* permettant de limiter l'attente de messages. Avec une borne ultimement respectée, on pourra assurer qu'à partir d'un certain temps ces *time-out* fonctionneront bien : si le délai est dépassé le message n'arrivera jamais. En augmentant au fil du temps la valeur de la borne supposée, on peut obtenir la même propriété avec une borne qui existe mais n'est pas connue. Aussi, ($USync$) (respectivement (US)) et ($Usync'$) (respectivement (US')) sont équivalents, dans le sens où on peut simuler [6] un système ($Usync$) (respectivement (US)) avec un système ($Usync'$) (respectivement (US')). Cependant, il faut noter qu'au moins du point de vue des performances les modèles sont réellement différents.

[4] Comme on le verra plus loin, ce modèle de système correspond aux détecteurs de défaillances de la classe $\mathcal{S}$ appelés *Strong* en anglais, ce qui justifie la notation et la terminologie adoptée ici.

[5] Avec quand même la restriction importante que, *stricto sensu*, il n'y a pas de notion de temps dans un système asynchrone!

[6] Avec les définitions de la section 2.6, on pourrait, avec lourdeur, donner un sens précis à cette équivalence. Intuitivement, tout algorithme défini dans ($USync$) peut être transformé en un algorithme de ($USync'$) qui le simule.

Bonnes et mauvaises périodes

On peut aussi considérer qu'un système asynchrone est, en définitive, la plupart du temps synchrone et alterne entre des périodes où il est réellement synchrone (les bornes sont respectées) et des périodes de défaillances pendant lesquelles des processus peuvent tomber en panne et les bornes sur les messages ou le temps des pas de calcul sont dépassées. De ce point de vue, de tels modèles peuvent être interprétés comme des systèmes synchrones avec des défaillances temporelles. Cependant, comme une borne dépassée n'est plus une borne (!), il s'agit réellement de systèmes asynchrones.

Cette approche correspond à celle de [29] et à celle que nous avons développée dans le cadre des Systèmes de Phases Synchronisées [32]. Elle a l'avantage d'être assez pratique et permet en particulier de faire des calculs explicites sur le temps.

On suppose connues les bornes sur les délais de communications (et sur la vitesse relative des processus), une période de temps est dite bonne ou *stable* (par rapport à cette borne) si, pendant cette période, les délais de communication respectent cette borne et si aucun processus ne tombe en panne. En fait, le choix de la borne sur les délais de communication est cruciale et, bien sûr, les propriétés des périodes stables (durée et répétition) dépendent de ce choix : la stabilité est toujours une stabilité relative à *une* borne supposée sur les délais de communication.

On peut caractériser le système partiellement synchrone par la durée et la répétition des périodes stables. Dans la suite, on ne s'intéressera qu'aux bornes sur les délais de communication δ et, pour simplifier, on supposera que le temps des pas de calcul des processus est le même pour tous et sans variations. On dira qu'un intervalle de temps $[\tau_0, \tau_1]$ est δ-stable si, pendant cet intervalle :

- aucun nouveau processus ne tombe en panne,
- tous les messages émis durant cette période sont reçus [7] avant δ.

D'un point de vue plus pratique on peut aussi faire dépendre les spécifications du fait que l'on est dans une période stable pour un certain temps. Comme on le verra plus loin, on peut de cette façon donner des conditions sur le système qui assurent certaines propriétés temps-réel. Nous avons développé cette approche dans [40].

[7] En fait, il faut distinguer entre la réception d'un message et son traitement, dire que le message est reçu en moins de δ signifie uniquement qu'en au plus δ il est disponible dans les buffers du destinataire.

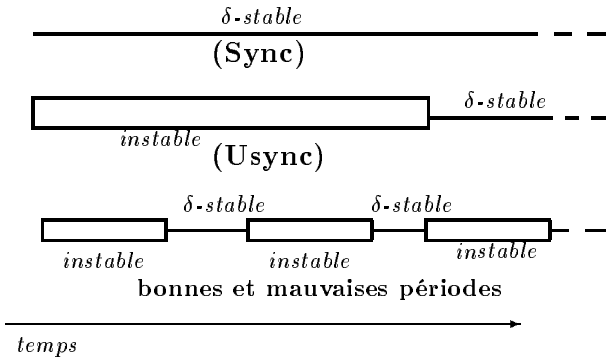


FIG. 5 – Modèles partiellement synchrones.

2.4 Les modèles de calcul : systèmes de rondes

En ce qui concerne les systèmes synchrones, la référence au temps (réel) n'est pas toujours souhaitable. On peut, par exemple, en s'inspirant de [43], supposer que toutes les exécutions vérifient des propriétés telles que :

- Il existe une constante α telle que, pour toute exécution, pour tout processus p entre deux pas de calculs de p, chaque autre processus a exécuté au plus α pas de calcul.
- Il existe une constante β telle que, pour toute exécution, entre l'émission et la réception d'un message, au plus β pas de calculs ont eu lieu.

Il est assez clair que, partant d'un système synchrone avec temps, on peut toujours réaliser ce modèle de calcul synchrone. Réciproquement, le problème est plus délicat : il faut réintroduire le temps. Pour cela on peut associer à chaque événement une date et vérifier qu'avec un temps virtuel défini ainsi, on a bien un système vérifiant, pour ce temps virtuel, les propriétés des systèmes synchrones.

En fait, pour simplifier, nous allons considérer un modèle de calcul un peu plus restreint : celui des rondes. Cela correspond à supposer que $\alpha = 1$ et que les messages sont immédiatement disponibles ($\beta = 0$). Dans le cas synchrone, le modèle des rondes synchronisées est assez naturel : il correspond au cas simplifié où l'émission d'un message, le temps de transmission et le temps de calcul local se font partout en une unité de temps.

Précisons un peu ce que l'on entendra par un modèle de calcul en rondes. Il ne s'agit pas d'une définition générale formelle, mais plutôt d'un cadre général dans lequel chaque modèle de rondes particulier aura une définition précise.

Dans un modèle de calcul en rondes, à chaque ronde est associé un ensemble de *participants* [8]. A chaque ronde, tous les participants envoient un message vers tous, ce message devant être reçu par les participants dans la même ronde. Avant de passer à la ronde suivante un participant attend une certaine condition portant en général sur la réception de messages de la ronde.

[8] Il n'est pas toujours nécessaire que tous les processus connaissent cet ensemble.

```
1 rondes génériques
2       round i :
3           Pour tous les participants p de la ronde i
4               envoyer message de la ronde m_i à tous
5               recevoir messages de la ronde jusqu'à cond
6               changer d'état
7               passer à la ronde suivante : i := i + 1
```

FIG. 6 – Modèle générique de calcul en rondes.

Les divers modèles de rondes définissent des modèles de calculs qui essayent d'être les plus synchrones possible, c'est-à-dire pour lesquels les processus essaient de travailler le plus possible ensemble. On fait, pour simplifier, une autre restriction importante : tous les messages sont *diffusés* : ils sont émis vers *tous* les processus. Pour cela, quitte à envoyer des messages vides, chaque émission est une diffusion vers *tous* les processus [9].

Dans ces modèles en rondes, à chaque ronde, *tous* les processus vivants émettent vers tous et reçoivent de tous avant de passer à la ronde suivante. Dans le cas synchrone, la ronde pourra être identitifée de façon unique (par un entier). Dans le cas asynchrone, la situation n'est pas aussi simple puisque les processus ne peuvent déterminer quels sont les processus vivants et donc à quelle condition ils peuvent passer à la ronde suivante. La section 4 est entièrement consacrée à l'étude des modèles de calculs pour l'asynchrone et le partiellement synchrone.

Pour le cas synchrone, en se restreignant aux spécifications qui sont indépendantes du temps [26], on admettra que ce qui peut être résolu dans un système synchrone peut aussi l'être dans un modèle en rondes.

Les modèles de calcul en rondes vont nous servir à comparer les divers modèles de systèmes (avec détecteurs de défaillances) proches du synchrone.

En particulier, le modèle des rondes permet d'évaluer les performances sans pour autant faire directement référence au temps. En effet, le nombre de rondes correspond à une mesure classique : la longueur des chaînes de messages (aussi appelée latence) [10, 86]. Si l'on montre qu'un certain problème ne peut pas être résolu en moins de k rondes, on en déduit aussi qu'il a une latence d'au moins k.

Pour l'essentiel, les divers types de rondes que l'on va considérer pour ces modèles diffèrent par les informations que les processus vivants ont à la fin de chaque ronde : en cas de pannes, certains processus reçoivent des messages que les autres ne reçoivent pas.

[9] Mais, il ne s'agit pas d'une diffusion fiable : si l'émetteur tombe en panne, de façon imprévisible, certains processus recevront le message et d'autres non.

ronde r et processus p :

$Viv(r)$	est l'ensemble des processus qui sont vivants à la fin de la ronde r,
$source(m)$	est l'émetteur du message m,
$M_p(r)$	est l'ensemble des messages de la ronde r reçus par p,
$M(r)$	est l'ensemble des messages de la ronde r ($M(r) = \bigcup M_p(r)$),
$V_p(r)$	est l'ensemble des processus dont p a reçu le message de la ronde r ($V_p(r) = source(M_p(r))$)
$V(r)$	est l'ensemble des processus dont au moins un message a été reçu dans la ronde r ($V(r) = \bigcup V_p(r)$)
$Der(p,q)$	est la dernière ronde si elle existe pour laquelle p a reçu un message de q ($= \infty$ si elle n'existe pas).

FIG. 7 – Notations pour les rondes.

2.4.1 Un peu de formalisme

Nous avons, jusqu'ici, essayé de limiter les notations et le formalisme. Nous allons, malgré tout, en introduire un peu dans cette section. Dans la mesure du possible, c'est-à-dire tant qu'il n'y pas de risque d'ambiguïté, nous essaierons dans la suite de rester le moins formel possible.

Les notations sont résumées dans la figure 7. Certaines de ces notations n'ont de sens que pour des rondes indexées par des entiers. $M_p(r)$ est l'ensemble des messages reçus par p dans la ronde r et $M(r) = \bigcup_{p \in P} M_p(r)$ est l'ensemble des *messages de la ronde* r. $V_p(r)$ représente l'ensemble des processus desquels p a reçu le message de la ronde r. De même, $V(r)$ est l'ensemble des processus dont au moins un processus a reçu le message de la ronde r. Intuitivement, $V_p(r)$ correspond aux participants de la ronde suivant la vue de p : dans le cas des rondes synchronisées, $V_p(r)$ correspondra aux processus que p considère comme vivants dans la ronde r. $V(r)$ correspond aux processus qui participent à la ronde (ceux dont au moins un message a été reçu). Enfin, $Der(p,q)$ est la dernière ronde au cours de laquelle p a reçu un message de q (si p a reçu un message de q pour toutes les rondes, on pose $Der(p,q) = \infty$).

Pour les modèles en rondes où un processus considère qu'il reçoit de tous les vivants, $Der(p,q)$ représente donc la ronde au cours de laquelle p a appris la mort de q et est, en quelque sorte, la vision de p de la date de la mort de q.

Introduisons aussi un peu de formalisme concernant les protocoles s'exécutant dans un modèle de rondes. Un protocole Π est la donnée pour chaque processus p d'un protocole Π_p :

$$\Pi = < \Pi_1, \ldots, \Pi_p, \ldots, \Pi_n >$$

Un protocole Π_p pour le processus p est un automate défini par un état initial s_p^0, une fonction de transition T_p, et une fonction de sortie O_p. La fonction de transition T_p détermine le nouvel état à partir de l'ancien, des messages reçus

$\mathcal{S}_p$ est l'ensemble des états p
$\mathcal{M}$ est l'ensemble des messages
$\mathcal{N}$ est l'ensemble des entiers

$\Pi_p =< T_p, O_p >$:

$T_p : \mathcal{S}_p \times \mathcal{M} \times \mathcal{N} \to \mathcal{S}_p$	$T_p(S_p, M_p, r)$ est l'état obtenu après la ronde r si l'état de p en ronde r était S_p et si p a reçu dans cette ronde l'ensemble de messages M_p
$O_p : \mathcal{S}_p \times \mathcal{N} \to \mathcal{M}$	$O_p(S_p, r)$ est l'ensemble des messages que p envoie dans la ronde r, si au début de cette ronde il est dans l'état S_p (et qu'il est vivant dans cette ronde)

FIG. 8 – Protocole Π_p pour p dans un modèle de rondes.

dans la ronde et de la ronde. O_p détermine, suivant l'état actuel et la ronde, quel message p envoie dans la ronde.

Ces notations sont résumées dans la figure 8 ; $\mathcal{S}_p$ est l'ensemble (qui peut être infini) des états pour p, $\mathcal{M}$ est l'ensemble des messages qui peuvent être échangés entre les processus. $\mathcal{N}$ est l'ensemble des rondes. Dans la suite, on ne s'intéressera qu'au cas où les rondes sont indexées par des entiers et on supposera que $\mathcal{N}$ est l'ensemble des entiers naturels.

On suppose d'autre part l'existence d'un état particulier $\perp$ (correspondant à la panne du processus). Dans cet état $\perp$, le processus ne fait plus rien et n'envoie plus de messages, on a donc toujours, pour tout M et tout r :

$$T_p(\perp, M, r) = \perp$$

On notera aussi $\perp$ le fait que p n'envoie pas de message ; pour tout r on aura :

$$O_p(\perp, r) = \perp$$

On suppose ici que les processus ne tombent pas volontairement en panne.

On peut, à partir de ces définitions et notations, définir une exécution en rondes. D'un point de vue opérationnel, à chaque ronde, il faut choisir quels processus tombent en panne et quels messages arrivent à leurs destinataires : le comportement de chaque processus p est alors déterminé par T_p.

Plus précisément, on suppose qu'à chaque ronde r, le système, se comportant comme un adversaire, décide de quels processus tombent en panne et quels sont les messages qui seront reçus [10].

[10] Ce choix est contraint : par exemple, on supposera qu'au plus t processus peuvent être en panne.

```
1 /* Protocole Π_p =< T_p, O_p > état initial s_p^0 */
2 S_p := s_p^0
3 for r := 0 to ∞ do
4    /* Ronde r */
5       send(O_p(S_p, r)) to all
6       /* recevoir les messages de la ronde */
7       /* M_p(r) est l'ensemble des messages reçus par p dans la ronde r */
8       S_p := T_p(S_p, M_p(r), r)
9 od
```

FIG. 9 – Exécution du protocole Π_p pour p dans un système de rondes.

Si l'adversaire décide de mettre un processus en panne, le nouvel état de ce processus est $\perp$.

Sinon, et si l'adversaire décide de faire participer le processus à la ronde, le nouvel état du processus est déterminé par T_p en fonction des messages reçus dans la ronde et les messages émis par p sont déterminés par O_p. Si un processus p meurt dans la ronde, l'adversaire, pourra suivant les modèles de rondes, éliminer certains des messages de p.

Les divers modèles de rondes considérés correspondent, pour l'essentiel, à des contraintes différentes du système agissant comme un adversaire.

Le modèle général du calcul en rondes de Π est décrit par la figure 9.

2.4.2 Les rondes parfaitement synchronisées RPS

Le modèle des rondes parfaitement synchronisées (RPS) est le plus puissant. Ici, non seulement tous les processus vivants participent à la ronde mais, en plus, ils reçoivent à chaque ronde *exactement* les mêmes messages. En d'autres termes, on a, pour tout p vivant à la fin de la ronde r :

$$p \in Viv(r) \Rightarrow p \in V(r) \tag{1}$$

$$M_p(r) = M(r) \tag{2}$$

Ce modèle correspond à un système synchrone dont la primitive de synchronisation serait la diffusion fiable [11].

La réalisation de rondes parfaitement synchronisées revient donc à implémenter une diffusion fiable en synchrone. Ce qui n'est pas tout à fait immédiat en cas de panne de l'émetteur. Toute implémentation de la diffusion fiable peut engendrer des chaînes de messages (latence) de longueur $t+1$ (cf., par exemple, [51, 76, 7]) . Comme on le verra, cette implémentation revient à simuler des rondes parfaitement synchronisées à partir de rondes synchronisées.

[11] Rappelons que dans la diffusion fiable, un message émis est soit reçu par *tous* soit par aucun. D'un point de vue plus technique, il s'agit de la version uniforme de la diffusion fiable : si un processus (correct ou non) délivre le message, tous les processus corrects délivrent le message.

```
1 Ronde r pour le processus p :
2   déterminer Message_p(r) (les messages de p pour la ronde r)
3   send Message_p(r) to all
4   receive les messages de la ronde r
5   changer d'état
6   passer à la ronde suivante : r := r + 1
```

FIG. 10 – Rondes synchronisées : description informelle.

2.4.3 Les rondes synchronisées RS

Le modèle des rondes synchronisées (RS) est celui qui correspond le plus directement aux systèmes synchrones. La figure 10 correspond à une description informelle de ce modèle.

Suivant la définition de Lynch [76], dans ce modèle de calcul, les processus progressent en *rondes synchronisées* : à chaque ronde, tous les processus envoient un message (éventuellement vide) vers tous les processus (il s'agit donc en fait d'une *diffusion*). Les processus, après avoir reçu tous les messages de la ronde, changent d'états suivant le protocole qu'ils exécutent et commencent la ronde suivante.

Le modèle de défaillances étant le modèle de pannes définitives des processus (*crash failure*), un message émis par un processus vivant à la fin de la ronde r sera reçu par tous les processus vivants à la fin de cette ronde, mais, si un processus meurt au cours de cette ronde, certains processus pourront avoir reçu son message alors que d'autres ne l'auront pas reçu. Ainsi, même avec les rondes synchronisées les informations partagées par les processus vivants à la fin d'une ronde ne sont pas toujours les mêmes. Aussi, on aura pour tout p vivant à la fin de la ronde r :

$$(1) \text{ et } q \in V(r) - V_p(r) \Rightarrow q \notin V(r+1) \tag{3}$$

Si un processus tombe en panne au cours de la ronde r, certains processus peuvent recevoir son dernier message et d'autres non. Ainsi, certains processus considèrent qu'il est mort au cours de la ronde r, alors que d'autres considèrent qu'il est mort au cours de la ronde $r+1$. Il peut donc y avoir un *décalage d'au plus 1 ronde entre la vision de la date des défaillances.* Avec les notations introduites dans les paragraphes précédents, on a, pour tout p, p' vivants quand le processus q tombe en panne :

$$|Der(p,q) - Der(p',q)| < 2 \tag{4}$$

Le modèle de rondes synchronisées est facilement implémentable dans un système synchrone : à chaque ronde, on attend suffisamment longtemps de façon à assurer que si on n'a rien reçu d'un processus cela indique qu'il est mort.

Inversement, le modèle des rondes synchronisées, permet de définir un temps virtuel pour lequel il existe des bornes sur les communications et les pas de calcul

des processus. En ce sens il est donc équivalent au modèle synchrone [12].

2.4.4 Autres modèles de rondes correspondant aux modèles synchrones (RAP, RAS)

On peut définir d'autres modèles de rondes. En fait, on en verra ici, deux. Le premier, *modèle en rondes asynchrones avec détecteur Parfait* (RAP), correspond exactement aux rondes que l'on obtient à partir d'un détecteur de défaillances de la classe $\mathcal{P}$. La caractéristique de ces rondes est qu'il ne suffit pas d'être vivant à la ronde r pour participer à cette ronde, il faut aussi être vivant à la fin de la ronde $r+1$:

$$p \in Viv(r+1) \Rightarrow p \in V(r) \tag{5}$$

La vision que les processus ont de la date des pannes est alors, pour tout p, p' vivants quand le processus q tombe en panne :

$$|Der(p,q) - Der(p',q)| < 3 \tag{6}$$

Le second, le *modèle en rondes asynchrones avec* $\mathcal{S}$ (RAS), correspond au modèle de rondes que l'on obtient à partir des détecteurs de défaillances de la classe $\mathcal{S}$. Il s'agit d'un modèle dégénéré de rondes, à chaque ronde les processus reçoivent de $n-t$ processus, mais, il existe un processus *correct* dont tout le monde reçoit les messages à chaque ronde :

$$\forall q \in Viv(r) : |V_q(r)| = n - t \tag{7}$$

$$\exists p \text{ correct } \forall r \forall q \in Viv(r) : p \in V_q(r) \tag{8}$$

Ces deux modèles de rondes ne sont pas très naturels, leur intérêt réside surtout dans le fait qu'ils vont permettre de préciser la puissance des modèles asynchrones utilisant des détecteurs de défaillances des classes $\mathcal{P}$ et $\mathcal{S}$.

2.4.5 Modèles de rondes correspondant à des systèmes partiellement synchrones

L'essentiel de l'étude de ces modèles est reporté à la section 4. Nous n'en présenterons ici que les grandes lignes.

Aux divers types de systèmes partiellement synchrones correspondent assez exactement divers modèles de calcul en rondes. Quand le système n'est plus synchrone, nous n'avons plus cette correspondance presque canonique.

Les divers types de *virtual synchrony* et les divers systèmes réalisant le *group membership* sont, sans doute, les modèles de calcul les plus usuels pour les systèmes partiellement synchrones. Il ne s'agit pas, en général de modèles théoriques mais plutôt de systèmes appelés *middleware* permettant à un utilisateur de travailler dans un univers ayant certains des avantages du monde synchrone

[12] On verra dans la section suivante que cette équivalence est modulo une simulation. Intuitivement, cette simulation consiste à ignorer ce qui dépend du temps.

(possibilité de détection de pannes, possibilité d'accord entre les processus, notion de groupes etc...) mais avec une implémentation reposant sur des systèmes asynchrones (en fait partiellement synchrones).

En essayant d'avoir un fonctionnement proche du synchrone, ces modèles de calcul peuvent être considérés comme des variantes de rondes synchronisées. Il existe de nombreuses variantes de ces systèmes de rondes. Dès que le système n'est plus synchrone, il est impossible d'avoir une vision exacte des pannes des processus : à un instant donné, on ne peut faire la différence entre un processus vivant mais très lent et un processus mort. On ne peut donc pas espérer avoir toujours un *seul* groupe de participants aux rondes contenant tous les processus corrects, il faudra se contenter d'un (ou plusieurs) sous-ensemble de ceux-ci. Il faudra aussi, pour assurer la présence de processus corrects, tout en garantissant la vivacité des rondes, modifier dynamiquement les participants des groupes.

Il existe deux grandes catégories de modèles concernant les groupes de processus : ceux qui imposent le présence simultanée d'*un seul* groupe (modèles *non-partitionnables*) et ceux qui autorisent la coexistence à un instant donné de *plusieurs* groupes (modèles *partitionnables*). Avec les modèles partitionnables, plusieurs groupes peuvent coexister sans se connaître et peuvent avoir des comportements différents, aussi il est assez difficile de trouver des applications pour ce type de modèles. Dans les modèles non partitionnables, pour assurer l'existence à un instant donné d'un seul groupe, on suppose généralement qu'il y a toujours une majorité de processus corrects, et on impose que le groupe des participants soit toujours un groupe majoritaire.

De plus, ces divers modèles varient suivant la connaissance que les processus ont des groupes. Si, en général, ils supposent que les groupes sont connus, ce n'est pas une nécessité. On peut aussi, à l'intérieur d'un groupe, avoir un fonctionnement plus ou moins proche des rondes synchronisées. La plupart de ces systèmes imposent certaines conditions qui doivent être maintenues en cas de changement de groupes (par exemple, si deux processus participent à deux groupes successifs ils doivent avoir la même vision de ce qui s'est passé dans le premier).

Terminons en notant que la définition formelle de ces modèles de calcul est assez délicate [9] et que leurs spécifications sont souvent incomplètes en particulier en ce qui concerne les propriétés de vivacité.

2.5 Systèmes avec oracles : détecteurs de défaillances

Les systèmes asynchrones présentent l'avantage de ne pas utiliser explicitement le temps et de ne pas supposer l'existence de bornes connues sur les délais de communication, mais le résultat d'impossibilité de Fischer, Lynch et Paterson [52] les rend insuffisants pour une algorithmique résistante aux pannes.

Ainsi, d'un côté on voudrait pouvoir rester dans des systèmes asynchrones, mais, d'un autre côté, de tels systèmes ne permettent pas d'envisager une algorithmique résistante aux pannes. Pour résoudre cette contradiction, au lieu d'atténuer les caractéristiques asynchrones du système comme avec les systèmes partiellement synchrones (section 2.3.3), on peut supposer que les processus disposent d'un mécanisme auxiliaire permettant de résoudre le problème du consensus. Bien entendu, ce mécanisme auxiliaire ne peut pas être implémenté dans un système réellement asynchrone, sinon le consensus pourrait être résolu.

Les détecteurs de défaillances définis par Chandra et Toueg [21] correspondent à cette démarche : les processus peuvent interroger des oracles locaux donnant des informations sur les pannes des autres processus.

Cette approche présente plusieurs avantages. Tout d'abord, elle est plus abstraite en ce sens que, séparant ce qui sert au consensus et à la détection de pannes du reste, elle ne modifie pas les caractéristiques du système qui reste asynchrone : l'impossibilité du consensus est contournée par l'usage de l'oracle. D'autre part, elle permet de comparer formellement les divers types d'oracles. En particulier Chandra, Hadzilacos et Toueg ont caractérisé le plus faible détecteur de défaillances permettant de résoudre le consensus [19] et ont défini une hiérarchie des détecteurs de défaillances.

Cette idée de détecteur de défaillances peut aussi apparaître comme une formalisation des *time-out* qui restent le mécanisme le plus courant pour traiter les pannes dans la programmation des systèmes réels. La notion de détecteur de défaillances permet de donner les propriétés exactes que doivent avoir ces *time-out*. Enfin, même si des détecteurs de défaillances permettant de résoudre le consensus ne peuvent pas être réalisés dans des systèmes asynchrones, on peut les réaliser dans des systèmes partiellement synchrones. Le problème de l'implémentation de ces détecteurs de défaillances est un domaine de recherche actif.

En résumé, pour développer une algorithmique tolérante aux pannes, on peut procéder ainsi :

1. développer une algorithmique résistante aux pannes dans un monde asynchrone avec des détecteurs de défaillances,
2. définir ensuite les propriétés des détecteurs de défaillances nécessaires à cette algorithmique,
3. dans le modèle particulier de système partiellement synchrone, implémenter efficacement les détecteurs de défaillances réalisant ces propriétés.

Cette façon de procéder est donc plus abstraite et plus générique que celle qui consisterait à rester du début à la fin dans un modèle partiellement synchrone particulier. Finalement, les approches de la *virtual synchrony* ne sont pas si diffé-

rentes. La différence réside dans ce que l'on rajoute aux systèmes asynchrones. Dans le premier cas, on ne rajoute que de la détection de pannes alors que, dans le deuxième, on rajoute tout un modèle de calcul. La question qui se pose est de savoir si, d'une part, ce qui est ajouté est réellement nécessaire et utile et, d'autre part, si en ajoutant trop de choses on ne restreint pas la classe de systèmes dans lesquels on va travailler.

2.5.1 Détecteurs de défaillances

Définitions

Les détecteurs de défaillances ont été introduits par Chandra et Toueg [21]. Il s'agit d'oracles que les processus peuvent consulter. Ces oracles donnent des informations non fiables sur les pannes des processus. Un détecteur de défaillances est défini par les sorties qu'il fournit aux processus à chaque consultation. Le point important est que ces sorties ne dépendent que des pannes des processus.

Plus formellement, on définit un *schéma de pannes* S comme étant une fonction du temps τ, indiquant à chaque instant τ quels sont les processus en panne. Étant donné un schéma de pannes S, un détecteur de défaillances fd fournit, pour chaque processus p, à chaque instant τ une sortie prise dans un certain ensemble O (une *histoire des sorties* du détecteur). Ainsi, un détecteur de défaillances est une *fonction* associant à chaque schéma de pannes un ensemble d'histoires de sortie. On trouvera toutes les définitions formelles dans [21]. La notion de détecteur peut se généraliser à une notion plus générale d'oracles dépendant des schémas de pannes (cf section 2.5.5 et [1]).

Propriétés

Pour l'instant, on n'a supposé aucune propriété particulière pour les détecteurs de défaillances. Pour résoudre le problème du consensus en asynchrone, on va imposer certaines propriétés aux détecteurs de défaillances.

Tout d'abord, on suppose que la sortie des détecteurs de défaillances est une liste de processus considérés comme *suspects*. Considérons les propriétés suivantes :

Complétude : il existe un temps τ à partir duquel tous les processus incorrects sont définitivement suspectés.

Exactitude forte : un processus correct n'est jamais suspecté.

Exactitude faible : il existe un processus correct qui n'est jamais suspecté.

Exactitude forte ultime : il existe un temps τ à partir duquel aucun processus correct ne sera plus jamais suspecté.

Exactitude faible ultime : il existe un temps τ à partir duquel il existera un processus correct qui ne sera plus jamais suspecté.

On définit alors les classes de détecteurs de défaillances :

– les détecteurs parfaits, $\mathcal{P}$, qui assurent la complétude et l'exactitude forte [13],

[13] En général, on suppose une propriété légèrement plus forte que l'exactitude : un processus vivant au temps τ n'est pas suspecté au temps τ.

- les détecteurs forts (*strong*), $\mathcal{S}$, qui assurent la complétude et l'exactitude faible,
- les détecteurs ultimement parfaits, $\diamond\mathcal{P}$, qui assurent la complétude et l'exactitude forte ultime,
- les détecteurs ultimement forts, $\diamond\mathcal{S}$, qui assurent la complétude et l'exactitude faible ultime.

Comme on peut le remarquer, les deux dernières classes de détecteurs ne sont définies que par des propriétés ultimes : pendant un temps arbitrairement long, ils peuvent avoir n'importe quelle sortie. Les deux premiers, donnent, par contre, en un certain sens, des informations exactes dès le début. On trouvera dans [57] une classification plus détaillée d'autres propriétés des détecteurs de défaillances.

Le résultat fondamental est que $\diamond\mathcal{S}$ est le plus faible détecteur de défaillances permettant de résoudre le consensus [19]. Plus faible est défini en termes de réduction : si on peut résoudre le consensus en utilisant un détecteur de défaillances, alors on peut construire avec ce détecteur de défaillances un détecteur de la classe $\diamond\mathcal{S}$.

Ω est un autre détecteur de défaillances qui présente un intérêt pratique. Ω correspond à une élection (ultime) de leader. Plus précisément, Ω est un détecteur de défaillance qui propose l'identité d'un processus et il existe un temps τ à partir duquel Ω proposera toujours à tous les vivants le *même* processus *correct*. Ainsi, ultimement, tous les processus corrects seront d'accord sur le *même* processus *correct*. En fait, Ω est équivalent à $\diamond\mathcal{S}$ et on donnera dans la section 4.2.4 une construction très simple de Ω à partir de $\diamond\mathcal{S}$. Dans une certaine mesure, on peut considérer aussi Ω et $\diamond\mathcal{P}$ comme des oracles proposant des groupes de processus corrects : ultimement, le premier propose un processus correct comme leader et le second assure qu'ultimement il proposera comme liste de processus *tous* les processus corrects. On reviendra sur cette question dans la section 4.2.4.

2.5.2 Implémentation des détecteurs de défaillances

Le résultat de minimalité pour le consensus de $\diamond S$ justifie à lui tout seul l'abstraction des détecteurs de défaillances. Il est toutefois intéressant de comparer cette approche avec celle des systèmes partiellement synchrones : utiliser l'abstraction des détecteurs de défaillances au lieu de considérer directement des systèmes partiellement synchrones peut-il présenter aussi un intérêt pratique ? Pour cela, il faut que, d'une part, on puisse implémenter effectivement dans le système partiellement synchrone considéré le détecteur de défaillances et que, d'autre part, on ne perde pas trop en passant par l'abstraction des détecteurs de défaillances : la puissance de calcul obtenue doit être la même (ou presque la même) que si on développe l'algorithmique directement dans le système partiellement synchrone. On s'intéresse dans cette section au premier de ces deux points. La section 3 est consacrée, en grande partie, à ce problème pour les systèmes synchrones.

En considérant les modèles de systèmes asynchrones présentés dans la section 2.3.3, nous allons voir comment ils permettent d'implémenter des détecteurs de défaillances.

```
1 Processus p :
2     A chaque η
3             send("I am alive") to all
4 ||
5     A chaque η + δ
6             forall q : Suspect_p := P − Rec_p
7             Rec_p := ∅
8 ||
9     receive("I am alive") from q : Rec_p := Rec_p ∪ q
```

FIG. 11 – Simple algorithme de détection de défaillances.

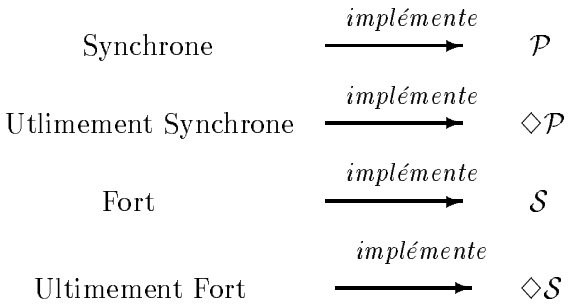


FIG. 12 – Implémentation des détecteurs de défaillances.

On ne s'intéresse pas ici à l'efficacité ni à quantifier la qualité de service [27] des détecteurs obtenus, mais uniquement à l'existence d'une réalisation. Pour cela, on considère une implémentation générique de détection de défaillances (figure 11) très simple [14] qui peut s'appliquer aux divers systèmes partiellement synchrones considérés ici. Régulièrement (à chaque η) les processus envoient à tous un message "I am alive". Si un processus p n'a rien reçu de q depuis $\eta + \delta$, il ajoute q à sa liste des suspects. Dès qu'il reçoit un message "I am alive" de q, il retire q de sa liste de suspects. Il est alors clair que, dans tous les cas, la complétude est assurée. On a d'autre part :

- si le système est synchrone ($Sync$), le détecteur de défaillances est parfait ($\mathcal{P}$),
- si le système est ultimement synchrone ($USync$), le détecteur de défaillances est ultimement parfait ($\Diamond\mathcal{P}$),
- si le système est *fort*, le détecteur de défaillances est *fort* ($\mathcal{S}$),
- si le système est *ultimement fort* (US), le détecteur de défaillances est *ultimement fort* ($\Diamond\mathcal{S}$),

[14] On suppose, bien que cela ne soit pas nécessaire, que les processus disposent d'horloges à dérive nulle.

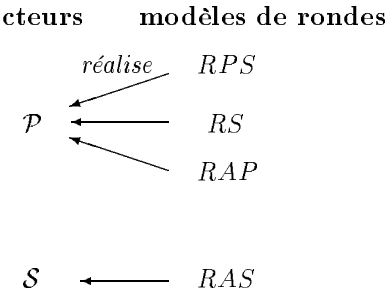


FIG. 13 – Réalisation des détecteurs de défaillances à partir de modèles de rondes.

2.5.3 Réalisation des détecteurs à partir de modèles de rondes

Les modèles de rondes décrits plus haut correspondent aussi aux détecteurs de défaillances. On verra plus loin (section 3 et section 4) d'autres éléments concernant les rapports entre les modèles de rondes et la détection de pannes.

Pour l'instant, nous allons juste indiquer comment on peut directement implémenter un détecteur de défaillances à partir d'un système de rondes : à chaque ronde un processus suspecte tout processus dont il n'a pas reçu de messages. De cette façon, par rapport à la hiérarchie définie plus haut, on obtient directement (figure 13) :

- avec des rondes parfaitement synchronisées (RPS), on obtient un détecteur dans la classe $\mathcal{P}$,
- avec des rondes synchronisées (RS), on obtient un détecteur dans la classe $\mathcal{P}$,
- avec des rondes asynchrones avec détecteur parfait (RAP), on obtient un détecteur de la classe $\mathcal{P}$,
- avec des rondes asynchrones avec $\mathcal{S}$ (RAS), on obtient un détecteur de la classe $\mathcal{S}$.

On verra plus loin (section 4), les détecteurs de défaillances que l'on obtient directement à partir des modèles de rondes pour d'autres systèmes partiellement synchrones.

2.5.4 Réalisation de modèles de rondes à partir de détecteurs de défaillances

Réciproquement, de façon directe, on peut réaliser des systèmes de rondes avec des détecteurs de défaillances. En considérant l'algorithme de la figure 14, on construit des systèmes de rondes. On obtient (ce qui justifie la terminologie !) :

- avec $\mathcal{P}$, des rondes asynchrones avec détecteur parfait (RAP),
- avec $\mathcal{S}$, des rondes asynchrones avec $\mathcal{S}$ (RAS).

On n'obtient pas *directement* des rondes synchronisées avec $\mathcal{P}$. Comme on le verra plus loin, le problème avec un détecteur de défaillances de la classe $\mathcal{P}$, est

```
1  ronde r :
2        send message de ronde r  to all
3        forall q
4             wait
5               until
6                     receive  un message de la ronde r  from q
7                     or q  Suspecté par F
8        changer d'état
9        passer à la ronde suivante : r := r + 1
```

FIG. 14 – Rondes avec détecteur de défaillances $\mathcal{F}$.

que si p est mort au temps τ mais a émis un message au temps $\tau - 1$, du fait de l'asynchronisme de la communication, ce message peut ne pas être reçu par q au temps τ, et il est possible que q soupçonne p *avant* d'avoir reçu ce message et n'attende pas la réception de ce message. La question qui se pose alors est de savoir dans quelle mesure on peut simuler des systèmes de rondes synchronisées avec ces systèmes de rondes. Nous reviendrons sur ce point dans la section 3.

Pour $\diamond\mathcal{P}$ ou $\diamond\mathcal{S}$ on étudiera dans la section 4 des systèmes de rondes que l'on peut obtenir.

Pour terminer, notons qu'il peut être intéressant de restreindre la puissance des détecteurs de défaillances. En effet, rappelons que les détecteurs de défaillances ne dépendent que du schéma de pannes de l'exécution. Il peut être utile de considérer des classes plus restreintes. En effet, certains détecteurs de défaillances peuvent ne pas très réalistes. Par exemple, les définitions n'excluent pas un détecteur de défaillances qui devine dès le premier instant toute l'histoire des pannes de l'exécution. Mais, une part de magie est nécessaire : pour résoudre le consensus en asynchrone, un détecteur de défaillances devra d'une façon ou d'une autre être devin.

Aussi, il peut être intéressant (en particulier pour être plus proche d'une implémentation dans des systèmes partiellement synchrones) de restreindre la classe des détecteurs de défaillances considérés. Par exemple, on peut définir la classe des détecteurs de défaillances *réalistes* [15] : un détecteur est réaliste si sa sortie au temps τ ne dépend que du schéma de pannes jusqu'au temps τ. En ce sens, un détecteur de défaillances est réaliste s'il ne devine pas ce qui va se passer plus tard. Il est alors intéressant d'étudier les hiérarchies de détecteurs de défaillances que l'on obtient.

2.5.5 Oracles

La notion d'oracle est une généralisation de celle des détecteurs de défaillances. Un oracle est un mécanisme extérieur que les processus peuvent

[15] Il s'agit de travaux actuellement en cours en collaboration avec C. Delporte-Gallet et R. Guerraoui.

consulter : un processus fait une requête à l'oracle et obtient ensuite une réponse à sa requête. L'oracle est caractérisé par son comportement dépendant des requêtes qui lui sont adressées et, comme les détecteurs de défaillances, du *schéma de pannes* (l'histoire des pannes correspondant à l'exécution). Cette dépendance du schéma de pannes est bien sûr le point important qui limite la puissance des oracles considérés.

D'un point de vue plus pratique, une question qui se pose dans la réalisation de systèmes asynchrones tolérants aux pannes est d'ajouter à ces systèmes la brique de base qui permettra de résoudre les problèmes usuels. L'oracle est un moyen abstrait de spécifier exactement ce que peuvent être ces briques de base et ensuite de pouvoir les comparer. Essentiellement, trois types d'oracles peuvent être pris en considération :

- les détecteurs de défaillances : la réponse de l'oracle ne dépend pas des requêtes et ne dépend que du schéma de pannes,
- les oracles de consensus : les processus proposent des valeurs pour le consensus et l'oracle répond par une valeur de consensus,
- les oracles de diffusion atomique : les processus proposent des valeurs à diffuser et l'oracle assure la diffusion atomique de ces valeurs.

On peut remarquer que, en tout cas, dans des système asynchrones, aucun de ces oracles n'est réalisable : il faudra ajouter des conditions supplémentaires sur le système pour les réaliser.

On peut aussi définir une notion de réduction entre oracles. Ainsi, directement, si la diffusion fiable est possible, un oracle de consensus peut être transformé en oracle de diffusion atomique et réciproquement [61]. De même, un oracle de détection de pannes de la classe $\diamond\mathcal{S}$ peut être transformé en oracle de consensus (s'il y a une majorité de processus corrects).

Les approches utilisant les oracles présentent plusieurs avantages. Le premier est que l'oracle ajoute exactement ce qui est nécessaire à la résolution de problèmes qui sinon ne pourraient pas l'être. Il restreint cet ajout et l'oblige à ne dépendre que du schéma de pannes et l'isole du reste du système. L'oracle permet de spécifier exactement ce qui est nécessaire pour la résolution des problèmes considérés.

On peut définir diverses propriétés générales concernant les oracles qui limitent plus ou moins leur caractère magique (non réalisable sans assertions supplémentaires sur le système) et leur puissance. En effet, le point important est que l'oracle, dépendant du schéma de pannes (et donc des pannes qui *vont* se produire) peut deviner ce qui va se passer. Ce caractère divinatoire peut être restreint de façon à minimiser ce qu'il faut ajouter au système pour les réaliser. C'est la même démarche qui amène à s'intéresser aux détecteurs de défaillances appelés ci-dessus réalistes, dans la mesure où leur comportement à un instant donné ne dépend que des défaillances jusqu'à cet instant.

On verra dans la section 6.5, une utilisation de divers types d'oracles. En particulier, dans cette section, suivant [1], l'objectif est de limiter l'usage de ces oracles.

2.6 Simulations, traductions et comparaison des modèles de calcul

2.6.1 Introduction

Supposons que je puisse choisir de réaliser dans mon système réel divers modèles de calculs, plusieurs questions se posent naturellement : avec lequel pourrais-je réaliser le plus, c'est-à-dire lequel est le plus puissant, lequel est le plus efficace (usage de ressources, temps, etc...), avec lequel l'algorithmique est-elle la plus simple etc... Par exemple, sachant que mon système est, disons, synchrone presque tout le temps, est-il préférable de choisir un modèle de calcul avec temps comme [29], ou de construire une couche spécifique permettant de réaliser des détecteurs de défaillances ou encore de faire un système de *virtual synchrony* ? En plus des considérations sur les conditions requises pour pouvoir réaliser ces divers modèles de calcul, il faut savoir, dans quelle mesure, on obtient la même puissance avec ces divers modèles.

Dans ce chapitre, il s'agit de préciser ce que signifie qu'un modèle de calcul est plus puissant qu'un autre. Essentiellement, un modèle est strictement plus puissant qu'un autre si tout ce qui peut être réalisé dans le premier peut aussi l'être dans le second et s'il existe des spécifications qui peuvent être résolues dans le premier et pas dans le second. Cependant, une telle comparaison est souvent trop restrictive : les modèles de calcul considérés étant différents ils deviennent vite tous incomparables. C'est pourquoi on considère plutôt une comparaison modulo une simulation c'est-à-dire en autorisant que certaines actions ou événements ne soient pas exactement réalisés mais uniquement simulés.

Pour mener à bien cette comparaison, il faut être très précis, car, sinon, on obtient vite des résultats contradictoires ou sans intérêt. Par exemple, *a priori*, en l'absence de pannes, les systèmes asynchrones sont aussi puissants que les systèmes synchrones, alors que, de façon évidente il existe des problèmes spécifiquement synchrones (par exemple, des problèmes où intervient explicitement le temps) que l'on ne peut pas résoudre avec un système asynchrone. Par contre, il est clair que l'on peut *simuler* tout protocole synchrone en asynchrone. De même, [26], allant contre une idée assez répandue, ont montré que certains problèmes résolubles en synchrone ne peuvent pas être résolus dans un système asynchrone avec des détecteurs de défaillances parfaits. Mais, d'autres résultats montrent que l'on peut toujours simuler une ronde du synchrone avec deux rondes construites en asynchrone avec des détecteurs de défaillances parfaits. Comme on le verra, il n'y a pas réellement de contradiction entre ces deux affirmations : tout dépend de ce que l'on impose à la simulation.

Pour éviter ces écueils, on voit bien l'intérêt qu'il y a à définir formellement les sémantiques des divers modèles de calcul. Cependant, une étude exhaustive nécessiterait de définir entièrement ces sémantiques formelles des divers modèles de calcul ce qui, outre le travail énorme que cela représente, est fastidieux et sans grand intérêt. On ne donnera ici que les grandes lignes en espérant que tous les détails puissent être reconstitués sans trop d'efforts...

La première chose à faire est donc de donner un sens général et suffisamment

précis à ce que signifie qu'une exécution dans un modèle de calcul *simule* une exécution d'un autre modèle de calcul. Ensuite, à partir de cette notion de simulation, il faut pouvoir éventuellement définir des *traductions* qui permettent de transformer un protocole Π_A d'un système donné A en un autre protocole Π_B d'un autre système B simulant les exécutions de Π_A dans A.

La notion de simulation est très généralement utilisée et souvent dans des contextes différents. Bien évidemment, suivant les propriétés requises par ces simulations on obtiendra des résultats différents : avec une définition très stricte aucune traduction d'un protocole d'un modèle dans un autre ne sera possible, à l'autre extrême, la traduction risque de ne pas présenter d'intérêt. Comme simuler un calcul n'est pas la même chose que le réaliser, toute la question est de définir ce que la simulation a le droit d'oublier. Cette question est à l'origine de l'existence de résultats apparemment contradictoires.

Prenons trois exemples simples :

- Il est généralement admis que, en l'absence de pannes, on peut *simuler* un système synchrone par un système asynchrone. En particulier, les synchroniseurs d'Awerbuch [10] permettent (dans un cadre non tolérant aux pannes) de simuler un protocole synchrone en rondes dans un système asynchrone. Cela signifie que, étant donné un protocole Π_{Sync} défini pour un système synchrone *Sync*, on peut l'exécuter dans un système asynchrone *ASync* en utilisant des synchroniseurs, qui permettront de faire comme si l'exécution était synchrone. On a donc ainsi *traduit* Π_{Sync} en un protocole Π_{ASync} (en fait Π_{Sync} + les synchroniseurs) : étant donné une exécution obtenue dans le système asynchrone en utilisant des synchroniseurs, on obtient une exécution (asynchrone) qui est intuitivement équivalente à une exécution synchrone obtenue par le même protocole. Bien entendu, certaines propriétés de cette exécution asynchrone, comme, par exemple, celles où interviendrait le temps, n'ont aucun sens dans le système asynchrone.
- Comme deuxième exemple, considérons les techniques de masquage d'erreurs comme celles définies dans [82]. On verra plus loin (section 3.3.1) des exemples de ces techniques. Ici, on part d'un système tolérant un certain type de défaillances (a priori des défaillances dites bénignes), on définit une traduction permettant de transformer un protocole défini pour ce type de défaillances en un protocole tolérant des pannes plus sévères. On augmente ainsi la tolérance aux pannes.
- [19] ont aussi montré que tout détecteur de défaillances permettant de réaliser le consensus permet d'implémenter un détecteur de défaillances de la classe $\diamond\mathcal{S}$, montrant ainsi que $\diamond\mathcal{S}$ est le plus faible détecteur de défaillances permettant de réaliser le consensus. Ce qui signifie que tous les détecteurs de défaillances permettant de résoudre le consensus peuvent simuler un détecteur de défaillances de la classe $\diamond\mathcal{S}$. Ici, un détecteur de défaillances en simule un autre s'il existe un algorithme (distribué) qui, utilisant les sorties du premier, permettra d'obtenir des sorties du second.

En fait, dans ces différents exemples, la question qui se pose est de déterminer quelles sont les propriétés vérifiées par la traduction. Par exemple, dans

le deuxième exemple, la simulation impose que si p est un processus correct dans l'exécution α qui simule une exécution β, p est aussi correct pour β, mais elle autorise que certaines défaillances d'un processus de α disparaissent dans la simulation. On exclut ainsi la possibilité qu'un processus correct ne meurt de lui-même (le suicide est interdit), mais on rend possible l'occultation de certaines pannes. Le troisième exemple suppose qu'il existe un algorithme réalisant la simulation, celle-ci est donc *calculable*. Il est aussi possible de définir, comme [62], une notion de simulation de détecteurs de défaillances, basée sur la *knowledge theory*, qui n'est pas toujours calculable.

On va donc, dans un premier temps, définir les notions de simulation et traduction. Dans les sections suivantes nous verrons comment on peut les appliquer aux cas réels.

2.6.2 Simulations

Il s'agit de donner des définitions assez générales permettant d'englober les diverses notions de simulation que l'on peut trouver dans la littérature. On va essayer aussi d'être le plus intuitif possible.

Etant donné un protocole Π défini dans un système S, on notera l'ensemble des exécutions ou histoires de Π, $R(\Pi, S)$[16]. Essentiellement, une exécution est une séquence (infinie) d'événements (ou d'événements et d'états) exécutés par les processus. En fait, certaines actions de ces exécutions ne jouent qu'un rôle purement interne et ne concernent pas la spécification que l'on veut résoudre. On s'intéresse donc à l'ensemble des *traces* des exécutions : une trace est l'image obtenue après effacement des actions que l'on désire rendre invisibles. Si Tr est l'application (*a priori* un morphisme) qui permet d'effacer certaines actions, l'ensemble des traces de Π est donc $Tr(R(\Pi, S))$. Par exemple, [76] considère que les traces sont réduites aux actions externes des *I/O automata*. Par abus de langage, le contexte permettant de préciser, on abrégera $Tr(R(\Pi, S))$ en $Tr(\Pi)$.

Une *spécification* Σ est la donnée d'un ensemble de traces. Dire qu'un protocole résout la spécification Σ signifie simplement que l'ensemble des traces générées par le protocole est inclus dans Σ :

Définition 1 *On dira que* Π *résout la spécification* Σ *(noté* $\Pi \models_{Tr} \Sigma$*)*[17] *si et seulement si :*

$$Tr(\Pi) \subseteq \Sigma$$

Il est bien clair que Tr représentant ce que l'on considère comme étant visible de l'extérieur, la signification pratique de cette définition dépend fortement de Tr.

Nous allons maintenant présenter la notion de simulation. Pour cela, considérons deux systèmes S_h et S_b ; intuitivement, S_h est un système haut niveau

[16] On peut naturellement définir formellement ces notions : un protocole est la donnée pour chaque processus d'un système de transitions vérifiant certaines propriétés de composition. Les *I/O automata* de Lynch et Tuttle [77, 76] représentent l'approche formelle la plus complète dans ce domaine. Dans la suite, on fera parfois référence à ce modèle.

[17] Comme on suppose Tr donné par le contexte, on écrira souvent $\Pi \models \Sigma$ au lieu de $\Pi \models_{Tr} \Sigma$

S	*Le système ou le modèle de calcul sous-jacent*
Π	un protocole défini dans S,
$R(\Pi, S)$	les exécutions (histoires) de Π dans S,
Tr_S	les traces dans S.
$Tr(R(\Pi, S))$ (ou $Tr(\Pi)$)	les traces des exécutions de Π : ce qui est visible des exécutions de Π ($Tr(R(\Pi, S) \subseteq Tr_S)$,
Σ	une spécification dans S : un ensemble de traces ($\Sigma \subseteq Tr_S$).

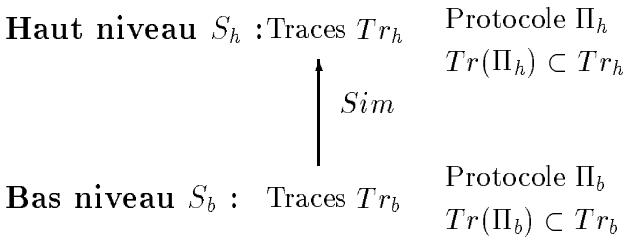
FIG. 15 – Notations.

que l'on veut simuler avec un système bas niveau S_b. Par exemple, S_h peut être un système synchrone avec des pannes bénignes (par exemple des pannes définitives de processus) et S_b un système du même type avec des pannes plus sévères (par exemple des pannes définitives et certaines pertes de messages). Ou, comme autre exemple, S_h peut être un système synchrone et S_b un système asynchrone disposant de détecteur de défaillances d'un certain type.

Dire qu'un protocole Π_b défini dans S_b simule un protocole Π_h défini dans S_h signifie essentiellement que chaque trace de l'ensemble des traces de Π_b correspond à une trace de l'ensemble des traces de Π_h. Cette correspondance n'est pas forcément une inclusion, car on peut interpréter dans S_h différemment certaines actions de S_b. On suppose donc l'existence d'une relation de simulation Sim qui transforme les histoires de Π_b dans S_b en traces dans S_h. On imposera, suivant les cas, des propriétés particulières à cette fonction de simulation des histoires. Par exemple, dans [76], Sim est supposée être définie par une relation entre les états des *I/O automata* correspondant à Π_b et à Π_h et Sim est l'identité pour les actions externes. Pour [82], Sim doit assurer qu'un processus correct ne peut se transformer en processus incorrect et doit aussi assurer la correspondance entre les états.

Considérons, maintenant la simulation de systèmes synchrones par des systèmes asynchrones utilisant les synchroniseurs d'Awerbuch. Dans ce cas, si α est une exécution utilisant un synchroniseur, même après effacement et/ou modification d'actions, α ne peut pas être considérée comme une exécution en rondes synchronisées. En effet, l'asynchronisme des actions de α ne peut directement se transformer en une exécution en rondes synchronisées. En fait, on montre qu'il existe une exécution α' correspondant à une exécution en rondes synchronisées qui est *indistinguable* de α. Indistinguable signifie ici que, pour chacun des processus, rien ne peut distinguer α de α' [18]. Plus précisément si α_i est

[18] Cette notion d'indistinguabilité joue un rôle important dans les preuves d'impossibilité.



$$\Pi_b \textit{ simule } \Pi_h : Sim(Tr(\Pi_b)) \subseteq Tr(\Pi_h)$$

FIG. 16 – Simulation du protocole Π_h par le protocole Π_b.

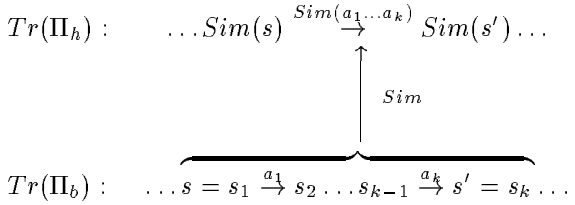


FIG. 17 – Simulation de k rondes en 1.

le comportement du processus i pour α et si α'_i est le comportement du processus i pour α', alors $\alpha_i = \alpha'_i$. En d'autres termes, deux comportements sont indistinguables si leurs projections sur chaque processus sont égales.

On en déduit donc que Sim n'est pas toujours une fonction transformant des actions et états en actions et états (c'est-à-dire un morphisme), mais peut être une relation entre exécutions ou plus exactement, en général, une transduction. Des propriétés de Sim dépendent les propriétés des simulations de protocoles.

Dans tous les cas, si α est une exécution, on supposera que Sim vérifie au moins :

1. si p est correct, alors il ne peut simuler un processus incorrect. Plus précisément, si α simule β ($\beta \in Sim(\alpha)$), alors si p est correct dans α, p est aussi correct dans β [19].
2. la simulation ne peut éliminer des comportements : $Sim(\alpha)$ ne peut être vide.

De plus, pour tous les cas intéressants, la relation de simulation devra aussi

On l'utilise pour les preuves des propriétés utilisées dans la section 3.4.

[19] Par contre, il est possible que la simulation occulte des pannes : un processus qui a commis des défaillances peut simuler un processus correct. Ce cas arrive surtout avec des pannes de type *send* ou *receive omissions* où un processus défaillant peut ne pas émettre ou recevoir certains messages.

correspondre à une relation de simulation entre états (comme c'est le cas pour les simulations définies pour les *I/O automata*, les masquages de pannes et la simulation décrite dans la figure 17), elle devra aussi être compatible avec une notion d'indistinguabilité (comme c'est le cas pour la simulation du synchrone dans des systèmes asynchrones avec les synchroniseurs d'Awerbuch).

Nous pouvons maintenant étendre la simulation aux protocoles : un protocole en simule un autre si toutes les traces du premier simulent des traces du second.

Définition 2 *Π_b simule Π_h pour (Tr, Sim) si et seulement si :*

$$Sim(Tr(\Pi_b)) \subseteq Tr(\Pi_h)$$

On étend ensuite la définition de la résolubilité d'une spécification à Sim :

Définition 3 *Π_b résout Σ pour la simulation Sim (ce que l'on notera $\Pi_b \models^{Sim} \Sigma$) si et seulement si :*

$$Sim(Tr(\Pi_b)) \subseteq \Sigma$$

On déduit immédiatement des définitions que :

Proposition 1 *Si Π_b simule Π_h pour Sim et si Π_h résout Σ alors $\Pi_b \models^{Sim} \Sigma$.*

Dans le cas des *I/O automata*, la simulation telle que définie dans [76] est une relation entre états correspondant à la notion classique de simulations entre automates. En particulier, la simulation pour les exécutions n'efface aucune action externe, aussi, pour des traces ne conservant que les actions externes, on aura la propriété : si Π_b simule Π_h et si Π_h résout Σ alors directement $\Pi_b \models_{Tr} \Sigma$. Mais, cette définition restrictive de la simulation ne permet pas de considérer que les synchroniseurs d'Awerbuch permettent de simuler un système synchrone par un système asynchrone.

Cette notion de résolubilité pour Sim correspond à la définition de *effectively solve* de [82].

Il faut noter que, mis à part certains cas simples, le fait que Π résolve une spécification Σ pour Sim ne permet pas de déterminer directement quelles spécifications sont résolues par Π (sans simulation). On peut toutefois noter que :

Proposition 2 *Si Π résout Σ pour la simulation Sim alors, pour toute spécification Σ' on a (Sim^{-1} étant la relation inverse de Sim) :*

$$\Sigma' \subseteq Sim^{-1}(\Sigma) \text{ alors } \Pi \models_{Tr} \Sigma'$$

2.6.3 Traductions

Pour comparer les modèles de calcul, il ne suffit pas de définir des simulations des exécutions d'un modèle de calcul dans un autre, il faut aussi traduire un protocole d'un modèle dans un autre.

Supposons que l'on sache résoudre une certaine spécification dans un modèle de haut niveau (exemples : en synchrone, respectivement avec des pannes bénignes) on veut pouvoir automatiquement modifier le protocole afin de résoudre la spécification dans un modèle de plus bas niveau (exemples : en asynchrone, respectivement avec des pannes plus sévères). En général, il ne s'agit pas d'une

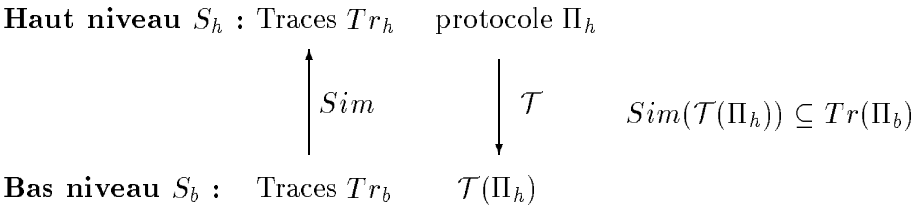


FIG. 18 – Traduction de protocoles.

résolution directe, mais plutôt d'une résolution pour une certaine simulation Sim.

En étant un peu plus précis, une traduction est un algorithme $\mathcal{T}$ qui transforme un protocole Π_h défini pour un modèle de calcul S_h en un protocole $\Pi_b = \mathcal{T}(\Pi_h)$ pour le modèle de calcul S_b. La traduction est compatible pour la simulation Sim si le protocole obtenu $\Pi_b = \mathcal{T}(\Pi_h)$ simule pour Sim celui de départ, Π_h (cf. figure 18). Toutes les traductions que l'on va considérer sont compatibles avec la simulation considérée Sim.

Reprenons nos exemples. Utiliser des synchroniseurs d'Awerbuch, permet d'exécuter n'importe quel algorithme synchrone dans un modèle de calcul asynchrone. Le point important est de déterminer exactement quelle spécification est alors résolue. En fait, l'algorithme traduit ne fait que simuler l'algorithme de départ et il résout la spécification d'origine *pour la simulation* Sim. Ainsi, pour les synchroniseurs d'Awerbuch, on montre que toute exécution du protocole $\mathcal{T}(\Pi_h)$ est indistinguable d'une exécution synchrone de Π_h et donc que $\mathcal{T}(\Pi_h)$ résout à l'indistinguabilité près les mêmes spécifications que Π_h.

Les méthodes de masquage de pannes ne permettent pas de résoudre dans le système de bas niveau n'importe quelles spécifications résolues par le protocole simulé. Cependant, les propriétés de la simulation permettent de montrer que l'on peut résoudre, par exemple, les problèmes de décisions. On peut d'ailleurs déduire cette propriété de la proposition 2. Mais comme ces méthodes simulent une ronde en plusieurs, la traduction d'un protocole résolvant une spécification contenant une notion de comptage de rondes ne pourra résoudre la même spécification. Il ne la résoudra que modulo la simulation.

2.6.4 Comparaisons entre modèles de calcul

Pour comparer la puissance des modèles de calculs on est amené à déterminer quelles sont les spécifications qui peuvent être résolues dans un modèle et qui ne peuvent l'être dans l'autre. Cependant, la résolution sans simulation ne donne pas des résultats très intéressants : très vite tous les modèles de calculs sont incomparables, il suffit de coder dans la spécification ce qui caractérise le modèle. Par exemple, même en l'absence de pannes, un système asynchrone est, intuitivement, moins puissant qu'un système synchrone puisque, quoiqu'il arrive, on aura des exécutions asynchrones qui, par définition, ne peuvent être des exécutions synchrones. C'est pourquoi on considérera plutôt la résolution

pour une simulation donnée.

On dira qu'un système S_1 est simulable par S_2 pour la simulation Sim (notation : $S_1 \preceq_{Sim} S_2$) si pour toute spécification Σ, pour tout protocole Π_1 de S_1, tel que $\Pi_1 \models \Sigma$ il existe Π_2 protocole de S_2 tel que $\Pi_2 \models^{Sim} \Sigma$.

Bien entendu, les propriétés de la simulation Sim sont fondamentales : on peut, par exemple, toujours choisir Sim de façon à ce que $S_1 \preceq_{Sim} S_2$. Encore une fois, Sim extrait ce qui est considéré comme significatif.

L'existence d'une traduction compatible avec Sim de S_1 vers S_2 assure naturellement que S_1 est simulable par S_2 :

Proposition 3 *Si $\mathcal{T}$ est une traduction de S_1 vers S_2 compatible avec Sim alors $S_1 \preceq_{Sim} S_2$.*

On peut maintenant revenir sur le paradoxe apparent signalé en début de chapitre. En considérant un système asynchrone avec détecteurs de défaillances parfaits, on verra plus loin que l'on peut définir une traduction qui, pour l'essentiel, transforme une ronde synchrone (avec pannes définitives) en deux rondes de simulation, cette traduction est compatible avec une simulation très naturelle. On peut donc simuler un système de rondes synchronisées avec un système asynchrone avec des détecteurs de défaillances parfaits. Cette simulation n'étant pas l'identité[20], cela ne veut pas dire pour autant que toute spécification résoluble dans un système synchrone avec pannes définitives l'est aussi dans le système asynchrone avec des détecteurs de défaillances parfaits. La proposition 2 permet de préciser, dans le cas des traductions, quelles spécifications sont résolues sans simulation.

Quand un modèle S_1 est un sous modèle d'un autre modèle S_2, dans le sens où toute exécution du premier est aussi une exécution du deuxième, on a une simulation triviale naturelle de S_2 dans S_1 et, bien entendu, $S_2 \preceq_{Id} S_1$ où Id est l'identité. Par exemple, le modèle en rondes parfaitement synchronisées est un sous modèle du modèle en rondes synchronisées et donc $RS \preceq_{Id} RPS$.

Dans la section suivante, nous allons présenter diverses simulations entre modèles de calcul. Sauf mention explicite du contraire, les simulations que nous allons étudier sont des simulations du type de celles de la figure 17. Dans ces simulations, on simulera *une* ronde du protocole de haut niveau par k rondes du protocole de bas niveau. Dans la relation de simulation, les processus corrects simulés ne peuvent correspondre qu'à des processus corrects. De plus, les états du système de bas niveau et les états du système de haut niveau se correspondent.

[20] En particulier, cette simulation autorise à décaler (d'au plus un) la ronde où intervient une panne de processus.

3 Synchrone : Utiliser des détecteurs de défaillances en synchrone ?

3.1 Introduction

Dans ce chapitre, nous allons comparer la puissance des systèmes synchrones (*Sync*) par rapport à celle de l'asynchrone avec détecteurs de défaillances. Les détecteurs de défaillances que nous considérerons sont ceux qui correspondent le mieux à des systèmes synchrones, à savoir ceux des classes $\mathcal{P}$ et $\mathcal{S}$ [21].

En dehors de l'intérêt théorique, la question qui se pose est la suivante : dans un système supposé être synchrone, est-il intéressant d'utiliser des détecteurs de défaillances ? On peut aussi, d'un point de vue plus pratique reformuler autrement la question : supposons que l'on dispose d'un système synchrone au sens classique du terme (avec bornes sur les temps d'exécution et de communication), une approche consistant à construire un système de détection de pannes pour ensuite l'utiliser dans l'algorithmique présente-t-elle une restriction par rapport à une approche utilisant directement les propriétés du système synchrone ? Si c'est le cas, l'utilisation de détecteurs de défaillances comme primitives pourrait s'avérer être un mauvais choix. En d'autres termes, on est amené à se poser les deux questions suivantes : existe-t-il des problèmes qui peuvent être résolus dans un cas et pas dans l'autre ? Le coût lié à la réalisation de détecteurs de défaillances peut-il être compensé ?

Ne pas faire d'assertion directe sur les délais de communication présente bien sûr l'intérêt de la modularité : l'algorithmique ne dépend pas de ces délais. Cette question des délais est cruciale. On peut raisonnablement estimer que les systèmes synchrones sont plus une abstraction de la réalité que la réalité elle-même : les délais de communication dépendent, entre autres, des conditions de charge du réseau et sont de ce fait difficilement prévisibles [22]. On ne peut, au mieux, qu'estimer des bornes sur ces délais. De cette estimation dépendront à la fois l'efficacité et la sûreté de l'algorithme : si on sous-estime ces délais, l'algorithme risque d'être incorrect, si on les surestime, les performances risquent d'être déplorables. Cet argument incite à une approche modulaire restreignant la partie de l'algorithmique dépendant de ces délais à la détection de pannes.

En ce sens, l'approche utilisant les détecteurs de défaillances est plus abstraite : on développe, d'une part, une algorithmique générale asynchrone utilisant des détecteurs de défaillances, et, d'autre part, indépendamment, une implémentation dans le système donné des détecteurs de défaillances vérifiant les propriétés requises. Ne dépendent alors des bornes sur les délais de communication que cette deuxième partie.

Donc, *a priori*, séparer la détection de pannes du reste semble, même dans les systèmes synchrones, une approche intéressante. Reste à savoir si, par cette

[21] On a vu dans la section 2.5.2 qu'ils sont implémentables respectivement dans (*Sync*) et dans (*Fort*), qui correspondent à des systèmes pour lesquels une borne existe *dès le début* sur les délais de communication de tous les processus dans le premier cas et de l'un des processus corrects dans le second cas.

[22] Ce même argument justifie l'intérêt des modèles partiellement synchrones.

approche, on perd quelque chose ou non. Pour cela, il faut comparer ces modèles du point de vue de la résolubilité et de l'efficacité.

Mais la comparaison *directe* des modèles n'est guère possible (on évolue dans un cas avec du temps et dans l'autre avec des détecteurs de défaillances). Aussi, on va avoir un objectif plus restreint : comparer les modèles de calcul en rondes correspondants, c'est-à-dire les rondes synchrones et les rondes avec $\mathcal{P}$ (RAP). Il faut bien insister sur les limites de cette comparaison. D'une certaine façon, le modèle de rondes synchronisées est naturel pour un système synchrone : la ronde est une sorte d'abstraction du temps. Par contre, les modèles de rondes ne sont pas aussi naturels pour un système asynchrone avec des détecteurs de défaillances parfaits. Il est probable que, si l'on dispose de détecteurs de défaillances parfaits, on ne développera pas nécessairement des algorithmes utilisant les modèles des rondes avec $\mathcal{P}$. La mesure de la latence, c'est-à-dire, ici, du nombre de rondes, n'est pas forcément la meilleure pour évaluer les performances. Ainsi, avec un détecteur de défaillances parfait, la connaissance des pannes peut remplacer l'échange de messages. Par exemple, supposons que l'on veuille faire une certaine action dès qu'une panne arrive. Un détecteur de défaillance donnant directement des informations sur les pannes, avec un détecteur parfait, il suffit d'attendre que le détecteur annonce une panne pour faire cette action : il n'y a pas eu besoin d'échanger des messages (la latence est ici nulle !) [23]. Par contre, dans un système synchrone, il faudra, continûment échanger des messages pour savoir qu'un processus est en panne (la latence n'est pas bornée !).

Néanmoins, à condition de se rappeler que la comparaison des modèles de calcul a ses limites et consiste, en quelque sorte, à utiliser les détecteurs de défaillances de façon non naturelle, cette comparaison est intéressante. On peut, malgré tout, en déduire des résultats généraux indépendants du modèle de calcul.

Il est clair qu'un système synchrone permet de faire tout ce que l'on peut faire (raisonnablement) avec de l'asynchrone utilisant des détecteurs de défaillances [24]. Supposons que l'on dispose d'un système pouvant être considéré comme synchrone avec des pannes franches de processus, il est clair qu'avec des *time-out* très simples, on peut construire des détecteurs de défaillances parfaits : par exemple (figure 11), à intervalle régulier chaque processus envoie un message "Je-suis-vivant" à tous ; si le processus est vivant, ce message arrivera partout dans un délai borné (et connu), aussi chaque processus pourra, après ce délai, savoir sans erreur si le processus est vivant ou mort. Ce procédé très simple a cependant l'inconvénient de nécessiter un échange important de messages (de

[23] C'est vrai parce qu'ici le détecteur est donné et qu'on ne prend pas en considération la façon dont il est *implémenté*.

[24] Il faut cependant nuancer le propos : on montre que dans un système synchrone on peut construire un détecteur de défaillances parfait (et donc un détecteur au sommet de la hiérarchie définie par [21]). Rappelons, cependant, qu'à partir de la définition de base, un détecteur est simplement un oracle qui ne dépend que du schéma de panne. On peut donc concevoir des détecteurs de défaillances irréalistes qui pourraient, par exemple, deviner à la première invocation toutes les pannes qui vont se produire. Il est bien évident qu'on ne pourra pas simuler un tel détecteur avec un système synchrone...

l'ordre de n^2 à chaque intervalle) avec le risque de saturer le réseau de communication. Même si certaines améliorations ou d'autres protocoles [27] permettent de diminuer le nombre de messages échangés [72, 2], si l'on veut maintenir une détection rapide des défaillances, il n'en reste pas moins que la détection restera coûteuse.

Si, maintenant, on considère un modèle de calcul en rondes synchronisées, on a directement des détecteurs de défaillances parfaits : chaque processus considérera comme mort tout processus dont il n'a pas reçu de messages à la fin de la ronde.

On déduit trivialement de ce qui précède qu'un système synchrone ou un modèle en rondes synchronisées pour des pannes franches de processus permet de simuler un système asynchrone avec des détecteurs de défaillances parfaits.

Réciproquement, il est généralement admis que supposer des détecteurs de défaillances parfaits revient au même que d'être dans un système synchrone. Bien entendu, par définition, un modèle synchrone fait référence au temps, ce qui n'a pas de sens en asynchrone, on doit donc se restreindre aux spécifications qui ne font pas intervenir le temps [26]. Pour simplifier, on considérera le modèle de rondes synchronisées comme abstraction du synchrone. On va donc comparer la puissance du modèle des rondes synchronisées avec celle de RAP.

On a vu dans la section 2.6 que la comparaison des modèles de calcul n'était pas simple et dépendait fortement des propriétés de la simulation considérée : d'un côté, [26] affirment "with regard to both solvability and efficiency, we have shown that the synchronous model [25] is better than the asynchronous model equipped with a perfect failure detector" alors que, d'un autre côté, on peut montrer que l'on peut toujours simuler des rondes synchrones par des rondes asynchrones avec $\mathcal{P}$. On va donc voir dans la suite plusieurs résultats permettant de donner un sens précis à la comparaison entre le modèle des rondes synchronisées et l'asynchrone utilisant des détecteurs de défaillances parfaits.

En résumé, les résultats obtenus montrent qu'il existe une traduction très simple de $Sync$ dans RAP : deux rondes obtenues avec des détecteurs parfaits simulent une ronde de $Sync$. La simulation correspondant à cette traduction est aussi une simulation des états : un état du protocole traduit correspond explicitement à un état du protocole simulé ; de plus, cette traduction conserve les processus corrects. Cependant, du fait de ce doublement des rondes, seules les spécifications insensibles [26] à ce doublement sont directement résolues par la traduction. De plus, [26] ont exhibé un problème simple, ayant un certain intérêt pratique en synchrone, qui ne peut pas être résolu directement dans RAP alors qu'il peut l'être dans $Sync$. L'exemple proposé par [26] revient à détecter qu'un processus est initialement mort et donc tel qu'aucun de ses messages n'est reçu [27]. Reste donc que, d'un point de vue pratique, quitte à doubler le nombre

[25] En fait, ce qu'ils appellent "synchronous model" correspond au modèle des rondes synchronisées.

[26] Ce qui est le cas pour de nombreux problèmes comme celui du consensus.

[27] Il s'agit clairement d'un problème qui, dans sa spécification, est sensible au doublement des rondes : avec la simulation esquissée plus haut, il est impossible de déterminer si un processus meurt au cours de la première ou de la deuxième ronde.

de rondes, on arrive à résoudre par *simulation* toutes les spécifications qui sont résolubles pour $Sync$.

Mais, du point de vue des performances, cette traduction double le temps. Peut-on améliorer ce coût et passer d'un doublement à un décalage constant ? On peut définir de la sorte une simulation avec décalage constant m : à la ronde $l+m$ chaque processus détermine (de façon cohérente entre les processus) quel était l'état de la l-ième ronde simulée. Le résultat essentiel est alors qu'il n'existe pas de traduction permettant un décalage de moins de $t+1$ rondes (t est le nombre de pannes que le protocole peut tolérer) même en se restreignant à des spécifications ne concernant que les processus corrects.

Il convient d'opposer à ce résultat plutôt négatif deux arguments.

Le premier est que, la classe de problèmes qui nécessite ce décalage de $t+1$ est relativement restreinte : en particulier, en ce qui concerne le problème du consensus, nous avons montré dans [41] que, quelque soit le nombre de pannes tolérées, le consensus dans RAP peut être résolu en $\min(t+1, f+2)$ [28] (f est le nombre de pannes *effectives* pour l'exécution). Cette borne est la même que celle que l'on obtient pour des rondes synchrones. Et donc, au moins en ce qui concerne le consensus, les performances sont aussi bonnes qu'en synchrone [29].

Le deuxième argument est que si l'on veut simuler des rondes *parfaitement* synchrones, il faudra une simulation avec *exactement* le même décalage à partir de rondes synchronisées ou à partir de rondes avec $\mathcal{P}$. Intuitivement, ce sont les rondes parfaitement synchronisées qui sont utiles, et pour de nombreux problèmes (par exemple, le consensus, la diffusion atomique, etc...), on est amené à réaliser des rondes parfaitement synchronisées à partir de rondes synchronisées.

Et donc, on peut considérer que les différences de puissance entre les deux modèles de calcul (rondes synchronisées et rondes avec $\mathcal{P}$) sont faibles ; elles concernent, en définitive, surtout des problèmes construits pour cela et ne sont donc pas fondamentales et peuvent être compensées par le caractère plus abstrait de l'approche utilisant les détecteurs de défaillances [30].

Enfin, nous terminerons cette section par des résultats concernant le détecteur de défaillances $\mathcal{S}$. Rappelons en quoi consiste le système qui correspond à ce détecteur : pour celui-ci (appelé $Fort$ dans la section 2.3.3), il n'existe pas de bornes sur les délais de communications de *tous* les processus, mais il existe au moins *un* processus correct qui respecte une certaine borne δ sur les délais de communication [31]. En utilisant des *time-out*, on peut construire, dans un tel

[28] On a montré, en fait, ce résultat pour le problème de l'accord global qui est plus fort que le consensus.

[29] La borne pour le synchrone provient de [46].

[30] Notons, d'autre part, que les rondes que l'on obtient avec un détecteur de défaillances parfait ont d'autres caractéristiques que l'on a pas notées ici et qui peuvent permettre, dans certains cas, comme nous l'avons indiqué au début de cette section, d'avoir des exécutions plus rapides qu'avec des rondes synchrones.

[31] Il ne s'agit pas à proprement parler d'un système synchrone, puisque la borne sur les délais de communication n'est pas respectée par tous les processus. Mais, le fait que cette borne soit connue et respectée *dès le début* rapproche un tel système d'un système synchrone (en particulier on peut résoudre le problème du consensus en temps borné). Nous réservons le terme partiellement synchrones aux systèmes pour lesquels, les bornes sont soit inconnues soit ultimement respectées.

système, des détecteurs de défaillances de la classe $\mathcal{S}$. On associe à $\mathcal{S}$, un modèle de calcul en rondes (RAS cf section 2.4.4).

Dans ce modèle, nous avons montré [34] que, quelque soit le nombre de pannes tolérées, le consensus ne peut être résolu en moins de n rondes (ou ce qui revient au même avec une latence inférieure à n). Par ailleurs, il existe des algorithmes atteignant cette borne [79, 21]. Le fait que le nombre de pannes tolérées par l'algorithme n'intervienne pas est plutôt étonnant.

Les résultats évoqués dans cette introduction sont développés dans les sections suivantes.

```
/* δ : borne sur les délais de communication */
/* γ : borne sur les temps de calcul de 1,2,3 */
/* protocole en rondes : Π_p = < T_p, O_p > */
Ronde r :
 Au temps t_r = r(δ + γ) faire :
1   send(O_p(S, r)) to all        /* O_p(S, r) : message de p pour la ronde r */
2   M_p(r) := { messages reçus dans l'intervalle [t_r, t_r + δ] }
3   S := T_p(S, M_p(r), r)                          /* S est le nouvel état */
4   r := r + 1                                      /* ronde suivante */
```

FIG. 19 – Rondes synchronisées dans un système synchrone.

3.2 Rondes synchronisées et détecteurs parfaits

3.2.1 De synchrone aux rondes synchronisées

Il faut tout d'abord insister sur le fait que réaliser des rondes synchronisées dans un système synchrone défini par des bornes sur les délais de communications et les pas de calcul des processus n'est pas sans coût. Le principe pour y arriver est simplement d'attendre à chaque fois les délais *maxima* donnés par les bornes [32]. La figure 19 présente une réalisation possible des rondes synchronisées à partir d'un système synchrone. Intuitivement ce que fait cette réalisation est clair. Cependant, on pourrait être plus précis et utiliser le contexte défini dans la section 2.6 : il s'agit d'une traduction d'un protocole quelconque Π d'un modèle de calcul en rondes synchronisées qui sera exécutée dans un système asynchrone. Ici la traduction est très simple et consiste simplement à réaliser des rondes synchronisées. De façon claire, tout protocole en rondes synchronisées peut être traduit, c'est-à-dire exécuté dans un système asynchrone. La relation de simulation est par ailleurs triviale : elle efface simplement les actions ajoutées pour réaliser les rondes. De ce fait, le formalisme introduit précédemment n'est pas très utile ici.

Le point important est que δ est la borne sur les délais des communication et γ une borne supérieure sur le temps que peuvent mettre les processus à exécuter les instructions des rondes. Pour simplifier, on a supposé que les processus disposaient d'une horloge globale parfaite.

Du point de vue des performances, on voit bien que la détermination de ces bornes est cruciale.

On vérifie aisément que l'on obtient les propriétés (3), des rondes synchronisées à savoir qu'à chaque ronde, les processus vivants reçoivent exactement les mêmes messages sauf en ce qui concerne le message d'un processus qui meurt dans la ronde (pour ceux-là certains peuvent le recevoir et d'autres non).

[32] On peut réaliser des rondes plus rapides telles que, s'il n'y a pas eu de pannes, le temps écoulé entre deux rondes ne dépende que du temps des communications *réalisées* entre ces deux rondes et non de la borne *maximale* sur les communications. Mais, en cas de pannes, on sera obligé d'attendre au moins cette borne avant de passer à la ronde suivante.

En particulier, comme l'indique la propriété (6) de la section 2.4.3, en se basant sur la ronde pendant laquelle pour la première fois un processus n'a pas reçu de message d'un processus q, la vision de la date de la mort de q (à quelle ronde q est mort) n'est pas exactement la même pour tous : elle peut différer d'au plus une ronde.

3.2.2 Des rondes synchronisées aux rondes parfaitement synchronisées

Rappelons que dans les rondes *parfaitement synchronisées*, à chaque ronde, tous les processus vivants, ayant reçu les mêmes messages, ont, à la fin de la ronde, les *mêmes* informations.

Le type de pannes que l'on considère a des conséquences concernant les rondes synchronisées. Avec ce modèle de pannes, un message émis par un processus vivant à la fin de la ronde i sera reçu par tous les processus vivants à la fin de cette ronde, mais, si un processus meurt au cours de cette ronde, certains processus pourront avoir reçu son message alors que d'autres ne l'auront pas reçu. Ainsi, contrairement aux rondes parfaitement synchronisées (section 2.4.2), avec des rondes synchronisées les informations partagées par les processus vivants à la fin d'une ronde ne sont pas toujours les mêmes.

Ce décalage peut être à l'origine d'un phénomène dit de *contamination* [56, 55] : au pire il faudra attendre $t + 1$ rondes pour pouvoir assurer que tous les processus aient la même information.

En effet, considérons la séquence : en ronde 1, p_1 tombe en panne, seul p_2 reçoit de p_1, p_2 tombe en panne dans la ronde 2 et seul p_3 reçoit de p_2 et ainsi de suite. Cette séquence ne peut pas être plus longue que $t + 1$ puisqu'à chaque pas un processus tombe en panne. Par ailleurs, à chaque ronde au moins un processus est contaminé : il possède des informations différentes des autres (il a, contrairement aux autres processus vivants, dans son passé causal le message issu de p_1).

Ce qui distingue le modèle des rondes synchronisées du modèle des rondes parfaitement synchronisées est l'accord entre les processus sur les messages reçus dans les rondes ; si r est la ronde parfaitement synchronisée que l'on veut simuler, le point essentiel pour la simulation est d'arriver à un accord entre les processus vivants sur les messages échangés en r. Pour cela définissons *l'accord fort sur un vecteur*(figure 20) : il s'agit d'un accord sur un vecteur de valeurs, tel que toutes les valeurs proposées par des processus corrects (et même simplement vivants au moment de la décision) figurent dans ce vecteur. Il s'agit d'un problème équivalent à celui du *Global data* de [65, 41] que nous reverrons dans la section 6.3.

Considérons une ronde r ; si, au cours de cette ronde, chaque p a reçu l'ensemble de messages M_p, en prenant ces M_p comme valeurs proposées, avec un algorithme réalisant un accord fort, les processus arriveront à un accord sur un *même* ensemble de messages reçus dans la ronde r, en assurant que cet ensemble contient (au moins) tous les messages émis par les processus corrects. De cette façon, à la fin de l'algorithme, ils peuvent reconstituer le même état simulé co-

$D_p =< d_1, \ldots, d_n >$ est le vecteur de décision de p.
v_p valeur proposée par p.

Intégrité : si $D_p =< d_1, \ldots, d_n >$ est un vecteur de décision, alors pour tout j, $1 \leq j \leq n$, $d_j = v_j$ ou $d_j = \perp$,

Accord : si p et q décident alors $D_p = D_q$,

Terminaison : tout processus correct p décide un jour,

Validité forte : si q est vivant lors de la décision de $D_p =< d_1, \ldots, d_n >$ de p, $d_q = v_q$.

FIG. 20 – Accord fort sur un vecteur.

```
/* Π_p^s =< T_p^s, O_p^s > protocole simulé par p */
/* S_r^s état simulé pour la ronde r */
Simulation de la ronde r :
1   E_p := O_p^s(S_r^s, r)          /* E_p : messages ronde r pour la simulation */
2   send(E_p) to all
3   /* M_p(r) : messages reçus par p dans la ronde r */
4   /* Se mettre d'accord sur l'ensemble des messages reçus pour r */
5   T_p := AccordFortVecteur(M_p(r))
6   S_{r+1}^s := T_p^s(S_r^s, T_p, r)                    /* nouvel état simulé */
```

FIG. 21 – Simulation générique des rondes parfaitement synchronisées.

hérent avec les rondes parfaitement synchronisées (mêmes messages reçus dans la ronde). La figure 21 présente cet algorithme.

Notons que, si on veut garder la structure des rondes il faut que, dans l'algorithme d'accord, les processus atteignent l'accord dans la *même* ronde.

Il reste donc à utiliser ce schéma d'algorithme dans le cadre des rondes synchronisées et à utiliser un algorithme d'accord fort sur un vecteur.

Pour cela, on peut utiliser l'algorithme du *Flood Set* [76] (figure 22). Pour assurer l'accord sur un vecteur de valeurs, à chaque ronde, les processus retransmettent et mettent à jour toutes les informations qu'ils ont. Au bout de $t + 1$ rondes, ils auront exactement le même vecteur. En effet, on vérifie aisément que, si au cours d'une ronde, aucun nouveau processus ne tombe en panne, à la fin de cette ronde tous les processus vivants ont exactement la même information. Comme au plus t processus peuvent tomber en panne, parmi les $t + 1$ rondes, il y en a au moins une pour laquelle aucun nouveau processus ne tombe en panne. De même, on vérifie que l'information ne peut que se maintenir, ce qui assure qu'à la fin de ces $t+1$ rondes tous les processus vivants ont la même information sur cette ronde initiale : ils sont tous d'accord sur les messages reçus dans cette ronde. On peut aussi voir cette transformation comme réalisant une couche de

```
   /* D_p := < d_p^1, ... , d_p^n > */
 1 D_p := < ⊥, ..., v_p^0, ..., ⊥ >                    /*initialisation de D_p */
 2 for r := 0 to t do
 3     /* Ronde r */
 4     send(D_p) to all
 5     /* v^q : messages reçus dans la ronde r de q */
 6     forall V =< v^1, ... , v^n >   reçu dans la ronde do
 7         forall i do
 8             if (d_p^i = ⊥) then d_p^i := v^i
 9         od
10     od
11 od
12 dec_p := ⋃ { d_p^i | d_p^i ≠ ⊥ }
13 /* dec_p est l'ensemble de décision */
```

FIG. 22 – Algorithme du *Flood Set*.

communication par diffusion *fiable* [33]. Ainsi, si p apprend par un autre processus que q a émis un message dans la ronde de départ, à la ronde suivante, il diffuse cette information.

En composant le *Flood Set* et l'algorithme précédent, on obtient la simulation de la figure 23.

Cet algorithme permet de traduire un protocole Π^s fonctionnant avec des rondes parfaitement synchronisées en un protocole fonctionnant avec des rondes synchronisées. Dans cette traduction, une ronde parfaitement synchronisée est transformée en $t+1$ rondes synchronisées : les rondes de $r(t+1)$ à $r(t+1)+t$ simulent la ronde r de Π^s. Il faut donc $t+1$ rondes synchronisées pour simuler une ronde parfaitement synchrone.

Si un processus meurt au cours de la ronde r, on peut ignorer le message qu'il a émis. Avec l'algorithme dérivé du *Flood Set* décrit plus haut, on obtient l'information maximale : si le message d'un processus mort à la fin de la ronde initiale (celle que l'on veut simuler) atteint au moins un correct, il sera reçu par tous. On peut aussi obtenir l'information en quelque sorte minimale : si le message d'un processus mort à la fin de la ronde initiale n'atteint pas tous les corrects, il est ignoré. Pour cela, il suffit de remplacer l'union dans la ligne 12 de l'algorithme de *Flood Set* (ou celle de la ligne 8 dans l'algorithme de la figure 23) par une intersection. La transformation reste correcte, puisque le comportement simulé reste compatible avec les rondes parfaitement synchronisées.

A partir de cette transformation, on peut aussi construire une traduction avec un *décalage* de $t+1$. L'écriture de cette traduction est assez fastidieuse. Elle consiste, en parallèle, à déterminer les messages de la simulation des $t+1$ rondes précédentes, et anticiper sur cette détermination en faisant tourner l'algorithme simulé sur toutes les valeurs possibles compatibles des messages et des états. Au

[33] Avec une diffusion fiable, un message est reçu par tous les processus ou par aucun.

```
   /* Π_p^s =< T_p^s, O_p^s >  protocole simulé par p  */
   /* S_r^s  état simulé pour la ronde r  */
   Ronde r(t+1) :
1     E_p  :=  O_p^s(S_r^s, r)          /* E_p  : messages ronde r pour la simulation */
2     send(E_p) to all
3     /* M_p(r(t+1))  : messages reçus par p dans la ronde r(t+1)  */
4     T_p := M_p(r(t+1))
5     for i := 1 to t do
   Ronde r(t+1)+i :
6        send(T_p) to all
7        /* M_p(r(t+1)+i)  : messages reçus par p dans la ronde r(t+1)+i  */
8        T_p := T_p  ∪  M_p(r(t+1)+i)
9     od
10    S_{r+1}^s  :=  T_p^s(S_r^s, T_p, r)
```

FIG. 23 – Simulation $t+1$ rondes en une.

bout de $r+t+1$ rondes, chaque processus a déterminé les messages de la ronde simulée r et peut choisir exactement quel était son état dans la ronde simulée r.

D'un point de vue pratique, cette transformation est, en général, peu intéressante car les processus doivent échanger des messages beaucoup plus grands (de l'ordre d'une exponentielle de $t+1$).

En conclusion, avec un décalage de $t+1$ dans le cas le pire, on peut réaliser des rondes parfaitement synchronisées avec des rondes synchronisées.

Réciproquement, on peut déduire des résultats classiques (par exemple [51, 46, 7]) que $t+1$ est une borne inférieure pour toute simulation : pour toute simulation de rondes parfaitement synchronisées par des rondes synchronisées, on ne pourra simuler une ronde r avant d'avoir atteint la ronde (synchronisée) $r+t+1$.

Notons cependant que l'on peut aussi utiliser des algorithmes d'accord fort sur un vecteur à terminaison au plus tôt (*early stopping*) (par exemple, [46]) c'est-à-dire qui terminent en un nombre de rondes dépendant du nombre de pannes effectives et non de celui des pannes tolérées. La borne prouvée par [46] montre alors que l'accord peut être atteint en $\min(f+2, t+1)$ où f est le nombre de pannes effectives pour l'exécution considérée.

3.3 Asynchrone avec détecteurs de défaillances $\mathcal{P}$

Dans ce paragraphe, nous allons montrer quel type de rondes on obtient directement avec des détecteurs de défaillances parfaits.

On a vu dans la section 2.5.3 comment on pouvait réaliser des modèles de rondes à partir de détecteurs de défaillances (algorithme de la figure 14).

Avec des détecteurs de la classe $\mathcal{P}$, on n'obtient pas exactement des rondes synchronisées. En effet, supposons qu'à la ronde $r+1$, le processus p tombe en panne. Il a reçu de tous les messages de la ronde r et a envoyé à certains ses messages de la ronde $r+1$; il est possible qu'un processus q détecte la panne de p avant même d'avoir reçu les messages de p pour la ronde r, de ce fait, q considérera que p est mort pendant la ronde $r-1$, par contre un autre processus q', mettant plus de temps à détecter la panne de p, recevra les messages de p pour la ronde r *et* la ronde $r+1$, par contre son détecteur de défaillances lui signifiera la mort de p au plus tard dans la ronde $r+2$, il considérera donc que p est mort dans la ronde $r+1$.

On a donc un décalage d'*au plus 2 rondes entre la vision de la date des défaillances*. Les propriétés (5) et (6) résument les propriétés des rondes que l'on obtient avec des détecteurs parfaits.

En fait, le problème provient de ce qu'il n'y a, dans les définitions, aucun lien entre la détection des pannes et l'envoi et la réception de messages [34]. Clairement, le système de rondes obtenues avec un détecteur de défaillances $\mathcal{P}$, RAP, est plus faible que les rondes parfaitement synchronisées. De plus, [26] ont exhibé une spécification relativement naturelle qui, du fait de ce décalage, ne peut être résolue dans RAP : avec des rondes utilisant des détecteurs parfaits, on ne peut pas distinguer entre un processus qui meurt après avoir envoyé un message (et qui donc a déjà exécuté un pas de calcul) d'un processus qui meurt sans faire de pas de calcul.

Si les rondes RAP obtenues directement en asynchrone en utilisant des détecteurs de défaillances parfaits, ne sont pas exactement des rondes synchronisées (on a un décalage de 2 entre la vision de la date des morts), on va dans la section suivante définir une traduction qui permet de simuler avec des détecteurs de défaillances des rondes synchronisées. La simulation sous-jacente n'est pas l'identité, mais elle a néanmoins toutes les bonnes propriétés (correspondance des états, un processus incorrect ne peut simuler un correct, etc...).

Ensuite, on montrera qu'on peut simuler des rondes parfaitement synchronisées avec des rondes avec $\mathcal{P}$ avec un décalage de $t+1$. On montrera ensuite qu'une simulation des rondes synchronisées par des rondes avec $\mathcal{P}$ nécessite un décalage de $t+1$. Le point est que, d'une part, de nombreux algorithmes (en particulier les problèmes de consensus) du modèle des rondes synchronisées peuvent être simulés avec un décalage de deux rondes [35] et que d'autre part, pour ob-

[34] Dans les implémentations "naturelles" des détecteurs de défaillances pour lesquelles la détection se fait en envoyant et en recevant des messages "je-suis-vivant" sur les *mêmes* canaux de communication que les autres messages, ce problème ne se pose pas : si q considère que p est mort, il ne peut y avoir de message en transit dans le canal de p vers q

[35] De plus, comme on le verra plus loin, on peut obtenir le consensus dans les rondes avec $\mathcal{P}$ en exactement le même nombre de rondes qu'avec des rondes synchronisées [41].

```
   /* Π_p^s =< T_p^s, O_p^s > protocole simulé par p */
   /* S_r^s état simulé pour la ronde r */
   ronde 2r :
1     E_p := O_p^s(S_r^s, r)          /* E_p : messages ronde r pour la simulation */
2     send(E_p) to all
3     /* M_p(2r) : messages reçus par p dans la ronde 2r */
   ronde 2r + 1 :
4     send(∅) to all
5     /* V_p(2r + 1) : processus dont p a reçu le message de la ronde 2r + 1 */
6     M_p^s(r) := { m ∈ M_p(2r)| source(m) ∈ V_p(2r + 1) }
7     S_{r+1}^s := T_p^s(S_r^s, M_p^s, r)          /* S_{r+1}^s est l'état simulée r pour Π_p^s */
```

FIG. 24 – Simulation deux rondes en une.

tenir des rondes *parfaitement* synchronisées, le *même* décalage est nécessaire à partir de rondes synchronisées qu'à partir de rondes avec $\mathcal{P}$. Ce qui montre que, malgré tout, la différence entre ces deux modèles est ténue.

3.3.1 Une traduction générale de RAP vers RS

Le principe de cette traduction est le même que celui des traductions définies dans [82]. Considérons un protocole Π défini pour un modèle de rondes synchronisées, à chaque ronde de ce protocole correspondront deux rondes dans la traduction. La traduction est décrite dans la figure 24.

La première de ces deux rondes ne fera que simuler exactement celle du protocole d'origine (envoi des mêmes messages suivant l'état du processus) ; dans la deuxième ronde, tous les processus envoient un message vide.

A la fin de la deuxième ronde, chaque processus considère tous les messages qu'il a reçus. Il considère un processus p comme étant correct s'il a reçu de ce processus p *les messages des 2 rondes*, dans tous les autres cas, il considère p comme incorrect et ignore le message de p qu'il a reçu dans la première ronde.

Il suivra ensuite le comportement du processus simulé (défini par l'état qu'il simule au cours de la première ronde et par les messages qu'il a reçus sans les ignorer au cours de la première ronde) pour changer d'état.

La simulation est alors très simple : les états de la traduction correspondent naturellement à ceux du protocole. La simulation est une fonction qui réduit les deux rondes en une seule. On vérifie aisément qu'après cette réduction, on a bien des rondes synchronisées, en particulier que la vision des rondes correspondant à la mort des processus est bien cohérente avec celle de rondes synchronisées.

Soit $\alpha = r_1^1 r_1^2 ... r_m^1 r_m^2 ...$ une exécution de la traduction simulant $\alpha' = r_1 ... r_m ....$ Les processus vivants simulent des processus vivants et les processus morts simulent aussi des processus morts. En effet, si p est vivant, pour α, à la fin de la ronde r_m^2, il est aussi considéré comme vivant à la fin de la ronde r_m de l'exécution simulée α'. Cependant, et c'est effectivement le cas considéré par [26], les dates des morts ne se correspondent pas exactement, et le fait que p meurt juste

après avoir fini r_m^1 n'est pas distingué du fait qu'il meurt dans la ronde r_m^2.

Par ailleurs, pour α', pour p et p' vivants dans la ronde où q tombe en panne pour α', on vérifie aisément que :

$$|Der(p,q) - Der(p',q)| < 2 \tag{9}$$

Ce qui permet de montrer que α' est bien un comportement de rondes synchronisées.

Cependant, il est clair que, d'une part, la simulation est coûteuse : elle double le nombre de rondes du processus qu'il simule, d'autre part, toute spécification sensible au doublement des rondes ne sera pas directement résolue par la traduction, mais uniquement simulée. Mais, pour de nombreux problèmes classiques comme ceux dérivés du consensus la résolution modulo la simulation considérée ici permet la résolution directe de la spécification. Ce qui n'est pas le cas, par contre, avec la spécification de l'exemple donné par [26] qui impose un comportement particulier si un processus meurt au cours de la première ronde sans atteindre aucun processus.

3.3.2 Simulation des rondes parfaitement synchronisées (RPS) par des RAP

Dans la section 3.2.2 on a montré que si l'on disposait d'un algorithme d'accord fort sur un vecteur on pouvait en déduire une simulation des rondes parfaitement synchronisées. Plus précisément, si cet algorithme d'accord fort terminait en k rondes, on pouvait construire une simulation des rondes parfaitement synchronisées soit en simulant une ronde par k rondes, soit par décalage de k rondes. On peut appliquer ce qui précède aux rondes avec $\mathcal{P}$.

Nous avons montré dans [41] que l'on peut réaliser l'accord fort sur un vecteur en $\min(f+2, t+1)$ rondes. On en déduit donc en appliquant cet algorithme à la traduction proposée dans la figure 21 que l'on obtient une simulation d'une ronde parfaitement synchronisée par $t+1$ rondes avec $\mathcal{P}$. On en déduit aussi que l'on peut simuler un modèle de rondes parfaitement synchronisées avec un modèle de rondes avec $\mathcal{P}$ et décalage de $t+1$. Il faut noter que, mis à part le cas où $t = n-1$, les performances sont donc exactement les mêmes qu'avec des rondes synchronisées, ce qui montre que l'on ne perd rien en utilisant des détecteurs de défaillances dans un système synchrone pour tous les problèmes qui nécessitent la construction de rondes parfaitement synchronisées.

3.3.3 Impossibilité d'une simulation des RS par des RAP avec décalage de moins $t+1$

Remarquons tout d'abord que les rondes parfaitement synchronisées étant un cas particulier de rondes synchronisées, la simulation précédente de rondes parfaitement synchronisées par des rondes avec $\mathcal{P}$ est aussi directement une simulation avec décalage de $t+1$ des rondes synchronisées. En montrant qu'il n'existe pas de simulation avec décalage inférieur à $t+1$, on montre que la simulation précédente est optimale en ce qui concerne le nombre de rondes.

$v_p \in 0, 1, 2$ valeur initiale p
d_p valeur de décision

Intégrité : la valeur de décision d_p une des valeurs initiales v_q,

Accord partiel : si p et q décident alors $|d_p - d_q| \leq 1$,

Terminaison : tous les processus corrects décident.

FIG. 25 – Accord partiel.

```
   /* v_p valeur initiale de p pour le consensus */
   /* transformation des valeurs initiales */
1  forall p
2       if v_p = 1 then V_p := 2

3  Appliquer un algorithme d'accord faible avec valeurs initiales V_1, ..., V_n
4  /* D_p décision de p pour l'accord faible est aussi la valeur de consensus */
```

FIG. 26 – Transformation du consensus en accord faible.

Pour montrer cette impossibilité nous allons exhiber un problème qui peut se résoudre immédiatement dans un système de rondes synchronisées et en au moins $t+1$ rondes dans un modèle de rondes avec $\mathcal{P}$. Le problème que nous allons considérer revient essentiellement à déterminer la date de la mort de processus. Comme on l'a vu précédemment, la vision de la date de la mort des processus diffère d'au plus une ronde dans le cas des rondes synchronisées alors qu'elle peut atteindre deux rondes pour des rondes avec $\mathcal{P}$. Avec des rondes avec $\mathcal{P}$, il faut $t + 1$ rondes pour arriver à une vision cohérente, à une ronde près, des dates des morts.

Plus précisément, considérons le problème d'accord partiel (figure 25) pour lequel les processus peuvent avoir uniquement trois valeurs initiales (0, 1 ou 2) et doivent décider sur une des valeurs initiales, de façon à ce que la différence entre les valeurs de décisions soit d'au plus un.

Il faut au moins $t+1$ rondes pour réaliser cet accord faible. Sinon, on pourrait résoudre le consensus binaire en moins de $t + 1$ rondes. En effet, en appliquant la transformation décrite dans la figure 26, on vérifie aisément que l'on obtient un consensus binaire (puisque la valeur de décision est, dans ce cas, unique). Or, il existe au moins une exécution en $t + 1$ rondes pour le consensus binaire (par exemple [51, 46, 7]).

Maintenant, considérons pour chaque processus p la valeur $D(p, q)$ (la dernière ronde pour laquelle p a reçu un message de q). Dans le cas des rondes synchronisées, pour chaque q incorrect, on a $|D(p, q) - D(p', q)| \leq 1$, ce qui signifie que l'accord faible est réalisé immédiatement, dès que p ne reçoit pas de message de q.

Rappelons qu'avec des rondes avec $\mathcal{P}$, on a $|D(p, q) - D(p', q)| \leq 2$. Pour

obtenir des valeurs simulées cohérentes, $D^s(p,q)$, de la mort de q pour une simulation des rondes parfaitement synchronisées (c'est-à-dire telles que $|D^s(p,q) - D^s(p',q)| \leq 1$) il faudra nécessairement d'après ce qui précède $t+1$ rondes.

Ainsi, il faudra dans certains cas un décalage de $t+1$ pour simuler des rondes synchronisées avec des rondes avec détecteur de défaillances parfait.

3.3.4 Consensus et autres problèmes

De façon assez étonnante, pour résoudre le consensus les rondes synchronisées ne sont pas plus rapides que les rondes avec $\mathcal{P}$. La raison en est que, en fait, pour réaliser le consensus on est amené à construire des rondes parfaitement synchronisées pour lesquelles, comme on vient de le voir, il faut dans les deux cas un décalage de $t+1$.

En fait, montrer que le consensus peut s'obtenir avec des rondes avec $\mathcal{P}$ en $t+1$ rondes est très simple : il suffit de vérifier que l'algorithme du *Flood Set* (figure 22) fonctionne aussi avec ce type de rondes.

Au lieu de compter le nombre de rondes en fonction du nombre de pannes tolérées par l'algorithme, on peut aussi compter en fonction du nombre de pannes *effectives* (c'est-à-dire en fonction du nombre de pannes qui se sont effectivement produites pour l'exécution considérée). C'est ce que l'on appelle les algorithmes à terminaison au plus tôt (*early stopping*), qui permettent d'obtenir le consensus avant les $t+1$ rondes pour les exécutions présentant un nombre de pannes inférieur à t. [45] ont montré qu'une borne inférieure pour ce genre d'algorithmes est $\min(f+2, t+1)$ où f est le nombre de pannes effectives dans l'exécution. Cette borne s'applique aux rondes synchronisées et une légère adaptation permet de l'étendre aux rondes avec $\mathcal{P}$.

Il existe plusieurs algorithmes atteignant cette borne pour des types de pannes plus sévères que les pannes définitives (pannes byzantines et *send* ou *receive omissions* [36]). On peut aisément montrer que ces algorithmes s'appliquent aussi à des rondes avec $\mathcal{P}$ [37]. Cependant, ces algorithmes soit ne fonctionnent pas pour un nombre quelconque de pannes, soit n'assurent pas l'uniformité. Dans [41], nous avons proposé un algorithme de consensus (uniforme) permettant d'atteindre ces bornes avec un nombre quelconque de processus en pannes [38]pour des rondes avec $\mathcal{P}$. Nous décrirons avec plus de détail cet algorithme dans la section 6.3.

On déduit de ce qui précède que, même si utiliser en asynchrone des détecteurs de défaillances parfaits ne permet pas, dans le cas général, d'assurer les mêmes performances que dans un système synchrone, en ce qui concerne le consensus il n'y a pas de pénalité à utiliser des détecteurs parfaits.

---

[36] Avec des pannes byzantines, un processus incorrect a la possibilité de faire n'importe quoi, avec des *send* ou *receive omission* il peut uniquement ne pas émettre ou recevoir certains messages.

[37] Avec des détecteurs parfaits, soit le dernier message émis, soit les deux derniers messages émis par un processus qui va mourir, peuvent ne pas être reçus par un processus correct : il s'agit donc d'un cas restreint de *send omission*.

[38] si $t < n-1$.

3.4 Rondes avec $\mathcal{S}$ (RAS)

Supposons maintenant que l'on dispose d'un système asynchrone utilisant des détecteurs de défaillances de la classe $\mathcal{S}$: ces détecteurs de défaillances finissent toujours par soupçonner les processus incorrects, mais peuvent aussi soupçonner des processus corrects, mais, *au moins un processus correct* ne sera jamais soupçonné par *aucun détecteur de défaillances*.

L'existence de ce processus correct (rappelons que correct signifie vivant pour toujours) en qui tout le monde a confiance *dès le début* permet de réaliser le consensus [21, 79].

Rappelons que le modèle de système correspondant à $\mathcal{S}$ est le système *Fort*. Plus précisément, pour réaliser naturellement un détecteur de défaillances de la classe $\mathcal{S}$ dans un système avec temps, on peut supposer qu'il existe un délai δ (connu) sur les temps de communications qui est respecté (pour toujours) pour au moins toutes les communications provenant d'un processus correct.

Comme précédemment, on peut aussi caractériser le système de rondes correspondant à ce système. Comme on l'a vu, la réalisation d'un système de rondes à partir de $\mathcal{S}$ (algorithme de la figure 14) permet de réaliser des rondes avec $\mathcal{S}$ (RAS). Pour obtenir le maximum possible d'information, sans pour autant attendre indéfiniment, à chaque ronde, les processus envoient à tous et attendent de recevoir d'au moins $n-t$ processus (le nombre minimal de processus corrects) *et* d'avoir reçu un message de tous les processus qui ne sont pas soupçonnés. (Dans un système avec temps et délai δ, à chaque ronde, chaque processus attend de recevoir d'au moins $n-t$ processus *et* attend un temps suffisamment long de façon à recevoir de tous les processus dont les messages respectent la borne δ de communication).

Une autre façon d'interpréter ce système de rondes consiste à considérer que l'on est dans un système synchrone avec des pertes de messages tel qu'à chaque ronde des messages peuvent se perdre (mais chaque processus en reçoit d'au moins $n-t$ processus) et tel qu'il existe un processus dont aucun message ne se perd.

Nous avons montré que dans un tel système, on ne peut pas obtenir le consensus en moins de n rondes (dans le pire cas) [34, 39]. La preuve repose sur les techniques classiques consistant à montrer que, sinon, on aurait des exécutions indistinguables menant à des décisions contradictoires.

Ce résultat permet de prouver que les algorithmes de [21, 79], ayant une latence de n sont optimaux. Mais le plus intéressant est qu'il montre qu'avec un détecteur de défaillances de la classe $\mathcal{S}$, le nombre de défaillances tolérées n'a pas d'impact sur les performances (pire cas) [34]. Ce qui montre l'originalité de cette classe de détecteurs de défaillances.

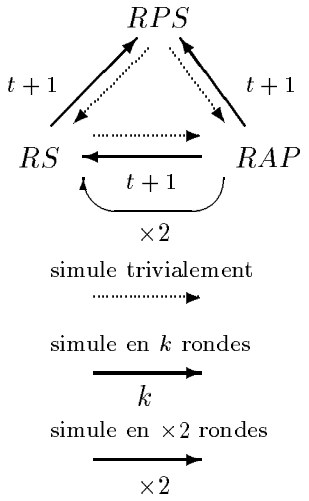


FIG. 27 – Simulations entre les modèles de rondes.

3.5 Conclusion

Résumons les résultats obtenus. Notons, d'abord, le résultat relativement étonnant concernant $\mathcal{S}$: le temps de décision ne dépend pas du nombre de pannes. Ce résultat a surtout un intérêt théorique dans la mesure où les systèmes correspondant à $\mathcal{S}$ (les systèmes $Forts$) ne sont pas très naturels.

D'un point de vue plus pratique, essayons de répondre à la question posée dans cette section : utiliser un détecteur de défaillances dans un système synchrone présente-t-il un intérêt ? En résumé, on voit donc que, pour l'essentiel, les rondes avec $\mathcal{P}$ et les rondes synchronisées reviennent au même. Plus précisément, en doublant toutes les rondes, on peut traduire tout protocole fonctionnant avec des rondes synchronisées en un protocole pour des rondes avec $\mathcal{P}$. La résolution modulo la simulation obtenue est raisonnable, mais ne permet pas de déterminer avec exactitude dans quelle ronde un processus mort a envoyé son dernier message.

Pour diminuer le coût de la simulation précédente, on peut utiliser une simulation avec décalage, mais on peut montrer dans ce cas qu'un décalage de $t + 1$ est nécessaire pour cette simulation. Cependant cette simulation permet d'obtenir directement des rondes *parfaitement* synchronisées à partir de rondes avec $\mathcal{P}$. Cela ne coûte donc pas plus cher d'obtenir à partir de rondes avec $\mathcal{P}$ des rondes parfaitement synchronisées que des rondes synchronisées et cela coûte aussi cher de construire des rondes parfaitement synchronisées à partir de rondes synchronisées. Or, *a priori*, ce sont les rondes parfaitement synchronisées qui semblent le plus naturellement utiles. En particulier, pour résoudre les problèmes de consensus, on est amené à construire ce type de rondes.

Tout ce qui précède peut donc justifier l'utilisation de détecteurs de dé-

faillances même pour des systèmes synchrones : il peut être intéressant de développer des algorithmes en supposant un mécanisme de détections de pannes sans pour autant supposer, pour l'algorithmique elle-même, l'existence de bornes connues sur les délais de communication.

4 Modèles en rondes de calcul pour l'asynchrone

Il s'agit essentiellement d'essayer d'être aussi proche que possible du synchrone avec de l'asynchrone ; pour cela, nous allons définir des modèles de calcul pour des systèmes (essentiellement) asynchrones qui reposent sur le paradigme des rondes synchronisées. Bien entendu, définir de tels modèles de calcul ne présente pas d'intérêt en soi ; pour être intéressant un modèle de calcul pour l'asynchrone doit :

1. (**puissance**) être suffisamment puissant pour permettre de résoudre des problèmes comme le consensus,
2. (**implémentation**) pouvoir être implémenté *efficacement* dans des systèmes partiellement synchrones avec ou sans notion de temps ou des systèmes asynchrones avec oracles suffisamment faibles (par exemple détecteurs de défaillances comme $\diamond\mathcal{P}$ ou $\diamond\mathcal{S}$),
3. (**conditions de faisabilité**) permettre de spécifier *explicitement* les conditions sur un système partiellement synchrone qui permettent leur réalisation. Notons que ce point n'est pas toujours réalisé comme le montre [20]. Ces conditions doivent pouvoir être vérifiées et être les plus réalistes possible.
4. (**abstraction**) pouvoir être utile pour les applications : le modèle doit simplifier le travail du programmeur et l'abstraire des conditions particulières d'un système donné.

Ces divers critères ne sont pas tous du même niveau, le dernier, par exemple, est d'ordre méthodologique (voire subjectif...). Cependant, comme nous allons le voir, ces critères peuvent servir de grille d'évaluation pour des modèles de rondes.

4.1 Présentation générale

Il est généralement admis que le développement des algorithmes est plus simple dans un cadre synchrone que dans un cadre asynchrone. Ainsi, les *synchroniseurs* d'Awerbuch [10] permettent, en l'absence de pannes, de simuler des rondes synchonisées dans un système asynchrone. De cette façon on pourra exécuter n'importe quel algorithme synchrone dans un système asynchrone. Cependant, si l'on admet des défaillances, même en se restreignant à des pannes définitives (*crash*) d'au plus un processus, de telles simulations sont impossibles. En effet, sinon on pourrait, par exemple, résoudre le consensus en asynchrone en exécutant un algorithme synchrone de consensus dans une telle simulation, ce qui est impossible.

En présence de défaillances on ne peut donc pas espérer obtenir une simulation complète. Tout au plus, peut-on essayer d'obtenir des exécutions le plus proche possible du synchrone : la simulation ne concernera qu'un sous-ensemble des processus corrects et ne durera pas toujours. Ainsi, les divers systèmes de *Virtual Synchrony* [8, 44, 91, 30, 48, 64, 67, 78, 85, 90, 93] vont essayer de maintenir un comportement synchrone à l'intérieur de groupes de processus.

En utilisant la terminologie des chapitres précédents, le problème qui nous intéresse ici consiste, d'une part, à définir des modèles de calculs suffisamment synchrones et, à étudier, d'autre part, dans quelle mesure ils sont réalisables dans un système asynchrone. Dans la mesure où l'on a vu précédemment que le modèle des rondes synchronisées est une bonne abstraction des systèmes synchrones, nous allons étudier des modèles basés sur le modèle des rondes synchronisées.

Propriétés des modèles de calcul en ronde pour l'asynchrone

Commençons par quelques remarques générales. Supposons que l'on essaie de réaliser des rondes. A chaque ronde, après avoir émis son message, un processus attend de recevoir le message de la ronde des autres processus. En présence de pannes de processus, en asynchrone, un processus ne pourra pas faire de distinction entre un processus mort et un processus dont le message est lent. Ainsi, on voit bien que le problème de la détection des pannes interviendra dans les propriétés que l'on veut assurer sur les rondes. Sans autres informations, pour assurer la progression des rondes, un processus ne pourra attendre que $n-t$ (t est le nombre maximum de processus qui peuvent tomber en panne) messages à chaque ronde avant de passer à la ronde suivante. Cependant, ces rondes ne sont pas très intéressantes : à chaque ronde, les processus ne reçoivent pas toujours les messages des *mêmes* processus. Remarquons que l'on peut obtenir un peu mieux [54] : à chaque ronde, deux processus vivants p et q reçoivent les messages d'au moins $n-t$ processus *et* soit l'ensemble des messages reçus par p est inclus dans celui des messages reçus par q, soit le contraire : $(M_p \subseteq M_q \vee M_q \subseteq M_p)$ [39]. Cependant, cette propriété ne permet, apparemment, pas grand chose de plus. En particulier, elle est insuffisante pour obtenir les propriétés d'accord entre processus. Pourtant, cet accord est nécessaire, et intuitivement, si, pendant suffisamment longtemps le système a un comportement proche du synchrone, le consensus devrait être réalisable.

Les modèles de calcul que nous allons considérer ici sont des modèles en rondes suffisamment puissants et proches des rondes synchronisées pour être utilisables (en particulier, ils permettront de réaliser le consensus) tout en restant le plus possible dans un cadre proche de l'asynchrone. D'un point de vue pratique, ils ne seront implémentables que dans des systèmes partiellement synchrones ou en utilisant des oracles comme les détecteurs de défaillances.

Précisons les diverses caractéristiques des modèles de calcul considérés. Tout d'abord, le comportement synchrone ne concernera pas tous les processus mais uniquement des *groupes* [40] de processus qui pourront changer. Les modèles de calcul doivent donc définir (a) ces groupes, ensuite, (b) une fois le groupe formé, ce que doit faire le groupe et quelles sont les propriétés de ces groupes et (c) comment on passe d'un groupe à l'autre. On aura donc :

[39] Suivant la notation introduite dans la section 2.4.1, M_p est l'ensemble des messages reçus par p dans la ronde.

[40] Insistons sur le fait que le terme de groupe est pris dans son sens le plus général de sous-ensemble de l'ensemble des processus. Il ne recouvre pas nécessairement le notion de groupes utilisée dans le *group membership*. Ici, les groupes peuvent être des groupes de fait définis pour une exécution, sans que les membres de ces groupes sachent s'ils en font partie ou non.

1. **Les membres :** alors que dans un système synchrone *tous* les processus peuvent participer aux rondes, il n'en va pas de même ici. En effet, on ne peut espérer que les processus obtiennent une information *exacte* sur les pannes, (sinon cela signifierait que l'on a une détection de pannes correspondant à des détecteurs parfaits $\mathcal{P}$, ce qui, comme on l'a vu, n'est pas très différent de supposer que le système est synchrone).

   Aussi, supposons que l'on réalise un système de rondes synchronisées (à chaque ronde les participants de la ronde émettent vers tous les autres et reçoivent de tous les autres avant de passer à la ronde suivante) : si *tous* les processus vivants participaient à ces rondes cela signifierait que tous les processus vivants savent exactement qui est vivant ou qui est mort. Aussi, les rondes ne concerneront pas forcément tous les processus vivants mais uniquement un sous-ensemble de *membres*.

   Le modèle de calcul aura donc à spécifier comment cet ensemble de membres est défini. Notons que, d'une part, les processus peuvent ne pas avoir connaissance des membres des groupes (par exemple, p et q peuvent ne pas savoir s'ils sont à un instant donné membres d'un même groupe). D'autre part, les groupes peuvent être constitués explicitement et volontairement par les processus, ou, au contraire, résulter d'évènements extérieurs (comme, par exemple, résultant de la vitesse de transmission des messages).

2. **Comportements synchrones dans les groupes :** une fois un groupe constitué, les processus doivent avoir un comportement proche de celui d'un système synchrone. Comme on s'intéresse à des modèles en rondes synchronisées, le comportement des membres du groupe sera, ici, du début à la fin de l'existence du groupe, un comportement en rondes synchronisées (à chaque ronde tous les participants envoient un message vers tous et attendent de recevoir de tous le message de la ronde avant de passer à la suivante). Comme pour les systèmes synchrones, cela présente l'avantage de la simplicité (on sait à chaque ronde que tous les membres doivent émettre un message). À cette approche plutôt opérationnelle, on peut préférer d'autres approches portant sur les propriétés des diffusions des messages dans le groupe. Dans ce cas, on supposera en général que tous les messages diffusés dans le groupe seront reçus par tous les membres du groupe dans le même ordre : tant que le groupe existe on a donc une sorte de diffusion atomique interne au groupe.

   Cependant, on ne pourra pas toujours assurer qu'un message proposé à la diffusion par un membre du groupe sera délivré par tous les membres du groupe. En effet, le groupe peut disparaître (par exemple, certains membres peuvent tomber en panne), et, dans ce cas, le message peut ne pas avoir été délivré par certains membres du groupe. Les propriétés garanties sont essentiellement des propriétés de sûreté (*safety*) (si m et m' sont délivrés par p et q, ils sont délivrés dans le même ordre). Il faut aussi spécifier ce qui doit se passer en cas de disparation du groupe.

3. **Propriétés des groupes :** la première propriété des groupes est la pos-

sibilité ou non de *partition* du système. En effet, les rondes ne concernant pas tous les processus corrects, on pourrait obtenir plusieurs rondes qui coexistent au même instant. On a alors une partition du système en plusieurs systèmes de rondes. Cette partition du système n'est pas, *a priori*, souhaitable [41], pour l'éviter, on supposera que les groupes comprennent un ensemble *majoritaire* de processus et qu'un processus n'est membre que d'au plus un groupe en même temps. Ainsi, il ne peut y avoir à un instant donné qu'un seul groupe actif.

La deuxième propriété des groupes concerne les propriétés qui doivent être assurées en cas de disparition/création des groupes. En effet, certains membres peuvent tomber en panne ou devenir trop lents. Dans ce cas, il faudra reconstruire un nouveau groupe pour recommencer des rondes. Certains modèles de calcul vont imposer certaines propriétés d'accord sur ce qui s'est passé dans le groupe précédent pour constituer le groupe suivant. On peut noter que, si on impose que tous les groupes successifs soient majoritaires, au moins un membre du précédent groupe est membre du groupe suivant ; ce membre peut servir à maintenir des informations sur le comportement du groupe précédent.

Virtual Synchrony

Les principes de divers systèmes de *Virtual Synchrony* sont, pour une grande part, les mêmes [42] : ce qui est appelé plus haut un groupe est appelé en *Virtual Synchrony* une *vue* (*view*). On peut alors considérer les systèmes de *Virtual Synchrony* comme réalisant les mêmes objectifs que ceux décrits précédemment : à l'intérieur d'une vue les processus ont un comportement synchrone. On peut donc les envisager suivant les trois critères que nous venons de voir. Il faut cependant garder à l'esprit que les divers systèmes de *Virtual Synchrony* ont des objectifs plus généraux que ceux développés ici et il n'est pas question de faire une analyse exhaustive de ces systèmes.

Il existe de nombreuses spécifications non équivalentes et souvent incomplètes ou même contradictoires de la *Virtual Synchrony* [9], ce qui fait qu'il est difficile de parler de *la Virtual Synchrony*. En s'inspirant de l'excellente synthèse de [92], nous allons néanmoins essayer d'en dégager les caractéristiques générales.

Concernant les membres des groupes, les systèmes de *Virtual Synchrony* supposent généralement que les processus savent s'ils sont membres d'une vue et qu'ils connaissent les identités des autres membres de la vue. De plus, pour la plupart de ces systèmes, un processus peut volontairement quitter la vue (ou en faire quitter d'autres). Certains systèmes insistent plus sur la notion de groupe : il s'agit de fournir un service permettant à des groupes de se constituer à la demande et ensuite d'avoir des primitives de communication de groupes. On

[41] Il existe cependant divers systèmes de *virtual synchrony* qui acceptent le principe de partitionnement du système.

[42] Les divers systèmes de *virtual synchrony* et/ou de group membership ont aussi souvent d'autres propriétés liées à la notion de groupes de processus.

parle alors de *group membership service*. De nombreux systèmes de *middleware* fonctionnent de cette façon [91, 44, 8, 18, 63]. Le problème avec ces systèmes est qu'ils ne donnent pas, en général, de façon explicite les conditions sur le système permettant d'assurer les propriétés qu'ils sont censés satisfaire. Comme on peut montrer que les spécifications, même faibles, de *group membership* sont impossibles à résoudre en asynchrone [20], il est nécessaire de pouvoir préciser quelles sont les propriétés supplémentaires que l'on suppose sur le système.

Concernant le comportement à l'intérieur des vues, on n'aura pas, en général, exactement des rondes au sens où on les a définies (tout le monde émet vers tous et reçoit de tous avant de passer à la ronde suivante), mais plutôt un comportement synchrone défini par les propriétés des diffusions dans le groupe (en fait, essentiellement, le fait que les messages diffusés sont reçus dans le même ordre par tous les membres de la vue).

Enfin concernant les propriétés des vues, certains modèles (en général les plus anciens) n'admettent pas la possibilité de partitions du système (par exemple ISIS [17], Consul [78] ou Ensemble [63]), alors que d'autres l'admettent (Transis [44], Horus [90], Totem [8], newtop [48], Relacs [13], RMP [93] ou Moshe [67]). L'existence de partitions rend difficile la spécification des propriétés de vivacité, mais on suppose alors en général que le système ultimement converge vers une seule vue (absence de partition ultime) ou vers plusieurs ; dans ce cas, ultimement, un processus vivant appartiendra définitivement à une seule vue (on peut imaginer que ce cas correspondrait à une sorte de partition physique du système en plusieurs composantes fortement connexes).

Une vue peut être abandonnée et une nouvelle vue installée. Dans ce cas, il faut pouvoir maintenir une certaine cohérence entre l'ancienne vue et la nouvelle. Par exemple, dans l'ancienne, certains processus peuvent s'être engagés dans telle ou telle décision, il faut alors que cet engagement soit maintenu et connu. Pour cela, on pourrait exiger que tous les membres de la vue aient la même "vision" de la vue c'est-à-dire qu'ils soient d'accord sur les messages reçus dans *la* vue. De cette façon, dans une vue suivante, en supposant que les vues contiennent toujours une majorité de processus, au moins un membre de l'ancienne vue sera un témoin de ce qui s'est passé dans cette ancienne vue et les engagements éventuels pourront être imposés dans la nouvelle vue. En fait, il suffit d'assurer la *propriété fondamentale* de la *virtual synchrony* d'après [92] : si deux processus participent à une vue v_1 et à la vue suivante v_2, alors ils ont la même vision de v_1.

Cependant, cette dernière condition est très forte [43], elle suppose un accord entre tous les membres de la vue et est, à notre avis, difficile à assurer. C'est pourquoi le système que l'on va proposer plus loin n'assurera pas *directement* cette propriété.

[43] Si, bien entendu, on suppose aussi des conditions de vivacité, à savoir que de nouvelles vues seront installées.

Phases synchronisées

Dans la section suivante nous allons étudier plus en détail un modèle de calcul en rondes synchronisées que nous avons défini et qui a fait l'objet d'une implémentation dans le cadre du projet ATR [11, 12]. Ce modèle est appelé systèmes de phases synchronisées [33] (SPS) et représente une des réalisations les plus importantes de ce rapport d'habilitation. L'idée du SPS découle simplement de ce qui a été présenté plus haut. Il s'agit d'obtenir un modèle de calcul en rondes synchronisées qui puisse être utile pour les applications. Pour cela, il faut, d'une part, définir une spécification du SPS qui soit suffisamment puissante pour être utilisable, et, d'autre part, il faut aussi que cette spécification puisse être résolue par une implémentation efficace dans un système suffisamment réaliste.

Précisons sommairement les caractéristiques d'un SPS. Pour simplifier, la description que nous allons faire des SPS est assez opérationnelle. Une exécution d'un SPS est une succession de *phases*, chaque phase ayant un ensemble de membres ; pendant une phase les membres de cette phase ont un comportement en rondes synchronisées tel qu'il a été défini précédemment. La notion de phase correspond à celle de groupe présentée plus haut. Le terme de phase sera dans la suite réservé aux groupes dans le modèle de calcul des phases synchronisées.

4.2 Phases synchronisées

4.2.1 Spécifications des phases synchronisées

Intuitivement, un système de phases synchronisées est une succession de *phases* ; durant chaque phase, les membres de la phase exécutent des rondes synchronisées. On trouvera les définitions formelles des systèmes de phases synchronisées dans [32, 36]. On trouvera aussi certaines applications des systèmes de phases dans [35, 40].

Plus précisément, considérons les trois critères exposés plus haut.

Membres : bien que cela ne soit pas indispensable pour spécifier les SPS, à une phase est associé un unique ensemble de membres et les processus savent quels processus sont membres d'une phase donnée. Dans les implémentations considérées, la constitution des phases dépend de mécanismes de détection de pannes et ce ne sont pas les processus qui décident de rejoindre ou de quitter une phase. Dans la suite, ph représentera une phase et $Member(ph)$ l'ensemble des membres de la phase ph.

Comportements synchrones dans les groupes : le comportement des processus durant une phase est un comportement en rondes synchronisées : à chaque ronde tous les membres de la phase envoient à tous et attendent de recevoir de tous les membres avant de passer à la ronde suivante. Les détails se trouvent Figure 28. Notons tout de suite deux propriétés importantes :

(P1) si deux membres d'une phase ph terminent la ronde k, ils ont reçu jusqu'à la ronde k les *mêmes* messages de ph.

(P2) si p termine la ronde $k+1$ de ph, tous les membres de ph ont *commencé* la ronde $k+1$.

Ainsi, jusqu'à ce que la phase échoue, tous les membres ont la même information ; seules les informations de la dernière ronde peuvent varier. En remplaçant "mort" par "changement de phase" et en arrêtant dès qu'il y a un changement de phase, on a une définition similaire à celle des rondes synchronisées du synchrone où les processus obtiennent exactement les mêmes informations à chaque ronde sauf si un processus meurt dans la ronde.

Propriétés des groupes : les phases contiennent une majorité de processus :

$$\forall ph : |Member(ph)| > \left\lfloor \frac{n}{2} \right\rfloor \tag{10}$$

et un processus p ne peut, au même instant, appartenir qu'à au plus une phase ph_p. Il s'agit donc d'un système sans partition. De plus, les phases sont identifiées de façon unique (et connues des processus) et peuvent être comparées (pour un ordre noté $\preceq$). On dira qu'une phase *réussit* si au moins un processus termine la première ronde. Soit $\mathcal{PR}$ l'ensemble des phases réussies pour une exécution, $\mathcal{PR}$ doit être un ensemble totalement ordonné :

$$\forall ph, ph' \in \mathcal{PR} : (ph \preceq ph') \vee (ph' \preceq ph) \tag{11}$$

```
1 for round := 0, 1, · , ∞                                   /* ronde round */
2     forall q ∈ Member(ph) send Mess_p(round) to q
3  /*Mess_p(round) est le message de p pour la ronde round de la phase ph */
4     wait until
5        receive(Mess_q) from all q ∈ Member(ph)
6     or
7        ph échoue
8     end wait
9 end for
```

FIG. 28 – Comportement rondes synchronisées pour le processus p

Remarquons que nous n'avons ici que des propriétés de sûreté (*safety*) : un système dans lequel $\mathcal{PR}$ est vide et dans lequel aucun processus ne fait rien est (trivialement) un système de phases synchronisées !

Nous allons maintenant considérer ce modèle de calcul du point de vue des critères indiqués au début de ce chapitre (puissance, implémentation et abstraction).

4.2.2 Puissance des systèmes de phases synchronisées : vivacité

Comme nous venons de le noter, il n'y a pas, pour l'instant, de propriétés de progression (*liveness*) dans les spécifications de notre modèle de calcul. De façon assez naturelle, les conditions de vivacité correspondront à la progression des rondes dans une phase : on dira qu'une phase ph est *k-synchronisée* si au moins un membre de cette phase termine la ronde k. Ainsi d'après (P1) et (P2), si ph est k-synchronisée, alors *tous* les membres de la phase *commencent* la ronde k et donc terminent la ronde $k-1$. Par extension, on dira qu'une phase ph est *définitivement synchronisée* si elle est k-synchronisée pour tout k.

On étend ces définitions aux exécutions et on dira qu'un système de phases synchronisées est :

- *k-synchronisé* si pour toute exécution il existe une phase k-synchronisée,
- *définitivement synchronisé* si pour toute exécution il existe une phase définitivement synchronisée,
- *infiniment souvent k-synchronisé* si pour toute exécution, il existe infiniment souvent des phases k-synchronisées ou une phase définitivement synchronisée.

Ce sont ces propriétés de vivacité qui vont permettre de résoudre les problèmes classiques d'accord.

Montrons comment on peut procéder pour résoudre le consensus.

A chaque ronde les processus proposent des valeurs pour le consensus. Si tous les membres d'une phase terminent la première ronde, il est clair qu'ils ont tous reçu les mêmes valeurs et pourraient donc prendre la même décision sur la base de ces valeurs reçues. S'il n'y a jamais d'autres phases, la décision ne pourra être que celle-là. Le problème se pose uniquement parce que cette

phase peut échouer. D'une part, un processus ne sait pas si les autres ont aussi terminé la ronde et, d'autre part, si la phase s'arrête, en cas de décision prise par certains durant cette phase, il faut assurer que toutes les autres phases ne pourront prendre que *cette* décision. Il faut donc un mécanisme qui permette d'assurer le maintien d'une décision potentielle entre les phases.

Pour cela, à la fin de la première ronde, un processus *s'engage* sur la valeur déterminée à partir de ce qu'il a reçu dans la ronde. Cet engagement signifie que s'il décide dans la même phase il décidera cette valeur et, si la phase échoue, quoi qu'il arrive il proposera dans les phases ultérieures cette valeur.

Il faut maintenant pouvoir garantir que si suffisamment de processus se sont engagés sur la même valeur, cette valeur sera la seule à pouvoir être la valeur de décision. Il faut faire en sorte que le choix se porte de préférence sur des valeurs sur lesquelles les processus se sont engagés et il faut, aussi, pouvoir choisir entre divers engagements possibles. Une solution simple consiste à choisir l'engagement le plus récent : chaque engagement sera étiqueté par la phase dans laquelle cet engagement a été pris. Les phases réussies étant totalement ordonnées (11), tous les processus pourront choisir la valeur provenant de l'engagement le plus récent.

Reste maintenant à déterminer les conditions qui permettent à un processus de décider. Si p termine la ronde 0, on a vu qu'il s'engage sur la valeur choisie à partir de ce qu'il a reçu dans cette ronde (celle étiquetée par la phase la plus récente). Si maintenant il termine la ronde 1 de cette phase, il sait que tous les membres de la phase ont aussi terminé la ronde 0 et se sont donc engagés sur la même valeur que lui. La propriété de majorité (10) assure qu'à chaque phase suivante au moins l'un d'entre eux participera avec sa valeur de décision et une étiquette qui assurera que seule cette valeur peut être choisie.

En résumé (cf. figure 29),

- Chaque processus propose une valeur à la première ronde de chaque phase. Cette valeur est étiquetée par la phase dans laquelle le processus s'est engagé sur cette valeur.
- Si cette ronde se termine, le processus s'engage sur la valeur reçue dans la ronde 0 étiquetée par le plus grand numéro de phase. Notons que dans ce cas tous les processus ont reçu le même ensemble de valeur et s'engagent sur la *même* valeur.
- Il participe ensuite à la ronde 1 en envoyant un message vide.
- Si cette ronde se termine, il décide avec la valeur sur laquelle il s'est engagé en ronde 0.

On voit assez immédiatement que le consensus est assuré dès que le système de phases synchronisées est 1-synchronisé.

Si l'on a présenté avec relativement de détails les principes qui permettent de résoudre le consensus dans un système de phases synchronisées, c'est que ces derniers sont tout à fait généraux et se retrouvent dans les divers algorithmes de consensus pour des modèles partiellement synchrones ou avec détecteurs de défaillances $\diamond\mathcal{S}$ ou $\diamond\mathcal{P}$ [21, 86, 70, 66, 79, 80] ou randomisés [15, 5]. Dans ces approches, les informations permettant d'obtenir le consensus sont incorrectes pendant un temps arbitrairement long. Ce n'est que de façon ultime que

```
   /* chaque valeur v a une étiquette label(v) */
   /* une étiquette est l'identification d'une phase */
 1 Pour chaque phase ph
 2     à la ronde 0 :
 3         proposer (v_p)
 4         soit M l'ensemble des messages reçus dans la ronde
 5         choisir v tel que label(v) = min{label(m)|m ∈ M}
 6         v_p := v ; label(v_p) := ph
 7     à la ronde 1 :
 8         proposer ∅
 9         if ronde 1 termine
10         then décider v_p
11                 diffuser la décision
```

FIG. 29 – Algorithme de consensus pour p

l'on obtient les conditions permettant d'assurer la propriété de terminaison du consensus (le détecteur de défaillances donne des informations exactes ou le tirage aléatoire finit par donner des valeurs assurant la terminaison). Dans ce cas, il faut assurer la validité (pas de décision contradictoire). Pour cela, on ne décidera d'une valeur que si on a la garantie que la valeur a été *verrouillée* chez suffisamment de processus (en fait un quorum [44]), c'est-à-dire si suffisamment de processus s'engagent à choisir cette valeur. Ainsi, si pour toute possibilité de décision on a un élément du quorum qui a verrouillé la valeur, on pourra, suivant les cas, imposer cette valeur verrouillée. Pour imposer cette valeur, soit on définira un ordre permettant de choisir entre des valeurs verrouillées (par exemple en définissant un ordre sur les dates du verrouillage des valeurs) , soit, dans le cas randomisé, on refusera de changer de valeur après le verrouillage.

Pour les phases synchronisées, suivant ce schéma, on a :

- **verrouillage :** le verrouillage de la valeur est assuré simplement par le fait que tous les processus ont terminé la ronde 0 d'une phase.
- **quorum :** la propriété de majorité permet d'assurer que, si une valeur est verrouillée, à chaque phase on en aura au moins un témoin du verrouillage.
- **imposer la valeur :** l'ordre total sur les phases permet d'imposer la valeur en choisissant celle étiquetée par la phase la plus récente.

Un avantage du modèle des phases synchronisées réside dans le fait que le caractère synchrone des rondes permet de simplifier les mécanismes de verrouillage et d'imposition de la valeur de décision. Cependant, l'avantage principal de ce modèle apparaîtra dans la facilité et la rapidité avec laquelle on pourra réaliser des consensus répétés (cf section 6.2.2).

Remarquons enfin que l'accord sur les membres de la phase permet une décision plus rapide : en cas de succès, dès la première ronde, la valeur de

[44] Un quorum est une famille d'ensembles telle que deux éléments de cette famille ont toujours un élément commun.

décision est acquise, la deuxième ronde ne sert qu'à confirmer la décision.

Terminons cette section par une remarque plus générale. Comme on l'a vu, les problèmes d'accord sont fondamentaux. Les divers algorithmes utilisés pour les résoudre présentent, comme on vient de le voir, de fortes similitudes. Il serait intéressant d'essayer de dégager les principes généraux des divers algorithmes d'accord. Les modèles de calcul peuvent permettre une approche uniforme de ces algorithmes de consensus ou plus généralement d'accord. Ces algorithmes sont finalement assez proches des algorithmes traditionnels de *commit* dans les bases de données [16].

4.2.3 Implémentation des phases synchronisées

Pour l'instant nous avons décrit uniquement les systèmes de phases en tant que modèle de calcul et montré comment ce modèle de calcul peut permettre de réaliser le consensus. Nous allons, dans ce paragraphe, nous intéresser à l'implémentation. Pour cela, on va procéder en deux temps ; dans un premier temps, on va montrer que ce modèle de calcul peut être réalisé dans un système asynchrone général mais en assurant uniquement les propriétés de sûreté (*safety*). Pour garantir des propriétés de vivacité (*liveness*), il faudra (bien sûr) des conditions supplémentaires sur le système. Il faudra soit supposer l'existence de détecteurs de défaillances de la classe $\diamond\mathcal{P}$, soit supposer que l'on a un système partiellement synchrone.

Algorithme des phases

Décrivons rapidement l'algorithme permettant d'obtenir des phases synchronisées (on peut trouver les détails dans [32]). On suppose que chaque processus p dispose à chaque instant d'une *liste de confiance*, $Trusted_p$, contenant les processus avec qui il pense établir une phase. Cette liste peut provenir, par exemple, d'un mécanisme de détection de défaillances. Elle n'est pas supposée, pour l'instant, avoir des propriétés particulières.

Pour essayer de réaliser des phases réussies, chaque processus maintient $Phase_p$ un vecteur d'entiers indexé par les processus. Essentiellement, à chaque fois que sa liste de confiance change, un processus p passe à une nouvelle version et incrémente $Phase_p[p]$, sa valeur dans le vecteur de phases. Les autres coordonnées de ce vecteur, $Phase_p[q]$, représentent la connaissance que p a de la version à laquelle est arrivé q (on trouvera plus de détails dans la figure 30).

Ce vecteur sert d'identification de la phase dans laquelle p est engagé (ou veut s'engager). Les membres de la phase associée à $Phase_p$ sont les éléments de la liste de confiance de p. Pour qu'une phase réussisse, il faut que les processus concernés agréent sur leur vecteur de phase et sur leur liste de confiance :

$$\forall q \in Trusted_p : Trusted_p = Trusted_q \wedge Phase_p = Phase_q \qquad (12)$$

Pour établir une nouvelle phase, les processus vont échanger et mettre à jour leur valeur de phase jusqu'à ce que la condition précédente soit réalisée.

```
   Initialisation :
   /* Phase_p est un vecteur d'entiers indicé par les processus */
1  /* Phase_p est initialisé par le vecteur nul */

   /* Informer les autres des changements */
2  une seule fois en cas de changement de Phase_p :
3     send(Phase_p, Trusted_p) to all

   /* incrémenter Phase_p[p] si Trusted_p change */
4  Trusted_p a changé :
5     Phase_p[p] := Phase_p[p] + 1

   /* une valeur de phase supérieure existe */
6  receive(ph) from q ∈ Trusted_p
7     if (ph[r] > Phase_p[r]) pour un r ∈ Trusted_p
8     then Phase_p := max(Phase_p, ph)
```

FIG. 30 – Principe de l'algorithme des phases.

Quand un accord s'est établi entre les participants, ils travaillent ensuite en rondes synchronisées jusqu'à ce que le vecteur de phase change pour l'un d'entre eux.

On montre aisément que cet algorithme assure toujours les propriétés de sûreté (*safety*) des phases. Les propriétés de vivacité dépendront de l'évolution des listes de confiance. Comme la 1-synchronisation permet le consensus, pour obtenir au moins la 1-synchronisation on sera obligé de faire des hypothèses supplémentaires sur le système.

Notons cependant que, implicitement, la liste de confiance correspond à la vision qu'un processus a des autres concernant les pannes, mais que cet algorithme pourrait s'utiliser aussi pour constituer des groupes de processus volontairement constitués (il faut néanmoins que ces groupes soient des groupes de processus vivants contenant une majorité de processus).

Remarquons, d'autre part, que, grâce à une légère modification technique de l'algorithme [32], on peut se restreindre à n'utiliser que des *buffers* à une place pour la communication entre les processus (un nouveau message de p à q écrase le message précédent de p dans le buffer de q). Cette remarque servira plus loin (section 4.2.5).

Comme on vient de le voir, il faut faire des hypothèses supplémentaires pour obtenir la vivacité. Un des avantages de cet algorithme est qu'il sépare explicitement ces conditions supplémentaires qui s'expriment en termes de détection de défaillances du reste.

4.2.4 Modèles de calculs et détecteurs de défaillances

Groupes et détecteurs de défaillances

Nous allons voir dans ce paragraphe qu'il n'y a pas tant de différences entre la notion de groupes telle qu'elle existe dans les divers modèles de calculs pour l'asynchrone (*virtual synchrony* et systèmes de phases synchronisées) et l'approche des détecteurs de défaillances. De fait, construire un groupe de processus corrects dans un modèle de calcul, ou proposer des listes de suspects avec des détecteurs de défaillances revient à peu près au même. La différence essentielle réside dans le fait que, dans le premier cas, il faut aussi réaliser un accord sur le groupe en question.

Comme on l'a vu, les divers modèles de calcul pour l'asynchrone contiennent généralement une notion de groupes de processus, chaque groupe contenant l'ensemble des processus qui participent à la vue ou la phase. Plus généralement, au lieu de considérer le groupe qui participe on peut aussi se restreindre à ceux qui prennent les décisions. Par exemple, le groupe peut être réduit à un seul processus décideur correct auquel les autres processus délèguent la responsabilité des choix ; on a alors un *leader*. De même, on peut considérer que les membres des phases constituent le groupe (ici majoritaire) des décideurs : les autres processus corrects s'adressent à ce groupe qui prendra toutes les décisions. En ne s'intéressant qu'à des propriétés ultimes, il s'agit dans ces divers modèles de calculs de déterminer (et de s'accorder) ultimement sur un groupe de processus corrects. Plus précisément, suivant les propriétés (appelons les PG) que l'on désire sur les groupes obtenus, on devra assurer qu'il existe un temps τ à partir duquel, il existe un ensemble G de processus tels que :

(propriété des groupes) tout $p \in G$ est correct et G vérifie PG,

(accord) pour tout $\tau' \geq \tau$, au temps τ', tous les processus corrects sont d'accord sur le fait que G est l'*unique* groupe choisi.

Par exemple, les algorithmes de leader (ou de construction de Ω) [28, 80, 2] vérifient exactement ces conditions pour des groupes contenant exactement un seul processus : le leader. De même, dans un système de phases ultimement synchronisées, on aura un unique groupe majoritaire.

Remarquons que, comme on exige que les membres du groupe choisi soient tous corrects, on peut, si l'on dispose d'un algorithme de groupe vérifiant les propriétés ci-dessus, réaliser (relativement facilement, par exemple, [80]) le consensus. De ce fait, sans autres hypothèses sur le système, il n'existera donc pas de tels algorithmes en asynchrone. On parlera donc plutôt *d'oracle de groupe vérifiant la propriété PG*. Il s'agit simplement de détecteurs de défaillances particuliers qui ultimement proposent la *même* liste de processus corrects à tous.

Par exemple, considérons la classe des détecteurs de défaillance $\diamond\mathcal{P}_k$ telle que l'on ait :

Complétude : tout processus incorrect finira un jour par être définitivement suspecté (c'est-à-dire ne figurera plus dans les listes proposées par le détecteur de défaillances),

```
   /* OF_p est la liste des suspects pour p donnée par ◇ P_k */
   /* NF_p est le groupe proposé pour p donné par ◇ G_k */
1  Infiniment souvent :
2      M = max{C_p[q]} + 1
3      forall q ∈ OF_p do
4          C_p[q] := M                          /* C_p[q] max pour les suspects */
5      send C_p to all
       /* choisir les k processus les moins souvent suspectés */
6      NF_p := { q : |{r : (C_p[r], r) ≤ (C_p[q], q)}| < k}

7  receive C
8      C_p = max (C_p, C)
       /* choisir les k processus les moins souvent suspectés */
9      NF_p := { q : |{r : (C_p[r], r) ≤ (C_p[q], q)}| < k}
```

FIG. 31 – Transformation du détecteur $\diamond\mathcal{P}_k$ en $\diamond\mathcal{G}_k$.

k-Précision faible : il existe un ensemble de k processus corrects qui, ultimement, ne seront plus jamais suspectés par aucun processus.

Un détecteur de la classe $\diamond\mathcal{S}$ est simplement un $\diamond\mathcal{P}_1$. De même, $\diamond\mathcal{P}$ assure, pour chaque exécution r, une $n-f$ précision faible. On peut aisément transformer de tels détecteurs de défaillances en oracles de groupes. Pour cela il faut éviter que les listes de suspects varient indéfiniment et faire en sorte qu'ultimement, les suspects soient partout les mêmes. Soit $\diamond\mathcal{G}_k$ la classe des oracles de groupes tels que le groupe ait k membres : ultimement, la sortie de tous les oracles est la même liste de k processus corrects. Notons que Ω [19, 73] est exactement $\diamond\mathcal{G}_1$.

L'algorithme de la figure 31 réalise une transformation très simple de $\diamond\mathcal{P}_k$ en $\diamond\mathcal{G}_k$. Nous avons proposé dans [2] (et section 6.6), une transformation pour Ω assurant d'autres propriétés comme la stabilité qui impose que l'on ne changera de leader que lorsque c'est nécessaire.

Systèmes de phases synchronisées et détecteurs de défaillances

En considérant la hiérarchie de [21], si on utilise un détecteur de défaillances de la classe $\diamond\mathcal{P}$, l'algorithme des phases assurera que le système est définitivement synchronisé. De plus, ultimement tous les processus corrects seront membres de la phase : tout le système a, ultimement, un comportement en rondes synchronisées.

Notons que $\diamond\mathcal{S}$ est suffisant pour réaliser le consensus, mais ne permettra pas de réaliser ultimement des rondes synchronisées. De ce point de vue, les systèmes de phases synchronisées ne sont pas optimaux : ils demandent pour être définitivement synchronisés des propriétés de détection de défaillances strictement plus fortes que ce qui est nécessaire pour le consensus.

Avec la terminologie précédente, l'algorithme des phases réalise une simulation du modèle des phases synchronisées définitivement synchronisé dans un

système asynchrone avec détecteurs de défaillances dans $\diamond\mathcal{G}_{maj}$ (ou ce qui revient au même $\diamond\mathcal{P}_{maj}$) pour $maj = \lfloor n/2 \rfloor + 1$.

On discutera dans la fin de ce chapitre de certains modèles de calcul plus adaptés à $\diamond\mathcal{S}$.

Réciproquement, supposons que l'on ait un système de phases définitivement synchronisé. Si un processus participant à la phase ph considère comme correct uniquement les membres de ph, on construit aisément ainsi un détecteur de défaillances dans $\diamond\mathcal{P}_{maj}$. Plus généralement, si ultimement la phase contient k membres on obtient un détecteur de défaillances de $\diamond\mathcal{P}_k$. Si, ultimement, tous les processus corrects appartiennent à la phase, on aura $\diamond\mathcal{P}$.

On a donc une correspondance exacte (si on a l'un on peut construire l'autre et réciproquement) entre le nombre de membres dans la phase ultime d'un système de phases synchronisées et les détecteurs de défaillances de $\diamond\mathcal{P}_k$ si $k > \lfloor n/2 \rfloor$.

4.2.5 Modèles de calcul pour l'asynchrone et systèmes asynchrones avec temps

Comme les modèles de calcul pour l'asynchrone considérés ici permettent de réaliser le consensus, ils ne peuvent pas être réalisés dans un système asynchrone. Il faut donc considérer des systèmes partiellement synchrones. On a vu précédemment certaines correspondances entre les modèles partiellement synchrones et les détecteurs de défaillances. Aussi, ces correspondances ainsi que les résultats de la section précédente peuvent permettre de donner des conditions suffisantes pour qu'un système partiellement synchrone permette de réaliser des modèles de calcul pour l'asynchrone.

Par exemple, si le système partiellement synchrone est ultimement synchrone ($USync$ tel que défini en section 2.3.3), on a vu que l'on pouvait construire $\diamond P$. D'après ce qui précède, cela permet d'obtenir un système de phases définitivement synchronisé (dont tous les processus corrects seront ultimement membres de la phase). Généralement, caractériser d'abord ces modèles de calcul en termes de propriétés requises concernant la détection de pannes et ensuite déterminer les conditions sur le système pour assurer ces propriétés est une approche efficace et suffisamment abstraite.

Dans cette section, nous allons suivre une approche plus pratique, en considérant un modèle proche du *Timed Asynchronous Distributed System Model* de [29].

Rappelons qu'une des caractéristiques essentielles que doit avoir un modèle de calcul est le fait que l'on puisse donner *explicitement* les conditions qui permettent son implémentation dans un système réel. En utilisant une terminologie utilisée dans le domaine temps-réel, il s'agit de pouvoir établir des *conditions de faisabilité* [75].

En termes de conditions ultimes, nous venons de voir que l'on pouvait donner des conditions de faisabilité pour construire des systèmes de phases définitivement synchronisées. Cependant, ces conditions ne sont pas très utiles pratiquement : affirmer, par exemple, qu'à partir d'un certain temps le système sera

toujours sans défaillance ne peut pas être testé. Il s'agit d'une vision de "Dieu" qui seul peut connaître ce qui se passe à l'infini. Il est plus utile de pouvoir affirmer des propriétés relatives à des périodes finies. Par exemple, affirmer que pour que le consensus soit réalisable, il faut qu'à partir d'un certain moment le système ait *pour toujours* un comportement synchrone, présente un intérêt surtout théorique. Par contre, affirmer que si pendant, au moins 200 ms, le système a un comportement synchrone alors le consensus est réalisable est plus intéressant du point de vue pratique. En reprenant l'approche méthodologique de [75], le vendeur proposant un produit s'engage à ce que son produit vérifie la spécification demandée par le client *à condition* que le système sous-jacent assure une condition de faisabilité. Dans le second cas, le client pourrait éventuellement se retourner contre le vendeur en exhibant une période de stabilité de 200 ms sans que le consensus ne soit réalisé, mais, dans le premier cas, avec des conditions ultimes il aura beaucoup de mal à exhiber une exécution contredisant la condition de faisabilité...

Un des problèmes des systèmes de *virtual synchrony* est qu'ils ne donnent pas, en général, de conditions explicites sur le système sous-jacent et que, quand ils en donnent, il s'agit de conditions uniquement ultimes. Un avantage des systèmes de phases synchronisées est que l'on peut donner explicitement de telles conditions.

En considérant l'algorithme de la figure 30, on peut montrer que quand le système reste stable on pourra en un temps constant Γ (ne dépendant que de la borne δ sur les délais de communication) obtenir une phase synchronisée. Dans ce qui suit, on suppose que le temps d'un pas de calcul d'un processus est négligeable (nous avons réalisé l'étude complète dans [32]). De plus, une fois qu'une phase est synchronisée, si le système reste δ-stable pendant cette période, en au plus $k \times \delta$, on réalisera k rondes dans cette phase.

Remarquons que la borne Γ ne dépend pas de ce qui s'est passé avant la période de stabilité. L'état du système peut être quelconque, cela n'empêchera ni la création d'une phase synchronisée ni le fait que le temps pour la constituer reste constant. Il s'agit dans un certain sens d'une propriété d'auto-stabilisation forte : dès que le système n'a plus de nouvelles défaillances, l'algorithme converge en temps constant. Notons aussi que pour obtenir cette propriété on n'a pas besoin de supposer qu'en un pas de calcul les processus peuvent effectuer un nombre arbitrairement grand d'actions. Pour cela, il faut en particulier que les processus ne soient pas obligés de traiter *tous* les messages reçus mais uniquement les plus récents (on a vu plus haut que l'algorithme pouvait n'utiliser que des buffers à une place). Sinon, pendant les périodes d'instabilité du système, les processus pouvant envoyer un nombre arbitrairement grand de messages, un processus serait amené à traiter en temps constant un nombre arbitraire de messages.

En résumé, à partir du moment où le système sous-jacent est δ-stable (cf. section 2.3.3) pour au moins $\Gamma + k \times \delta$, le système est k-synchronisé. On a vu que le consensus est réalisable avec les phases si le système de phases synchronisées est 1-synchronisé. En d'autres termes, le consensus sera possible si le système sous-jacent assure que dans toute exécution, pendant au moins une durée de $\Gamma + \delta$, le système est δ-stable. On a donc une condition explicite de faisabilité.

Dans certains cas, comme pour réaliser la diffusion atomique (section 6.2.2), la condition de faisabilité ne porte pas uniquement sur un intervalle mais sur la répétition de tels intervalles : l'algorithme de diffusion atomique de la figure 34 suppose un système de phases infiniment souvent 1-synchronisé. Pour assurer cette propriété, il faudra donc que le système sous-jacent assure que toute exécution est infiniment souvent δ-stable pour au moins $\Gamma + \delta$.

4.2.6 Conclusion

Dans cette section, nous avons étudié l'intérêt des systèmes de phases synchronisées en tant que modèle de calcul. En reprenant les critères présentés au début du chapitre, nous avons montré que :

- ils offrent suffisamment de puissance pour pouvoir résoudre (en supposant des conditions de vivacité) le consensus,
- ils sont implémentables efficacement dans de nombreux systèmes partiellement asynchrones supposés être réalistes,
- on peut donner des conditions explicites de faisabilité.

Il sont aussi suffisamment abstraits pour simplifier le travail du programmeur (on reverra ce point dans le chapitre suivant qui donnera un exemple de l'utilisation des phases pour la réalisation d'une couche de diffusion atomique).

Dans une certaine mesure, ce modèle de calcul correspond à la volonté d'essayer de ressembler au modèle des rondes synchronisées du synchrone. Au delà de l'intérêt théorique que cela représente, il faut cependant noter que cette approche présente aussi quelques inconvénients. *A priori*, on peut en considérer deux principaux : les conditions requises sur le système et la charge élevée de communication entre les processus.

Conditions sur le système

Le premier est que l'on demande trop : pour réaliser le consensus ou la diffusion atomique, des détecteurs de défaillances de la classe $\diamond\mathcal{S}$ suffisent alors qu'ici il faut, essentiellement, des détecteurs de $\diamond\mathcal{P}$. De même, en terme de périodes de stabilité, il faut ici que *tous* les processus communiquent dans les délais pendant les périodes stables, alors qu'il suffirait que ce ne soit le cas que pour un seul.

En fait, si l'on demande trop, c'est parce que l'on voulait avoir plus. Le but du système de phases synchronisées n'est pas uniquement de faire du consensus ou de la diffusion atomique, mais aussi d'obtenir un comportement en rondes synchronisées. On a montré plus haut que, dans un certain sens, ce comportement exige ce type de détection de pannes (on a montré qu'à partir d'un système de phases définitivement synchronisées, avec tous les corrects on pouvait construire un détecteur de défaillances dans $\diamond\mathcal{P}$ et que réciproquement avec $\diamond\mathcal{P}$ on obtenait un tel système de phases). On pourrait obtenir un résultat similaire par rapport aux périodes de stabilité.

La vraie question est donc plutôt : qu'obtient-on de plus en voulant obtenir de tels systèmes ? Après tout, quelle est l'utilité pour les processus de travailler

en rondes synchronisées ou de connaître l'identité des divers participants à ces rondes ? La réponse (partielle) à cette question se trouvera dans le chapitre suivant où on montrera comment on obtient grâce à cela une meilleure efficacité avec la notion de groupe. Par ailleurs, le temps réel, qui était, rappelons-le un des objectifs du projet qui a permis de définir les systèmes de phases synchronisées, oblige à supposer que *tous* les processus respectent les délais durant les périodes de stabilité.

Coût élevé de communication

Ce deuxième point est, en fait, similaire au précédent : le coût élevé de communication est intrinsèque au modèle des rondes synchronisées. De fait, pour passer à la ronde suivante avec n processus, comme chacun doit envoyer un message à chacun, il faudra n^2 messages. Cependant, on peut remarquer qu'on peut aussi diminuer ce coût en augmentant la *latence* : dans une phase synchronisée, comme tous les membres de la phase ont la connaissance des membres de la phase, ils peuvent choisir tous le même processus (par exemple celui ayant la plus petite identité) et le désigner comme leader de la phase. Ensuite, ils s'adressent tous à ce leader, et c'est le leader qui retransmet les messages à chacun. Ainsi, pour réaliser une ronde, il faudra une latence de deux (du processus vers le leader et ensuite du leader vers les processus), mais on sera passé de n^2 messages à $2 \times n$ messages. (Notons cependant qu'il existe un risque de congestion des canaux menant ou provenant du leader).

Comme on le voit, il est possible d'adapter ces systèmes de rondes de façon à les rendre moins gourmands en messages. Mais, à vrai dire, ce genre de systèmes est surtout intéressant pour des systèmes de processus (processeurs) fortement couplés dans lesquels la communication de tous vers tous est économique. C'est le cas sur des réseaux locaux (LAN) mais pas sur des réseaux globaux (WAN). Cependant, il n'est pas certain que dans des réseaux globaux on ait besoin de ce genre de systèmes et probablement les problèmes d'accord deviennent impraticables.

4.3 Conclusion

Dans ce chapitre on a essayé de montrer ce que peut être un modèle de calcul pour des systèmes (partiellement) asynchrones. Nous nous sommes presque exclusivement intéressés à des modèles de calcul en rondes qui sont des modèles assez fortement synchrones.

La démarche utilisée est intéressante car relativement abstraite.

Partant de la nécessité de résoudre le consensus, on définit un modèle de calcul suffisamment puissant pour cela. De plus, partant du principe qu'il est plus simple de travailler dans un modèle proche du synchrone, le modèle des rondes et des phases synchronisées permet de pouvoir développer plus aisément une algorithmique distribuée tolérante aux pannes.

Cette démarche n'a pas seulement un intérêt théorique : elle devient aussi pratique quand il s'agit d'implémenter le modèle de calcul. Un de ses avantages est que l'on peut choisir, suivant le niveau de généralité et d'abstraction désiré, soit de réaliser cette implémentation directement dans un modèle de système donné, soit de la réaliser dans un modèle de système avec détecteurs de défaillances.

Notons enfin que cette démarche peut s'appliquer plus généralement à des modèles de calcul qui seraient moins proche du synchrone comme par exemple des modèles avec leader.

5 Conclusion des comparaisons et perspectives

Le résultat de ces diverses études et comparaisons est résumé dans la figure 32.

Les systèmes synchrones avec diffusion fiable, synchrones et forts sont des systèmes pour lesquels il existe une borne sur les délais de communication qui est respectée *dès le début*. Ces systèmes correspondent bien aux divers modèles de calcul en rondes (respectivement rondes parfaitement synchronisées, rondes synchronisées et rondes avec $\mathcal{S}$). Réciproquement, il est clair qu'à partir d'un de ces modèles en rondes on pourrait construire des modèles de systèmes correspondants. La difficulté est essentiellement technique : il faut définir la notion de temps.

Travailler directement avec des modèles de systèmes ne présente, *a priori*, d'intérêt que par rapport au temps réel. Dans tout ce qui a été fait précédemment, un temps virtuel nous suffit et il est donc préférable de considérer les modèles de calcul correspondants qui font abstraction du temps en utilisant la notion de rondes. Cependant, partant d'une algorithmique développée dans des modèles de rondes, il serait intéressant d'étudier le coût du point de vue des performances temps réel du passage à des modèles de systèmes avec temps. Dans ce cadre, des modèles comme les automates temporisés ou les systèmes hybrides sont, sans doute, de bons modèles pour l'étude des systèmes synchrones.

Si l'on considère maintenant les détecteurs de défaillances, $\mathcal{P}$ et $\mathcal{S}$ correspondent intuitivement respectivement aux systèmes synchrones et forts. Ils peuvent effectivement être réalisés (simplement) à partir de tels systèmes. On a vu que, dans une grande mesure, l'abstraction des détecteurs de défaillances n'était pas très coûteuse. D'une part, même si certains problèmes ne peuvent être résolus avec les modèles de rondes correspondants à $\mathcal{P}$, on a vu que l'on pouvait, malgré tout, les résoudre par des simulations raisonnables. D'autre part, et c'est sans doute le point le plus important, le coût de l'utilisation de ces modèles de rondes est faible : le consensus se résout avec la même complexité qu'avec les modèles de rondes dérivés des systèmes synchrones et ce même en ce qui concerne les algorithmes à terminaison au plus tôt (*early stopping*). Par ailleurs, on a montré que pour simuler des rondes parfaitement synchronisées (qui, *a priori*, est le modèle le plus adapté aux problèmes d'accord), le coût est exactement le même en partant de rondes avec détecteur de défaillances parfait que de rondes obtenues à partir d'un système synchrone. Ces résultats justifient donc l'intérêt des détecteurs de défaillances même pour des systèmes synchrones.

Il serait donc intéressant d'affiner les détecteurs de défaillances correspondant essentiellement aux systèmes synchrones. On peut, par exemple, s'intéresser à la classe des détecteurs de défaillances qui peuvent être réalisés dans des systèmes synchrones et essayer pour diverses classes de problèmes correspondants d'autres formes d'accord (par exemple, le *commit*, l'accord sur un ensemble de valeurs etc.) de définir quels sont les détecteurs de défaillances les plus faibles permettant de résoudre ces problèmes.

Cependant, l'intérêt principal des détecteurs de défaillances réside dans les

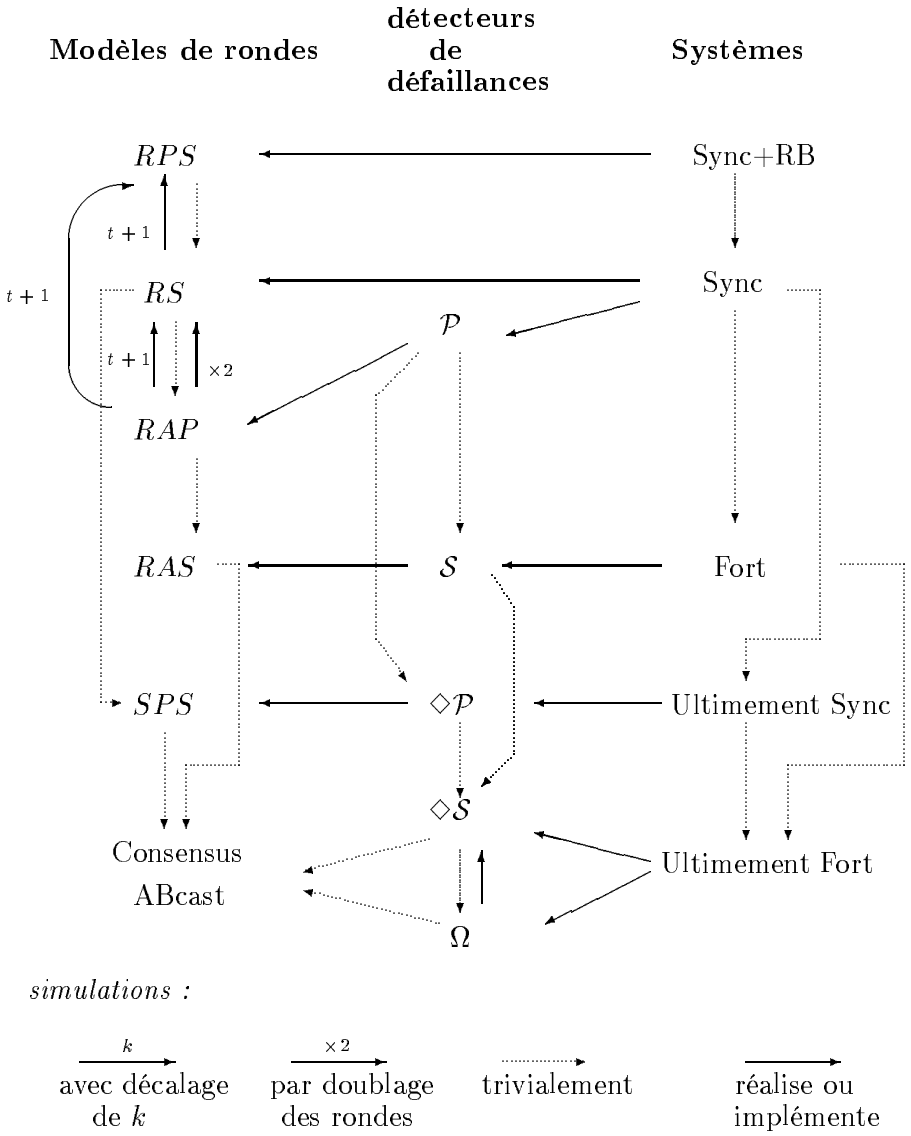


FIG. 32 – Résumé des comparaisons.

systèmes asynchrones : ils permettent d'ajouter au système ce qu'il faut pour résoudre les problèmes de consensus. Cependant, on peut se poser les mêmes questions que pour les systèmes synchrones : quel est le bon niveau d'abstraction, les modèles de calcul comme la *virtual synchrony*, les détecteurs de défaillances, ou les modèles de systèmes partiellement synchrones ? L'étude qui précède ne résout pas entièrement ce problème. On a essayé de montrer que ces diverses approches ne sont pas orthogonales. Essentiellement, au niveau d'abstraction le plus bas, on peut considérer directement le modèle partiellement synchrone, à un niveau intermédiaire, les détecteurs de défaillances et, à un niveau plus fort, des modèles de calcul comme les systèmes de phases synchronisées. L'étude qui précède montre que l'on peut construire en couche ces divers niveaux. Cependant, le point important est que, contrairement au cas des modèles de rondes issus du synchrone, il n'y a pas un modèle de calcul naturel. Le choix qui a été fait dans ce qui précède est de s'approcher le plus possible des systèmes synchrones et, pour cela, on a dans les systèmes de phases synchronisées, comme dans les diverses formes des *Virtual Synchrony*, la notion de groupe. Bien entendu, d'autres approches se justifient. En particulier, au lieu d'essayer de se mettre d'accord tous ensemble (construire une phase), on peut aussi s'en remettre à un *leader*. Clairement, si on a tous ultimement un leader (donné, par exemple, par un détecteur de défaillances), la notion de groupes de processus n'est plus très importante : le groupe est implicitement défini par le leader.

De cette étude partielle, je pense que l'on peut retenir plusieurs points. Tout d'abord, il existe de nombreuses approches pour travailler avec des systèmes distribués. Pour les comparer, il est nécessaire de développer un cadre formel suffisant. En particulier, seul un cadre formel raisonnable permet de préciser quelles sont les conditions sur le système sous-jacent qui assureront les propriétés des algorithmes. Ce cadre formel doit permettre aussi de comparer les approches, et c'est partiellement ce que l'on a essayé de faire. Enfin, bien entendu, il ne s'agit pas uniquement de développer des modèles plus ou moins intéressants, mais aussi de les réaliser.

N'oublions pas que les modèles de calcul correspondent aussi à des systèmes de *middleware* qui doivent permettre au programmeur d'utiliser efficacement le système sous-jacent. Il s'agit donc aussi de développer des algorithmes efficaces permettant leur implémentation.

6 Autres travaux algorithmiques

6.1 Introduction

Nous allons ici passer en revue la partie plus algorithmique de mes travaux. Certains de ces algorithmes ont déjà été évoqués dans la première partie, mais dans un esprit un peu différent, dans la mesure où c'étaient plus les conséquences de ces algorithmes que les algorithmes eux-mêmes qui nous intéressaient.

Ici, il s'agira plutôt d'essayer de comprendre et d'expliquer les intuitions et les mécanismes généraux qu'ils mettent en oeuvre.

Dans une certaine mesure, chacun de ces algorithmes correspond à une idée ou une préoccupation principale.

La première section correspond à l'application des systèmes de phases décrits dans la première partie. On y trouve un algorithme qui met en oeuvre des méthodes relativement simples pour réaliser la diffusion atomique. Mais cette section correspond à un projet *pratique*. Les questions de spécifications et de méthodologie y sont importantes.

L'algorithme du *global data* présenté ensuite est plus technique : il s'agit d'appliquer et de traduire des méthodes d'arrêt au plus tôt (*early stopping*) utilisées en synchrone à des modèles de rondes basées sur des détecteurs de défaillances parfaits.

Les deux algorithmes suivants concernent des problèmes de diffusion atomiques. Pour résoudre ces problèmes, il faut obligatoirement faire appel à un mécanisme supplémentaire (des oracles). Dans les deux cas, on voudra limiter cet usage de ce mécanisme à ce qui est réellement nécessaire. Le premier des deux algorithmes peut aussi être interprété comme une sorte d'exercice sur la réplication active. Le deuxième correspond à une préoccupation pratique importante : la diffusion atomique est manifestement trop coûteuse dans un réseau de grande taille et comme elle n'est généralement pas nécessaire tout le temps, il s'agit d'essayer de limiter son usage.

Enfin le dernier algorithme présenté est assez pratique, il s'agit de réaliser de façon efficace un mécanisme de détection de pannes dans un modèle partiellement synchrone avec temps.

La diffusion atomique temps-réel est définie par deux primitives *A-diffuse* et *A-délivre* qui vérifient :

Validité : si un processus correct A-diffuse m, tous les processus corrects A-délivrent m,

Accord Uniforme : si un processus A-délivre m alors tous les processus corrects A-délivrent m,

Intégrité Uniforme : m est A-délivré au plus une fois par chaque processus uniquement si m a été A-diffusé,

Ordre Uniforme : si p et q A-délivrent m et m' alors ils A-délivrent m et m' dans le *même* ordre,

Ponctualité : tout message m a une *échéance*, m est A-délivré avant son échéance.

FIG. 33 – Diffusion atomique temps réel.

6.2 Les phases synchronisées

6.2.1 Le projet ATR

Le projet ATR [11, 22, 23, 24, 12] est un projet conjoint (DRET, MENESR, CNRS) regroupant plusieurs laboratoires (INRIA, LIAFA, LIX, LMC-IMAG) et des partenaires industriels (Axlog, Dassault-Aviation, Thomson-CSF). On trouvera dans [35] une présentation de l'utilisation des algorithmes de phases dans un cadre proche du projet ATR.

Le but de ce projet était, dans un premier temps, d'identifier avec les partenaires industriels un problème d'algorithmique distribuée tolérant aux pannes. Une fois le problème identifié, il s'agissait de développer des solutions algorithmiques qui devaient, dans un troisième temps, être mises en oeuvre par Axlog.

La première phase (capture), utilisant partiellement la méthode TRDF [75], a permis de spécifier le problème à traiter. Il s'agit du problème de la *diffusion atomique temps réel résistant aux pannes* (cf. figure 33).

Essentiellement, il s'agit d'une diffusion atomique pour laquelle on doit pouvoir assurer des délais de livraison des messages.

La première tâche est de définir le modèle de système correspondant à ATR. Comme on veut assurer des propriétés de ponctualité, il est bien évident que le modèle considéré sera un modèle avec temps (réel). Le système ATR assure que :

- la vitesse relative des processeurs est bornée,
- chaque processeur a accès à une horloge locale et ces horloges sont ϵ-synchronisées,
- la communication est point à point et sans perte de messages,
- il n'y a pas de borne connue pour les délais d'acheminement des messages,
- les processus ne peuvent défaillir que par des pannes franches (*crash*).

Ainsi, un système ATR est simplement un système asynchrone avec temps réel.

En fait, pour simplifier, on supposera généralement que les processus ont accès à des horloges parfaitement synchronisées.

De façon claire (résultat d'impossibilité du consensus [52] !), même sans la propriété de ponctualité, il est impossible de réaliser la diffusion atomique dans de tels systèmes. La démarche du projet ATR sera de définir des *conditions de faisabilité* qui devront assurer la diffusion atomique temps réel. Cela revient à donner des conditions explicites *suffisantes* supplémentaires sur le système sous-jacent. Cette démarche est d'un usage courant dans le domaine du temps réel : si une tâche doit être effectuée au plus tard au temps τ, on détermine les conditions sur l'ordonnancement des tâches pour assurer cette échéance. Par exemple, la condition de faisabilité peut être que, si telle ou telle condition sur les lois d'arrivée des tâches est vérifiée, alors l'ordonnancement des tâches peut assurer cette échéance.

Pour le problème de la diffusion atomique temps réel, on va procéder de la même façon : on va déterminer des conditions de faisabilité sur le système qui permettent d'assurer ces propriétés. L'objectif est de pouvoir explicitement quantifier les conditions de validité de la diffusion atomique temps-réel. Essentiellement, il s'agit d'un engagement : si le système sous-jacent vérifie ces conditions de faisabilité, nos algorithmes assurent que toutes les propriétés de la diffusion atomique temps réel seront garanties. Bien entendu, ces conditions dépendent des échéances choisies.

Cette quantification des conditions de faisabilité et l'engagement qui va avec sont fondamentaux, car ils obligent à définir au moins des conditions suffisantes que doit assurer le système. Rappelons que la plupart des systèmes de *middleware* ne précisent pas les conditions qu'ils supposent sur le système sous-jacent, et laissent l'utilisateur gérer ses divers problèmes de réglage de *time-out*.

Si les conditions de faisabilité ne sont pas réunies par le système sous-jacent, *a priori*, on ne s'engage plus à rien. Cependant, dans le projet ATR, on assure que le moins possible de propriétés seront violées (*gracefully degradation*). Plus précisément, on assure que :

1. les propriétés de sûreté sont toujours assurées. Par exemple, l'ordre de livraison des messages est toujours le même pour tous,
2. on peut définir des conditions de faisabilité plus faibles qui assurent la diffusion atomique mais ne garantissent plus la ponctualité.

D'un point de vue pratique, cela signifie que, si le système assure toutes les conditions de faisabilité, alors le service est parfait. Si, certaines de ces conditions ne sont plus assurées, alors le service sera partiellement dégradé : les échéances ne sont plus respectées, et si plus aucune condition de faisabilité n'est assurée, certains messages risquent de ne pas être délivrés, mais, s'ils le sont, ce sera dans le même ordre partout [45]. On a donc une dégradation progressive : on perd d'abord la ponctualité, puis la vivacité. Mais la sûreté est toujours garantie.

Le projet ATR a décidé de réaliser la diffusion atomique temps réel à partir des systèmes de phases synchronisées que nous avons décrits dans la section 4.1.

[45] Cela revient à considérer qu'il vaut mieux ne rien faire plutôt que de faire des erreurs.

Comme on le verra, les systèmes de phases synchronisées permettent de réaliser, sous des conditions assez larges (il suffit que le système soit infiniment souvent 1-synchronisé), la diffusion atomique. Pour assurer des conditions de faisabilité concernant les échéances, il faut, au minimum, pouvoir donner une borne pour la livraison de tout message : il faut donc, sous les conditions de faisabilité, que l'algorithme de livraison des messages délivre les messages en temps borné.

Comme il s'agit de temps réel, les performances sont essentielles. Les systèmes de phases synchronisées permettent d'assurer des délais très courts : si les messages mettent un temps δ, en l'absence de pannes, le temps de la diffusion d'un message est de l'ordre de $2 \times \delta$. Cependant, ce délai très court se paie par un échange important de messages (de l'ordre de n^2 messages à chaque ronde). Aussi, la réalisation d'ATR a surtout un intérêt pour des processeurs fortement couplés, ou sur des réseaux à large bande passante ayant, par exemple, des primitives de communication par diffusion.

L'implémentation expérimentale et les tests de simulations de Axlog ont montré que l'algorithmique mise en oeuvre correspond bien à ce qui était attendu [12].

Pour terminer, notons que la spécification de la diffusion atomique temps réel choisie par ATR nécessite essentiellement que le système soit *parfaitement* synchrone : on vérifie aisément qu'il faut au moins que les délais de communication entre processus soient bornés (et que cette borne soit connue). Nous verrons dans la sous-section 6.2.3 comment définir une spécification moins restrictive.

6.2.2 Diffusion Atomique et SPS

On a vu (figure 29) comment réaliser le consensus à partir d'un système de phases synchronisées. On peut, bien entendu, utiliser une transformation standard permettant d'implémenter la diffusion atomique à partir du consensus (par exemple [61]), cependant les systèmes de phases synchronisées permettent de réaliser directement la diffusion atomique de façon efficace et simple.

En effet, tant que la phase n'échoue pas, les participants de la phase progressent en rondes synchronisées et ont un fonctionnement synchrone. L'algorithme est décrit dans la figure 34.

Expliquons intuitivement son fonctionnement.

L'algorithme va constituer des lots de messages indexés par les entiers et les processus délivreront les lots suivant leur indice ; à l'intérieur d'un lot, les messages sont délivrés suivant un ordre préétabli (par exemple, grâce à un ordre déterminé par les identités des processus).

Pour réaliser la diffusion atomique des messages, un processus membre peut proposer, comme message de la ronde, le message qu'il doit diffuser (atomique) : si la ronde n'échoue pas, tous les processus membres recevront le même ensemble de messages (un lot), et peuvent alors les délivrer dans le *même* ordre.

Cependant, si la ronde échoue, il est possible que certains processus terminent cette ronde alors que d'autres non. On ne peut pas délivrer *tout de suite* après cette première ronde le lot constitué des messages de cette ronde. En effet, avant de décider de délivrer un lot, il faut être sûr que, soit tous les processus

```
   /* ADiff_p : messages que p doit diffuser (atomique) */
   /* ProposeADiff_p = (Mnext, N, ph) :*/
   /*     Mnext la liste des messages proposés par p */
   /*             pour la prochaine ronde de diffusion */
   /*     N numéro du lot proposé pour Mnext */
   /*     ph la phase correspondant à Mnext */
   /*Initialisation :*/
 1 ADiff_p := ∅
 2 ProposeADiff_p := (∅, 0 , ⊥)

 3 on ADiffusion(m)                    /* m est proposé pour une diffusion atomique */
 4    do ADiff_p := ADiff_p ∪ {m} od

   /* DelivrerFiable/DiffuserFiable réalise une diffusion fiable */
 5 on DelivrerFiable(DELIVRER, Mnext, N)
 6    do
 7        Délivrer Mnext dans le lot N
 8        ProposeADiff_p := ProposeADiff_p − Mnext_p
 9        ADiff_p := ADiff_p − Mnext_p
10    od

11 ronde 2r de la phase ph
12    /* Proposer ses valeurs pour la diffusion */
13    send (ProposeADiff_p)              /* envoyer à tous les membres de la phase */
14    if (la ronde réussit)
15    then
16    /* Choisir les messages avec les numéros de lot et de phase max */
17        N_p := max { numéros de lots reçus dans la ronde }
18        LM_p := { m reçus dans la ronde| numero de lot = N_p
19                                       ∧ valeur de phase maximale}
20        ProposeADiff_p := (LM_p, N_p, ph)
21        /* si la ronde suivante échoue, p proposera LM_p avec ph */
22    fi

23 ronde 2r + 1 de la phase ph
24    /* Confirmer */
25    send ∅                             /* envoyer à tous les membres de la phase */
26    if (la ronde réussit)
27    then
28        /* Délivrer les messages choisis dans la ronde 2r */
29        /* Informer tout le monde */
30        DiffuserFiable(DELIVRER, LM_p, N_p)
31        ProposeADiff_p := ProposeADiff_p − LM_p
32        ADiff_p := ADiff_p − LM_p
33        N_p := N_p + 1
34        /* Ajouter les messages à diffuser par p */
35        ProposeADiff_p := (ProposeADiff_p ∪ Adiff, NP_p, ph)
36    fi
```

FIG. 34 – algorithme de diffusion atomique utilisant les phases.

membres de la phase délivrent ces mêmes messages dans la même ronde, soit que ce sont ces messages qui seront nécessairement choisis pour être délivrés en premier ensuite.

Pour cela, on procède de la façon suivante : dans un premier temps (les rondes paires) les processus proposent leurs messages à diffuser. Dans ce cas, tous les processus qui terminent cette ronde auront connaissance du *même* ensemble de messages. Dans un deuxième temps (la ronde impaire suivante), les processus envoient un message vide de confirmation.

Les processus qui terminent cette deuxième ronde peuvent délivrer l'ensemble des messages reçus dans la première ronde. En effet, si tous terminent cette deuxième ronde, ils délivrent nécessairement le même lot [46].

Si tous ne terminent pas la ronde, il faut assurer que le choix de diffuser ce lot sera assuré dans la suite.

Notons que tous les processus qui terminent cette deuxième ronde savent en plus que *tous les membres* de la phase ont reçu le même ensemble de messages dans la première ronde. Comme l'ensemble des membres est un ensemble majoritaire, cet ensemble de messages sera nécessairement connu par au moins un des participants des phases suivantes : on est donc sûr qu'il en restera une trace. Avoir une trace ne suffit pas tout à fait, il faut aussi assurer que c'est *ce* lot qui sera choisi. Comme les phases synchronisées sont totalement ordonnées, cela ne pose pas de problème : les messages seront étiquetés par la phase au cours de laquelle ils ont été reçus (ou proposés pour la première fois).

Plus précisément, au cours de la ronde paire, les processus choisissent les messages étiquetés par une valeur de phase *maximale*. Les messages ainsi choisis seront maintenant étiquetés par la phase courante, ce qui assure que, s'ils sont proposés dans une phase ultérieure, ils seront choisis de préférence à des messages provenant d'une phase antérieure. On assure ainsi, qu'à partir du moment où tous les membres d'une phase ont terminé la ronde paire, nécessairement, le premier lot délivré après cette ronde ne peut être que celui qui a été choisi dans cette ronde.

On assure ainsi les propriétés de sûreté de l'algorithme. Pour assurer la vivacité de la diffusion atomique (tout message proposé par un processus correct sera délivré), il faut supposer que le système de phases est infiniment souvent 1-synchronisé, c'est à dire que, infiniment souvent, deux rondes successives réussissent.

Implicitement, on a supposé dans la présentation ci-dessus que tous les processus corrects participaient infiniment souvent aux rondes des phases 1-synchronisées. En toute généralité, si ce n'est pas le cas, il faudrait, pour être sûr d'atteindre un membre de la phase, ajouter un envoi préalable vers tous les processus des messages qui doivent être diffusés ; chaque processus membre proposant, ensuite, non seulement ses propres messages mais aussi ceux dont il a appris l'existence.

[46] Si tous les processus corrects ne sont pas membres de la phase il faut aussi faire une diffusion fiable du lot de façon à assurer que même les processus qui ne sont pas membres de la phase délivreront le lot.

Dans la hiérarchie classique des détecteurs de défaillances, on obtiendra ces deux propriétés de vivacité (infiniment souvent 1-synchronisé et tout processus correct est infiniment souvent membre de phases 1-synchronisées) avec une implémentation des phases utilisant des détecteurs de défaillances ultimement parfaits.

Il faut aussi remarquer que, dans tous les cas, les propriétés de sûreté sont assurées. Seules les propriétés de vivacité nécessitent des propriétés supplémentaires sur le système sous-jacent.

Notons que, tant que la phase n'échoue pas, l'algorithme est très simple et tout message proposé par un membre dans la ronde $2r$ sera délivré dans la ronde $2r + 1$. On a donc dans ce cas une latence de 2. Cette latence 2 est optimale si l'on veut assurer l'uniformité [47]. Cette optimalité a cependant un prix : les processus sont fortement couplés (*closely coupled*) et, à chaque ronde, de l'ordre de n^2 messages sont échangés.

D'autre part, cet algorithme construit sur un système de phases a surtout un intérêt dans le cas où tous les processus ont souvent (à chaque ronde) des messages à diffuser ; si ce n'est pas le cas, dans la version présentée ici, les processus devront émettre des messages vides.

Remarquons enfin que certaines optimisations sont possibles : par exemple, les messages vides de confirmation peuvent aussi contenir les propositions pour la ronde suivante. De même, on peut aussi montrer que de légères modifications permettent de faire en sorte que l'algorithme résiste à certains types de pertes de messages.

6.2.3 Systèmes de phases et temps réel

Cette section correspond à [40].

Pour assurer *tout le temps* les propriétés de la diffusion atomique temps-réel de la figure 33, il faut que le système soit synchrone : il doit y avoir des délais connus sur le temps de transmission des messages. On peut affaiblir ces conditions en n'assurant ces propriétés que pendant certaines périodes. Pour cela, considérons le modèle décrit dans la section 2.3.3 et correspondant au *timed asynchronous distributed system model* de [29]. Dans ce modèle, on suppose qu'il existe une sorte de borne connue δ sur les délais de communication [48]. Cette borne n'est pas toujours respectée. En s'intéressant aux périodes où ces bornes sont respectées, on dira que l'intervalle de temps $[\tau, \tau']$ est δ-*stable* si, pendant cet intervalle, les délais de communications δ sont assurés, et, s'il n'y a pas de pannes de processus durant tout cet intervalle.

On définit ensuite un prédicat de stabilité dépendant du temps : $Stable_{\gamma,\delta}(\tau)$. $Stable_{\gamma,\delta}(\tau)$ signifie que l'intervalle $[\tau, \tau + \gamma]$ est δ-stable. On peut alors relativiser les propriétés à ces prédicats de stabilité. Par exemple, au lieu d'assurer

[47] C'est-à-dire que les processus incorrects ne peuvent pas délivrer des messages que les processus corrects ne délivrent pas.

[48] Pour simplifier, on suppose, ce qui n'est pas nécessaire, que les processus ont accès à des horloges parfaitement synchronisées et que le temps des pas de calcul des processus est négligeable.

la ponctualité à tout instant, elle ne sera assurée que pendant les bonnes périodes, c'est-à-dire les périodes δ-stables. La forme générale de ces propriétés relativisées sera de la forme, pour un prédicat P :

$$Stable_{\gamma,\delta}(\tau) \Longrightarrow P \tag{13}$$

Ce qui signifie que, si le système est stable par rapport à la borne δ, pour au moins γ, alors P est assurée à l'instant τ. Par exemple, la ponctualité relativisée sera une propriété assurant que, dans une période où le système est δ-stable pour au moins une certaine durée γ, alors les messages sont délivrés avant leur échéance.

D'un point de vue pratique, ces prédicats de stabilité permettent de déterminer des conditions de faisabilité : pratiquement, étant donné une propriété P, il s'agit de déterminer γ et δ qui assurent (13).

Pour pouvoir traiter des propriétés faisant intervenir le temps avec des systèmes de phases synchronisées, nous allons introduire des conditions de vivacité faisant intervenir le temps. Nous dirons qu'une exécution est α-k-synchronisée au temps τ, si pendant l'intervalle $[\tau, \tau + \alpha]$ au moins k rondes successives ont réussi. On notera ce prédicat $Sync_{\alpha,k}(\tau)$.

Ces prédicats de vivacité faisant intervenir le temps pour les systèmes de phases correspondent aux prédicats de stabilité sur le système sous-jacent. Les premiers correspondent à des propriétés sur le système de phases (haut niveau), alors que les seconds correspondent à des propriétés du système sous-jacent (bas niveau). Cette approche est modulaire : on a deux couches, celle des phases synchronisées et celle du système sous-jacent. *A priori*, on peut résoudre la spécification au haut niveau en restant dans le système de phases et déterminer des conditions suffisantes assurant cette spécification. Ensuite, il faut traduire ces conditions sur le système de phases en prédicats de stabilité sur le système sous-jacent. De cette façon, on obtiendra des conditions de faisabilité sur le système sous-jacent pour la spécification.

Le point important est qu'avec l'algorithme réalisant les phases on peut effectivement déterminer des prédicats de stabilité assurant les propriétés de vivacité sur le système de phases faisant intervenir le temps. En fait, quand le système se stabilise (les bornes sont respectées), on peut majorer par une constante le temps qu'il faut pour qu'une phase k-synchronisée soit réalisée [49].

En particulier, étant donné α et k, on peut déterminer les conditions de faisabilité sur le système sous-jacent, c'est-à-dire, calculer γ (durée de la période de stabilité) et δ (borne sur les délais de communication) tels que :

$$Stable_{\gamma,\delta}(\tau) \Longrightarrow Sync_{\alpha,k}(\tau) \tag{14}$$

Par ailleurs, on peut aussi donner des conditions de faisabilité assurant la vivacité (sans temps) du système de phases synchronisées : pour un δ donné, on

[49] Cette propriété n'est pas évidente : elle signifie que le temps de re-synchronisation des phases est indépendant de la durée de la période de non stabilité. Elle provient, en particulier, du fait que l'algorithme des phases peut ignorer tous les messages anciens.

peut calculer ν tel que, s'il existe une période δ-stable de taille au moins ν alors le système de phase est k-synchronisé.

En appliquant ce qui précède à la diffusion atomique temps-réel, à partir des conditions de ponctualité, on détermine une propriété de vivacité avec temps sur le système de phase qui assure la ponctualité. Ensuite, on peut grâce à la propriété 14, traduire cette condition sur les phases en une condition de stabilité sur le système sous-jacent.

Notons que cette démarche est générale. Pour résoudre une spécification en utilisant un système de phases synchronisées, on peut procéder comme suit :

1. Résoudre, dans le système de phases, la spécification en s'intéressant aux conditions de sûreté.
2. Déterminer les conditions de vivacité sur le système de phase (pour la diffusion atomique, il fallait que le système soit infiniment souvent 1-synchronisé).
3. Traduire ces conditions de vivacité en conditions de stabilité (pour la diffusion atomique, cela revient à l'existence infiniment souvent de périodes stables d'une certaine taille Δ).

En procédant de la même façon, on peut aussi traiter des propriétés faisant intervenir des notions de ponctualité.

$D_p =< d_1, \ldots, d_n >$ est le n-vecteur de décision de p.
v_p valeur proposée par p.

Intégrité : si p décide D_p, pour tout $k \in [1 \ldots n]$ $d_k \in \{\perp, v_k\}$

Accord : si p et q décident alors $D_p = D_q$,

Terminaison : tout processus correct p décide un jour,

Obligation : si p décide D_p, alors $d_p = v_p$.

FIG. 35 – Spécification du *global data*.

6.3 *Global data* avec terminaison au plus tôt

Cette section correspond à un travail réalisé avec C. Delporte, J.M. Hélary et M. Raynal [41].

Dans la première partie, nous avons évoqué ce problème de *global data* : il s'agit d'obtenir un accord sur un n vecteur tel que, si le processus k décide, la k-iéme coordonnée du vecteur soit la valeur proposée par k. La figure 35 précise la spécification de ce problème sous une forme légèrement différente de celle de la figure 20.

Ce problème est assez directement équivalent à celui dit du *Uniform Terminating Reliable Broadcast* (UTRB) [61] à la base des problèmes de généraux byzantins [50]. Avec UTRB, l'initiateur doit émettre un message m, il faut qu'à la fin du protocole, tous les processus corrects arrivent à un accord, soit sur m, soit sur $\perp$, tel que si l'initiateur est correct la valeur d'accord doit être m. On peut montrer que ce problème est strictement plus difficile que le consensus [21, 65]. En particulier, dans la hiérarchie des détecteurs de défaillances proposée par Chandra et Toueg, $\mathcal{P}$ est nécessaire pour le résoudre.

Dans ce qui suit, on s'intéresse à résoudre ce problème dans un système asynchrone disposant de détecteurs de défaillances parfaits.

Ce problème a un intérêt pratique clair, en effet, il renforce le consensus en décidant sur un vecteur dans lequel figurent *toutes* les valeurs des processus corrects (et même, en fait, vivants au moment de la décision). Avec la spécification du consensus, on n'a aucune garantie sur le choix de la valeur initiale : on sait juste qu'il s'agit d'une valeur proposée par un processus. Aussi, à partir de la spécification du consensus, pour imposer une certaine valeur proposée par un processus correct, on sera amené à diffuser cette valeur par une diffusion fiable qui assure que, un jour, *tous* les processus corrects auront cette valeur, de façon à ce que cette valeur soit proposée par *tous* les processus, garantissant ainsi qu'elle fera partie des valeurs choisies pour la décision [51]. Ici, avec la spécification du *global data*, on a directement un accord pour *toutes* les valeurs proposées par des processus corrects.

[50] le caractère uniforme est, ici, important : il affirme que, si un processus incorrect décide, il doit décider de la même façon que les processus corrects ;

[51] C'est effectivement ce qui est fait pour réaliser la diffusion atomique à partir du consensus [21].

D'autre part, comme on l'a vu dans la section 3.2.2, ce problème peut aussi servir à simuler des rondes parfaitement synchronisées à partir de rondes avec $\mathcal{P}$.

Résoudre le problème du *global data* avec des détecteurs parfaits est simple. Le problème posé, ici, est de décider au plus tôt (*early stopping*) c'est-à-dire avec une latence qui est fonction du nombre de pannes effectives de *l'exécution* et non du nombre de pannes *tolérées*.

Dans un système synchrone, avec $f \leq t < n-1$, [46] ont montré que la borne inférieure pour la terminaison est $\min(t+1, f+2)$ rondes.

Avec un système asynchrone disposant de détecteurs de défaillances parfaits, on peut remarquer que le système de rondes que l'on obtient est au moins aussi puissant que celui obtenu avec des *send-omission* [52]. On peut alors utiliser, pour les rondes utilisant des détecteurs parfaits, l'algorithme de [89], pour résoudre la version *non-uniforme* de UTRB en $\min(t+1, f+2)$ quel que soit le nombre de pannes. Mais, il n'existe pas de moyens simples et efficaces pour transformer cet algorithme en un algorithme pour UTRB.

On peut aussi résoudre le problème du *global data* avec des détecteurs de défaillances parfaits en utilisant la simulation décrite figure 24. En traduisant ainsi un algorithme synchrone pour le *global data* optimal et en remarquant que, dans tous les cas, on résout le problème en au plus $t+1$ rondes, on obtiendra un algorithme qui résout le problème en asynchrone avec détecteurs de défaillances parfaits en $\min(t+1, 2(f+2))$. C'est l'approche suivie dans [65].

Mais, comme on va le voir, on peut faire mieux et obtenir, si $f \leq t < n-1$, la même borne qu'en synchrone : $\min(t+1, f+2)$.

Nous n'allons pas donner ici les détails de l'algorithme que l'on pourra trouver dans [41]. Par contre, on va essayer d'en expliquer les mécanismes essentiels.

Commençons par rappeler les principes à la base des protocoles à terminaison au plus tôt (*early stopping*) dans les systèmes synchrones. Pour cela, considérons l'algorithme du *Flood Set* (figure 22). Dans cet algorithme, à chaque ronde tous les processus envoient à tous leur vecteur et le mettent à jour avec les nouvelles valeurs reçues durant cette ronde. Avec un système synchrone, si un processus p tombe en panne dans la ronde r, tous les processus vivants dans cette ronde recevront tous les messages de p jusqu'à la ronde $r-1$, mais, il peut arriver que certains reçoivent le message de p de la ronde r alors que d'autres ne le recevront pas. Mais, si dans la ronde r, il n'y a pas de nouvelles pannes, à la fin de cette ronde, tous les processus vivants auront le *même* vecteur et ce vecteur ne changera plus. Au bout de $t+1$ rondes, on est sûr qu'il y a bien eu au moins une ronde sans pannes ce qui assure l'accord.

Mais pour arriver à cet accord *avant* la ronde $t+1$, un processus p peut décider à la fin de la ronde r si la condition suivante est réalisée :

(C) : p a reçu dans les rondes $r-1$ et r des messages provenant d'*exactement* le même ensemble de processus.

[52] Avec des pannes de ce type, un processus *incorrect* peut non seulement tomber en panne définitive mais aussi oublier d'envoyer des messages.

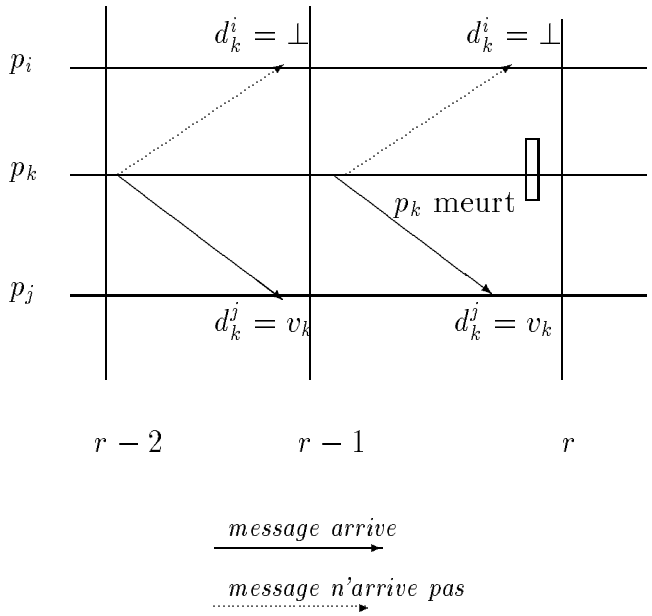


FIG. 36 – r et $r-1$ identiques pour p_i et p_j.

Dans ce cas, du point de vue de p, il n'y a pas eu de nouvelles pannes dans la ronde r. Il est facile de vérifier que, dans ce cas, p ne pourra pas obtenir de nouvelles valeurs et que le vecteur de p à la fin de cette ronde r *est* la valeur de décision. En $f+2$ rondes s'il y a f pannes, la condition (C) est nécessairement assurée.

Supposons maintenant que nous avons des rondes avec des détecteurs de défaillances parfaits. Comme le montre la figure 36, La condition (C) n'est plus suffisante, en effet, supposons que p_k envoie des messages jusque dans la ronde r, p_i peut soupçonner p_k trop tôt en $r-1$ et recevoir de ce fait des mêmes processus dans la ronde r et la ronde $r-1$, alors que p_j soupçonne p_k dans la ronde $r+1$, et en ronde r, p_j peut apprendre de p_k une nouvelle valeur que p_k a lui-même apprise dans la ronde $r-1$. Ainsi, au début de la ronde $r+1$, p_j peut avoir strictement plus de valeurs que p_i.

Pour éviter ce problème, si p_j sait que quelqu'un a soupçonné p_k dans la ronde $r-1$, il ne tiendra pas compte des informations issues de p_k dans la ronde r. Pour cela, il suffit, à chaque ronde, d'échanger les listes des processus qui ont été soupçonnés dans la ronde précédente et d'éliminer dans la ronde tous les messages issus de ces processus. De cette façon, si p_i et p_j sont vivants dans la ronde $r+2$, ils ont une vue cohérente de ce qui s'est passé jusqu'à la ronde r. Plus exactement, ils sont d'accord avec un décalage d'au plus une ronde sur la ronde pendant laquelle un processus mort a émis son dernier message [53].

[53] Ici, contrairement au cas général, la *contamination* ne pose pas de problème. Il y a conta-

```
   /* initialisation */
1  D_p =< ⊥,...,⊥,v_p,⊥,...,⊥ >                          /* D_p : le vecteur */
2  LP_p(0) = Π          /* LP_p(r) : liste des participants pour la ronde r  */
3  for r := 1 to t+1 do
      /* round r */
4     send(D_p, LP_p(r-1)) to all
5     LRP := ∅ /* liste des processus suspectés dans la ronde précédente */
6                          /* par aucun processus en qui p a confiance *
7     LP_p(r) := ∅       /*     LP_p(r) :liste des processus ''acceptables'' :*/
8                        /*              q ∈ LP_p(r) : q non suspecté par p */
9                        /*                        et, pour la ronde r-1, q*/
10                       /*suspecté par aucun processus en qui p a confiance */
11    forall q do
         /* attendre suivant détecteur de défaillances */
12       wait until
13          receive(D, L) from q
14                LP_p(r) := LP_p(r) ∪ q
15                D(q) := D                          /* D(q) : le vecteur de q */
16                AC := AC ∪ L                       /* D(q) : le vecteur de q */
17          or
18            q suspecté
19          end
20    LP_p(r) := LP_p(r) ∩ LRP
21    forall q ∈ LP_p(r) do
          /* les données de q sont ''acceptables'' */
22          forall s do
23            if D(q)[s] ≠ ⊥ then D_p[s] := D(q)[s]        /* mettre à jour D_p */
24    if COND then décider D_p
25                    send(Decide, D_p) to all
26 od
```

FIG. 37 – Trame de l'algorithme pour le *global data*.

Le texte de l'algorithme se trouve dans la figure 37.

La transformation évoquée plus haut élimine les messages de ceux que certains ont vu mourir plus tôt. $LP_p(r)$ correspond à la liste des participants vus par p pour la ronde r après cette transformation. Le processus p décide quand $COND$ est assurée. Cette condition $COND$ est simplement la disjonction des quatre conditions $(C1)$, $(C2)$, $(C3)$, $(C4)$ suivantes :

$(C1)$ est simplement l'arrêt quand la ronde $t+1$ est atteinte,

$(C2)$ est la traduction de (C) avec un décalage de 2 rondes : en ronde $r+2$, du point de vue de p et après la transformation, il n'y pas eu de nouvelles

mination si p_i a reçu des messages d'un processus (nécessairement défaillant) qui, lui, n'aurait pas la même vision. Ici, si quelqu'un a anticipé la mort de p_k, cela ne diminuera pas l'ensemble des valeurs initiales connues par p_i.

pannes en ronde r. Dans ce cas, p peut décider dans la ronde $r+2$. Avec f pannes, il faudra au plus $f+3$ rondes pour assurer cette condition.

Plus formellement, $(C2)$ est la condition :

$$LP_p(r-3) = LP_p(r-2)$$

$(C3)$ correspond à ce que pour p, les deux dernières rondes sont identiques : p n'a entendu parler d'aucune nouvelle panne dans les deux dernières rondes. Avec f pannes, pour assurer avec des rondes avec des détecteurs parfaits cette condition, il faut au plus $2f$ rondes. Il s'agit d'une condition correspondant au doublement des rondes.

Plus formellement, $(C3)$ est la condition :

$$LP_p(r-2) = LP_p(r-1) \wedge \forall j \in LP_p(r) : LP_j(r-1) = LP_p(r-1)$$

$(C4)$ enfin est une condition plus technique qui permet de terminer tout de suite si la *première* ronde est sans pannes.

En décidant dès qu'une de ces quatre conditions est assurée, on résout le problème du global data. Le point plus étonnant est que $(C2)$ *ou* $(C3)$ sont réalisées au plus tard dans la ronde $f+2$.

On peut résumer l'algorithme de la façon suivante : on fait feu de tout bois, et miracle, on décide aussi vite qu'en synchrone.

6.4 Multi-diffusion ordonnée

Cette section correspond à ce que j'ai fait avec C. Delporte dans [38, 37].

En quelque sorte, il s'agit d'une application des principes de la redondance active : rendre tolérant aux défaillances un algorithme qui ne l'est pas en réalisant chaque *processus* par un *groupe* qui a le même comportement que le processus. Cette démarche correspond à celle de la *state-machine approach* [68, 87]. Par ailleurs, cet exemple montre aussi que, dans certains cas, une solution fortement asymétrique peut être très efficace et simple.

Tel qu'il a été défini la première fois par [58], il s'agit d'un problème pratique de serveurs répliqués. Chaque serveur est réalisé par *réplication active* par un groupe de processus ou de processeurs : tous les processus du groupe exécutent le même code et doivent avoir vis-à-vis de l'extérieur le même comportement. Généralement, pour réaliser cette réplication active, on utilise la diffusion atomique. En effet, par la diffusion atomique, toutes les requêtes (les messages) sont délivrées exactement dans le même ordre par tous les processus, et, si ces processus exécutent le même code, à partir de la même histoire des requêtes, ils auront le même comportement.

Dans le cas où l'on a plusieurs services et donc plusieurs groupes de processus, on peut, bien entendu, utiliser encore la diffusion atomique et ordonner *toutes* les requêtes. Mais, dans ce cas, *toutes* les requêtes sont diffusées à *tous*, même si le service demandé ne concerne qu'un seul groupe. A priori, si m ne concerne que le service s et le service s', seuls les membres du groupe de s et de s' (ceux qui implémentent s et s') devraient participer à la diffusion atomique de m.

Plus formellement (cf figure 38), on suppose que l'on a des groupes de processus disjoints [54] $\{g_1, \ldots, g_m\}$, et que chaque message m a un ensemble de groupes destinataires $dest(m)$. On veut réaliser une multi-diffusion ordonnée : il faut que tous les messages soient délivrés (arrivent à leur destinataire) et que, de plus, les messages soient délivrés dans le *même* ordre par tous les destinataires. En fait, on veut une propriété légèrement plus forte : on veut que la relation induite par les dates de délivrances de messages soit acyclique. De plus, l'algorithme doit être *authentique* (*genuine*) dans le sens où *seuls les destinataires et la source d'un message participent à l'algorithme de diffusion.*

Concernant les pannes, on suppose, d'une part qu'il y a toujours *au moins un processus correct* dans chaque groupe et que le *consensus uniforme est réalisable à l'intérieur de chaque groupe* (par exemple, on peut disposer de détecteurs de défaillances internes à chaque groupe qui permettent de réaliser le consensus). La première chose à noter est que ces conditions concernant les groupes de processus et les pannes sont très fortes : elles permettent d'assurer que chaque groupe peut avoir le comportement d'un processus sans pannes. En effet, comme on peut faire du consensus à l'intérieur d'un groupe, on peut aussi réaliser la diffusion atomique à l'intérieur du groupe. On peut donc assurer que tous les processus recevront et traiteront les messages entrant dans le groupe dans le même ordre. De cette façon, tous les processus corrects d'un groupe peuvent

[54] Cette condition est nécessaire [60].

La multi-diffusion ordonnée est définie par deux primitives *MO-diffuse* et *MO-délivre* qui vérifient :

Validité : si un processus correct MO-diffuse m, tous les processus corrects des groupes destinataires de m MO-délivrent m,

Accord Uniforme : si un processus p MO-délivre m alors tous les processus corrects des groupes destinataires de m MO-délivrent m,

Intégrité Uniforme : m est MO-délivré au plus une fois par p et uniquement si m a été MO-diffusé et si p appartient à un des groupes destinataires.

Soit :

$m \prec m'$ si et seulement si un processus a MO-délivré m avant m'

Acyclicité : $\prec$ est acyclique.

FIG. 38 – Multi-diffusion Ordonnée.

La diffusion ordonnée sans pannes est définie par deux primitives *MO-diffuse* et *MO-délivre* qui vérifient :

Diffusion : tout message MO-Diffusé vers $p_1, .., p_l$ est MO-délivré exactement une fois par chaque p_i, $1 \leq i \leq l$. Tout message MO-Délivré a été MO-Diffusé.

Soit :

$m \prec m'$ si et seulement si un processus a MO-délivré m avant m'

Acyclicité : $\prec$ est acyclique.

FIG. 39 – Multi-diffusion Ordonnée version sans pannes.

avoir le même comportement et simuler le comportement d'un processus sans panne.

Considérons le problème de la multi-diffusion ordonnée (figure 39) pour des processus *sans défaillances* : chaque message a comme destinataire un ensemble de processus (sans pannes), il s'agit de délivrer tous les messages de façon à ce que la relation induite par la livraison des messages ($m \prec m'$ si et seulement si un processus délivre m avant de délivrer m') soit sans cycles.

En simulant un processus sans pannes par un groupe de processus, la version sans pannes de la spécification correspond exactement à la spécification avec pannes de la figure 38. Avec les hypothèses faites sur les groupes, étant donnée une solution à la version sans pannes de la spécification, on peut en dériver directement par application de la transformation esquissée dans la figure 40 une solution de la version avec pannes.

On va donc résoudre le problème dans un environnement sans pannes et ensuite traduire cette solution en suivant le schéma précédent. De fait, la plupart

Sans pannes		**Avec pannes**	commentaires
Processus p	$\longrightarrow$	Groupe $G(p)$	Chaque processus de $G(p)$ exécute le même code que p
send vers q	$\longrightarrow$	*A-diffuse* (dans $G(q)$)	en remplaçant les *send* par des *A-diffuse*
receive	$\longrightarrow$	*A-délivre* (dans $G(p)$)	et les *receive* par des *A-délivre*

FIG. 40 – Implémentation d'un processus par un groupe.

des solutions proposées [58, 59, 60, 53] reposent sur des adaptations de l'algorithme de Skeen [88] qui fonctionne dans un environnement sans défaillances et utilise fortement le relation de causalité. Cependant ces adaptations (à l'exception partielle de [60]) ne sont pas générales et il en résulte des algorithmes assez complexes dont certains sont même incorrects. Il est préférable de procéder comme suit :

- développer une solution pour une spécification en ne prenant pas en considération les pannes de processus,
- traduire automatiquement cette solution en une solution correspondant aux groupes de processus (avec pannes),
- faire ensuite quelques optimisations finales.

Considérons donc la spécification sans pannes. L'algorithme que nous avons proposé est très simple. Supposons que l'on veuille diffuser m vers les processus $p_1, \ldots, p_k$ [55] , on peut procéder comme suit :

1. on envoie m à p_1 qui délivre m, puis p_1 transmet m à p_2, qui lui-même le délivre et le retransmet au suivant p_3 jusqu'à atteindre p_k,
2. $p_1, \ldots, p_{k-1}$ *attendent* avant de faire quoi que ce soit que p_k ait délivré m : une fois que p_k délivre m, il envoie à tous un message libérant $p_1, \ldots, p_{k-1}$ de leur attente.

On peut vérifier assez aisément que cet algorithme assure les propriétés requises. Contrairement à l'algorithme de Skeen, cet algorithme est fortement asymétrique et c'est cette asymétrie qui le rend si simple. L'absence de cycle est claire : un cycle ne peut arriver puisque l'on attend avant de progresser que le message soit délivré partout.

Dans la transformation, les *send* de p_i vers p_{i+1} correspondent donc à une diffusion atomique vers le groupe $G(p_{i+1})$ correspondant. Enfin, et c'est une optimisation, on vérifie aussi que le message de p_k émis vers les autres n'a pas besoin d'être transformé en une diffusion *atomique* : un simple *send* à tous les processus des divers groupes suffit.

On peut, enfin, remplacer les diffusions atomiques vers les divers groupes par des diffusions (fiables) et un consensus dans le groupe. On vérifie alors que pour

[55] On suppose toujours que les processus sont ordonnés suivant leurs identités : $p_1 < p_2 ... < p_k$.

diffuser (MO-diffuser) m vers $G(p_1), \ldots, G(p_{k-1})$, chaque processus de chaque groupe ne participe qu'à *un seul* consensus, ce qui est optimal. Ce dernier point est important : la réalisation du consensus et/ou de la diffusion atomique à l'intérieur d'un groupe ne pouvant être réalisée si le système est asynchrone, elle nécessite l'appel à un oracle (ou, de façon moins abstraite, à des hypothèses supplémentaires sur les système). Cette appel à la magie de l'oracle étant, *a priori*, coûteux et risqué, il semble raisonnable de vouloir en limiter l'usage. On y reviendra dans la section suivante.

6.5 Diffusion générique

Ce travail, réalisé avec M. Aguilera, C. Delporte et S. Toueg, correspond à [1].

Le problème de la diffusion générique a été proposé par A. Schiper et F. Pedone dans [83, 84].

Comme dans la section précédente, le but est, dans une certaine mesure, de limiter l'usage des oracles. Dans la section précédente, on réalisait, en utilisant des oracles restreints uniquement aux groupes destinataires, une multi-diffusion ordonnée. Ici, on affaiblit les propriétés requises par la diffusion atomique : certains messages devront être délivrés partout dans le même ordre, alors que d'autres pourront être délivrés dans n'importe quel ordre. En relâchant, pour certains messages, la condition d'ordre sur les délivrances, on va essayer de limiter l'usage de l'oracle.

Dans un mécanisme de diffusion de messages, il n'est pas, en général, nécessaire d'assurer que *tous* les messages soient délivrés dans le même ordre par tous les processus. Par exemple, dans le cas de serveurs répliqués, des messages *read* n'ont pas besoin d'être traités dans le même ordre sur chaque serveur, mais, par contre, dès qu'un message de type *write* est concerné, il faut maintenir le même ordre de traitement sur tous les serveurs.

Pour cela, on définit une *relation de conflit* $\mathcal{C}$ entre les messages : si deux messages m et m' sont en conflit, alors m et m' doivent être délivrés dans le même ordre par tous les processus. Dans l'exemple précédent, m sera en conflit avec m' si et seulement si m *ou* m' est un message de type *write*. Notons que cette relation de conflit n'est généralement pas transitive. On supposera que cette relation est toujours symétrique et non réflexive.

Dans la diffusion *générique*, il s'agit d'assurer, dans tous les cas, la diffusion fiable des messages, et, pour deux messages en conflit, leur diffusion ordonnée (cf figure 41).

Remarquons que, si la relation de conflit est vide, alors la diffusion générique est simplement une diffusion fiable et, à l'autre extrême, si tous les messages sont en conflit, la diffusion générique est une diffusion atomique.

Notons que la diffusion atomique assure directement la diffusion générique. Mais, une telle implémentation va clairement à l'encontre de l'idée de la diffusion générique : seuls les messages en conflit devraient être délivrés dans le même ordre. Par ailleurs, assurer l'ordre de délivrance des messages est impossible sans un mécanisme externe (un oracle) dans un système asynchrone en présence de pannes de processus. Aussi, il paraît assez naturel d'essayer de limiter l'usage de cet oracle, qui, pour sa réalisation supposera des assertions supplémentaires sur le système, et qui, d'autre part, coûte, *a priori*, cher.

L'implémentation aura donc accès à un oracle (section 2.5.5) qui pourra être un oracle de diffusion atomique ou un oracle de consensus ou encore un détecteur de défaillances. Une implémentation devra limiter autant que possible l'appel à cet oracle.

On dira que l'implémentation est *non triviale pour un oracle* si elle assure que, pour une exécution sans aucun conflit, elle ne fait jamais appel à l'oracle.

La diffusion générique est définie par deux primitives *g-diffuse* et *g-délivre* qui vérifient :

Validité : si un processus correct g-diffuse m, tous les processus corrects g-délivrent m,

Accord Uniforme : si un processus g-délivre m alors tous les processus corrects g-délivrent m,

Intégrité Uniforme : m est g-délivré au plus une fois par chaque processus uniquement si m a été g-diffusé,

Ordre uniforme généralisé : si m et m' sont en conflit, et un processus g-délivre m avant m', alors un processus ne g-délivrera m' qu'après avoir g-délivré m.

(notons que dans tous les cas on a au moins une *diffusion fiable*)

FIG. 41 – Diffusion générique.

En d'autre termes, si l'oracle n'est pas nécessaire, on ne doit pas l'utiliser.

Cependant, cette définition permet d'utiliser sans limite l'oracle dès qu'un conflit survient. C'est la raison pour laquelle on propose une autre définition : on dira que l'implémentation est *économique pour un oracle*, si elle est non triviale et si, pour toute exécution telle qu'à partir d'un certain moment il n'y a plus de conflit, alors, à partir d'un certain moment, on ne fera plus appel à l'oracle.

Ces définitions font référence à un oracle donné, cependant on peut aisément montrer que, pour les oracles usuels permettant de résoudre la diffusion atomique, elles dépendent relativement peu de l'oracle choisi. Plus précisément, toute implémentation économique par rapport à un oracle de diffusion atomique peut être transformée en une implémentation économique par rapport à un oracle de consensus et réciproquement. De même, en présence d'une majorité de processus corrects, une telle implémentation peut aussi être transformée en une implémentation économique par rapport à un détecteur de défaillances $\diamond\mathcal{S}$ (le plus faible détecteur de défaillances permettant de résoudre le consensus).

Nous avons proposé deux implémentations de la diffusion générique utilisant un oracle de diffusion atomique, la première, relativement simple, est non-triviale, la deuxième est plus difficile et est économique par rapport à l'oracle de diffusion atomique. Toutes les deux fonctionnent dans un système asynchrone avec une majorité de processus corrects [56].

Les algorithmes proposés sont intéressants car ils mettent en oeuvre le coeur des techniques utilisées dans les algorithmes tolérant des pannes. Aussi, nous allons essayer de décrire la première implémentation et les intuitions qui permettent de la comprendre.

Quand un processus prend connaissance d'un message m qui doit être g-diffusé, il va, en deux rondes, déterminer si on doit faire appel à l'oracle pour diffuser le message (diffusion atomique pour le message), ou si on le diffuse

[56] Ces solutions améliorent la borne de $n > 3t$ proposée par [83].

par diffusion fiable simple. La difficulté est bien évidemment, que du fait de l'asynchronisme, certains processus peuvent avoir connaissance d'un message en conflit et d'autres non.

Supposons donc que m et m' soient en conflit. Si m et m' sont tous les deux diffusés par diffusion atomique, il n'y a pas de problèmes : la diffusion atomique assurera l'ordre. Par contre, il ne faut pas qu'un processus décide de diffuser m par une diffusion fiable alors qu'un autre processus déciderait de diffuser m' par diffusion fiable. Mais, à cause de la non-trivialité, on ne peut éviter que l'un des deux (disons m) soit diffusé par une diffusion fiable alors que l'autre (m') sera diffusé par une diffusion atomique. Dans ce cas, il faut, pour assurer l'ordre, qu'au moment de délivrer m' on connaisse l'existence de m et que l'on force la délivrance de m avant celle de m'.

Plus précisément, dans la version donnée figure 42, le message $FIRST$ informe tous les processus du message m qui doit être diffusé.

Si m n'est en conflit avec aucun message déjà rencontré par le processus p, p répond *à tous* par un message $GOOD$, sinon il répond par un message BAD.

Dans la ronde suivante, si un processus p a reçu d'au moins une majorité de processus un message $GOOD$ pour m, alors p considère que m pourrait être diffusé par une diffusion fiable. Remarquons que, dans ce cas, aucun message m' en conflit avec m n'a pu et ne pourra jamais obtenir une majorité de réponses $GOOD$. Ainsi, ayant obtenu une majorité de $GOOD$, on élimine toute possibilité de diffusion fiable sur un message en conflit avec m.

Si on ne peut donc plus avoir une diffusion fiable à la fois pour m et m', on peut encore avoir une diffusion fiable de m *et* une diffusion atomique de m'. Dans ce cas, il faut assurer que la délivrance (atomique) de m' et la délivrance (fiable) de m aient lieu dans le même ordre pour tous. Pour cela, avant de délivrer (atomique) m', chaque processus devra délivrer m. Aussi, si on décide de délivrer par diffusion atomique m', il faudra en même temps connaître tous les messages en conflit qui ont pu être délivrés par diffusion fiable. Chaque processus maintiendra donc la liste des messages qu'il considère comme pouvant être délivrés par diffusion fiable (DF).

Avant de pouvoir décider de diffuser m par une diffusion fiable, il faut que m soit connu au cas où un m' en conflit avec m arrive ultérieurement. Pour cela, il faut une ronde supplémentaire ($THIRD$) pour assurer que m figure dans la liste DF chez suffisamment de processus dont au moins un processus correct. Comme, on a supposé qu'il y a une majorité de processus corrects, il suffit d'assurer que m figure dans une majorité de listes DF.

Si un m' en conflit arrive ensuite, m' sera diffusé par diffusion atomique *avec* la réunion des listes $DL \cup C(m')$ des messages en conflit avec m'. D'après ce que l'on vient de voir m figurera forcément dans cette liste. Il suffit, alors, au moment de la délivrance, de délivrer *d'abord* les éléments de cette liste et *ensuite* m'.

Il est intéressant de noter, que, dans un certain sens, les rondes $SECOND$ et $THIRD$ ont des fonctions légèrement différentes :

- La ronde $SECOND$, attendant une majorité de réponses, assure qu'il ne peut y avoir une majorité de $GOOD$ à la fois pour m et pour m' en conflit.

Son rôle est d'éliminer une possibilité. Pour cela, on utilise la propriété de majorité : il ne peut y avoir à la fois une majorité de $GOOD$ pour m et une majorité de $GOOD$ pour m'.

- La ronde $THIRD$ doit assurer que si la décision est prise de diffuser m par diffusion fiable, alors il y en aura toujours une trace. Si un m' en conflit arrive, on saura toujours que m a pu être diffusé par diffusion fiable. Comme dans le cas précédent, une majorité de réponses suffit.

Pour conclure, nous allons essayer de dégager quelques principes généraux.

Dans une certaine mesure, ces deux rondes ressemblent un peu aux deux rondes que l'on a dans l'algorithme de consensus probabiliste de Ben-Or [15, 6] (ou dans des algorithmes dits hybrides [5, 81], c'est-à-dire utilisant à la fois des tirages aléatoires et des détecteurs de défaillances). Dans cet algorithme, au cours de la première ronde, les processus envoient leur valeur. Un processus choisit à la fin de cette première ronde, soit v s'il a reçu cette valeur v d'une majorité de processus, soit " ?". Cette ronde correspond à la ronde $SECOND$, dans le sens où si v est choisi, aucune autre valeur n'a pu être choisie.

Dans la deuxième ronde, les processus échangent leur (nouvelle) valeur, si un processus reçoit d'une majorité la même valeur (différente de " ?"), un processus décide de cette valeur. Sinon, il change sa valeur pour la valeur (différente de " ?") qu'il a reçue. Si p décide sur v, il sait que, nécessairement, *tous* les processus corrects auront aussi reçu v et auront donc changé leur valeur pour v. Avec la ronde $THIRD$ c'est similaire, si m figure dans une majorité de liste DF, dans tous les cas, si un m' en conflit arrive, m sera toujours connu comme pouvant avoir été diffusé par diffusion fiable. Pour l'algorithme de Ben-Or, si à la fin de la deuxième ronde on n'a obtenu que des " ?", on tire la nouvelle valeur, alors qu'ici on fera appel à l'oracle de diffusion atomique.

Notons enfin que, si $3t < n$, on peut fusionner les rondes $SECOND$ et $THIRD$: si au cours de la ronde $SECOND$ on reçoit plus des deux tiers de réponses $GOOD$ pour m, m peut être diffusé par diffusion fiable. En effet, aucun autre message m' en conflit ne pourra être diffusé par diffusion fiable, et si un m' en conflit avec m arrive, m figurant chez une majorité de processus corrects sera toujours dans plus du tiers des messages donnant la liste des messages en conflit avec m'.

6.5.1 Diffusion atomique *frugale*

Une application de la diffusion générique concerne la diffusion atomique elle-même.

Pour réaliser la diffusion atomique en asynchrone, il faut, comme on l'a vu, faire appel à un oracle. Comme précédemment, on veut, ici, limiter l'usage de cet oracle. On va donc implémenter une diffusion atomique grâce à un oracle de diffusion atomique, *mais en appelant l'oracle le moins souvent possible* !

Rappelons qu'à partir de l'ordre causal entre les événements [69] on peut ordonner certaines délivrances de messages. Plus précisément, si les émissions de m et m' sont causalement ordonnées, on peut, en utilisant des *horloges vectorielles* [50], même en présence de pannes, assurer que les réceptions ont lieu

```
   /* DF  les  messages pour lesquels p propose d'utiliser */
   /* la diffusion fiable */
   /* C(m) messages en conflit avec m */

 1 to g − diffuser(m)
 2    send(m, FIRST) to all

 3 upon receive(m, FIRST)
 4    if m  en conflit avec aucun des messages vus
 5    then send(m, GOOD, SECOND)
 6    else send(m, BAD, SECOND)

 7 upon receive(m, ∗, SECOND) from n − t  processus
 8    if ∗ = GOOD  pour plus de  n/2 processus
 9         ∧ m  en conflit avec aucun messages vus
10    then DF := DF ∪ {m}          /* OK pour diffusion fiable de m */
11         send(m, OK, ∅, THIRD) to sender(m)
12    else              /* NOK pour diffusion fiable à cause de DF ∩ C(m) */
13         send(m, NOK, DF ∩ C(m), THIRD) to sender(m)

14 upon receive(m, ∗, E, THIRD) from n − t  processus
15     if ∗ = OK   pour plus de  n/2 processus
16     then
17         DiffusionFiable(m)
18     else
19         Avant := ⋃ { L | (m, NOK, L, THIRD)  reçu}
20         /* Avant : messages à délivrer avant m */
21         DiffusionAtomique(Avant, m)


   /* DiffusionFiable et DiffusionAtomique ne délivrent que */
   /* les messages qui n'ont pas déjà été délivrés */
   /* DiffusionAtomique(L, m) délivre L avant m */
```

FIG. 42 – Diffusion générique non triviale.

pour tous les processus dans le même ordre [61]. Si on veut réaliser une diffusion atomique, on peut utiliser cet ordre causal et n'utiliser l'oracle que pour les messages qui sont *concurrents*.

Pour cela, on associe à chaque message m son estampille vectorielle (codage de l'ordre causal) $ts(m)$. Les estampilles définissent une relation d'ordre partiel : si $ts(m)$ et $ts(m')$ sont ordonnées, on peut assurer l'ordre de délivrance de m et m' sans faire appel à un oracle. Si $ts(m)$ et $ts(m')$ sont incomparables, il faudra alors faire appel à l'oracle de diffusion atomique. On définit donc la relation de conflit : $(m, ts(m))$ et $(m', ts(m'))$ sont en conflit si et seulement si $ts(m)$ et $ts(m')$ sont incomparables. Ainsi, deux messages sont en conflit si et seulement si ils sont concurrents (causalement incomparables). En utilisant un algorithme économique de diffusion générique avec cette relation de conflit, on obtiendra une implémentation de la diffusion atomique qui n'utilise l'oracle de diffusion atomique que pour des messages concurrents.

6.6 Élection stable de leader

Cette section correspond à [2, 3].

Nous avons vu précédemment l'intérêt théorique que présentaient les détecteurs de défaillances. Reste maintenant à étudier leur intérêt pratique.

On peut, tout d'abord, considérer la détection comme un service et s'intéresser à la réalisation et l'évaluation de ce service. C'est ce qui est fait dans [27], où les auteurs définissent, dans un modèle probabiliste, diverses mesures pour la qualité d'un service de surveillance (*monitor*) des pannes et indiquent comment mettre en oeuvre un tel service. Cependant, du point de vue de l'algorithmique distribuée, assurer une certaine qualité de service sur la détection des pannes de *chaque* processus n'est pas tout à fait suffisant. Pour résoudre le consensus, les détecteurs de défaillances ont besoin de propriétés plus globales, comme, par exemple, assurer que, à partir d'un certain moment, au moins un processus correct, le *même* pour tous, ne sera plus soupçonné. Pour obtenir ces propriétés globales de détection de pannes, on est naturellement amené à développer une algorithmique plus spécifique pour réaliser des détecteurs de défaillances permettant de résoudre le consensus. La première chose est de déterminer quel est, du point de vue pratique, le détecteur de défaillances le plus intéressant à implémenter. Ensuite, il faut trouver des algorithmes efficaces pour son implémentation.

Concernant le premier point, Ω est un détecteur de défaillances particulièrement intéressant : il correspond à une élection ultime de leader. Plus exactement, avec Ω, la sortie du détecteur de défaillances de chaque processus est un *unique* processus. Ultimement, tous les détecteurs vont s'accorder sur le *même* processus *correct*.

On peut aussi présenter Ω (cf aussi la section 4.2.4) comme un oracle de groupe qui constitue ultimement un groupe de processus corrects (groupe à un seul élément [57] : le leader). Comme on le voit, Ω réalise une élection (au sens informatique du terme : tout le monde s'accorde sur qui est l'élu) ultime.

Par ailleurs, Ω est équivalent à $\diamond\mathcal{S}$, le plus faible des détecteurs de défaillances : cela signifie que si un algorithme résout le consensus dans un système asynchrone avec un détecteur de défaillances, alors ce détecteur de défaillances peut se transformer en Ω. Comme Ω est dans la classe $\diamond\mathcal{S}$, tous les algorithmes de consensus utilisant $\diamond\mathcal{S}$ (par exemple, [19, 4, 86, 79, 66]) s'appliquent directement à Ω. Il existe aussi des algorithmes utilisant explicitement Ω ([70, 80]). D'autre part, il existe plusieurs travaux concernant la réalisation de Ω, par exemple [28, 73, 74].

Ce qui nous intéresse ici [2] est d'obtenir une algorithmique efficace pour réaliser cette élection de leader. Comme Ω permet de réaliser le consensus, cette implémentation ne peut pas se faire dans un système purement asynchrone. En termes de systèmes partiellement synchrones, pour implémenter Ω, il suffit d'avoir un système ultimement fort, c'est-à-dire tel qu'ultimement tous les messages issus d'un certain processus correct arrivent dans les délais. Ici, nous

[57] Ce qui est souvent un avantage : il est beaucoup plus facile de prendre des décisions quand il n'y a qu'une seule personne!

aurons une condition légèrement plus forte, mais, d'un autre côté, nous autorisons des pertes de messages. On dira qu'un processus p est *accessible* à l'instant τ, si toutes les communications de p et vers p à l'instant τ respectent les délais. On supposera qu'ultimement il existe un processus correct définitivement accessible [58].

Sous cette hypothèse, il s'agit d'implémenter *efficacement* Ω, c'est-à-dire de trouver un leader ultime. Pour cela, on veut :

- **(stabilité)** ne pas changer de leader si le leader actuel est satisfaisant pendant suffisamment de temps,
- **(économie)** ne pas échanger trop de messages, en particulier, on veut qu'ultimement l'algorithme n'utilise plus qu'un nombre *linéaire* de canaux de communications [59].

Une des difficultés de ce travail est de donner les bonnes définitions correspondant à ces deux propriétés.

Nous renvoyons à [2] pour les algorithmes vérifiant ces deux propriétés.

On applique, ensuite, cette élection stable de leader à la construction d'un détecteur de défaillances $\diamond\mathcal{P}$. L'idée est assez simple : le leader détermine quels sont les processus corrects. Ainsi, uniquement les liens de communication du leader et vers le leader sont utilisés et on obtient ainsi une implémentation de $\diamond\mathcal{P}$ qui sera ultimement linéaire avec des conditions plus simples que dans [72]. Comme on l'a noté plus haut, $\diamond\mathcal{P}$ est particulièrement intéressant car il peut être directement considéré comme un oracle de groupe. En effet, ultimement, tous les processus font partie du même groupe ; on peut ainsi en déduire un service de *group membership*.

[58] Il faut noter que les diverses implémentations proposées de Ω supposent une propriété plus forte qui correspond aux systèmes que l'on a appelés *ultimement synchrones*.

[59] En fait, dans l'algorithme proposé, ultimement, le leader seul émet des messages vers tous.

6.7 Autres travaux

6.7.1 Bornes inférieures pour le consensus

Dans [34], nous avons étudié des bornes inférieures pour obtenir le consensus avec des détecteurs de défaillances. Le but de ce travail est de pouvoir comparer du point de vue quantitatif les diverses approches utilisant les détecteurs de défaillances. Pour cela, il faut tout d'abord définir des mesures raisonnables. La mesure généralement utilisée est dérivée de la notion de *latence*, c'est-à-dire la longueur des plus longues chaînes de messages. Cette mesure est assez raisonnable [60]. Si l'on peut ainsi définir une mesure de complexité pour une exécution, le problème est d'obtenir une mesure pour les performances d'un algorithme. Choisir une analyse *meilleur cas* amène, avec des pannes, à des résultats un peu absurdes, permettant, par exemple, d'atteindre le consensus plus rapidement en présence de pannes. De même, une analyse *pire cas* donne rapidement des mesures non bornées.

Un autre problème est celui de la comparaison des algorithmes de consensus utilisant des détecteurs de défaillances dans les classes $\diamond\mathcal{S}$ ou $\diamond\mathcal{P}$: une bonne mesure devrait prendre en compte, non le cas où il n'y a aucune panne, mais, plutôt, le temps nécessaire pour arriver à converger une fois que le détecteur de défaillance vérifie ses propriétés ultimes.

Dans [34], nous donnons des bornes inférieures pour obtenir le consensus avec des détecteurs de défaillances $\mathcal{P}$ et $\mathcal{S}$. Le résultat le plus surprenant (cf section 3.4) est que, avec $\mathcal{S}$, le nombre de rondes nécessaire pour atteindre le consensus est indépendant du nombre de pannes tolérées [61]. En effet, quel que soit le nombre de pannes tolérées, n rondes peuvent être nécessaires pour atteindre le consensus avec des détecteurs de défaillances $\mathcal{S}$. Ce résultat est relativement difficile à démontrer : après s'être ramené à un système de rondes, on montre que si toutes les décisions s'obtiennent en moins de n rondes, on peut construire une séquence d'exécutions indistingables [62] telle qu'une de ces exécutions décide 0 alors qu'une autre décidera 1. Ce procédé de preuve est classique.

6.7.2 Travaux plus anciens

Terminons en rappelant quelques travaux plus anciens.

L'objectif de [49] était de définir une sémantique pour les comportements infinis de langages du type CSP. Le principal problème était de définir une sémantique pour les problèmes d'équité. Pour cela, l'utilisation des langages infinitaires s'est révélée très intéressante. Dans une certaine mesure, ce genre de sémantique, peut-être plus en terme de systèmes de transition que de langage, pourrait s'avérer aussi très intéressante concernant la tolérance aux pannes.

[60] Cependant, d'autres mesures comme celles définies dans [10], peuvent mieux prendre en compte le parallélisme des calculs.

[61] A condition que ce nombre ne soit pas 0 !

[62] Intuitivement, deux exécutions sont indistingables, si pour au moins un processus rien ne peut les distinguer.

En effet, les définitions des différentes classes de détecteurs de défaillances s'expriment dans une forme assez proche de la logique temporelle. On pourrait alors intégrer cette approche dans le cadre des systèmes de transition comme ceux des I/O automata [77]. De cette façon, une partie de l'algorithmique tolérante aux pannes pourrait être traitée en utilisant les méthodes formelles de la vérification.

Dans [42], il s'agissait d'étudier dans quelle mesure certains problèmes peuvent se paralléliser avec un coût de communication faible et, finalement, d'obtenir une sorte de critère de parallélisation des algorithmes.

Dans [25], nous étudions des propriétés assez fines des exécutions asynchrones : il s'agit par exemple de pouvoir affirmer qu'une exécution est nécessairement passée par un état global vérifiant telle ou telle propriété. L'intérêt de cette étude réside aussi dans le caractère géométrique et visuel de la représentation des exécutions. Pour ces travaux, on suppose qu'il n'y a pas de pannes. Il pourrait être intéressant de regarder ce qui se passe en présence de pannes et d'étudier quelle est alors la signification des divers transformateurs de prédicats qui sont considérés dans cet article. D'une manière plus générale, il serait intéressant d'étendre à la tolérance aux pannes les travaux et résultats concernant les modèles pour les systèmes asynchrones sans pannes. Et, en ce qui concerne les systèmes de transition, il serait intéressant de définir, par exemple, une logique temporelle adaptée à l'algorithmique tolérante aux pannes. En effet, de nombreuses propriétés de l'algorithmique distribuée résistante aux défaillances proviennent de propriétés générales bien étudiées dans les logiques temporelles.

Enfin, [31] contient un algorithme de consensus basé sur le paradigme du coordinateur tournant. Cet algorithme a la propriété de fonctionner pour des modèles de pannes plus généraux que les pannes définitives de processus. En particulier, il tolère les pertes de messages à condition qu'une infinité de messages d'un lien de communication entre processus corrects arrivent à leur destinataire (*fair lossy links*). Bien entendu, il s'agit d'échanger le moins de messages possible.

7 Conclusion

Au cours des dernières années, de nombreux progrès ont été réalisés concernant l'algorithmique distribuée résistante aux pannes.

D'une part, la situation s'est clarifiée entre ce qui est possible et ce qui ne l'est pas. Cette clarification permet de mieux cerner les caractéristiques essentielles des grands types de systèmes (synchrone, asynchrone et partiellement synchrone). D'autre part, on sait ce qu'il faut ajouter pour rendre réalisable ce qui ne l'était pas. De ce point de vue, la notion de détecteur de défaillances (et plus généralement d'oracle) permettant de continuer à pouvoir faire abstraction du temps est un apport fondamental sur le plan théorique.

Cette notion d'oracle est une abstraction qui permet de bien cerner ce que l'on rajoute au système. En ce sens, l'existence d'une plus faible classe de détecteur de défaillances pour le consensus est un résultat important. Il serait intéressant d'étendre ce genre de résultats à d'autres classes de problèmes (exclusion mutuelle, élection de leader, accord plus fort par exemple). Pour cela, il faut, sans doute, affiner la hiérarchie des détecteurs de défaillances, progresser dans la comparaison de ces classes et essayer de montrer la minimalité de certaines classes pour un problème donné.

De même, on peut étendre cette démarche avec la notion d'oracle dépendant du schéma de pannes qui est une généralisation du concept de détecteur de défaillances. N'oublions pas que les primitives utiles pour la résistance aux défaillances des divers systèmes de *middleware* peuvent être considérées comme des oracles. Grâce à ces derniers, on a un cadre général formel pour définir et étudier ces primitives.

Comme l'atteste, par exemple, le grand nombre de systèmes de *middleware* reposant sur le concept parfois imprécis de *virtual synchrony* existant actuellement, une certaine clarification est nécessaire dans ce domaine. Pour l'instant, les outils pour les étudier et les comparer sont encore limités. Pourtant cette comparaison est utile aussi bien d'un point de vue théorique (quel est le meilleur des mondes ?) que pratique (quel est le meilleur des mondes *possible* ?).

Il faut donc continuer à développer des modèles formels qui pourraient, un jour, permettre d'appliquer certains des résultats et des pratiques issus du domaine de la vérification à l'algorithmique distribuée tolérante aux pannes. Cela concerne aussi la prise en compte du temps réel qui est nécessaire pour de nombreuses applications.

Enfin, grâce aux progrès réalisés, se pose maintenant le problème plus pratique de la réalisation et de l'implémentation. Par exemple, si les détecteurs de défaillances sont un outil théorique puissant, pour prouver leur utilité pratique, il faut les réaliser pratiquement et *efficacement*. C'est, partiellement ce qui est tenté (mais à un niveau encore assez théorique) dans [3].

Le dernier point concerne plus directement l'algorithmique. Comme, je l'espère, on l'a remarqué, les algorithmes distribués tolérants aux pannes utilisent souvent les mêmes mécanismes. Pour transmettre et augmenter le savoir-faire dans le domaine, il faudrait arriver à répertorier ces mécanismes et à comprendre leur caractère universel. Par exemple, une question intéressante serait

de déterminer quels sont les rapports entre la résistance aux pannes telle que nous l'avons présentée ici et la notion d'auto-stabilisation.

Il est vraisemblable que de nombreux concepts utilisés en algorithmique distribuée résistante aux défaillances dépassent largement ce cadre.

Table des matières

1 Introduction **2**

2 Les divers modèles **6**
2.1 Introduction 6
2.1.1 Systèmes, modèles de calcul, systèmes avec oracles 6
2.1.2 Modèles 8
2.1.3 Comparaisons 10
2.2 Généralités 12
2.3 Modèles de systèmes 13
2.3.1 Systèmes synchrones 13
2.3.2 Systèmes asynchrones 13
2.3.3 Systèmes partiellement synchrones 13
2.4 Les modèles de calcul : systèmes de rondes 16
2.4.1 Un peu de formalisme 18
2.4.2 Les rondes parfaitement synchronisées RPS 20
2.4.3 Les rondes synchronisées RS 21
2.4.4 Autres modèles de rondes correspondant aux modèles synchrones (RAP, RAS) 22
2.4.5 Modèles de rondes correspondant à des systèmes partiellement synchrones 22
2.5 Systèmes avec oracles : détecteurs de défaillances 24
2.5.1 Détecteurs de défaillances 25
2.5.2 Implémentation des détecteurs de défaillances 26
2.5.3 Réalisation des détecteurs à partir de modèles de rondes . 28
2.5.4 Réalisation de modèles de rondes à partir de détecteurs de défaillances 28
2.5.5 Oracles 29
2.6 Simulations, traductions et comparaison des modèles de calcul . . 31
2.6.1 Introduction 31
2.6.2 Simulations 33
2.6.3 Traductions 36
2.6.4 Comparaisons entre modèles de calcul 37

3 Synchrone : Utiliser des détecteurs de défaillances en synchrone ? 39
3.1 Introduction 39
3.2 Rondes synchronisées et détecteurs parfaits 44
3.2.1 De synchrone aux rondes synchronisées 44
3.2.2 Des rondes synchronisées aux rondes parfaitement synchronisées 45
3.3 Asynchrone avec détecteurs de défaillances $\mathcal{P}$ 49
3.3.1 Une traduction générale de RAP vers RS 50
3.3.2 Simulation des rondes parfaitement synchronisées (RPS) par des RAP 51

3.3.3 Impossibilité d'une simulation des RS par des RAP avec décalage de moins $t+1$ . . . 51
3.3.4 Consensus et autres problèmes . . . 53
3.4 Rondes avec $\mathcal{S}$ (RAS) . . . 54
3.5 Conclusion . . . 55

4 Modèles en rondes de calcul pour l'asynchrone 57
4.1 Présentation générale . . . 57
4.2 Phases synchronisées . . . 63
4.2.1 Spécifications des phases synchronisées . . . 63
4.2.2 Puissance des systèmes de phases synchronisées : vivacité 64
4.2.3 Implémentation des phases synchronisées . . . 67
4.2.4 Modèles de calculs et détecteurs de défaillances . . . 69
4.2.5 Modèles de calcul pour l'asynchrone et systèmes asynchrones avec temps . . . 71
4.2.6 Conclusion . . . 73
4.3 Conclusion . . . 75

5 Conclusion des comparaisons et perspectives 76

6 Autres travaux algorithmiques 79
6.1 Introduction . . . 79
6.2 Les phases synchronisées . . . 80
6.2.1 Le projet ATR . . . 80
6.2.2 Diffusion Atomique et SPS . . . 82
6.2.3 Systèmes de phases et temps réel . . . 85
6.3 *Global data* avec terminaison au plus tôt . . . 88
6.4 Multi-diffusion ordonnée . . . 93
6.5 Diffusion générique . . . 97
6.5.1 Diffusion atomique *frugale* . . . 100
6.6 Élection stable de leader . . . 103
6.7 Autres travaux . . . 105
6.7.1 Bornes inférieures pour le consensus . . . 105
6.7.2 Travaux plus anciens . . . 105

7 Conclusion 107

Références

[1] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In Springer-Verlag, editor, *Proceeding of the 14th International Symposium on Distributed Algorithms(DISC00)*, Toledo, Oct. 2000.

[2] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. Research Report 2001/04, LIAFA, Paris, France, May 2001.

[3] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In Sringer-Verlag, editor, *Proceeding of the the 15th International Symposium on Distributed Algorithms(DISC01)*, 2001. to appear.

[4] M. K. Aguilera, W. Chen, and S. Toueg. Quiescent reliable communication and quiescent consensus in partitionable networks. Technical Report TR 97-1632, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, June 1997.

[5] M. K. Aguilera and S. Toueg. Randomization and failure detection : A hybrid approach to solve consensus. *Lecture Notes in Computer Science*, 1151, 1996.

[6] M. K. Aguilera and S. Toueg. Correctness proof of Ben-Or's randomized consensus algorithm. Technical Report TR98-1682, Cornell University, Computer Science, May 17, 1998.

[7] M. K. Aguilera and S. Toueg. A simple bivalency proof that t-resilient consensus requires t + 1 round. *Information Processing Letters*, 71((3-4)) :155–158, 1999.

[8] Y. Amir, P. Moser, L.E. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4) :311–342, Nov. 95.

[9] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical report, INRIA, Rocquencourt, July 1995.

[10] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4) :804–823, Oct. 1985.

[11] AXLOG, D. Aviation, INRIA, LIAFA, LIX, LMC-IMAG, and Thomson-CSF. Exemple de problèmes applicatifs de type consensus ou de type diffusion atomique temps-réel, spécification du problème étudié, rapport tranche ferme contrat atr. Technical report, DRET/MENESR/CNRS, 1997.

[12] AXLOG, D. Aviation, INRIA, LIAFA, LIX, LMC-IMAG, and Thomson-CSF. Rapport final. Technical report, DRET/MENESR/CNRS, 1999. Rapport tranche conditionnel contrat ATR.

[13] O. Babaoğlu, R. Davoli, L.-A. Giachini, and M. G. Baker. *RELACS : a communications infrastructure for constructing reliable applications in large-scale distributed systems.* BROADCAST Project deliverable report, 1994. Department of Computing Science, University of Newcastle upon Tyne, UK.

[14] R. Bazzi and G. Neiger. Simulating crash failures with many faulty processors. In A. Segal and S. Zaks, editors, *Proceedings of the Sixth International Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes on Computer Science*, pages 166–184. Springer-Verlag, 1992.

[15] M. Ben-Or. Another advantage of free choice : Completely asynchronous agreement protocols. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 27–30, Aug. 1983.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[17] K. P. Birman. *Reliable Distributed Computing with the ISIS Toolkit.* IEEE Computer Society Press, 1994.

[18] K. P. Birman, R. Cooper, T. A. Joseph, K. P. Kane, and F. B. Schmuck. *Isis - A Distributed Programming Environment*, June 1990.

[19] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4) :685–722, July 1996.

[20] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 322–330, Philadelphia, Pennsylvania, USA, May 1996.

[21] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2) :225–267, Mar. 1996.

[22] B. Charron-Bost, C. Delporte, H. Fauconnier, G. Le Lann, and P. Minet. Rapport intermediaire 1.1. Technical report, DRET/MENESR/CNRS, 1997. Rapport tranche ferme contrat ATR.

[23] B. Charron-Bost, C. Delporte, H. Fauconnier, G. Le Lann, and P. Minet. Rapport intermediaire 1.2a. Technical report, DRET/MENESR/CNRS, 1997. Rapport tranche ferme contrat ATR.

[24] B. Charron-Bost, C. Delporte, H. Fauconnier, G. Le Lann, and P. Minet. Rapport final. Technical report, DRET/MENESR/CNRS, 1999. Rapport tranche ferme contrat ATR.

[25] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed system. *ACM TOPLAS*, jan 1995.

[26] B. Charron-Bost, R. Guerraoui, and A. Schiper. Synchronous system and perfect failure detector : Solvability and efficiency issues. In *International Conference on Dependable Systems and Networks (IEEE Computer Society)*, 2000.

[27] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *International Conference on Dependable Systems and Networks (ICDSN/FTCS-30)*, June 2000.

[28] F. Chu. Reducing Ω to $\diamond W$. *Information Processing Letters*, 67(6) :298–293, Sept. 1998.

[29] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6) :642–657, 1999.

[30] F. Cristian and F. Schmuck. Agreeing on processor-group membership in asynchronous distributed systems. Technical Report CSE95-428, UCSD, 1995.

[31] C. Delporte-Gallet and H. Fauconnier. Un algorithme de consensus résistant aux pannes définitives de processus et aux pertes de messages. In *RenPar'9*, 1997.

[32] C. Delporte-Gallet and H. Fauconnier. Synchronized phases systems. Technical Report 11, LIAFA, université Denis Diderot Paris 7, 1998. Rapport LIAFA 98.

[33] C. Delporte-Gallet and H. Fauconnier. Système de phases. Technical report, LIAFA, université Denis Diderot Paris 7, LIAFA, université Denis Diderot, 2, pl. Jussieu, 75221 Paris cedex 05, 1998.

[34] C. Delporte-Gallet and H. Fauconnier. Greatest lower bounds for consensus using unreliable failure detectors. In *Brief Annoucement the 13th International Symposium on Distributed Algorithms(DISC99)*, Bratislava, 1999.

[35] C. Delporte-Gallet and H. Fauconnier. Real-time fault tolerant atomic broadcast. In *18 th IEEE Symposium on Reliable Distributed Systems (SRDS 99)*, pages 48–55, 1999.

[36] C. Delporte-Gallet and H. Fauconnier. Synchronized phases systems. In *3th International Conference on Principles of Distributed Computing (OPODIS 99)*, 1999.

[37] C. Delporte-Gallet and H. Fauconnier. Fault-tolerant genuine atomic multicast to multiple group. Technical Report 11, LIAFA, Paris, 2000.

[38] C. Delporte-Gallet and H. Fauconnier. Fault-tolerant genuine atomic multicast to multiple groups. In *4th International Conference on Principles of Distributed Computing (OPODIS 2000)*, 2000.

[39] C. Delporte-Gallet and H. Fauconnier. Greatest lower bounds for consensus using unreliable failure detectors. Technical Report 2000/07, LIAFA, Paris, France, 2000.

[40] C. Delporte-Gallet and H. Fauconnier. An example of real-time group communication system. In *WARG01, IEEE*, Phoenix, april 2001.

[41] C. Delporte-Gallet, H. Fauconnier, J.-M. Hélary, and M. Raynal. Early stopping in global data computation. Research Report 2001/03, LIAFA, Paris, France, May 2001.

[42] C. Delporte-Gallet, H. Fauconnier, and M. Nivat. Parallélisation d'algorithmes avec un nombre fixe de processeurs. *rairo*, 24(4) :353–386, 1990.

[43] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1) :77–97, Jan. 1987.

[44] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Commun. ACM*, 39(4), Apr. 96. NOM.

[45] D. Dolev and R. Reischuk. Bounds on information exchange for Byzantine agreement. *J. ACM*, 32(1) :191–204, Jan. 1985.

[46] D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in Byzantine agreement. *J. ACM*, 37(4) :720–741, Oct. 1990.

[47] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2) :288–323, Apr. 1988.

[48] P. D. Ezhilchelvan, R. A. Macêdo, and S. K. Shrivastava. Newtop : a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, Vancouver, BC, Canada, June 1995.

[49] H. Fauconnier. Sémantique asynchrone et comportement infinis en CSP. *Theoretical Comput. Sci.*, 54 :277–298, 87.

[50] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Eleventh Australian Computer Science Conference*, University of Queensland, Australia, 1988.

[51] M. J. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14 :183–186, 1982.

[52] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, Apr. 1985.

[53] U. Fritzke, P. Ingels, A. Mostéfaoui, and M. Raynal. Fault-tolerant total order multicast to asynchronous groups. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, pages 578–585, West Lafayette, Indiana, Oct. 1998.

[54] E. Gafni. Round-by-round fault detectors (extended abstract) : Unifying synchrony and asynchrony. In *PODC : 17th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1998.

[55] A. Gopal. *Fault-Tolerant Broadcasts and Multicasts : The Problem of Inconsistency and Contamination.* PhD thesis, cornell, Jan. 1992.

[56] A. Gopal and S. Toueg. Inconsistency and contamination. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 257–272, Aug. 1991.

[57] R. Guerraoui. Indulgent algorithms (preliminary version). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, Portland, Oregon, USA*, pages 289–297. ACM, July 2000.

[58] R. Guerraoui and A. Schiper. Genuine atomic multicast. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG-11)*, Saarbrücken, Germany, Sept. 1997.

[59] R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS-17)*, pages 578–585, Baltimore, USA, May 1997.

[60] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Comput. Sci.*, to appear, 2000.

[61] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, May 1994.

[62] J. Y. Halpern and A. Ricciardi. A Knowledge-Theoretic Analysis of Uniform Distributed Coordination and Failure Detectors. In *18th ACM Symposium on the Principles of Distributed Computing (PODC99)*, pages 73–82, May 1999.

[63] M. Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, Computer Science, Jan. 8, 1998.

[64] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for ensemble layers. In *Proceedings of the FifthInternational Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes on Computer Science. Springer-Verlag, March 1999.

[65] J.-M. Hélary, J.-M. Hurfin, M. Mostéfaoui, M. Raynal, and F. Tronel. Computing global functions in asynchronous distributed systems with perfect failure detectors. *IEEE Transactions on Parallel and Distributed Systems*, 11(9), September 2000.

[66] M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4) :209–223, 1999.

[67] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe : A group membership service for wans. Technical Report MIT-LCS-TM-593a, Laboratory for Computer Science, Massachusetts Institute of Technology, 2000.

[68] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2 :95–114, 1978.

[69] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, July 1978.

[70] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2) :133–169, May 1998.

[71] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst.*, 4(3) :382–401, July 1982.

[72] M. Larrea, A. Fernández, and S. Arévalo. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In LNCS, editor, *DISC 1999*, volume 1693, pages 34–48, Bratislava, 1999. Springer.

[73] M. Larrea, A. Fernández, and S. Arévalo. Eventually consistent failure detectors. In *Brief Annoucement the 14th International Symposium on Distributed Algorithms(DISC00)*, Toledo, Oct. 2000.

[74] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th*

IEEE Symposium on Reliable Distributed Systems (SRDS2000), Nürnberg, Germany, Oct. 2000. IEEE Computer Society Press.

[75] G. Le Lann. Proof-based system engeneering and embedded systems. In Springer-Verlag, editor, *Proceedings of the European School on Embedded Systems, Nov. 96*, 1494, pages 208–248, October 1998.

[76] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[77] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[78] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul : a communication substrate for fault tolerant distributed programs. *Distributed Systems Engineering*, pages 87–103, 1993.

[79] A. Mostefaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors : A general quorum-based approach. In L. Springer, editor, *Proceedings of the 13th International Symposium on DIStributed Computing*, volume 1693, pages 49–63, 1999.

[80] A. Mostéfaoui and M. Raynal. Leader-based consensus. Research Report 1372, IRISA, Dec. 2000.

[81] A. Mostéfaoui, M. Raynal, and F. Tronel. The best of both worlds : A hybrid approach to solve consensus. In *Proceedings of the International Conference on Dependable Systems and Networks, DSN2000*. IEEE, 2000.

[82] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3) :374–419, Sept. 1990.

[83] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, Sept. 1999.

[84] F. Pedone and A. Schiper. Generic broadcast. Technical Report SSC/1999/012, École Polytechnique Fédérale de Lausanne, Switzerland, Apr. 1999.

[85] L. Rodrigues and P. Verissimo. xAMp, a protocol suite for group commuinication. Technical Report RT/43-92, INESC, 1992.

[86] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3) :149–157, 1997.

[87] F. B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys*, 22(4) :299–319, Dec. 1990.

[88] D. Skeen. unpublished communication. The Skeen's algorithm, Feb. 1985.

[89] S. Toueg, K. J. Perry, and T. K. Srikanth. Fast distributed agreement. *SIAM J. Comput.*, 16(3) :445–457, June 1987.

[90] R. van Renesse, K. Birman, and S. Maffeis. Horus : A flexible group communication system. *Commun. ACM*, 39(4), Apr. 96. NOM.

[91] R. van Renesse, K. P. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus system. In K. P. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[92] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group communication specifications : A comprehensive study. Technical Report MIT-LCS-TR-790, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1999.

[93] B. Whetten, T. Montgomery, and R. Kaplan, S. A hight performance totally ordered multicast protocol. In K. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems : International Workshop*, pages 33–57. Springer Verlag, 1995.