Sylvain MARCHAND

Le problème du consensus en environnement distribué asynchrone

Utilisation de détecteurs de défaillance

DEA Informatique

Université Bordeaux I

Juin 1996

Directeurs du mémoire

Robert CORI

Bernadette CHARRON-BOST

Résumé

Réaliser un consensus entre plusieurs processus est un problème extrêmement important en algorithmique distribuée, mais qui n'est malheureusement pas réalisable dans un modèle asynchrone à partir du moment où un des processus est susceptible de tomber en panne. Afin de contourner cette impossibilité, plusieurs voies ont été étudiées et deux grandes tendances se sont dégagées : affaiblir le problème ou enrichir le modèle avec des oracles, des détecteurs de défaillance non fiables. Le but de ce mémoire est d'explorer ces deux voies afin de les rapprocher sur le plan théorique et de les tester en pratique.

Remerciements

C'est très sincèrement que je tiens à remercier mon directeur de mémoire Monsieur Robert Cori ainsi que Madame Bernadette Charron-Bost pour toute l'attention qu'ils ont portée à mon travail ainsi que pour m'avoir guidé et fait bénéficier de leur expérience.
Je tiens à exprimer à Monsieur Yves Métivier toute ma reconnaissance pour l'intérêt qu'il a témoigné envers ce mémoire et pour ses nombreux et judicieux conseils.
Je remercie tout particulièrement Nathalie Geoffrin pour les intéressantes discussions que nous avons eues au sujet de l'implémentation, qui m'ont fait gagner un temps précieux.
Enfin, je ne voudrais surtout pas oublier les amis avec lesquels j'ai discuté, et qui ont d'une manière ou d'une autre contribué à l'aboutissement de ce mémoire.

Sommaire

1 Introduction **5**

2 Modèle de base **7**

2.1 Le réseau 7

2.1.1 Les connexions 7

2.1.2 Les messages 8

2.2 Les processus 10

2.2.1 Modélisation des processus 10

2.2.2 Modélisation des pannes 11

2.2.3 La diffusion fiable 12

2.2.4 Atomicité 13

3 Le problème du consensus **19**

3.1 Spécification 19

3.2 Uniformité 20

3.3 Algorithmes de consensus 22

3.3.1 Solution dans un cas simple 22

3.3.2 Impossibilité dans le modèle considéré 22

4 Consensus probabiliste **23**

4.1 La terminaison probabiliste 23

4.2 Spécification 26

4.3 Algorithme de Ben-Or 26

4.4 Algorithme de Bracha et Toueg 30

4.5 Résultat 34

5 Détecteurs de défaillance **35**

5.1 Définitions 35

5.2 Propriétés 36

5.2.1 Complétude 37

5.2.2 Précision 37

5.3 Réduction et équivalence 38

5.4 Classes de base 39

5.5 Un exemple de détecteur de défaillance 43

6 Consensus avec détecteurs de défaillance **47**

6.1 Principe 47

6.2 Algorithmes 47

6.2.1 Classes $\mathcal{S}$ et $\mathcal{P}$ 47

6.2.2 Classes $\diamond\mathcal{S}$ et $\diamond\mathcal{P}$ 51

6.3 Résultats 56

6.4 Cas du consensus probabiliste 57

7 Implémentation **59**

7.1 Choix généraux 59
7.2 Description 60
7.2.1 Mécanisme d'abonnement 60
7.2.2 Primitives de communication 61
7.2.3 Appel à l'oracle 61
7.3 Respect du modèle 62
7.4 Implémentation des détecteurs de défaillance 62
7.5 Tests et résultats 64
7.5.1 Algorithme de consensus 64
7.5.2 Algorithme de consensus probabiliste 64
7.5.3 Comparaison consensus / consensus probabiliste 65

8 Conclusion et perspectives **67**

Bibliographie **69**

Annexe A : Preuve d'impossibilité FLP **73**

Annexe B : Ben-Or $\rightarrow$ Bracha et Toueg **81**

Annexe C : Résultats des tests **93**

Annexe D : Sources de l'implémentation **99**

Chapitre 1

Introduction

La programmation d'applications réparties consiste à concevoir un ensemble d'entités indépendantes qui communiquent entre elles. On peut ainsi répartir les données et les traitements entre un ensemble de processeurs localisés sur des machines différentes. Mais la conception et la vérification d'applications distribuées sont en général des tâches complexes. De plus, certaines défaillances peuvent se produire en pratique, comme des pannes de processus, des ruptures de connexions, des messages perdus, dupliqués ou dont le contenu a été altéré, voire même des processus qui se comportent anormalement (fautes "byzantines"). Dans le cas d'une application centralisée, la moindre faute est en générale fatale. En revanche, on peut penser que le fait de disposer d'entités indépendantes qui collaborent peut permettre de concevoir des applications qui tolèrent ces fautes. Mais on se retrouve alors souvent en face de problèmes extrèmement difficiles, voire impossibles.

Durant ces dernières années, plusieurs paradigmes simplifiant la conception des applications distribuées ont été identifiés. Parmi eux on trouve le problème du consensus, dont le principe est simple : chaque processus propose une valeur, et tous les processus corrects doivent décider une valeur commune parmi toutes celles proposées. Le problème du consensus présente un grand intérêt à la fois pratique et théorique. En pratique, on doit en effet souvent décider de manière unanime. Par exemple, lorsqu'on veut mesurer un phénomène, on peut par soucis de sécurité utiliser plusieurs capteurs identiques, et il leur faut alors pouvoir s'accorder sur la valeur enregistrée. De même, lorsqu'on veut réaliser une modification dans une base de données distribuée, il faut que tous les participants décident de manière unanime si la transaction est validée ou non afin de maintenir la cohérence des informations. Sur le plan théorique, le consensus joue également un rôle fondamental car beaucoup d'autres problèmes s'y ramènent ou lui sont équivalents.

Dans un environnement asynchrone on ne fait d'hypothèses ni sur le temps de transmission des messages ni sur les vitesses des processeurs. Par conséquent, il est *a priori* très intéressant de concevoir des algorithmes dans un modèle asynchrone, ne dépendant pas de ces paramètres qui peuvent varier énormément d'un système à l'autre, ce qui leur confèrerait une portabilité maximale. De plus, en pratique, toute surcharge du système engendre tôt ou tard de l'asynchronisme. Malheureusement il est impossible de réaliser un consensus entre des processus dans un système asynchrone si on suppose qu'un de ces processus est susceptible de tomber en panne. Cette impossibilité constitue un obstacle majeur à l'utilisation du modèle asynchrone.

En fait, les contraintes du problème du consensus sont trop fortes pour que celui-ci puisse être réalisé dans le modèle asynchrone, et cela principalement à cause du fait que l'on ne dispose pas d'une détection précise des défaillances dans un tel modèle (il est par exemple impossible de décider si un processus est définitivement en panne ou si il est seulement dans une phase d'exécution très lente). Afin de contourner cet obstacle, plusieurs possibilités ont été étudiées et deux grandes tendances se sont dégagées : affaiblir le problème ou enrichir le modèle. La première voie de recherche est la plus ancienne. L'idée est d'affaiblir le problème du consensus, en définissant notamment le problème du consensus probabiliste. L'idée de l'enrichissement du modèle avec des détecteurs de défaillance est plus récente. Elle est due à Sam Toueg et son équipe de l'Université de Cornell.

Le but de ce mémoire est d'étudier ces deux approches, afin de les comparer et de mettre en évidence d'éventuelles relations. Il convient donc d'étudier et d'approfondir les résultats connus. Il est également intéressant de les tester en pratique, en programmant les algorithmes qui ont été proposés.

La suite de ce document est organisée de la manière suivante. Le modèle de base est présenté dans le chapitre 2, et le problème du consensus est défini dans le chapitre 3. Ensuite la première approche pour contourner l'impossibilité du consensus est abordée dans le chapitre 4. Le problème du consensus probabiliste y est défini et des algorithmes pour le résoudre y sont proposés. La seconde approche, plus récente, est basée sur des détecteurs de défaillance qui sont exposés dans le chapitre 5. Puis des algorithmes résolvant le consensus à l'aide de ces détecteurs sont proposés dans le chapitre 6. On verra également à la fin de ce chapitre que cette approche peut englober la première. Après ces résultats théoriques on trouve dans le chapitre 7 la description d'une réalisation pratique ainsi que les résultats qui ont été obtenus en l'utilisant. En annexe, on trouvera également une reformulation de la preuve d'impossibilité du consensus [FLP85], des algorithmes réalisant le consensus probabiliste, les résultats des tests effectués en pratique ainsi qu'une petite partie des sources de l'implémentation en C++.

L'étude s'appuie principalement sur les articles suivants : [FLP85], [CT94], [BO83] et [BT85].

Chapitre 2

Modèle de base

On considère un ensemble de n processus ($n > 1$) communiquant entre eux par envoi et réception de messages (les notions de processus et de message seront précisées plus loin).

Soit $\Pi = \{p_1, \cdots, p_n\}$ l'ensemble des n processus, et soit M l'ensemble des messages en transit (c'est-à-dire envoyés mais pas encore reçus).

On considère le *système* constitué par l'ensemble Π des processus et l'ensemble M des messages en transit. Un *état du système* (ou *état global*) est constitué du contenu de M et de l'état interne de chacun des processus de Π. Un *état initial du système* est un état du système où chaque processus est lui-même dans un état initial et où M est vide.

On s'intéresse ici au cas d'un système *asynchrone*, c'est-à-dire que les processus n'ont pas accès à une horloge globale et n'ont aucune contrainte temporelle, ce qui peut se résumer ainsi : on ne fait aucune hypothèse sur les délais d'acheminement des messages et sur la vitesse relative des processus.

2.1 Le réseau

2.1.1 Les connexions

Pour tout couple de processus (p_i, p_j) il existe un canal de communication acheminant des messages de p_i vers p_j.

Les connexions sont fiables, c'est-à-dire que :

- tout message envoyé à un processus arrive à destination au bout d'un temps fini,
- tout message reçu par un processus a été envoyé précédemment :

il n'est pas apparu spontanément et n'est pas un "écho" d'un autre message.

Remarque
On peut considérer que chaque *paire* de processus est reliée par un canal de communication *bidirectionnel* ou, de manière équivalente, que chaque *couple* de processus est relié par un canal *unidirectionnel*. Le graphe modélisant les connexions entre les processus est alors complet. Pour un ensemble de n processus, on considère donc le graphe orienté complet à n sommets K_n.

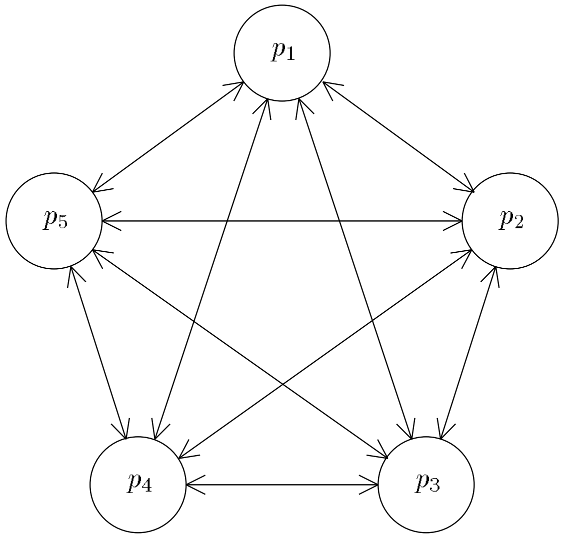

Figure 1: *Graphe K_5 pour 5 processus*

En fait, la condition "*chaque paire de processus est reliée par un canal de communication fiable*" ne nécessite pas que le graphe des connexions soit réellement complet. Bien que ce soit l'hypothèse avancée dans la plupart des articles pour l'étude théorique, cette condition est suffisante mais n'est pas nécessaire en pratique. On peut en effet utiliser des protocoles de communication en mode connecté comme TCP, qui assurent des connexions point-à-point fiables tout en faisant abstraction de la topologie du réseau. Dans la partie suivante, concernant les messages, on verra que l'on peut s'affranchir de telles considérations, même sur le plan théorique, en considérant une approche à un plus haut niveau.

2.1.2 Les messages

Un *message* est formé d'un couple (q, m) où :

- m est le contenu du message (de n'importe quel type),
- q désigne le processus destinataire du message (récepteur).

Par abus de langage, on appelera parfois "message" le contenu m d'un message (q, m), le processus destinataire étant alors implicite.

Remarque
Il est parfois nécessaire de pouvoir retrouver l'émetteur d'un message lors de la réception. Cela peut se faire très facilement avec le modèle précédent. En effet, il suffit que chaque processus émetteur p envoie $m' = (p, m)$ à la place de m. Alors un processus récepteur qui recevra le message m' n'aura qu'à extraire les champs nécessaires. En revanche, ce n'est qu'une fois qu'il a pris connaissance du message qu'il peut connaître le nom de l'émetteur, le modèle ne permettant pas à un processus de choisir la provenance des messages lors de la demande de réception. Mais un processus peut émuler ce comportement s'il le désire, avec un algorithme très simple (il lui suffit en effet de mémoriser les messages avec une provenance non souhaitée, pour les traiter plus tard, et de traiter les autres directement).

L'*ensemble des messages en transit* (c'est-à-dire émis mais pas encore reçus) est noté M. En fait, M est un multi-ensemble (non ordonné), puisque :

- il peut y avoir plusieurs messages identiques (si un même message a été envoyé plusieurs fois),
- les messages sont reçus dans un ordre quelconque qui n'a pas *a priori* de rapport avec celui des émissions.

Pour chaque processus p, on dispose de deux primitives de communication :

- **envoyer**(q,m)
  le message (q, m) est placé dans l'ensemble M des messages en transit,
- **recevoir**$()$
  deux situations peuvent alors se produire :
  1. p obtient en retour le "message nul" $\emptyset$, et M demeure inchangé,
  2. p obtient la valeur m, et un couple (p, m) est retiré de M.

  La première situation se produit évidemment s'il n'y a pas de message (p, m) dans M, mais ce n'est pas le seul cas. En fait, la seule condition est que si le processus p fait une infinité de demandes **recevoir**$()$, il reçoit tout message qui lui a été envoyé. M a donc un comportement *non déterministe.*

Le système considéré étant asynchrone, un message peut être retardé pendant un intervalle de temps fini. Ceci est modélisé par le comportement particulier de la primitive **recevoir**$()$.

Ces primitives permettent d'échanger des messages en faisant abstraction du réseau d'interconnexion entre les processus.

Remarque importante

Les communications sont point-à-point. On ne dispose pas d'une primitive "atomique" pour diffuser un message à tous les processus en même temps (*broadcast*). Dans la suite, "**envoyer à tous** m" sera équivalent à :

pour chaque processus q
  envoyer(q,m)

On ne sait rien *a priori* de l'ordre dans lequel les messages seront envoyés aux n processus. Il est d'ailleurs impossible de prévoir dans quel ordre ils seront reçus à cause du comportement non déterministe de l'ensemble des messages M. De plus, si "**envoyer à tous** m" est interrompu en cours d'exécution, seuls certains des n messages seront effectivement envoyés, et les autres ne le seront jamais.

2.2 Les processus

2.2.1 Modélisation des processus

Les processus sont modélisés par des automates *déterministes* qui communiquent entre eux au moyen de messages.

Un *processus* est une entité composée des éléments suivants :

- un programme,
- des données, l'espace de stockage disponible étant supposé non borné.

Parmi les données contenues dans l'espace de stockage interne d'un processus on distinguera :

- un registre d'entrée x, prenant ses valeurs dans $\{0, 1\}$,
- un registre de sortie y, prenant ses valeurs dans $\{\perp, 0, 1\}$, initialisé à $\perp$ et dont l'accès est verrouillé en écriture dès qu'une valeur (0 ou 1) y est écrite.

L'*état interne* d'un processus est constitué par :

- la position atteinte dans son programme (en pratique le compteur de programme),
- l'image de son espace de stockage interne (en pratique la pile, le tas, les registres, ...).

L'*état initial* d'un processus est caractérisé par :

- la valeur initiale de son registre d'entrée x qui vaut 0 ou 1,
- la valeur de son registre de sortie y à $\perp$,
- le fait qu'on se trouve au début du programme,
- l'espace de stockage interne qui a un contenu prédéfini (contenu initial).

L'exécution d'un processus se déroule par *pas atomique*. A chaque pas, un processus effectue en séquence les actions suivantes :

1. recevoir un message,
2. effectuer une action interne (qui équivaut à une transition de son automate),
3. éventuellement envoyer un message.

2.2.2 Modélisation des pannes

Les processus ne sont pas fiables : ils peuvent à tout moment tomber en *panne franche*, c'est-à-dire suspendre leur exécution de manière définitive. Les pannes franches seront les seules fautes considérées par la suite.

Un processus est dit *correct* si il effectue un nombre infini de pas au cours de son exécution, dans le cas contraire il est dit *défectueux* . Un processus est donc défectueux si il existe une date à partir de laquelle il ne fait plus aucun pas. A partir de cette date, le processus est définitivement “mort” et ne peut plus “ressusciter”. Les autres processus peuvent envoyer des messages à p, qui vont s'accumuler dans l'ensemble M des messages en transit puisque p, qui ne fait plus aucun pas, ne les recevra jamais. En revanche, il est clair que p n'envoie plus de message.

On note f un majorant du nombre de processus défectueux dans le système.[1]

Remarque
Il est important de bien distinguer par la suite les termes “défectueux” et “défaillant”. On dira qu'un processus est *défaillant* s'il ne fait plus aucun pas, et qu'il est défectueux s'il devient défaillant au cours de son exécution.

En pratique, il est impossible de décider si un processus est correct ou défectueux, puisqu'on retombe sur le problème d'arrêt, indécidable.

[1] Ce majorant est souvent noté t dans les articles, mais dans la suite t sera utilisé pour désigner le temps.

2.2.3 La diffusion fiable

La *diffusion fiable* (*Reliable Broadcast*) est une primitive de communication définie dans [CT94][2] et qui sera utilisée dans les algorithmes par la suite. Il garantit que :

1. tous les processus corrects prennent connaissance du même ensemble de messages (émis par des processus corrects ou défectueux),
2. tout message diffusé par un processus correct est reçu par tous les processus corrects,
3. si un processus prend connaissance d'un message, c'est que ce dernier a été précédemment diffusé (il n'est pas apparu spontanément et n'est pas un "écho" d'un autre message).

L'algorithme réalisant la diffusion fiable est simple :

```
R-broadcast(m):
  {
    envoyer à tous m
  }
```

et un processus p prend connaissance d'un message diffusé ainsi :

```
quand m est reçu pour la première fois
  {
    si m n'a pas été émis par le processus p
      envoyer à tous m
    R-deliver(m)
  }
```

C'est grâce à **R-deliver**(), qui est une "fonction réflexe" (*callback function*), que le processus prend connaissance d'un message diffusé par **R-broadcast**().

Remarque
Pour simplifier, on suppose en plus que chaque message diffusé est unique.

La diffusion fiable possède les propriétés suivantes :

- **Validité** :
  si un processus correct diffuse m, alors tous les processus corrects prennent tôt ou tard connaissance de m,
- **Accord**[3] :
  si un processus correct prend connaissance de m, alors tous les processus corrects pren-

[2] C'est en fait une version simplifiée (limitée aux pannes franches) d'une primitive de diffusion asynchrone définie dans [BT85] pour les fautes "byzantines".

[3] En fait, on a même ici une propriété un peu plus forte : si un processus (correct ou non) prend connaissance de m, alors tous les processus corrects prennent tôt ou tard connaissance de m.

nent tôt ou tard connaissance de m,

- **Intégrité uniforme** :
  pour tout message m, chaque processus prend connaissance de m au plus une fois, et seulement si m a été diffusé par un des processus.

Il est facile de voir que l'algorithme précédent satisfait les propriétés de validité, d'accord et d'intégrité uniforme dans tout système asynchrone pouvant comporter jusqu'à $n-1$ processus défectueux. La preuve est triviale et par conséquent omise (elle découle de la définition même de l'algorithme).

On a vu précédemment que les processus ne pouvaient pas envoyer plus d'un message lors d'un pas *atomique*. Cette modélisation est bien adaptée aux systèmes pratiques, qui disposent rarement d'une primitive de diffusion. On peut cependant se servir de la diffusion fiable pour implanter un modèle *a priori* plus puissant, qui permet à un processus d'envoyer un nombre fini de messages lors d'un pas "atomique".

2.2.4 Atomicité

Lorsqu'on définit un algorithme, il convient de bien préciser quels sont les pas atomiques réalisés par les processus. Or la définition d'un pas atomique diffère suivant les articles. De plus, les pas proposés ne sont en général pas directement réalisables en pratique sur un système réel. Puisque les processus peuvent être défectueux (et par conséquent suspendre définitivement leur exécution entre deux pas), il est donc très important de bien préciser l'atomicité des pas.

2.2.4.1 Modèle de base

Dans le modèle de base exposé précédemment, on a vu que lors de chaque pas atomique un processus peut effectuer la séquence indivisible constituée des étapes suivantes :

1. recevoir 1 message en utilisant la primitive **recevoir**(),
2. effectuer une action interne,
3. éventuellement envoyer *un* message en utilisant la primitive **envoyer**().

C'est ce modèle, que l'on peut retrouver notamment dans [CT91] et [CT94], qui sera utilisé pour tous les algorithmes présentés dans les chapitres suivants. Cependant, le grain d'atomicité de ce modèle est *a priori* trop grossier pour l'implémentation des algorithmes sur des systèmes réels.

2.2.4.2 Modèle “d'implémentation”

En pratique, dans la plupart des systèmes réels, on peut atomiquement effectuer chacune des trois étapes d'un pas du modèle de base, mais pas le pas en entier. Il faudrait donc qu'un pas atomique soit, au choix : recevoir un message, effectuer une action interne, envoyer un message. On obtient le *modèle d'implémentation* en considérant qu'un pas atomique est l'une de ces trois actions.

En fait, on peut montrer que ces deux modèles sont “équivalents” en ce sens qu'en cas de panne franche d'un processus le comportement du système reste le même.

En effet, on voit facilement que tout algorithme conçu pour le modèle d'implémentation peut être réutilisé sans modification dans le modèle de base, plus puissant. Mais c'est l'inverse qui est intéressant en pratique : étant donné un algorithme conçu pour le modèle de base, peut-on l'utiliser dans le modèle d'implémentation ? C'est en fait possible sans même modifier l'algorithme.

Décomposons un pas du modèle de base dans le modèle d'implémentation :

(**1**)
- recevoir 1 message en utilisant la primitive **recevoir**()

(**2**)
- effectuer des calculs internes

(**3**)
- éventuellement envoyer 1 message en utilisant la primitive **envoyer**()

(fin du pas = début (**1**) du pas suivant)

Les processus ne peuvent communiquer entre eux que par envoi et réception de messages. Examinons les différents cas qui peuvent se produire lorsqu'une panne franche survient chez un processus p lors d'un pas du modèle de base exécuté dans le modèle d'implémentation. Si cette panne survient :

- en **1** :
  la panne survient donc entre deux pas du modèle de base (comme elle aurait très bien pu le faire dans ce même modèle),
- dans $]\mathbf{1}, \mathbf{2}[$:
  c'est impossible puisque la primitive **recevoir**() est supposée atomique également dans le modèle d'implémentation,
- dans $[\mathbf{2}, \mathbf{3}]$:
  c'est exactement comme si la panne franche était survenue en **1**, mis à part le fait qu'il y a peut-être un message m en moins pour p dans l'ensemble M des messages en transit. Mais cela ne change en rien l'évolution future du système, puisque le seul processus qui aurait pu être géné par l'absence de m est le processus p, qui est maintenant défaillant,
- après **3** :

si le pas ne devait pas faire appel à la primitive **envoyer**() on est dans le même cas que précédemment, et sinon c'est impossible puisque cette primitive est supposée atomique également dans le modèle d'implémentation.

Dans tous les cas, le système continue à évoluer de la même manière, et cela quel que soit le modèle considéré.

Remarque
Le modèle d'implémentation sera celui utilisé pour la réalisation pratique des algorithmes présentés dans les chapitres suivants. Les pas atomiques dans ce modèle étant très simples, cela évite d'avoir à préciser systématiquement les pas dans les algorithmes présentés.

2.2.4.3 Modèle FLP

Le modèle de base n'est pas le modèle le plus fréquemment rencontré dans les articles. On trouve en effet plus souvent le *modèle FLP*, que l'on retrouve notamment dans la preuve d'impossibilité [FLP85]. Dans ce modèle, un pas atomique pour un processus est constitué de la séquence indivisible suivante :

1. recevoir un message,
2. effectuer une action interne,
3. envoyer un nombre fini ($k \geq 0$) de messages.

Or, dans le modèle de base, on n'a pas *a priori* la possibilité de diffusion "atomique" de la troisième étape.

Remarquons d'abord que tout algorithme conçu pour le premier modèle peut être clairement réutilisé sans modification dans le modèle FLP, plus puissant. En revanche, l'inverse n'est pas vrai. Pour voir cela, il suffit de considérer un système de trois processus : p_1 qui diffuse deux messages vers p_2 et p_3, ces derniers attendant un message avant de poursuivre. Il est clair que si p_3 défaille entre les deux envois, un de ces deux processus attendra indéfiniment et ne poursuivra jamais son traitement.

On pourrait donc croire dans un premier temps que le second modèle est strictement plus puissant que le premier. En fait, les deux modèles sont "équivalents", en ce sens qu'on peut implanter le modèle FLP à l'aide du premier, en utilisant la diffusion fiable vue précédemment.

Remarque
Il est clair que, lorsqu'on implémente le modèle FLP, les pas ne sont plus vraiment atomiques. Mais on obtient bien un modèle "équivalent" en ce sens que le comportement du système reste le même, et cela même en cas de panne franche.

L'idée est la suivante :

On veut diffuser “simultanément” un ensemble fini de k messages, chaque message m_i étant destiné à un processus p_i. On implante pour cela deux nouvelles primitives **envoyer+**() et **recevoir+**() (qui remplacent respectivement **envoyer**() et **recevoir**()), et on fait alors transiter 2 types de messages :

1. les messages “normaux” : $m' = (normal, m)$
   m est alors retourné par la primitive **recevoir+**() comme il l'aurait été précédemment par la primitive **recevoir**(),
2. les messages “diffusés” : $m' = (diffusion, (d_1, s_1, h_1, m_1), \cdots, (d_k, s_k, h_k, m_k))$
   les m_i ne sont pas retournés par la primitive **recevoir+**(), mais le processus p prend connaissance de tout message m_i qui lui est destiné (i.e. $d_i = p$) lors d'un appel à **R-deliver**(m_i).

Dans le second cas, les s_i et h_i permettent d'assurer l'unicité des messages diffusés par **R-broadcast** (unicité qui avait été supposée lors de la définition des primitives de diffusion fiable). En effet, si deux processus différents s et s' diffusent un même message m on aura bien $(s, \cdots, m) \neq (s', \cdots, m)$, et si un même processus s diffuse le même message m à deux reprises son horloge h_s sera incrémentée entre les deux diffusions et on aura à nouveau $(s, h_s, m) \neq (s, h'_s, m)$. L'ensemble $E_{broadcast}$ sert à mémoriser les messages, afin de savoir si un certain message a déjà été reçu auparavant.

Des algorithmes pour implémenter les primitives **envoyer+**() et **recevoir+**() sont proposés dans ce qui suit (leur validité est une conséquence des propriétés de la diffusion fiable et de la remarque précédente, en notant bien qu'un processus correct effectue une infinité de pas, et donc une infinité de demandes de réception).

On constate cependant que le nombre de messages échangés lors d'une diffusion est grand (jusqu'à n^2). Or les algorithmes présentés dans les chapitres suivants ne nécessiteront au pire que la diffusion d'un *même* message m ($\forall i,\ m_i = m$). Par conséquent, l'implémentation exposée ici ne sera réalisée que partiellement, suivant les besoins de chaque algorithme.

envoyer+$(d_1,m_1,\cdots,d_k,m_k)$:

```
{
  si k = 1 /* un seul message */
    {
      m' ← (normal, m1)
      envoyer(d1,m')
    }
  sinon /* diffusion */
    {
      incrémenter h
      m' ← (diffusion, (d1, p, h, m1), ···, (dk, p, h, mk))
      envoyer à tous m'
    }
}
```

recevoir+():

```
{
  m' ← recevoir()
  si m' est de type normal
    {
      extraire m de m'
      retourner m
    }
  sinon /* message de type diffusion */
    {
      pour tout (di,si,hi,mi) dans m'
        si di = p et (di, si, hi, mi) ∉ E_broadcast
          {
            E_broadcast ← E_broadcast ∪ {(di, si, hi, mi)}
            si si ≠ p
              envoyer à tous m'
            R-deliver(mi)
          }
      retourner ∅ /* pas de message */
    }
}
```

Chapitre 3

Le problème du consensus

Le problème du consensus s'énonce simplement : chaque processus correct propose une valeur, et tous ces processus corrects doivent décider de manière irrévocable une valeur commune parmi toutes celles proposées.

3.1 Spécification

Plus formellement, on peut définir le problème du consensus en terme de deux primitives :

- **propose**(v)
- **décide**(v)

où v est une valeur quelconque dans l'ensemble $\{0, 1\}$ (consensus binaire).

Dans le modèle présenté au chapitre 2, **propose**(v) équivaut à déterminer la valeur initiale du registre d'entrée x (initialement, on a donc $x = v$) et **décide**(v) à fixer la valeur du registre de sortie y (en faisant $y \leftarrow v$).

Quand un processus exécute **propose**(v) (resp. **décide**(v)), on dit qu'il propose (resp. décide) la valeur v.

La spécification du *problème du consensus* s'exprime selon quatre propriétés :

- **Terminaison** :
  chaque processus correct doit finalement décider une valeur,
- **Intégrité uniforme** :
  chaque processus décide au plus une valeur,

- **Accord** :
  il n'existe pas deux processus corrects qui décident des valeurs différentes,

- **Validité** :
  si un processus décide une valeur v, c'est que v a été proposée par un des processus.

Remarque
Dans ce qui précède, rien n'empêche un processus d'effectuer plusieurs appels à **décide**(v), du moment que la valeur v soit toujours la même (sinon la propriété d'intégrité n'est pas vérifiée). C'est le cas par exemple de l'algorithme de Ben-Or présenté au chapitre 4. Or, dans le modèle de base défini au chapitre 2, un processus ne peut affecter qu'une seule fois une valeur à son registre de sortie y. On peut supprimer cet inconvénient en définissant une nouvelle primitive **décide+**(v) ainsi :

```
décide+(v):
  {
    si valeur = ⊥
      {
        valeur ← v
        décide(v) /* y ← v */
      }
    sinon si valeur ≠ v
      {
        ERREUR FATALE /* violation du modèle */
      }
    /* si valeur = v ne rien faire */
  }
```

où *valeur* est une variable qui ne fait que refléter l'état de y, et qui contient donc initialement la valeur $\perp$ (i.e. indéfinie).

3.2 Uniformité

Dans les propriétés énoncées précédemment, le terme *uniforme* indique que tous les processus sont concernés, et non pas seulement les processus corrects. On peut donc se demander ce qu'il advient du problème du consensus si on modifie l'uniformité de ces propriétés.

Puisque les seules fautes considérées ici sont les pannes franches, on peut tout d'abord remarquer que la terminaison *uniforme* n'a pas de sens (puisqu'un processus défectueux peut à tout moment tomber en panne franche, et ne jamais décider) et que l'intégrité uniforme est équivalente à l'intégrité (*non uniforme*) (puisque si un processus défectueux décide plus d'une valeur, c'est une faute plus grave que la panne franche, et on sort du modèle considéré ici).

La validité relie la valeur décidée aux valeurs proposées. Elle peut être modifiée, mais on obtient alors d'autres formes du problème du consensus. En particulier, il ne faut pas la confondre avec une validité plus complexe qui exigerait que v ait été proposée par un processus *correct*.

En revanche, on aurait pu remplacer l'accord par l'accord *uniforme* : il n'existe pas deux processus qui décident différemment (sans préciser si ces processus sont corrects ou non). Ce problème a été soulevé par Toueg et Neiger. On aurait alors obtenu le consensus *uniforme* au lieu du consensus, or, les seules pannes envisagées ici étant les pannes franches, on peut montrer ([Tou]) que sous réserve qu'on a une majorité de processus corrects les problèmes du consensus et du consensus uniforme sont équivalents.

Preuve

Soit A_u un algorithme résolvant le consensus uniforme. Il est clair que cet algorithme résout également le consensus (les processus corrects étant des processus...).

Réciproquement, soit A un algorithme résolvant le consensus. Supposons qu'on a une majorité de processus corrects. On montre alors qu'il est possible de transformer A en un algorithme A_u résolvant le consensus uniforme (par une réduction d'algorithme).

En utilisant la diffusion fiable vue en 2.2.3, l'algorithme annoncé A_u peut être obtenu à partir de A ainsi :

- on rajoute un tableau d'entiers $compteur[]$
  (initialement $compteur[0] = compteur[1] = 0$),
- on remplace tout appel à **décide**(v) par un appel à **R-broadcast**(v),
- et on a aussi :

  quand R-deliver(v)
  {
  incrémenter $compteur[v]$
  si $compteur[v] > \lceil \frac{n}{2} \rceil$
  décide(v)
  }

En effet, comme on a au moins une majorité absolue de processus corrects, et que ces derniers diffuseront la même valeur, cette valeur sera décidée par tous les processus encore en vie, et cela en dépit des autres valeurs "parasites" provenant des processus défaillants. De plus, les processus défaillants, définitivement morts, ne décideront jamais. Par conséquent, il n'existe pas deux processus qui décident des valeurs différentes.

Remarque
Les algorithmes proposés par la suite pour la résolution du consensus seront en fait des algorithmes réalisant le consensus uniforme...

3.3 Algorithmes de consensus

Si l'énoncé du problème du consensus est simple, sa résolution est, elle, bien plus difficile...

3.3.1 Solution dans un cas simple

Dans le cas idéal d'un réseau et de processus fiables, le problème se résout simplement.

En effet, dans ce cas simple, l'algorithme suivant résout de manière évidente le problème du consensus :

- *initialisation*
  l'espace de stockage de chaque processus comporte un vecteur de taille n initialisé à $< \perp, \cdots, \perp >$
- *phase 1*
  chaque processus propose une valeur v en envoyant le message (q, v) pour tout destinataire $q \in \{1, \cdots, n\}$
- *phase 2*
  chaque processus attend les propositions des n processus, en plaçant la valeur v dans son vecteur à l'indice q lors de la réception d'un message v provenant de q $(q \in \{1, \cdots, n\})$
- *phase 3*
  chaque processus décide la plus petite valeur qui se trouve dans son vecteur une fois que toutes les valeurs contenues sont différentes de $\perp$

3.3.2 Impossibilité dans le modèle considéré

Le cas idéal d'un réseau et de processus parfaitement fiables n'est pas réalisable en pratique. Le modèle défini au chapitre 2 est plus réaliste. En effet, même si le réseau est toujours supposé parfaitement fiable, les processus, eux, peuvent être défectueux. Malheureusement, on peut montrer qu'à partir du moment où un processus est susceptible d'être défectueux, le problème du consensus n'a pas de solution algorithmique déterministe en environnement asynchrone (voir l'annexe A). Cela est dû au fait que l'on peut toujours faire évoluer le système de manière à ce qu'aucun des processus ne décide jamais, et cela quels que soient les algorithmes considérés.

Par conséquent, il est inutile de chercher des algorithmes pour résoudre le problème du consensus dans le modèle étudié, vu qu'il n'en existe pas...

Il faut donc soit considérer une variante du problème plus simple, soit modifier le modèle.

Chapitre 4

Consensus probabiliste

Le problème du consensus ne peut pas être résolu à l'aide d'algorithmes déterministes dans un système asynchrone à partir du moment où un processus est susceptible d'être défectueux. En fait, on peut réaliser sans difficulté les propriétés d'intégrité, d'accord et de validité énoncées au chapitre 3, mais la propriété de terminaison est alors irréalisable.

Par conséquent, on peut affaiblir cette propriété de terminaison et alors s'intéresser à un problème plus simple : le consensus probabiliste.

Cette approche se retrouve principalement dans les trois articles [BO83], [BT85] et [Rab83]. Les deux premiers sont exposés rapidement dans ce chapitre. Le troisième [Rab83] traite le cas plus complexe des fautes "byzantines".

4.1 La terminaison probabiliste

Sous certaines hypothèses, il se peut qu'un problème donné n'admette pas de solution car la propriété de terminaison n'est pas réalisable. C'est le cas pour le problème du consensus dans le modèle de base. On peut alors faire appel à une propriété plus faible, appelée terminaison probabiliste.

Dans le cas du problème du consensus, on a vu que la propriété de terminaison pouvait s'énoncer ainsi : tout processus correct décide une valeur au bout d'un temps fini.

On définit alors la *terminaison probabiliste* ainsi : tout processus correct décide une valeur au bout d'un temps fini avec probabilité 1 ; ainsi la probabilité pour qu'un processus correct ne décide pas au bout d'un temps fini est nulle.

Dans ce qui suit, on dira qu'un processus a terminé lorsqu'il a décidé. Le problème du consensus est alors terminé lorsque tous les processus corrects ont eux-mêmes terminé.

On se place dans l'univers de toutes les exécutions possibles pour un certain algorithme.

Soit $S(t)$ la probabilité de terminer avant la date t.
Alors $S = \lim_{t \to +\infty} S(t)$ est la probabilité de terminer (au bout d'un temps fini).

La terminaison probabiliste s'exprime par $S = 1$.

De nombreux algorithmes consistent en une boucle principale, chaque tour de boucle étant appelé un *"round"* . A chaque *round* r, on peut terminer (la plupart du temps en sortant de la boucle) si une condition $A(r)$ est vérifiée. Dans la suite, $A(r)$ sera appelée la *condition d'arrêt* (puisqu'elle conduit en général à l'arrêt de l'algorithme). De tels algorithmes sont donc de la forme :

...

$r \leftarrow 1$

répéter `/* boucle */`
{
...
si $A(r)$ **alors** $\cdots$ `/* terminé */`
...
incrémenter r
}

...

Pour $r \geq 1$, on définit alors :

- $a(r)$ la probabilité pour que la condition d'arrêt $A(r)$ soit vraie
(i.e. la probabilité pour que, une fois le *round* r atteint, on y termine),
- $s(r)$ la probabilité de terminer exactement au *round* r,
- $S(r)$ la probabilité de terminer au plus tard au *round* r,
- S la probabilité de terminer au bout d'un temps fini.

En remarquant que pour terminer lors du *round* $k+1$ il faut ne pas avoir déjà terminé lors des *rounds* antérieurs et vérifier $A(k+1)$, on trouve :

$$s(1) = a(1) \quad \text{et} \quad \forall k \geq 1,\ s(k+1) = (1 - S(k)) \cdot a(k+1)$$

et :

$$s(r) = (\prod_{l=1}^{r-1} (1 - a(l))) \cdot a(r)$$

De plus, puisque pour terminer au plus tard au *round* $k+1$ il faut soit terminer au plus tard au *round* k, soit terminer exactement au *round* $k+1$, on a $S(1) = s(1)$ et $\forall k \geq 1$, $S(k+1) = S(k) + s(k+1)$, soit :

$$S(r) = \sum_{k=1}^{r} s(k)$$

Et comme S est la probabilité de terminer au bout d'un temps fini, c'est-à-dire lors d'un certain *round* r, on a évidemment :

$$S = \lim_{r \to +\infty} S(r)$$

De plus,

si

$$\exists \alpha, \beta : \forall r \geq 1,\ 0 < \alpha \leq a(r) \leq \beta < 1$$

alors

$$\forall l \geq 1,\ 0 < 1 - \beta \leq 1 - a(l) \leq 1 - \alpha < 1$$

$$\forall r \geq 1,\ 0 < (1-\beta)^{r-1} \leq \prod_{l=1}^{r-1} (1 - a(l)) \leq (1-\alpha)^{r-1} \leq 1$$

et donc

$$1 - (1-\alpha)^{r-1} \leq 1 - \prod_{l=1}^{r-1} (1 - a(l)) < 1$$

et comme

$$S(r) = 1 - \prod_{l=1}^{r-1} (1 - a(l))$$

on a donc

$$1 - (1-\alpha)^{r-1} < S(r) < 1$$

et comme $0 < \alpha < 1$, on a $|1 - \alpha| < 1$ et en passant à la limite :

$$S = 1$$

On a donc le résultat suivant :

Théorème 4.1.1
Si il existe une constante α strictement positive telle que, pour tout round r, la probabilité $a(r)$ de terminer à ce round lorsqu'on l'exécute est minorée par α, alors la propriété de terminaison probabiliste est vérifiée (i.e. $S = 1$).

Remarque

$$Terminaison \Rightarrow Terminaison\ probabiliste$$

Cette implication est triviale (elle découle des définitions), et l'implication réciproque est clairement fausse. En effet, considérons par exemple l'algorithme suivant avec un pièce de monnaie non truquée :

> $r \leftarrow 1$
> **répéter**
> {
> lancer la pièce
> **incrémenter** r
> }
> **tant que** la pièce tombe sur pile

Pour chaque *round* $r \geq 1$, la probabilité de terminer alors qu'on est dans ce *round* est $a(r) = \frac{1}{2}$. Par conséquent, la probabilité de terminer au plus tard au *round* r est $S(r) = 1 - (1 - \frac{1}{2})^{r-1}$ et on a bien $S = \lim_{r \to +\infty} S(r) = 1$ comme précédemment. Cet algorithme se termine donc au bout d'un temps fini avec une probabilité de 1, et poutant il ne termine pas dans un cas (de probabilité certes nulle, mais cependant possible) : si on tombe infiniment souvent sur face !

4.2 Spécification

La spécification du problème du *consensus probabiliste* est obtenue à partir de celle du consensus (voir chapitre 3) en remplaçant la terminaison par la terminaison probabiliste (plus faible) présentée précédemment.

4.3 Algorithme de Ben-Or

L'algorithme de Ben-Or présenté dans ce qui suit résout le consensus probabiliste dans les systèmes asynchrones avec une majorité stricte de processus corrects.

Comme on note f un majorant du nombre de processus défectueux dans le système, on considère donc que $f < \lceil \frac{n}{2} \rceil$.

On ne s'intéresse ici qu'au cas des pannes franches, mais l'algorithme de Ben-Or se généralise dans le cas de pannes quelconques ("byzantines"), à condition cependant d'inclure le numéro du processus émetteur dans chaque message (pour pouvoir le retrouver à la réception) et d'attendre plus de messages (et vérifier leur provenance) avant de proposer ou de décider une valeur. Bien que l'algorithme reste assez simple, la preuve en est plus compliquée...

Le principe de l'algorithme est le suivant :
chaque processus dispose initialement d'une valeur x_0 qu'il voudrait voir adoptée par les autres processus. Cette valeur est initialement placée dans sa variable d'entrée x. Tous les processus répètent alors séquentiellement 3 phases :

1. **rapporter** aux autres la valeur x que l'on veut proposer,

2. **proposer** aux autres une valeur en fonction des rapports reçus,
3. **ajuster** sa variable x si il est impossible de décider une valeur en fonction des propositions reçues, quitte à faire appel au hasard en cas d'indétermination totale.

Les processus échangent des messages sous forme de triplets (tag, r, v) où :

- tag indique la nature du message : `R` (*Rapport*) ou `P` (*Proposition*),
- r est le numéro du *round* lors duquel il a été envoyé,
- v est une valeur (**0** ou **1**, ainsi que **?** pour les propositions).

Remarque
Dans chacune des deux premières phases, un processus p attend des messages en provenance de $n - f$ processus. Dans le modèle asynchrone, p ne peut pas en attendre un de plus sans risquer de se bloquer (en attendant en vain, indéfiniment...).

Chaque processus p effectue l'algorithme suivant :

$x \leftarrow x_0$ `/* valeur initialement proposée */`

$r \leftarrow 1$ `/* numéro du` *round* `*/`

répéter
{
`/* Phase 1: Rapports */`

envoyer à tous $(\mathtt{R}, r, x)$
attendre $(n - f)$ messages de la forme $(\mathtt{R}, r, \cdots)$
si on a reçu plus de $(n/2)$ messages $(\mathtt{R}, r, v)$ pour un même v
$proposition \leftarrow v$
sinon
$proposition \leftarrow ?$

`/* Phase 2: Propositions */`

envoyer à tous $(\mathtt{P}, r, proposition)$
attendre $(n - f)$ messages de la forme $(\mathtt{P}, r, \cdots)$
si on a reçu au moins $(f + 1)$ messages $(\mathtt{P}, r, v)$ pour un même $v \neq ?$
décide(v)

`/* Phase supplémentaire: Ajustement */`

si il y avait un $(\mathtt{P}, r, v)$ où $v \neq ?$
$x \leftarrow v$
sinon
$x \leftarrow 0$ ou 1, au hasard (probabilité $\frac{1}{2}$ pour chaque valeur)

incrémenter r
}

On note de manière classique v_p la variable v du processus p.

La preuve suivante est extraite de [Had95].

Lemme 1
Il est impossible qu'à l'intérieur d'un même round deux processus proposent deux valeurs différentes.

Preuve
Raisonnons par l'absurde, en supposant que lors d'un certain *round* r on ait deux processus p_0 et p_1 qui proposent respectivement les valeurs 0 et 1. Comme p_0 (resp. p_1) propose la valeur

0 (resp. 1), c'est qu'il a reçu auparavant au moins $n/2$ rapports pour cette valeur 0 (resp. 1). Par conséquent, puisqu'il n'y a au total que n processus, il y a nécessairement un processus q qui a rapporté la valeur 0 à p_0 et la valeur 1 à p_1, ce qui est impossible. $\diamond$

Lemme 2
Pour $k \geq 1$, si à la fin du round $k-1$ il existe une valeur $v \in \{0,1\}$ telle que $x_p = v$ pour tous les processus p envoyant un rapport lors du round suivant k, alors tous les processus corrects décideront la valeur v lors du round k (en considérant que pour un processus p la valeur de x_p au "round 0" est la valeur initiale proposée par p).

Preuve
Si on se trouve dans la situation indiquée précédemment au *round* $k-1$, alors il est clair que tous les rapports reçus lors du *round* k seront pour la valeur v (d'après la définition même de l'algorithme). Lors du *round* k, chaque processus, si il propose une valeur, proposera la valeur v, puisque qu'il aura préalablement reçu $n-f > n/2$ rapports pour v. Par conséquent, toutes les propositions faites lors du *round* k seront en faveur de la valeur v. Enfin, puisqu'au moins les processus corrects enverront des propositions lors du *round* k et qu'il y en a au moins $n-f > f$, tout processus correct recevra au moins $f+1$ propositions en faveur de v au cours du *round* k et par conséquent décidera v lors du *round* k. $\diamond$

Lemme 3
Si un processus décide v lors du round k alors tous les processus corrects décideront également v lors du round suivant $k+1$.

Preuve
Si un processus p décide v lors du *round* k, c'est qu'il a reçu au moins $f+1$ propositions en faveur de v lors de ce *round* k. Soit A_p l'ensemble des processus qui ont émis ces propositions. Soit q un des processus qui enverra un rapport lors du *round* $k+1$. Soit A_q l'ensemble des processus desquels q a reçu ses $n-f$ propositions lors du *round* k. Alors A_p et A_q ont au moins un processus en commun. En effet,

$$|A_p| + |A_q| = |A_p \cup A_q| + |A_p \cap A_q| \leq n + |A_p \cap A_q|$$

et donc

$$|A_p \cap A_q| \geq |A_p| + |A_q| - n \geq (f+1) + (n-f) - n = 1$$

Par conséquent, q a reçu au moins une des $\geq f+1$ propositions en faveur de v que p a reçu. Donc q, ayant reçu une proposition en faveur de v, va procéder à l'affectation $x_q \leftarrow v$ (notons que, d'après le lemme 1, q n'a pas pu recevoir de proposition pour $1-v$). Par conséquent, tout processus q qui envoie un rapport lors du *round* $k+1$ aura $x_q = v$ à la fin du *round* k. Alors d'après le lemme 2, tout processus correct décidera v lors du *round* $k+1$. $\diamond$

Théorème 4.3.1 *([Had95])*
L'algorithme de Ben-Or résout le problème du consensus probabiliste dans tout système asynchrone où $f < \lceil \frac{n}{2} \rceil$.

Preuve
D'après le lemme 2 (avec $k = 1$), si toutes les valeurs proposées sont les mêmes alors tous processus corrects décident cette valeur au *round* 1, et sinon la propriété de validité est clairement satisfaite.
D'après le lemme 3, la propriété d'accord uniforme est aussi satisfaite. De plus, d'après ce même lemme, si un processus correct décide v lors d'un certain *round*, il continuera à décider v dans tous les *rounds* suivants, et par conséquent la propriété d'intégrité uniforme est aussi satisfaite.
Il reste donc à montrer la propriété de terminaison probabiliste. D'après le lemme 1, tous les processus corrects qui reçoivent au moins une proposition en faveur d'une certaine valeur v affectent tous leur variable x avec cette même valeur v, et les autres processus choisissent chacun une valeur (0 ou 1 avec une probabilité $\frac{1}{2}$ pour chaque valeur). Par conséquent, avec une probabilité au pire de $1/2^n$ ils choisissent également la valeur v. D'après le lemme 2, si tous les processus corrects ont choisi la même valeur v, tous les processus corrects décident au *round* suivant. Par conséquent, à chaque *round* exécuté, la probabilité de terminer est minorée par $\alpha = 1/2^n$ (et $0 < \alpha < 1$ pour $n \geq 1$), et d'après le théorème 4.1.1, la terminaison probabiliste est bien satisfaite. $\diamond$

4.4 Algorithme de Bracha et Toueg

On constate que l'on peut apporter certaines améliorations à l'algorithme de Ben-Or vu précédemment. On pourrait tout d'abord éviter qu'un processus s'envoie des messages à lui-même, mais, si la couche qui assure la transmission des messages est bien conçue, ce n'est qu'une amélioration mineure. En revanche, dans le cas asynchrone, un message peut mettre beaucoup de temps pour arriver à destination. On peut donc légèrement améliorer l'algorithme en n'attendant pas inutilement des messages.

Ainsi, dans les deux premières phases de l'algorithme (rapports et propositions), on a des constructions du type :

envoyer à tous $(\cdots)$
attendre $(n - f)$ messages de la forme $(\cdots)$
si on a reçu au moins k messages vérifiant $P(k)$
  ...

que l'on peut avantageusement remplacer par :

envoyer à tous $(\cdots)$
répéter
  attendre un message de la forme $(\cdots)$
jusqu'à avoir k messages vérifiant $P(k)$ ou $(n - f)$ messages au total
si on a reçu au moins k messages vérifiant $P(k)$
  ...

Puis on peut changer les imbrications des structures de contrôle.

En fait, on peut améliorer l'algorithme de Ben-Or en appliquant une stratégie qui fait que l'on retombe sur un autre algorithme très connu : l'algorithme de Bracha et Toueg [BT85]. Pour cela, on applique la stratégie suivante :

1. L'exécution de l'algorithme de Ben-Or ne se termine jamais, puisque ce dernier consiste en une boucle infinie dont on ne sort jamais. Mais on peut la faire se terminer très facilement grâce au résultat du lemme 3. En effet, on a montré que si un processus décide lors du *round* r alors tous les processus corrects décideront lors du *round* suivant $r+1$. Il suffit donc d'envoyer "prématurément" les messages qu'on aurait dû envoyer lors du *round* $r+1$ pour que les autres processus effectuent ce *round* normalement et décident.

2. On explicite les attentes de messages en comptant ces messages, sans oublier le "nettoyage" de l'ensemble des messages (ignorer les messages périmés, remettre les messages "futurs").

3. On fusionne les deux types de messages (`R` : Rapports, `P` : Propositions) en un seul de la forme $(cardinalité, \cdots)$, la cardinalité étant le nombre de rapports qui ont été reçus pour la valeur contenue dans le message. Ainsi, si la cardinalité d'un message est supérieure à $\lceil \frac{n}{2} \rceil$, c'est qu'il s'agit d'un rapport qui tient également lieu de proposition ("super-rapport"), sinon ce n'est qu'un simple rapport (et l'éventuelle proposition sera faite plus tard). Lorsqu'on fusionne ces deux types de messages, les deux phases d'un *round* r se retrouvent séparées en deux étapes $s = 2r - 1$ et $s + 1$. En fait, le lien entre *round* (Ben-Or) et étape (Bracha et Toueg) est donné par : $s = 2 \cdot (r - 1) + phase$ (avec $phase = 1$ ou 2).

4. On change les imbrications des structures de contrôle (et on renomme également une variable).

Il reste alors à remplacer le hasard pur par le "hasard" introduit par le système des messages, ce qui est un point fondamental.

Chacun des quatre stades de modification donne un algorithme "hybride", entre l'algorithme de Ben-Or et celui de Bracha et Toueg. Ces algorithmes peuvent être trouvés dans l'annexe B.

On aboutit donc à l'algorithme de Bracha et Toueg, où chaque processus p effectue l'algorithme suivant :

$x \leftarrow x_0$ `/* valeur initialement proposée */`

$s \leftarrow 1$ `/* numéro de l´étape */`

$cardinalité \leftarrow 1$

$nb\_messages[2] \leftarrow \{0,0\}$ `/* nombre de messages reçus */`
$nb\_témoins[2] \leftarrow \{0,0\}$ `/* nombre de témoins reçus */`

tant que $nb\_témoins[0] < (f+1)$ **et** $nb\_témoins[1] < (f+1)$
{
$nb\_messages \leftarrow \{0,0\}$
$nb\_témoins \leftarrow \{0,0\}$

envoyer à tous $(cardinalité, s, x)$
tant que $(nb\_messages[0] + nb\_messages[1]) < (n-f)$
{
attendre un message de la forme (c, k, v)

si $k = s$
{
incrémenter $nb\_messages[v]$
si $c > (n/2)$ `/* super-rapport ? */`
incrémenter $nb\_témoins[v]$
}
sinon si $k > s$
{
`/* message pour une étape future: à conserver */`
envoyer$(p,(c,k,v))$ `/* remettre le message dans` M `*/`
}
`/* si` $k < s$`, c´est un message périmé... */`
}

```
/* Ajustement */
```

si il existe v tel que $nb\_témoins[v] > 0$
  $x \leftarrow v$
sinon
  {
  si $nb\_messages[1] > nb\_messages[1]$
    $x \leftarrow 1$
  sinon
    $x \leftarrow 0$
  $cardinalité \leftarrow nb\_messages[x]$

  incrémenter s
  }
}

soit v tel que $nb\_témoins[v] \geq (f+1)$

envoyer à tous $(n-f, s, v)$
envoyer à tous $(n-f, s+1, v)$

décide(v)

Théorème 4.4.1
L'algorithme de Bracha et Toueg résout le problème du consensus probabiliste dans tout système asynchrone où $f < \lceil \frac{n}{2} \rceil$.

Preuve
La preuve est très similaire à celle faite pour l'algorithme de Ben-Or, ce qui est normal compte tenu du fait que l'on peut passer de l'un à l'autre en appliquant une stratégie qui "préserve la preuve". Ainsi on peut montrer les propriétés d'accord et d'intégrité uniformes pratiquement de la même façon. La propriété de validité découle, elle, de la définition même de l'algorithme. La seule difficulté est de trouver un minorant pour la probabilité de terminer lorsqu'on exécute une étape, afin de montrer la propriété de terminaison probabiliste à l'aide du théorème 4.1.1. En fait, on considère un système qui assure l'existence de ce minorant...Soit S un ensemble de $(n-f)$ processus corrects. Supposons que tous ces processus aient atteint l'étape s_0 sans avoir décidé. On montre alors qu'il y a une constante $\theta > 0$ telle que tous les processus de S décident à l'étape $s_0 + 2$ avec une probabilité supérieure à θ. On se place pour cela dans un système où il y a une constante $\rho > 0$ telle que tous les processus de S reçoivent lors d'une étape les $(n-f)$ messages envoyés par ces mêmes processus de S avec une probabilité supérieure à ρ. Par conséquent, les processus de S s'échangent des messages uniquement entre eux durant 3 étapes consécutives avec une probabilité supérieure à $\theta = \rho^3$. Et si cela se produit, il est clair que tous les processus de S décident à l'étape $s_0 + 2$. $\diamond$

4.5 Résultat

Théorème 4.5.1
Le consensus probabiliste peut être résolu si et seulement si $f < \lceil \frac{n}{2} \rceil$.

Preuve
On a déjà vu précédemment (en 4.3 et 4.4) des algorithmes résolvant le consensus probabiliste si $f < \lceil \frac{n}{2} \rceil$. Or cette condition est nécessaire ([BO83]), ca sinon on peut simuler une bipartition du réseau.

Chapitre 5

Détecteurs de défaillance

Il est impossible de résoudre le problème du consensus à l'aide d'algorithmes déterministes dans un système asynchrone à partir du moment où un processus est susceptible d'être défectueux. Cette impossibilité réside dans le fait que, dans de tels systèmes, il est impossible de savoir de manière précise si un processus est défaillant ou seulement dans une phase d'exécution très lente. Aussi Toueg et Chandra [CT94] ont-ils eu l'idée d'introduire des détecteurs de défaillance non fiables.

Un détecteur de défaillance doit être vu comme un "oracle", qui n'est en général pas un algorithme. En effet, un détecteur de défaillance qui serait un algorithme déterministe ne peut en aucun cas aider à résoudre le problème du consensus dans le modèle considéré, puisque sinon on contredirait le théorème d'impossibilité. On se limite donc, du moins dans un premier temps, à des définitions formelles.

5.1 Définitions

Un *détecteur de défaillance* (FD) peut être vu comme un oracle distribué qui, lorsqu'il est consulté, retourne l'ensemble des processus qu'il suspecte d'être défaillants (un FD pouvant très bien ne pas être fiable).

On étend alors le modèle de base présenté au chapitre 2 avec des FD, chaque processus disposant localement d'un de ces FD. Lors d'un pas atomique un processus peut, en plus, questionner son FD local.

On suppose également l'existence d'une horloge discrète globale. En fait, cette horloge ne sert qu'à simplifier la notion de temps dans les définitions et les preuves formelles. Les processus n'y ont naturellement pas accès.[1] Cette horloge prend ses valeurs dans l'ensemble $\mathcal{T}$, qui sera

[1]Les processus peuvent cependant posséder des horloges locales, qui sont indépendantes les unes des autres et ne sont pas liées à l'horloge globale.

en fait ici l'ensemble des entiers naturels.

Il convient maintenant de définir de manière plus formelle ce qu'est un détecteur de défaillance, et il faut pour cela quelques définitions préliminaires...

Un *motif de défaillance* F (*failure pattern*) est une fonction de $\mathcal{T}$ dans l'ensemble des parties de Π telle que, pour toute date t, $F(t)$ est l'ensemble des processus défaillants à cette date. Comme on ne considère ici que le cas des pannes franches, et que par conséquent un processus tombé en panne le reste indéfiniment, on a :

$$\forall t \in \mathcal{T}, F(t) \subseteq F(t+1)$$

On définit alors les deux ensembles suivants :

$$défectueux(F) = \bigcup_{t \in \mathcal{T}} F(t) \quad \text{et} \quad correct(F) = \Pi \setminus défectueux(F)$$

Si $p \in défectueux(F)$, on dit que p est *défectueux dans* F, et si $p \in correct(F)$, on dit que p est *correct dans* F. On ne s'intéresse ici qu'à des motifs de défaillance F dans lesquels au moins un processus est correct, c'est-à-dire pour lesquels $correct(F) \neq \emptyset$.

Un *historique de détecteur de défaillance* H (*failure detector history*) est une fonction de $\Pi \times \mathcal{T}$ dans l'ensemble des parties de Π telle que, pour tout processus p et pour toute date t, $H(p,t)$ est la valeur de sortie du FD local au processus p, à la date t. Lorsque $q \in H(p,t)$, on dit que *p suspecte q à la date t dans H*. Lorsqu'aucune ambiguïté n'est possible, les références à H seront implicites.

Remarque
Il se peut très bien que deux processus différents p et q ne suspectent pas les mêmes processus au même instant, c'est-à-dire que $H(p,t) \neq H(q,t)$ $(p \neq q)$.

Un FD $\mathcal{D}$ fournit donc des informations, éventuellement fausses, sur le motif de défaillance F qui se produit lors de l'exécution. Formellement, un détecteur de défaillance $\mathcal{D}$ est une fonction qui à chaque motif de défaillance F fait correspondre un ensemble d'historiques $\mathcal{D}(F)$: l'ensemble de tous les historiques qui peuvent se produire dans une exécution avec le motif de défaillance F et le détecteur de défaillance $\mathcal{D}$.[2]

Il ne faut pas chercher à deviner une éventuelle implémentation concrète des FD, qui sont des oracles, mais plutôt raisonner sur ces FD en se basant sur des propriétés abstraites...

5.2 Propriétés

On s'intéresse tout particulièrement à deux familles de propriétés : les propriétés de complétude et les propriétés de précision. Intuitivement, la complétude assure qu'un processus correct finit

[2] Il y a en général beaucoup d'exécutions possibles pour un même motif de défaillance F (les temps de transmission et les vitesses des processus pouvant varier). $\mathcal{D}$ peut alors donner un historique de FD différent pour chacune de ces exécutions.

par suspecter tout processus défectueux, tandis que la précision assure qu'un processus correct ne suspecte pas un processus correct. Complétude et précision qui semblent à première vue s'opposer diamétralement sont en fait orthogonales.

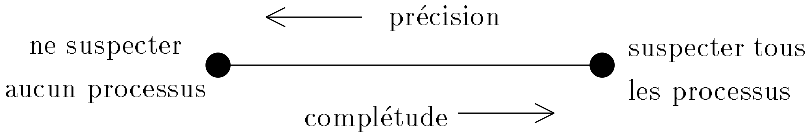

Figure 2: *Complétude vs précision.*

En fait, il convient de raffiner ces propriétés et de doser harmonieusement complétude et précision afin d'obtenir des FD utiles à la résolution du consensus.

5.2.1 Complétude

On considère les deux propriétés de complétude suivantes :

• la *complétude forte* (*strong completeness*) : tout processus défectueux finit tôt ou tard par être continuellement suspecté par *tout* processus correct. Formellement, $\mathcal{D}$ satisfait la complétude forte si et seulement si :

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathcal{T}, \forall p \in défectueux(F), \underline{\forall q} \in correct(F), \forall t' \geq t : p \in H(q, t')$$

• la *complétude faible* (*weak completeness*) : tout processus défectueux finit tôt ou tard par être continuellement suspecté par *un* certain processus correct. Formellement, $\mathcal{D}$ satisfait la complétude faible si et seulement si :

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathcal{T}, \forall p \in défectueux(F), \underline{\exists q} \in correct(F), \forall t' \geq t : p \in H(q, t')$$

La complétude en elle-même n'est pas une propriété intéressante. En effet, il suffit de considérer un FD qui fait que chaque processus suspecte en permanence tout autre processus dans le système. Un tel FD satisfait clairement la complétude forte, mais est clairement inutile (puisqu'il n'apporte aucune information sur les défaillances). Pour être utile, un FD doit *également* satisfaire une propriété de précision qui restreint les erreurs qu'il peut faire.

5.2.2 Précision

5.2.2.1 Précision perpétuelle

On considère dans un premier temps les deux propriétés de précision suivantes :

• la *précision (perpétuellement) forte* (*(perpetual) strong accuracy*) : aucun processus n'est suspecté avant de défaillir. Formellement, $\mathcal{D}$ satisfait la précision forte si et seulement si :

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \underline{\forall p, q \in \Pi \setminus F(t)} : p \notin H(q,t)$$

• la *précision (perpétuellement) faible* (*(perpetual) weak accuracy*) : il existe un processus correct qui n'est jamais suspecté. Formellement, $\mathcal{D}$ satisfait la précision faible si et seulement si :

$$\forall F, \forall H \in \mathcal{D}(F), \underline{\exists p \in correct(F)}, \forall t \in \mathcal{T}, \underline{\forall q \in \Pi \setminus F(t)} : p \notin H(q,t)$$

En fait, ces propriétés sont souvent trop fortes pour être systématiquement réalisables en pratique lorsqu'on exige également la complétude. En effet, il est alors difficile de garantir qu'un certain processus ne soit *jamais* suspecté. On considère alors des FD qui peuvent suspecter tout processus à un moment ou à un autre, en demandant seulement que les propriétés de précision vues précédemment soient vérifiées à partir d'une certaine date...

5.2.2.2 Précision "à la longue"

On obtient ainsi deux nouvelles propriétés de précision, encore plus faibles. En remarquant que tous les processus défectueux vont défaillir au bout d'un temps fini, on a :

$$\forall F, \exists t \in \mathcal{T}, \forall t' \geq t : \Pi \setminus F(t') = correct(F)$$

qui permet de simplifier les énoncés des propriétés suivantes :

• la *précision finalement forte* (*eventual strong accuracy*) : il existe une date après laquelle aucun processus correct n'est suspecté par un processus correct. Formellement, $\mathcal{D}$ satisfait la précision finalement forte si et seulement si :

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

• la *précision finalement faible* (*eventual weak accuracy*) : il existe une date après laquelle un certain processus correct n'est plus jamais suspecté par aucun processus correct. Formellement, $\mathcal{D}$ satisfait la précision finalement faible si et seulement si :

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

5.3 Réduction et équivalence

Soient $\mathcal{A}$ et $\mathcal{A}'$ deux algorithmes. Si il existe un algorithme $T_{\mathcal{A}\rightarrow\mathcal{A}'}$ qui utilise l'algorithme $\mathcal{A}$ pour émuler une exécution de $\mathcal{A}'$, on dit que $\mathcal{A}'$ peut se réduire à $\mathcal{A}$ et que $T_{\mathcal{A}\rightarrow\mathcal{A}'}$ est un algorithme de réduction.

On généralise la réduction d'algorithmes aux détecteurs de défaillance[3] ainsi : soient $\mathcal{D}$ et $\mathcal{D}'$ deux FD. Si il existe un algorithme $T_{\mathcal{D}\to\mathcal{D}'}$ qui utilise la sortie de $\mathcal{D}$ pour émuler une sortie possible[4] de $\mathcal{D}'$, on dit que $\mathcal{D}'$ peut se réduire à $\mathcal{D}$ et que $T_{\mathcal{D}\to\mathcal{D}'}$ est un *algorithme de réduction.*

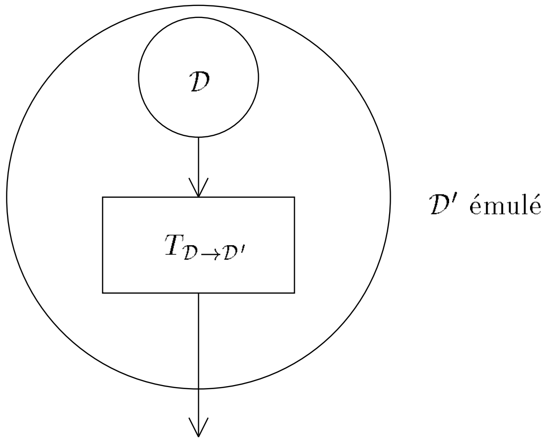

Figure 3: *Transformer $\mathcal{D}$ en $\mathcal{D}'$.*

Etant donné un algorithme de réduction $T_{\mathcal{D}\to\mathcal{D}'}$, tout problème pouvant être résolu en utilisant le FD $\mathcal{D}'$ peut être résolu en utilisant $\mathcal{D}$ à la place. Intuitivement, comme $T_{\mathcal{D}\to\mathcal{D}'}$ est capable d'utiliser $\mathcal{D}$ pour émuler $\mathcal{D}'$, c'est que $\mathcal{D}$ fournit au moins autant d'informations sur les défaillances que $\mathcal{D}'$. Par conséquent, on dit que $\mathcal{D}'$ est plus faible que $\mathcal{D}$, et on note $\mathcal{D} \succeq \mathcal{D}'$. Si $\mathcal{D} \succeq \mathcal{D}'$ et $\mathcal{D}' \succeq \mathcal{D}$, on dit que $\mathcal{D}$ et $\mathcal{D}'$ sont équivalents, et on note $\mathcal{D} \cong \mathcal{D}'$. Si $\mathcal{D} \succeq \mathcal{D}'$ et $\mathcal{D}$ et $\mathcal{D}'$ ne sont pas équivalents, on dit que $\mathcal{D}'$ est strictement plus faible que $\mathcal{D}$, et on note $\mathcal{D} \succ \mathcal{D}'$.

On définit de plus l'algorithme de réduction *identité* qui consiste à recopier la sortie de $\mathcal{D}$.

On voit aisément que $\succeq$, $\cong$ et $\succ$ sont respectivement des relations d'ordre, d'équivalence et d'ordre strict (en utilisant l'algorithme de réduction identité et la composition d'algorithmes). Cependant, on ne s'intéressera pas dans la suite aux classes d'équivalence modulo $\cong$, mais aux classes de FD obtenues à partir des propriétés de complétude et de précision. On se servira alors de la généralisation des propriétés $\succeq$, $\cong$ et $\succ$ aux classes de FD afin de les comparer.

5.4 Classes de base

On s'intéresse dans la suite à des FD satisfaisant des propriétés du type complétude *et* précision. On regroupe en classes les FD ayant les mêmes propriétés. En combinant les 2 propriétés de complétude avec les 4 propriétés de précision énoncées en 5.2, on obtient donc 8 classes.

[3] Un détecteur de défaillance est un oracle de nature *a priori* quelconque, et pas un algorithme.

[4] $T_{\mathcal{D}\to\mathcal{D}'}$ n'a pas à émuler toutes les historiques de $\mathcal{D}'$. Il suffit que toutes les historiques qu'il émule soient des historiques de $\mathcal{D}'$.

Complétude	Précision			
	Forte	Faible	Finalement Forte	Finalement Faible
Forte	Parfait *Perfect* $\mathcal{P}$	Fort *Strong* $\mathcal{S}$	Finalement Parfait *Eventually Perfect* $\diamond\mathcal{P}$	Finalement Fort *Eventually Strong* $\diamond\mathcal{S}$
Faible	$\mathcal{Q}$	Faible *Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	Finalement Faible *Eventually Weak* $\diamond\mathcal{W}$

Figure 4: *Les 8 classes de FD modulo la complétude et la précision.*

Remarque
Comme on a clairement complétude forte $\Rightarrow$ complétude faible, on a $\mathcal{P} \subseteq \mathcal{Q}$, $\mathcal{S} \subseteq \mathcal{W}$, $\diamond\mathcal{P} \subseteq \diamond\mathcal{Q}$ et $\diamond\mathcal{S} \subseteq \diamond\mathcal{W}$. De même, précision forte $\Rightarrow$ précision faible donne $\mathcal{P} \subseteq \mathcal{S}$, $\mathcal{Q} \subseteq \mathcal{W}$, $\diamond\mathcal{P} \subseteq \diamond\mathcal{S}$ et $\diamond\mathcal{Q} \subseteq \diamond\mathcal{W}$. Et enfin, précision (perpétuelle) $\Rightarrow$ précision "à la longue" donne $\mathcal{P} \subseteq \diamond\mathcal{P}$, $\mathcal{Q} \subseteq \diamond\mathcal{Q}$, $\mathcal{S} \subseteq \diamond\mathcal{S}$ et $\mathcal{W} \subseteq \diamond\mathcal{W}$.

On peut étendre les relations $\succeq$, $\cong$ et $\succ$ aux classes de FD de façon naturelle. Etant donné deux classes $\mathcal{C}$ et $\mathcal{C}'$, si pour chaque $\mathcal{D} \in \mathcal{C}$ il existe un $\mathcal{D}' \in \mathcal{C}'$ tel que $\mathcal{D} \succeq \mathcal{D}'$, on dit que $\mathcal{C}'$ est plus faible que $\mathcal{C}$ (noté $\mathcal{C} \succeq \mathcal{C}'$). Si $\mathcal{C} \succeq \mathcal{C}'$ et $\mathcal{C}' \succeq \mathcal{C}$ on dit que $\mathcal{C}$ et $\mathcal{C}'$ sont équivalentes (noté $\mathcal{C} \cong \mathcal{C}'$). Si $\mathcal{C} \succeq \mathcal{C}'$ et si $\mathcal{C}$ et $\mathcal{C}'$ ne sont pas équivalentes, on dit que $\mathcal{C}'$ est strictement plus faible que $\mathcal{C}$ (noté $\mathcal{C} \succ \mathcal{C}'$).

Théorème 5.4.1
$\mathcal{C} \subseteq \mathcal{C}' \Rightarrow \mathcal{C} \succeq \mathcal{C}'$ *et* $\mathcal{C} = \mathcal{C}' \Rightarrow \mathcal{C} \cong \mathcal{C}'$.
Les implications réciproques sont fausses en général.
En particulier l'implication : $\mathcal{C} \subset \mathcal{C}' \Rightarrow \mathcal{C} \succ \mathcal{C}'$ *est fausse.*

Preuve
Soient $\mathcal{C}$ et $\mathcal{C}'$ deux classes de FD, et soit $\mathcal{D}$ un FD dans $\mathcal{C}$. En considérant l'algorithme de réduction identité (consistant à recopier la sortie de $\mathcal{D}$) on voit immédiatement que l'on émule un FD $\mathcal{D}'$ ayant les mêmes propriétés que $\mathcal{D}$, et donc $\mathcal{D}' \in \mathcal{C}$. Si on suppose de plus $\mathcal{C} \subseteq \mathcal{C}'$, on a alors $\mathcal{D}' \in \mathcal{C}'$. Pour tout FD $\mathcal{D} \in \mathcal{C}$, on a donc un algorithme de réduction (trivial) qui transforme $\mathcal{D}$ en $\mathcal{D}' \in \mathcal{C}'$, et par conséquent $\mathcal{C} \succeq \mathcal{C}'$. D'où l'implication $\mathcal{C} \subseteq \mathcal{C}' \Rightarrow \mathcal{C} \succeq \mathcal{C}'$.
Comme $\mathcal{C} \subseteq \mathcal{C}' \Rightarrow \mathcal{C} \succeq \mathcal{C}'$ et que $\mathcal{C}' \subseteq \mathcal{C} \Rightarrow \mathcal{C}' \succeq \mathcal{C}$, on a également $\mathcal{C} = \mathcal{C}' \Rightarrow \mathcal{C} \cong \mathcal{C}'$.
Considérons alors deux classes $\mathcal{C}$ et $\mathcal{C}'$ telles que $\mathcal{C} = \mathcal{C}' \cup \{\mathcal{D} : \mathcal{D} \succeq \mathcal{D}' \text{ et } \mathcal{D}' \in \mathcal{C}'\}$. On peut de plus trouver de telles classes où $\mathcal{C}' \subset \mathcal{C}$. Or on a clairement $\mathcal{C} \succeq \mathcal{C}'$. L'implication $\mathcal{C} \subseteq \mathcal{C}' \Leftarrow \mathcal{C} \succeq \mathcal{C}'$ est donc fausse.
Considérons maintenant deux classes $\mathcal{C}$ et $\mathcal{C}'$ telles que $\mathcal{C} = \mathcal{C}' \cup \{\mathcal{D} : \mathcal{D} \cong \mathcal{D}' \text{ et } \mathcal{D}' \in \mathcal{C}'\}$. On peut de plus trouver de telles classes où $\mathcal{C}' \subset \mathcal{C}$. Or on a clairement $\mathcal{C} \cong \mathcal{C}'$. L'implication $\mathcal{C} = \mathcal{C}' \Leftarrow \mathcal{C} \cong \mathcal{C}'$ est donc également fausse.
Enfin, comme l'implication $\mathcal{C}' \subseteq \mathcal{C} \Leftarrow \mathcal{C}' \succeq \mathcal{C}$ est fausse, sa contraposée $\mathcal{C} \subset \mathcal{C}' \Rightarrow \mathcal{C} \succ \mathcal{C}'$ l'est également. $\diamond$

$$\begin{tikzcd}
\mathcal{P} \cong \mathcal{Q} \arrow[r] \arrow[d] \arrow[dr] & \diamond\mathcal{P} \cong \diamond\mathcal{Q} \arrow[d] \\
\mathcal{S} \cong \mathcal{W} \arrow[r] & \diamond\mathcal{S} \cong \diamond\mathcal{W}
\end{tikzcd}$$

Figure 5: *Relations entre les 8 classes de base.*

On veut comparer deux à deux les 8 classes de base. On cherche donc, pour tout couple de classes, quelle relation les lie ($\succeq$, $\cong$, $\succ$ ou aucune des trois). Par conséquent, on devrait avoir 28 (C_8^2) relations. La figure précédente (Figure 5) en représente 24 ($4 + 5 \times 4$). Dans [CT94] il est dit que $\mathcal{S}$ et $\diamond\mathcal{P}$ sont incomparables, d'où les 4 "non relations" manquantes.

Pour prouver ces relations, on peut montrer que :

1. $\mathcal{P} \cong \mathcal{Q}$, $\mathcal{S} \cong \mathcal{W}$, $\diamond\mathcal{P} \cong \diamond\mathcal{Q}$ et $\diamond\mathcal{S} \cong \diamond\mathcal{W}$
2. $\mathcal{P} \succ \mathcal{S}$ et $\diamond\mathcal{P} \succ \diamond\mathcal{S}$
3. $\mathcal{P} \succ \diamond\mathcal{P}$ et $\mathcal{S} \succ \diamond\mathcal{S}$

puis que $\mathcal{S}$ et $\diamond\mathcal{P}$ sont incomparables.

Toutes les autres relations se déduisent facilement des précédentes (notamment en utilisant la transitivité de $\succ$).

Le fait que des classes soient deux à deux incomparables ($\mathcal{S}$ et $\diamond\mathcal{P}$, $\mathcal{S}$ et $\diamond\mathcal{Q}$, $\mathcal{W}$ et $\diamond\mathcal{P}$, $\mathcal{W}$ et $\diamond\mathcal{Q}$) ne sera pas utilisé par la suite, et par conséquent la preuve en est omise.
En revanche, on utilisera dans le chapitre 6 le fait que $\mathcal{P} \succeq \mathcal{S}$ et $\diamond\mathcal{P} \succeq \diamond\mathcal{S}$. Ces relations sont directement obtenues avec le théorème 5.4.1 après avoir remarqué que $\mathcal{P} \subseteq \mathcal{S}$ et $\diamond\mathcal{P} \subseteq \diamond\mathcal{S}$. De manière générale, on peut montrer les relations des troisième et deuxième points en appliquant la même stratégie : on veut montrer que $\mathcal{C} \succ \mathcal{C}'$, alors que le théorème 5.4.1 donne $\mathcal{C} \succeq \mathcal{C}'$. Il reste à montrer que les classes $\mathcal{C}$ et $\mathcal{C}'$ ne sont pas équivalentes, en exhibant un FD $\mathcal{D}' \in \mathcal{C}'$ qui ne peut pas être émulé par un FD $\mathcal{D} \in \mathcal{C}$ (on revient pour cela aux propriétés définissant les classes $\mathcal{C}$ et $\mathcal{C}'$, soient respectivement P et P', et on se sert du fait que P' est "strictement plus forte" que P).
Les quatre relations du premier point sont, elles, très utiles puisqu'elles permettent de limiter

l'étude aux 4 classes avec complétude forte ($\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$ et $\diamond\mathcal{S}$) au lieu des 8 initiales. On peut donc obtenir la complétude forte à partir de la complétude faible (par réduction) sans toucher à la propriété de précision.

On s'intéresse par conséquent aux résultats suivants :

Théorème 5.4.2 *([CT94])*
$\mathcal{P} \cong \mathcal{Q}$, $\mathcal{S} \cong \mathcal{W}$, $\diamond\mathcal{P} \cong \diamond\mathcal{Q}$ *et* $\diamond\mathcal{S} \cong \diamond\mathcal{W}$.

Preuve

Comme $\mathcal{P} \subseteq \mathcal{Q}$, $\mathcal{S} \subseteq \mathcal{W}$, $\diamond\mathcal{P} \subseteq \diamond\mathcal{Q}$ et $\diamond\mathcal{S} \subseteq \diamond\mathcal{W}$, d'après le théorème 5.4.1, on a $\mathcal{P} \succeq \mathcal{Q}$, $\mathcal{S} \succeq \mathcal{W}$, $\diamond\mathcal{P} \succeq \diamond\mathcal{Q}$ et $\diamond\mathcal{S} \succeq \diamond\mathcal{W}$.

Réciproquement, montrons que $\mathcal{Q} \succeq \mathcal{P}$, $\mathcal{W} \succeq \mathcal{S}$, $\diamond\mathcal{Q} \succeq \diamond\mathcal{P}$ et $\diamond\mathcal{W} \succeq \diamond\mathcal{S}$.
Considérons pour cela l'algorithme de réduction suivant (pour chaque processus p) :

$sortie \leftarrow \emptyset$

répéter
{
$suspects \leftarrow \mathcal{D}$ `/* questionner son FD local */`
envoyer à tous $(p, suspect)$
}

quand on reçoit un message du type $(q, suspects_q)$
$sortie \leftarrow (sortie \cup suspects_q) \setminus \{q\}$

où *suspects* est une liste de suspects maintenue par le processus.

Montrons alors les trois points suivants :

• Cet algorithme transforme bien la complétude faible en complétude forte.
En effet, soit p un processus qui défaille. Supposons que $\mathcal{D}$ soit faiblement complet, c'est-à-dire qu'il y ait une date t après laquelle un certain processus correct q suspecte continuellement p dans $H_{\mathcal{D}}$. Nous devons alors montrer qu'il y a une date après laquelle tout processus correct suspecte continuellement p dans *sortie*. Comme p est défaillant, il y a une date t' après laquelle aucun processus ne reçoit plus de message en provenance de p. Après la date $t'' = max(t, t')$, le processus q envoie à tous les processus un message de la forme $(q, suspect_q)$ avec $p \in suspect_q$. Au bout d'un certain temps, tout processus correct recevra ce message et ajoutera p à sa *sortie*. Comme aucun processus ne reçoit plus de message en provenance de p après la date t' et comme $t'' \geq t'$, aucun processus correct ne retirera p de sa *sortie* après la date t''. Par conséquent, il y a une date à partir de laquelle tout processus correct suspecte continuellement p, d'où la complétude forte.

• Cet algorithme préserve la précision perpétuelle.
Soit p un processus quelconque. Supposons qu'il existe une date t avant laquelle aucun processus ne suspecte p dans $H_{\mathcal{D}}$. Alors aucun processus n'envoie de message du type $(\cdots, suspects)$ avec $p \in suspects$ avant cette date t. Par conséquent, aucun processus n'ajoute p à sa *sortie* avant la date t.

• Cet algorithme préserve la précision "à la longue" (finalement forte et finalement faible).
Soit p un processus correct quelconque. Supposons qu'il existe une date t après laquelle aucun processus correct ne suspecte p dans $H_{\mathcal{D}}$. Alors il existe une date t' après laquelle aucun processus ne reçoit plus de message du type $(\cdots, suspects)$ avec $p \in suspects$. Soit q un processus correct quelconque. Il nous faut montrer qu'il existe une date après laquelle q ne suspecte plus p dans sa *sortie*. Après la date t', le processus p envoie un message $(p, suspects_p)$ à q. Quand q reçoit ce message, il retire p de sa sortie $sortie_q$ (s'il y a lieu). Comme q ne reçoit pas un message du type $(\cdots, suspects)$ avec $p \in suspects$ après la date t', q n'ajoute plus p à sa sortie $sortie_q$ après cette date. Par conséquent, il existe une date après laquelle q ne suspecte plus jamais p dans *sortie*.

Soit $\mathcal{D}$ un détecteur de défaillance dans $\mathcal{Q}$ (resp. $\mathcal{W}$) ; les deux premiers points montrent que $\mathcal{Q} \succeq \mathcal{P}$ (resp. $\mathcal{W} \succeq \mathcal{S}$).
Soit $\mathcal{D}$ un détecteur de défaillance dans $\diamond\mathcal{Q}$ (resp. $\diamond\mathcal{W}$) ; les premier et dernier points montrent que $\diamond\mathcal{Q} \succeq \diamond\mathcal{P}$ (resp. $\diamond\mathcal{W} \succeq \diamond\mathcal{S}$). $\diamond$

5.5 Un exemple de détecteur de défaillance

Jusqu'à présent, les FD ont été manipulés formellement, sans que leur nature soit précisée. Dans le modèle considéré, il est d'ailleurs impossible d'implémenter un FD dans l'une des 8 classes de base (puisqu'on verra plus tard qu'elles permettent de résoudre le consensus).

On se place ici dans le cas d'un système dans lequel les temps nécessaires pour transmettre un message et pour exécuter un pas sont *bornés*, mais les bornes ne sont pas connues (*synchronisation partielle*). C'est une restriction du modèle qui est raisonnable en pratique, puisque dans la plupart des systèmes réels la probabilité que ces temps soient bornés est *a priori* forte.

Sous de telles hypothèses, on considère alors l'algorithme suivant (pour chaque processus p) :

```
sortie ← ∅
pour tout q ∈ Π
  {
    /* Δ(q) est la durée du time-out pour q */
    Δ(q) ← time-out par défaut
  }

faire en parallèle
  {
    || Tâche 1:
      répéter
        envoyer à tous "p est vivant"

    || Tâche 2:
      répéter
        pour tout q ∈ Π
          si q ∉ sortie
          et p n'a pas reçu "q est vivant"
             pendant les derniers Δ(q) tops d'horloge (locale)
            {
              /* q est maintenant suspecté */
              sortie ← sortie ∪ {q}
            }

    || Tâche 3:
      quand on reçoit un message "q est vivant"
        si q ∈ sortie
          {
            /* on a suspecté q à tort */
            sortie ← sortie \ {q}   /* réhabiliter q */
            incrémenter Δ(q)   /* augmenter le time-out pour q */
          }
  }
```

Théorème 5.5.1 *([CT94])*
Supposons que les temps nécessaires pour transmettre un message et pour exécuter un pas sont bornés (mais on ne connaît pas ces bornes). Alors l'algorithme précédent implémente un FD $\mathcal{D} \in \Diamond\mathcal{P}$.

Preuve
Montrons tout d'abord la complétude forte : tout processus défaillant finit tôt ou tard par être continuellement suspecté par tous les processus corrects. Supposons qu'un certain processus

q défaille. Par conséquent, q n'envoie plus aucun message, et il y a donc une date à partir de laquelle aucun processus ne reçoit plus de messages "q est vivant". Pour chaque processus correct p, il y a donc une date après laquelle le *time-out* pour q est écoulé et plus aucun message "q est vivant" n'est reçu. Par conséquent, il est clair que tout processus correct finit tôt ou tard par suspecter continuellement q, et donc la complétude forte est satisfaite.
Il reste alors à montrer la précision finalement forte : au bout d'un certain temps, aucun processus correct n'est suspecté par un processus correct. Soient p et q deux processus corrects. p ne peut pas voir une infinité de fois son *time-out* pour q atteint. En effet, à chaque fois que ce dernier est atteint, il est incrémenté. Si cela se produisait une infinité de fois, $\Delta_p(q)$ ne serait pas borné, ce qui contredit le fait que les vitesses des processus et des transmissions soient bornées. Considérons alors la dernière fois que le processus p voit son *time-out* pour q atteint. Comme q est correct, il continue indéfiniment à envoyer "q est vivant", et p finit par recevoir un tel message et réhabilite q. Comme p ne verra plus son *time-out* pour q atteint, il ne le suspectera plus jamais, et donc la précision finalement forte est satisfaite. $\diamond$

Remarque
Cette implémentation n'est pas utilisée en pratique car ses performances ne sont pas suffisantes. De plus, on ne peut en général pas garantir qu'un système réel permette de satisfaire à la fois la complétude forte et la précision finalement forte...

Chapitre 6

Consensus avec détecteurs de défaillance

Certains détecteurs de défaillance (FD) présentés au chapitre 5 permettent de résoudre le problème du consensus. Ce résultat établi dans [CT94] est présenté dans ce chapitre, et le cas de consensus probabiliste est examiné. Des algorithmes résolvant le consensus en utilisant ces FD sont égalements présentés.

6.1 Principe

On se place dans le modèle de base enrichi avec un FD $\mathcal{D}$ (chaque processus p accédant à un FD local $\mathcal{D}_p$). On cherche alors des algorithmes réalisant le consensus en tolérant le plus grand nombre de processus défectueux possible.

6.2 Algorithmes

On cherche à résoudre le problème du consensus dans chacune des 4 classes mises en évidence dans le chapitre 5.

Rappel: On note v_p la variable v du processus p.

6.2.1 Classes $\mathcal{S}$ et $\mathcal{P}$

Soit $\mathcal{D} \in \mathcal{S}$ (c'est-à-dire que le FD $\mathcal{D}$ est fortement complet et faiblement précis).

Le principe de l'algorithme proposé est le suivant : lors d'une première phase chaque processus effectue $n-1$ *rounds* en diffusant à chaque *round* les nouvelles valeurs qu'il a apprises lors du précédent et attend d'éventuelles nouvelles valeurs, puis lors d'une deuxième phase les processus accordent leurs estimations, et finalement lors d'une troisième et dernière phase ils décident une certaine valeur parmi celles qui restent à la fin de la phase précédente.

Chaque processus p exécute l'algorithme suivant :

proposer(v):
{
  $V \leftarrow [\perp, \cdots, \perp]$ `/* estimations des valeurs proposées */`
  $V[p] \leftarrow v$
  $\Delta \leftarrow V$

  `/* Phase 1 (`*rounds* `asynchrones) */`

  pour r allant de 1 à $n-1$
    {
      envoyer à tous (r, Δ, p)
      attendre $\forall q : (r, \Delta_q, q)$ reçu $\vee\ q \in \mathcal{D}$
      $\Delta \leftarrow [\perp, \cdots, \perp]$
      pour k allant de 1 à n
        si $V[k] = \perp$ **et** on a reçu un message (r, Δ_q, q) où $\Delta_q[k] \neq \perp$
          {
            $V[k] \leftarrow \Delta_q[k]$
            $\Delta[k] \leftarrow \Delta_q[k]$
          }
    }

  `/* Phase 2 */`

  envoyer à tous V
  attendre $\forall q : V_q$ reçu $\vee\ q \in \mathcal{D}$
  pour k allant de 1 à n
    si on a reçu un V_q où $V_q[k] = \perp$
      $V[k] \leftarrow \perp$

  `/* Phase 3 */`

  $v \leftarrow$ la plus petite valeur dans V différente de $\perp$
  décide(v)
}

Notons que $V_p[q]$ est l'estimation faite par p de la valeur proposée par q. De plus, $\Delta_p[q] = v_q$ à la fin du *round* r si et seulement si p a reçu v_q (la valeur proposée par q) pour la première fois lors du *round* r.

Lemme 4
Pour tous processus p et q, et pour chaque phase, $V_p[q]$ vaut soit v_q soit $\perp$.

Preuve
Ce résultat est trivial (il découle de la définition même de l'algorithme). $\diamond$

Lemme 5
Tout processus correct finit tôt ou tard par atteindre la phase 3.

Preuve
Soit p un processus correct. La seule façon pour p de ne pas atteindre la phase 3 est de rester indéfiniment bloqué lors d'une des attentes de messages des phases 1 ou 2. Cela ne peut se produire que si p attend indéfiniment un message en provenance d'un certain processus q qui, après une certaine date, ne serait jamais suspecté. Or q ne peut pas être correct, car sinon il finirait par envoyer le message que p attend, et p le recevrait tôt ou tard. Mais alors, si q est défectueux, comme $\mathcal{D}$ est fortement complet, il y a une date après laquelle il est continuellement suspecté (donc $q \in \mathcal{D}_p$). Il est donc impossible que p reste indéfiniment bloqué. $\diamond$

Comme $\mathcal{D}$ est faiblement précis, il existe un processus correct c qui n'est jamais suspecté. Soit Π_1 l'ensemble des processus qui ont effectué la totalité des $n-1$ *rounds* lors de la phase 1, et soit Π_2 l'ensemble des processus qui ont passé la phase 2. On dira que $V_p \leq V_q$ si et seulement si pout tout $k \in \Pi$, $V_p[k]$ vaut $V_q[k]$ ou $\perp$.

Lemme 6
A l'intérieur de chaque round r, $1 \leq r \leq n-1$, tout processus $p \in \Pi_1$ reçoit (r, Δ_c, c) (émis par c).

Preuve
Comme $p \in \Pi_1$, p a effectué tous les *rounds* de la phase 1. A chaque *round* r, comme $c \notin \mathcal{D}_p$, p attend et reçoit le message (r, Δ_c, c) envoyé par c. $\diamond$

Lemme 7
Pour tout processus $p \in \Pi_1$, $V_c \leq V_p$ à la fin de la phase 1.

Preuve
Supposons que pour un certain processus q on a $V_c[q] \neq \perp$ à la fin de la phase 1. D'après le

lemme 4 on a $V_c[q] = v_q$. Soit $p \in \Pi_1$. Il nous faut alors montrer que $V_p[q] = v_q$ à la fin de la phase 1. C'est évident si $p = c$. Considérons donc le cas $p \neq c$. Soit r le premier *round* lors duquel c reçoit v_q (et si $c = q$, on considère que $r = 0$). D'après la définition de l'algorithme, il est clair que $\Delta_c[q] = v_q$ à la fin du *round* r. Considérons alors les deux cas suivants :

1. $r \leq n-2$. Alors, lors du *round* $r+1 \leq n-1$, c diffuse la valeur v_q en envoyant à tous le message $(r+1, \Delta_c, c)$ où $\Delta_c[q] = v_q$. D'après le lemme 6, p reçoit $(r+1, \Delta_c, c)$ lors du *round* $r+1$. D'après la définition de l'algorithme, il est clair que $V_p[q] = v_q$ à la fin du *round* $r+1$.

2. $r = n-1$. Alors c a reçu v_q pour la première fois lors du *round* $n-1$. D'après la définition de l'algorithme, chaque processus peut relayer une nouvelle valeur dans son Δ lors d'un *round* au plus. Comme c a reçu v_q pour la première fois lors du *round* $n-1$, c'est que cette valeur a été relayée à chacun des $n-1$ *rounds*, et donc nécessairement par $n-1$ processus différents. Par conséquent, $n-1$ processus ont reçu v_q avant c. Il s'agit nécessairement des $n-1$ processus de $\Pi \setminus \{c\}$, qui contient p (puisque $p \neq c$). Comme p affecte la valeur v_q à $V_p[q]$ avant de relayer v_q, il est clair que $V_p[q] = v_q$ à la fin de la phase 1. $\diamond$

Lemme 8
Pour tout processus $p \in \Pi_2$, $V_c = V_p$ à la fin de la phase 2.

Preuve
Soit $p \in \Pi_2$ et $q \in \Pi$ quelconque. Il nous faut montrer que $V_p[q] = V_c[q]$ à la fin de la phase 2. Considérons alors les deux cas suivants :

1. $V_c[q] = v_q$ à la fin de la phase 1. D'après le lemme 7, pour tout processus $p' \in \Pi_1$ (p et c sont dans Π_1), on a $V_{p'}[q] = v_q$ à la fin de la phase 1. Par conséquent, on a $V[q] = v_q$ pour tous les V envoyés lors de la phase 2. On a donc $V_p[q] = V_c[q] = v_q$ tout au long de la phase 2.

2. $V_c[q] = \perp$ à la fin de la phase 1. Comme $c \notin \mathcal{D}_p$, p attend et finit par recevoir V_c lors de la phase 2. Comme $V_c[q] = \perp$, p affecte la valeur $\perp$ à $V_p[q]$. $\diamond$

Lemme 9
Pour tout processus $p \in \Pi_2$, $V_p[c] = v_c$.

Preuve
On a clairement $V_c[c] = v_c$ à la fin de la phase 1. D'après le lemme 7, pour tout $q \in \Pi_1$, $V_q[c] = v_c$ à la fin de la phase 1. Par conséquent, aucun processus n'envoie un V où $V[c] = \perp$ lors de la phase 2, et il est donc clair que pout tout $p \in \Pi_2$, $V_p[c] = v_c$ à la fin de la phase 2. $\diamond$

Théorème 6.2.1 *([CT94])*
L'algorithme présenté précédemment résout le consensus dans tout système asynchrone où $f \leq (n-1)$ en utilisant tout détecteur de défaillance dans les classes $\mathcal{S}$ ou $\mathcal{P}$.

Preuve
Comme $\mathcal{P} \subseteq \mathcal{S}$, il suffit de montrer que cet algorithme résout bien le problème du consensus en utilisant tout FD $\mathcal{D} \in \mathcal{S}$.
D'après la définition même de l'algorithme, il est clair qu'un processus ne peut pas décider plus d'une fois, et que par conséquent la propriété d'intégrité uniforme est satisfaite.
D'après le lemme 5, tout processus correct atteint tôt ou tard la phase 3, et comme d'après le lemme 9 le vecteur V_p de tout processus correct p contient au moins une composante différente de $\perp$ (puisque $V_p[c] = v_c \neq \perp$), il est clair que tout processus correct décide une valeur (différente de $\perp$) lors de la phase 3, et la propriété de terminaison est bien satisfaite.
D'après le lemme 8, tous les processus qui atteignent la phase 3 ont le même vecteur V. Par conséquent, tous les processus corrects décident la même valeur, et la propriété d'accord est bien satisfaite.
Enfin, d'après le lemme 4, toute valeur placée dans un vecteur V a été proposée par un des processus, et donc la propriété de validité est également satisfaite. $\diamond$

6.2.2 Classes $\diamond\mathcal{S}$ et $\diamond\mathcal{P}$

Soit $\mathcal{D} \in \diamond\mathcal{S}$ (c'est-à-dire que le FD $\mathcal{D}$ est fortement complet et faiblement précis "à la longue").

Le principe de l'algorithme proposé est le suivant : chaque processus répète quatre phases jusqu'à ce qu'il décide, chaque itération constituant un *round*. A chaque *round* r un processus particulier, $c = (r \bmod n) + 1$, désigné comme étant le coordinateur de ce *round*, joue un rôle particulier. Lors de la première phase, tous les processus envoient la valeur qu'ils veulent voir décidée à ce coordinateur. Lors de la deuxième phase ce dernier récolte ces messages et en déduit une nouvelle valeur, qu'il tente d'imposer aux autres processus. Lors de la troisième phase, les processus répondent au coordinateur. Si une majorité absolue de processus adopte cette valeur, alors le coordinateur diffuse la valeur de décision, qui sera tôt ou tard décidée par tous les processus corrects.

Cette technique, qui consiste à balayer circulairement tous les processus, est appelée technique du "coordinateur tournant" (*rotating coordinator*).

Chaque processus p exécute l'algorithme suivant :

proposer(v):
{
  $estimation \leftarrow v$
  $\acute{e}tat \leftarrow$ indécis
  $r \leftarrow 0$ `/* numéro du` *round* `*/`
  $ts \leftarrow 0$ `/*` *time-stamp* `*/`

  tant que $\acute{e}tat =$ indécis
  {
    incrémenter r
    $c \leftarrow (r \bmod n) + 1$ `/*` c `est l´actuel coordinateur */`

    `/* Phase 1 */`

    envoyer$(c,(p, r, estimation, ts))$

    `/* Phase 2 */`

    si $p = c$
    {
      attendre $\lceil \frac{(n+1)}{2} \rceil$ messages de la forme $(q, r, estimation_q, ts_q)$
      $t \leftarrow$ le plus grand ts_q parmi les messages reçus précédemment
      $estimation \leftarrow$ un $estimation_q$ d'un message reçu où $ts_q = t$
      envoyer à tous $(p, r, estimation)$
    }

    `/* Phase 3 */`

    attendre un message de la forme $(c, r, estimation_c)$ **ou** $c \in \mathcal{D}$
    si on a reçu $(c, r, estimation_c)$
    {
      $estimation \leftarrow estimation_c$
      $ts \leftarrow ts_c$
      envoyer$(c,(r, \texttt{ACK}))$
    }
    sinon
      envoyer$(c,(r, \texttt{NACK}))$

`/* Phase 4 */`

si $p = c$
{
attendre $\lceil\frac{(n+1)}{2}\rceil$ messages de la forme $(q, r, \texttt{ACK})$ ou $(q, r, \texttt{NACK})$
si on a reçu $\lceil\frac{(n+1)}{2}\rceil$ messages de la forme $(q, r, \texttt{ACK})$
R-broadcast$((p, r, estimation))$
}
}
}

et il faut également préciser :

quand R-deliver(m)
{
si *état* = indécis
{
extraire q, r_q et $estimation_q$ de m
décide$(estimation_q)$
état $\leftarrow$ décidé
}
}

On dira que l'on se trouve à l'étape (r, p) lorsqu'on est à la phase p ($p \in \{1, 2, 3, 4\}$) du *round* r de l'algorithme précédent.

Lemme 10
Deux processus ne peuvent pas décider des valeurs différentes.

Preuve
Si aucun processus ne décide, le lemme est clairement vrai. Si un processus décide, c'est qu'un coordinateur a diffusé un message de décision en utilisant **R-broadcast**(). Ce coordinateur a donc nécessairement reçu $\lceil\frac{(n+1)}{2}\rceil$ accusés de réception de la forme $(\cdots, \cdots, \texttt{ACK})$. Soit r le premier *round* lors duquel $\lceil\frac{(n+1)}{2}\rceil$ messages de cette forme sont envoyés (lors de la phase 3), et soit c le coordinateur lors de ce *round* r. Soit enfin $estimation_c$ l'estimation que c a envoyée à l'étape $(r, 2)$.
Nous allons montrer la propriété suivante : pour tout *round* $r' \geq r$, si un coordinateur c' envoie $estimation_{c'}$ à l'étape $(r', 2)$, alors $estimation_{c'} = estimation_c$. La preuve se fait par récurrence sur $r' \geq r$. La propriété est clairement vraie pour $r' = r$ (car alors $c' = c$). Supposons maintenant que la propriété soit vraie pour tout $r \leq r' < k$. Soit c_k le coordinateur lors du *round* k. Nous allons montrer que la propriété est également vraie pour $r' = k$, c'est-à-dire que si c_k envoie $estimation_{c_k}$ à l'étape $(k, 2)$, alors $estimation_{c_k} = estimation_c$. D'après la définition de l'algorithme, il est clair que si c_k envoie $estimation_{c_k}$ à l'étape $(k, 2)$, c'est qu'il

a précédemment reçu $\lceil\frac{(n+1)}{2}\rceil$ estimations. Comme il n'y a au total que n processus, il existe donc un certain processus tel que d'une part p a envoyé $(p, r, \texttt{ACK})$ à l'étape $(r, 3)$ et d'autre part c_k a reçu $(p, k, estimation_p, ts_p)$ à l'étape $(k, 2)$. Comme p a envoyé $(p, r, \texttt{ACK})$ à c lors de l'étape $(r, 3)$, $ts_p = r$ à la fin de cette même étape. Comme ts_p est une fonction croissante du temps, $ts_p \geq r$ à l'étape $(k, 1)$. Par conséquent, c_k reçoit $(p, k, estimation_p, ts_p)$ où $ts_p \geq r$ à l'étape $(k, 2)$. On voit facilement que c_k n'a pas pu recevoir de message $(q, k, estimation_q, ts_q)$ où $ts_q \geq k$ à l'étape $(k, 2)$. Soit t le plus grand des ts_q présents dans les messages $(q, k, estimation_q, ts_q)$ reçus à l'étape $(k, 2)$. On a $r \geq t < k$. Lors de l'étape $(k, 2)$, c_k exécute l'affectation "$estimation \leftarrow estimation_q$", où $(q, k, estimation_q, ts_q)$ est l'un des messages reçus au début de cette étape. D'après la définition de l'algorithme, il est clair que l'estimation adoptée par q lors de l'étape $(ts_q, 3)$ est $estimation_{c_k} = estimation_q$. Par conséquent, le coordinateur du *round* ts_q a envoyé $estimate_q$ à q lors de l'étape $(ts_q, 2)$. Puisque $r \leq ts_q < k$ on a, avec l'hypothèse de récurrence, $estimation_q = estimation_c$. Par conséquent, c_k effectue "$estimation \leftarrow estimation_c$" lors de l'étape $(k, 2)$. La propriété est donc prouvée.
Finalement on montre que si un processus décide une valeur, alors il décide $estimation_c$. Supposons qu'un certain processus p prenne connaissance d'un message $(q, r_q, estimation_q)$ lors d'un appel à **R-deliver**(), et décide donc la valeur $estimation_q$. Le processus q a nécessairement diffusé le message $(q, r_q, estimation_q)$ en utilisant **R-broadcast**() lors de l'étape $(r_q, 4)$. D'après la définition de l'algorithme, q a dû recevoir $\lceil\frac{(n+1)}{2}\rceil$ messages de la forme $(\cdots, r_q, \texttt{ACK})$ à l'étape $(r_q, 4)$. Par définition de r, on a $r \leq r_q$ et donc $estimation_q = estimation_c$. ◇

Lemme 11
Tout processus correct finit tôt ou tard par décider une valeur.

Preuve
Si un des processus correct décide, c'est qu'il a pris connaissance d'un message de décision lors d'un appel à **R-deliver**(), et, d'après la propriété d'accord de la diffusion fiable, tous les processus corrects prendront connaissance de ce même message et décideront donc.
Montrons tout d'abord par l'absurde qu'aucun processus correct ne peut rester bloqué lors d'une des attentes de messages. Supposons donc qu'un processus se bloque lors d'une attente de messages. Soit r le premier *round* lors duquel un processus correct se bloque. Comme tous les processus corrects ont nécessairement passé l'étape $(r, 1)$, ils ont tous envoyé un message de la forme $(\cdots, r, estimation, \cdots)$ au coordinateur c du *round* r. Comme on a une majorité de processus corrects (par hypothèse), au moins $\lceil\frac{(n+1)}{2}\rceil$ messages de cette forme sont envoyés à c. Deux situations sont possibles :

1. c finit par recevoir ces messages et répond en envoyant le message $(c, r, estimation_c)$. Par conséquent, c passe à la phase 3 sans se bloquer,

2. c défaille.

Dans la première situation, tout processus correct finit par recevoir $(c, r, estimation_c)$. Dans le second cas, puisque $\mathcal{D}$ est fortement complet, pour tout processus correct p il y a une date

après laquelle c est continuellement suspecté par p (i.e. $c \in \mathcal{D}_p$). Par conséquent, dans les deux cas, aucun processus correct ne peut se bloquer lors de la phase 3. Comme il y a au moins $\lceil\frac{(n+1)}{2}\rceil$ processus corrects, c ne peut pas se bloquer lors de la phase 4. Par conséquent tous les processus corrects terminent le *round* r sans se bloquer, ce qui est en contradiction avec l'hypothèse initiale (r est le premier *round* lors duquel un processus correct se bloque).
Montrons enfin, toujours par l'absurde, qu'il est impossible qu'aucun processus correct ne décide. Supposons donc qu'aucun processus correct ne décide. Comme $\mathcal{D}$ est faiblement précis "à la longue", il existe une date t et un processus correct q tels qu'aucun processus correct ne suspecte plus q après la date t. Donc tous les processus qui suspectent q après la date t vont tôt ou tard défaillir et il existe une date t' telle qu'aucun processus n'envoie de message de la forme $(\cdots, r, \texttt{NACK})$ à q alors que ce dernier est le coordinateur du *round* r. Comme de plus aucun processus ne peut rester bloqué lors d'une des attentes de messages, il y a nécessairement un *round* r tel que :

1. tous les processus corrects parviennent au *round* r après la date t' (après laquelle aucun processus correct ne suspecte plus q),
2. q est le coordinateur du *round* r.

A l'étape $(r, 1)$, tous les processus corrects envoient leurs estimations à q. A l'étape $(r, 2)$, q reçoit donc $\lceil\frac{(n+1)}{2}\rceil$ estimations, et envoie $(q, r, estimation_q)$ à tous les processus. A l'étape $(r, 3)$, puisque q n'est suspecté par aucun des processus corrects, tout processus correct attend l'estimation de q, finit par la recevoir, et y répond par un message de la forme $(\cdots, r, \texttt{ACK})$. De plus, aucun processus n'envoie de message de la forme $(\cdots, r, \texttt{NACK})$ à q (car q n'est suspecté par aucun processus "vivant" après la date t'). Par conséquent, à l'étape $(r, 4)$, q reçoit $\lceil\frac{(n+1)}{2}\rceil$ messages de la forme $(\cdots, r, \texttt{ACK})$, et diffuse le message $(q, r, estimation_q)$ en utilisant **R-broadcast**(). Comme q est un processus correct, et d'après la propriété de validité de la diffusion fiable, tout processus correct finit par prendre connaissance de ce message et décide, d'où une contradiction avec l'hypothèse qu'aucun processus ne décide. ◇

Théorème 6.2.2 *([CT94])*
L'algorithme présenté précédemment résout le problème du consensus dans tout système asynchrone où $f < \lceil\frac{n}{2}\rceil$ *en utilisant tout détecteur de défaillance dans les classes* $\diamond\mathcal{S}$ *ou* $\diamond\mathcal{P}$.

Preuve
Comme $\diamond\mathcal{P} \subseteq \diamond\mathcal{S}$, il suffit de montrer que cet algorithme résout bien le problème du consensus en utilisant tout FD $\mathcal{D} \in \diamond\mathcal{S}$.
Les propriétés de terminaison et d'accord (uniforme) découlent respectivement des lemmes 11 et 10.
Les deux autres propriétés découlent, elles, de la définition même de l'algorithme. En effet, il est clair qu'aucun processus ne peut décider plus d'une fois, et que par conséquent l'intégrité uniforme est satisfaite. De plus, toutes les estimations sont des valeurs proposées, et la validité est également satisfaite. ◇

6.3 Résultats

Classe	$\mathcal{P} \cong \mathcal{Q}$	$\mathcal{S} \cong \mathcal{W}$	$\diamond\mathcal{P} \cong \diamond\mathcal{Q}$	$\diamond\mathcal{S} \cong \diamond\mathcal{W}$
Soluble	oui	oui	oui	oui
Condition	$f \leq (n-1)$	$f \leq (n-1)$	$f < \lceil \frac{n}{2} \rceil$	$f < \lceil \frac{n}{2} \rceil$

Figure 6: *Le consensus en utilisant les classes de FD de base.*

Théorème 6.3.1
Le consensus peut être résolu dans les systèmes asynchrones en utilisant des détecteurs de défaillance dans les classes :

- $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$ *ou* $\mathcal{W}$ *si et seulement si* $f \leq (n-1)$,
- $\diamond\mathcal{P}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$ *ou* $\diamond\mathcal{W}$ *si et seulement si* $f < \lceil \frac{n}{2} \rceil$.

Preuve
D'après le théorème 6.2.1, et comme $\mathcal{Q} \cong \mathcal{P}$ et $\mathcal{W} \cong \mathcal{S}$, le problème du consensus peut être résolu avec les classes $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$ et $\mathcal{W}$ (avec $f \leq (n-1)$).
D'après le théorème 6.2.2, et comme $\diamond\mathcal{Q} \cong \diamond\mathcal{P}$ et $\diamond\mathcal{W} \cong \diamond\mathcal{S}$, le problème du consensus peut être résolu avec les classes $\diamond\mathcal{P}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$ et $\diamond\mathcal{W}$ si $f < \lceil \frac{n}{2} \rceil$.
Il reste donc à montrer que cette condition est nécessaire, et alors le théorème sera démontré. En fait, on utilise la même technique de partitionnement de réseau que dans le cas du consensus probabiliste [BO83], [BT85]. Raisonnons par l'absurde en supposant qu'il existe des algorithmes résolvant le consensus avec $f \geq \lceil \frac{n}{2} \rceil$. Partitionnons alors l'ensemble Π des processus en deux sous-ensembles Π_0 et Π_1 comportant chacun au moins f processus. Considérons alors les deux scénarios (symétriques) suivants :

1. Tous les processus proposent la valeur 0, tous les processus de Π_0 sont corrects et tous ceux de Π_1 sont initialement défaillants. Comme il n'y a pas plus de f processus défectueux, par hypothèse et d'après les propriétés de terminaison et de validité, il existe une exécution où tous les processus de Π_0 décident la valeur 0 au bout d'un certain temps t_0.

2. Tous les processus proposent la valeur 1, tous les processus de Π_1 sont corrects et tous ceux de Π_0 sont initialement défaillants. Comme il n'y a pas plus de f processus défectueux, par hypothèse et d'après les propriétés de terminaison et de validité, il existe une exécution où tous les processus de Π_1 décident la valeur 1 au bout d'un certain temps t_1.

Considérons alors le troisième scénario suivant : tous les processus de Π_0 (resp. Π_1) proposent la valeur 0 (resp. 1), et tous les processus sont corrects. On retarde la transmission des messages entre Π_0 et Π_1 d'au moins $t = \max(t_0, t_1)$ (on peut retarder des messages pendant un temps fini puisqu'on est dans un système asynchrone). Il existe une exécution où tous les processus de Π_0 se comportent comme dans le scénario 1 et où tous les processus de Π_1 se comportent comme dans le scénario 2 (car jusqu'à la date t les processus de Π_0 (resp. de Π_1)

ne peuvent pas différencier le scénario 3 du scénario 1 (resp. 2)). Après la date t, on transmet les messages qui étaient retardés. Cette exécution est bien correcte, mais la propriété d'accord n'est pas respectée (puisque les processus de Π_0 ont décidé la valeur 0 et ceux de Π_1 la valeur 1). Reprenons alors le raisonnement précédent en ajoutant un FD $\mathcal{D}$ se comportant ainsi :

- dans le scénario 1, $\mathcal{D}$ suspecte tous les processus de Π_0,
- dans le scénario 2, $\mathcal{D}$ suspecte tous les processus de Π_1,
- dans le scénario 3, $\mathcal{D}$ suspecte tous les processus jusqu'à la date $t = \max(t_0, t_1)$, et plus aucun processus après cette date.

Il existe bien un tel FD $\mathcal{D} \in \diamond\mathcal{P}$, puisque dans les trois scénarios considérés, tout processus correct finit par suspecter continuellement tout processus défaillant, et qu'il existe une date après laquelle aucun processus correct n'est jamais suspecté. On peut ainsi montrer que la condition $f < \lceil \frac{n}{2} \rceil$ est nécessaire pour la résolution du consensus avec la classe $\diamond\mathcal{P}$. Et comme $\diamond\mathcal{Q} \cong \diamond\mathcal{P}$, $\diamond\mathcal{P} \succeq \diamond\mathcal{S}$ et $\diamond\mathcal{W} \cong \diamond\mathcal{S}$, cette condition est également nécessaire pour les classes $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$ et $\diamond\mathcal{W}$. $\diamond$

6.4 Cas du consensus probabiliste

On a vu dans le chapitre 4 qu'il existait des algorithmes déterministes réalisant le consensus probabiliste, et cela sans faire appel aux détecteurs de défaillance. Par conséquent, il est clair que le problème du consensus probabiliste, ne nécessitant aucun détecteur de défaillance pour être résolu, peut se faire *a fortiori* avec n'importe quel détecteur de défaillance, et donc toute classe de détecteurs, même la classe vide $\emptyset$...

Mais en rapprochant les résultats trouvés ici pour le consensus (Figure 6) et le théorème 4.5.1, on constate que le consensus probabiliste et le consensus avec FD finalement précis sont réalisables si et seulement si $f < \lceil \frac{n}{2} \rceil$. On peut donc se demander si on ne pourrait pas affaiblir encore plus la propriété de précision des FD afin de réutiliser l'algorithme du "coordinateur tournant" utilisé pour le consensus pour résoudre uniquement le consensus probabiliste.

On a introduit la précision "à la longue" en disant que l'on voulait que la précision soit tôt ou tard satisfaite. On définit maintenant la précision probabiliste : la probabilité pour que la précision ne soit pas satisfaite au bout d'un temps fini est nulle.

Plus précisemment, on définit deux nouvelles propriétés de précision, encore plus faibles que les précédentes :

• la *précision forte probabiliste* : la probabilité pour qu'il existe une date après laquelle aucun processus correct n'est suspecté par un processus correct vaut 1. Formellement, $\mathcal{D}$ satisfait la précision forte probabiliste si et seulement si :

$$\forall F, \forall H \in \mathcal{D}(F), \Pr(\exists t \in \mathcal{T} : \forall t \geq t', \underline{\forall p, q \in correct(F)}, p \notin H(q, t')) = 1$$

• la *précision faible probabiliste* : la probabilité pour qu'il existe une date après laquelle un certain processus correct n'est plus jamais suspecté par aucun processus correct est 1. Formellement, $\mathcal{D}$ satisfait la précision faible probabiliste si et seulement si :

$$\forall F, \forall H \in \mathcal{D}(F), \Pr(\exists t \in \mathcal{T} : \underline{\exists p \in correct(F)}, \forall t \geq t', \underline{\forall q \in correct(F)}, p \notin H(q, t')) = 1$$

Remarque

Précision (perpétuelle) $\Rightarrow$ Précision "à la longue" $\Rightarrow$ Précision probabiliste

mais les implications inverses sont clairement fausses.

En combinant ces 2 propriétés avec les 2 propriétés de complétude, on obtient donc 4 nouvelles classes : $\circ\mathcal{P}$, $\circ\mathcal{Q}$, $\circ\mathcal{S}$ et $\circ\mathcal{W}$.

Complétude	**Précision**	
	Forte Probabiliste	Faible Probabiliste
Forte	$\circ\mathcal{P}$	$\circ\mathcal{S}$
Faible	$\circ\mathcal{Q}$	$\circ\mathcal{W}$

Figure 7: *4 nouvelles classes de FD.*

On a évidemment $\circ\mathcal{P} \cong \circ\mathcal{Q}$ et $\circ\mathcal{S} \cong \circ\mathcal{W}$, et comme $\circ\mathcal{P} \subseteq \diamond\mathcal{P}$ et $\circ\mathcal{S} \subseteq \diamond\mathcal{W}$, on a $\circ\mathcal{P} \succeq \diamond\mathcal{P}$ et $\circ\mathcal{S} \succeq \diamond\mathcal{W}$ (en fait, on a même $\circ\mathcal{P} \succ \diamond\mathcal{P}$ et $\circ\mathcal{S} \succ \diamond\mathcal{W}\ldots$).

Il est clair que l'algorithme du "coordinateur tournant" associé à un FD dans la classe $\circ\mathcal{S}$ résout le problème du consensus probabiliste si et seulement si $f < \lceil \frac{n}{2} \rceil$ (il suffit de reprendre la preuve faite pour le théorème 6.2.2 en introduisant les probabilités). Mais ce qui est intéressant, c'est que dans ces conditions il ne résout plus le problème du consensus. En effet, il est dit dans [CHT94] que le plus faible FD utilisable pour résoudre le problème du consensus est un certain FD $\diamond\mathcal{W}_0 \in \mathcal{W}$, qui n'appartient pas à $\circ\mathcal{S}$.

En fait, on verra dans le chapitre 7 que lorsqu'on veut réaliser en pratique un FD, la propriété de précision n'est obtenue que de manière probabiliste, qui peut dépendre de l'état du système (charge,...). Suivant le comportement du FD, on réalise soit le consensus, soit un consensus qui termine avec probabilité τ. Si $\tau = 1$, on retombe sur le consensus probabiliste. Mais rien n'empêche *a priori* d'avoir $\tau = 0\ldots$

Chapitre 7

Implémentation

Il est intéressant de tester en pratique les différents algorithmes de consensus et de consensus probabiliste exposés précédemment. Mais avant d'implémenter ces algorithmes, il a fallu réaliser l'implémentation d'un système distribué de communications fiables avec détecteurs de défaillance non fiables, c'est-à-dire l'implémentation pratique du modèle théorique. Cette implémentation a été baptisée **rcUFD** (*reliable communication with Unreliable Failure Detection*), et a été écrite en C++ sous UNIX, notamment grâce aux informations contenues dans [Rif94]. Ce chapitre présente les principales caractéristiques du système rcUFD, et expose rapidement quelques résultats obtenus par les algorithmes en utilisant ce système (des sources extraites de cette implémentation sont fournies dans l'annexe D).

7.1 Choix généraux

Le système rcUFD a été réalisé en C++ sous UNIX, et testé avec succès sous SunOS, Solaris et Linux.

Il se présente sous la forme d'un fichier en-tête `rcufd.h` qui donne accès aux fonctionnalités d'une bibliothèque `librcufd.a`. Ainsi, si on veut compiler un programme `exemple.c` utilisant le système rcUFD, il suffit de faire : `cc exemple.c -lrcufd -o exemple`

Outre la facilité d'accès à Internet (grâce aux *sockets*), UNIX possède des propriétés très intéressantes, et notamment :

- les appels systèmes (et notamment `read()` et `write()`) sont atomiques,
- l'état d'un processus local peut être connu à tout moment de manière fiable (en utilisant `kill()`).

Nous verrons dans la suite comment toutes ces propriétés on été utilisées afin de garantir le respect du modèle.

7.2 Description

Le but du système rcUFD est de fournir en pratique les fonctionnalités requises en théorie. On lance sur chaque machine $\mathcal{M}_i$ du réseau un "démon" UNIX local $rcUFD_i$. Tous ces démons s'interconnectent lors d'une phase d'initialisation, puis ils attendent et traitent les requêtes des processus clients.

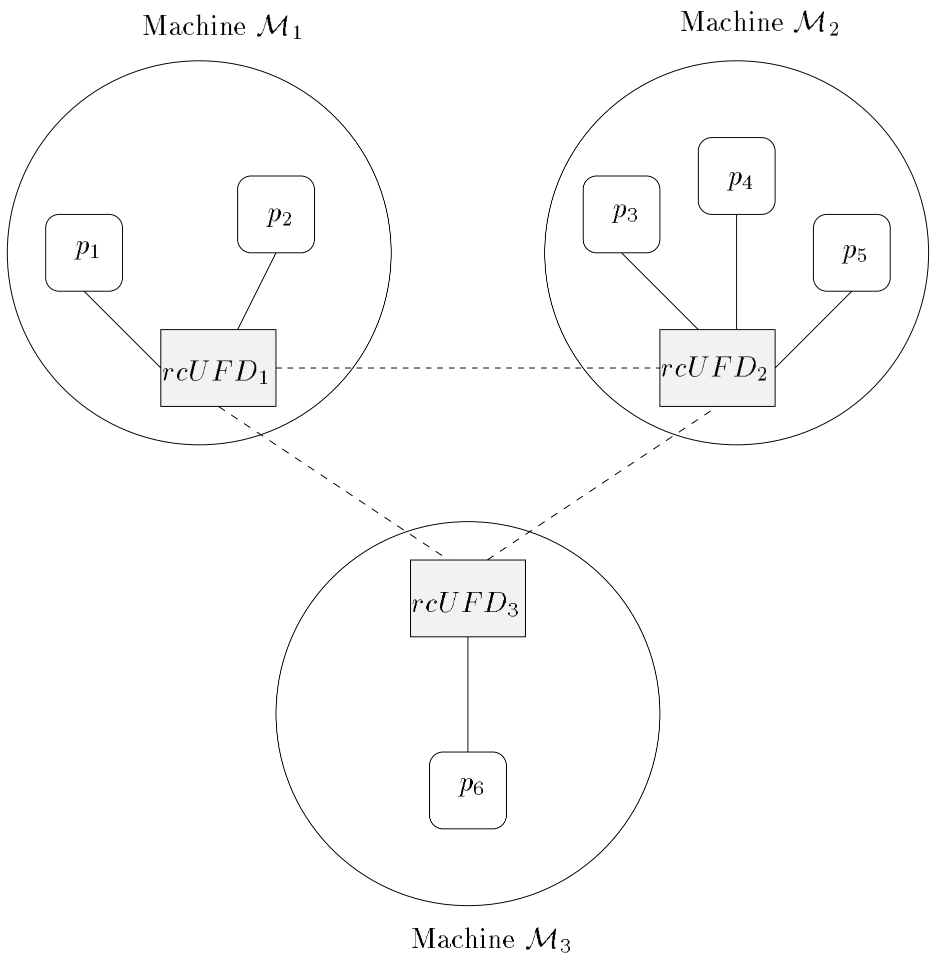

Figure 8: *6 processus répartis sur 3 machines.*

7.2.1 Mécanisme d'abonnement

Quand n processus répartis sur plusieurs machines veulent s'échanger des messages, le problème de l'identification des processus se pose naturellement. On peut tout d'abord remarquer que deux processus distincts doivent évidemment posséder des noms différents (afin d'éviter tout conflit). Sous UNIX, les processus localisés sur une même machine ont des *pid* différents, et comme sur Internet les machines ont toutes des adresses différentes, `pid@machine` est une identification unique des processus. Cependant, il faut que l'identification d'un processus ne dépende ni de son *pid* ni de sa localisation sur le réseau. En effet, si un processus p_i veut envoyer un message à un autre processus p_j il doit connaître le nom de p_j, or il n'a *a priori* aucun moyen de connaître la localisation de p_j, et encore moins son *pid* (qui varie d'une exécution à l'autre). La solution proposée est la suivante : un abonnement avec un nom d'immatriculation (*registration name*) défini par le processus lui-même.

Pour mettre en oeuvre ce mécanisme d'abonnement, on utilise les deux primitives suivantes :

• `void` **rcufd_register** `(char *name)`
Pour pouvoir accèder au système rcUFD, un processus doit s'abonner en appelant cette primitive avec le nom d'immatriculation qui lui sera attribué. Si deux processus veulent s'abonner avec le même nom, le système considèrera qu'il s'agit d'un faute grave (*byzantine*), qui sort de notre cadre d'étude.

• `void` **rcufd_leave** `(void)`
Un processus abonné doit résilier son abonnement lorsqu'il veut quitter le système rcUFD. Il utilise pour cela cette primitive. Cette résiliation est toutefois automatique en cas de panne du processus.

Une fois qu'un processus s'est abonné auprès du système rcUFD, il a alors accès à plusieurs autres primitives...

7.2.2 Primitives de communication

• `void` **rcufd_send** `(char *name, char *message)`
Un processus appelle cette primitive lorsqu'il veut envoyer un message `message` à un processus ayant pour nom d'immatriculation `name`. Le système garantit que ce message arrivera intact à destination au bout d'un temps fini. Si un message est envoyé à un processus qui n'est pas encore abonné, il est conservé dans un tampon et est livré dès que cela devient possible.

• `char` ***rcufd_receive** `(void)`
Un processus qui décide de recevoir un message utilise cette primitive. Lorsqu'il n'y a pas de message pour lui lors de sa demande, la valeur `NULL` lui est retournée. Sinon, il obtient en retour un message (sous forme d'une chaîne de caractères).

7.2.3 Appel à l'oracle

• `int` **rcufd_suspected** `(char *name)`
Un processus appelle cette primitive lorsqu'il veut savoir si un processus est suspecté d'être

défaillant. La valeur retournée est une valeur booléenne qui est la réponse à la question : "`name` est-il actuellement suspecté ?". Il faut bien noter que les détecteurs de défaillance ne sont pas fiables en général, et que l'exactitude de cette réponse est toute relative.

7.3 Respect du modèle

L'implémentation respecte évidemment les caractéristiques du modèle d'implémentation vu en 2.2.4.2, c'est-à-dire :

- la fiabilité des communications, qui repose sur le protocole TCP (*Transmission Control Protocol*), ce protocole assurant également l'absence d'interblocages grâce à un mécanisme de tampons en entrée et en sortie qui permettent d'envoyer un message de manière non bloquante,
- l'atomicité des primitives **envoyer**() et **recevoir**(),
- le comportement de l'ensemble M des messages en transit.

Les seules défaillances tolérées ici sont les pannes franches de processus, et si le système détecte une faute qui sort du cadre d'étude, il signale une erreur fatale et "se suicide". Parmi ces fautes, on trouve entre autres la rupture de la connexité entre machines (connexion non fiable, effondrement du système sur une des machines, destruction du démon local sur une des machines), deux processus distincts essayant de s'abonner sous le même nom d'immatriculation, etc.

Le soucis de conformité avec le modèle a été constant, parfois au dépend de la recherche d'une performance maximale.

7.4 Implémentation des détecteurs de défaillance

L'implémentation de détecteurs de défaillance doit avoir de bonnes propriétés de détection (complétude, précision) tout en étant la plus efficace possible.

Le FD qui a été réalisé est "localement parfait" en ce sens que si un processus questionne son FD local pour un processus situé sur la même machine, alors la réponse qu'il obtient est exacte.

On s'intéresse maintenant au comportement global du FD. Chaque FD local diffuse périodiquement la liste des processus locaux non défaillants à l'attention de chacun des FD distants. Chaque FD écoute par ailleurs les messages en provenance des autres FD, et possède un *time-out*, c'est-à-dire un délai maximum d'attente. Si un message attendu n'arrive pas à temps, les processus de la machine en cause sont suspectés de défaillance. Si ce message arrive par la suite, le FD reconnaît son erreur et réhabilite les processus concernés, du moins jusqu'à la prochaine échéance du *time-out*.

La propriété de complétude forte est clairement satisfaite. En effet, tout processus qui défaille sera soit "dénoncé" par son FD local, qui diffusera périodiquement cette information : si cette information est reçue par les autres, ils suspectent définitivement le processus, et sinon le *time-out* arrive à écheance et le processus est quand même suspecté.

En revanche, le FD ne possède *a priori* aucune des propriétés de précision énoncées dans les chapitres précédents (ce qui est normal car sinon on contredirait le résultat d'impossibilité [FLP85]). Cependant, le but est de s'en approcher le plus possible, par des méthodes statistiques.

Soit X une variable aléatoire. On définit :

- l'espérance mathématique : $E(X) = (\sum_{i=1}^{n} x_i)/n$
- la variance : $V(X) = (\frac{1}{n} \cdot \sum_{i=1}^{n} {x_i}^2) - E(X)^2$
- l'écart type : $\sigma = \sqrt{V(X)}$

L'inégalite de Bienaymé-Tchebychev

$$\forall \epsilon > 0, \quad probabilit\acute{e}(|X - E(X)| \geq \epsilon) \leq \frac{\sigma^2}{\epsilon^2}$$

donne un majorant de la probabilité pour que la variable aléatoire X s'écarte, en valeur absolue, de son espérance mathématique $E(X)$ de plus de ϵ, en fonction de $\epsilon > 0$ et de l'écart type σ de X.

En prenant $\epsilon = h\sigma$, avec $h > 0$,
puisque $probabilit\acute{e}(|X - E(X)| < \epsilon) = 1 - probabilit\acute{e}(|X - E(X)| \geq \epsilon)$, on obtient :

$$probabilit\acute{e}(|X - E(X)| < h\sigma) \geq 1 - \frac{1}{h^2}$$

Ainsi, à chaque réception, chaque FD local utilise ce résultat pour adapter au mieux son *time-out*. On peut décider du taux de réussite du FD, $1 - \frac{1}{h^2}$, et le *time-out* $t = E(X) + h\sigma$ s'en déduit.

La valeur des la précision utilisée en pratique pour les tests est de 95%.

La validation de l'efficacité pratique de cette méthode a fait l'objet d'un important travail par Nathalie Geoffrin lors d'un stage dirigé par Sam Toueg, au sein de l'équipe du projet REFLECS de l'INRIA. Les performances ont été mesurées sur un réseau local à l'Ecole Polytechnique. Les résultats peuvent être trouvés dans [Geo95].

7.5 Tests et résultats

Il faut bien noter que tous les résultats exposés dans ce qui suit sont très dépendants du système d'exploitation sur lequel les algorithmes sont exécutés, ainsi que de la charge des machines et de la saturation du réseau les reliant. Cependant, par soucis d'exactitude, les tests ont été réalisés sur trois systèmes différents, à des moments différents, et sous des conditions différentes, et cela un grand nombre de fois (une centaine en pratique). Les valeurs considérées par la suite sont celles qui sont apparues le plus fréquemment. Ce sont en général les valeurs obtenues avec le système Linux, car les résultats se sont avérés être très réguliers, et le contrôle de la charge des machines et de la saturation du réseau était très facilité. En fait, la détermination de jeux de tests représentatifs pourrait à elle seule faire l'objet d'une étude approfondie. Les résultats "bruts" des tests sont fournis dans l'annexe C.

7.5.1 Algorithme de consensus

L'algorithme de consensus retenu pour les tests est le "coordinateur tournant" vu au chapitre 6. Ce n'est certes pas le plus rapide des algorithmes proposés, mais il présente l'énorme avantage de ne nécessiter que des propriétés de détection très faibles. Il fonctionne parfaitement, même si en pratique le FD utilisé ne possède pas la bonne propriété de précision. Cela provient du fait que si le FD ne se comporte pas correctement, l'algorithme perd en vivacité (*liveness*) mais pas en sécurite (*safety*). Il peut donc ne pas décider, mais il ne prend en tout cas jamais de mauvaise décision. En fait, il se bloque momentanément et se débloque dès que le FD se comporte à nouveau correctement. Comme en pratique la probabilité que le FD se comporte mal en permanence est *a priori* faible (du moins pour un système normalement chargé), l'algorithme progresse pendant les périodes "favorables" et finit par décider. En pratique, on a donc un FD qui comporte comme un FD de la classe $\diamond\mathcal{S}$ (ou plutôt $\circ\mathcal{S}$) par intervalles de temps. Il faut également noter que si la propriété de complétude n'est pas respectée, l'algorithme se "bloque" lors d'une attente de message, alors que si la précision n'est pas respectée, on observe une attente active, chaque processus parcourant sans cesse circulairement la liste de tous les processus pour chercher un coordinateur valide.

7.5.2 Algorithme de consensus probabiliste

L'algorithme de consensus probabiliste retenu pour les tests est l'algorithme de Bracha et Toueg pour les raisons vues au chapitre 4 (c'est une version très améliorée de l'algorithme de Ben-Or...). Dans le meilleur cas, le consensus probabiliste est réalisé en deux étapes. Cependant le nombre de messages échangés est toujours très grand, et cela même dans les cas les plus favorables.

7.5.3 Comparaison consensus / consensus probabiliste

On constate que le nombre de messages échangés lors de l'algorithme de Bracha et Toueg ne varie pas tellement en cas de pannes de processus, ce qui est normal. L'algorithme du "coordinateur tournant" doit, lui, échanger plus de messages afin de trouver un coordinateur. Cependant, le nombre de messages échangés reste tout à fait raisonnable, alors que dans le cas de l'algorithme de Bracha et Toueg ce nombre était élevé même dans les cas les plus favorables. En revanche, on peut trouver étonnant que le "coordinateur tournant" soit un peu plus lent alors qu'il échange moins de messages. En fait, ceci est dû à l'utilisation de la diffusion fiable (vue au chapitre 2), qui est très coûteuse. On gagnerait sans doute à implanter directement dans le système rcUFD une telle diffusion : c'est d'ailleurs une des extensions à envisager dans un futur proche.

Chapitre 8

Conclusion et perspectives

L'impossibilité du problème du consensus dans un système asynchrone avec pannes est un résultat décourageant. Le fait que l'on puisse résoudre des problèmes légèrement plus faibles comme le consensus probabiliste est un espoir, mais l'approche probabiliste est particulière et certains en refusent l'utilisation.

On a vu au chapitre 6 que des détecteurs de défaillance non fiables permettent de résoudre le problème du consensus, même si on se place dans les conditions les plus défavorables (identifiées dans [DDS87]). De plus, il apparaît que le problème du consensus probabiliste peut également être résolu en reprenant un algorithme de consensus tout en utilisant des détecteurs de défaillance plus faibles (moins précis).

L'idée de résoudre des problèmes légèrement différents, en modifiant certes le comportement des détecteurs de défaillance, mais avec les mêmes algorithmes est une idée très séduisante.

La modularité apportée par les détecteurs de défaillance est très utile. En effet, il suffit de concevoir de manière théorique des algorithmes distribués en se basant sur des propriétés abstraites, puis d'utiliser une implémentation des détecteurs de défaillance qu'on aura au préalable certifiée "correcte" (peut-être statistiquement) par des jeux de tests adéquats.

Mais compte tenu du fait que les propriétés des détecteurs de défaillance sont définies en théorie, on pourrait penser que leur réalisation pratique n'est qu'une utopie. En fait, l'implémentation effectuée montre que ce n'est pas le cas, les propriétés abstraites pouvant souvent être obtenues avec une forte probabilité (qui dépend notamment du système d'exploitation utilisé).

De plus, dans ce qui précède, un détecteur de défaillance (FD) a été présenté comme un oracle renvoyant en sortie l'ensemble des processus qu'il suspecte d'être défaillants. On peut généraliser la notion de FD en disant que la sortie d'un FD peut très bien être un ensemble de processus quelconques, et pas nécessairement des processus de Π. Une autre généralisation consiste à supposer que cette sortie peut être quelconque.[1] On pourrait ainsi prendre en compte

[1] Cette généralisation a déjà été utilisée dans [CHT92] et [CHT94].

des défaillances du système plus complexes que les pannes franches de processus. De plus, rien n'empêche d'imaginer un FD qui s'amuse à sortir le texte de ce rapport en boucle.... Un tel FD peut-il permettre de résoudre le problème du consensus ? Bien sûr que non, puisqu'il peut être facilement modélisé sous forme d'un automate déterministe. Mais si on complique légèrement le FD en faisant en sorte que sa sortie soit choisie de façon non déterministe, la réponse n'est plus du tout évidente...

Enfin, les tests réalisés en pratique sont assez rapides en moyenne, mais les performances s'écroulent souvent dans les cas extrèmement défavorables. Garantir un consensus en temps réel, c'est-à-dire un consensus qui se termine avant une certaine date t_{max} fixée, serait un résultat extrèmement intéressant. Essayer de respecter une telle contrainte temporelle dans un système asynchrone peut sembler *a priori* paradoxal, et donc impossible. Toutefois l'identification de nouvelles propriétés et de nouvelles classes de détecteurs de défaillance peuvent peut-être résoudre ce problème de manière probabiliste. C'est une question qui mériterait d'être étudiée.

Bibliographie

[BO83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, août 1983.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, octobre 1985.

[CHT92] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, pages 147–158, août 1992. Version préliminaire.

[CHT94] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. Technical report, Cornell University, mai 1994. URL: ftp://ftp.cs.cornell.edu/pub/chandra/failure.detectors.weakest.dvi.Z.

[CT91] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340, août 1991. Version préliminaire.

[CT94] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. Technical report, Cornell University, 1994. URL: ftp://ftp.cs.cornell.edu/pub/chandra/failure.detectors.algorithms.dvi.Z.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, janvier 1987.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, avril 1985.

[Geo95] Nathalie Geoffrin. Etude de l'utilisation des détecteurs de défaillance pour résoudre le problème du consensus en algorithmique distribuée. Rapport de stage, Ecole Polytechnique, F-91128 Palaiseau CEDEX (France), juillet 1995.

[Had95] Vassos Hadzilacos. Randomized consensus in asynchronous systems. Lecture notes, Cornell University, Department of Computer Science, Cornell University, Ithaca, New York 14853, USA, 1995.

[Rab83] Michael O. Rabin. Randomized byzantine generals. *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, novembre 1983.

[Rif94] Jean-Marie Rifflet. *La Communication sous Unix (Applications réparties)*. Ediscience international, deuxième édition edition, 1994.

[Tou] Sam Toueg. Communication personnelle.

Index

algorithme de Ben-Or, 26
algorithme de Bracha et Toueg, 30

complétude faible, 37
complétude forte, 37
configuration, 71
connexions, 7
consensus, 19
consensus probabiliste, 23
coordinateur tournant, 51

détecteur de défaillance, 35
diffusion fiable, 12

FD, 35

message, 8
modèle d'implémentation, 14
modèle de base, 7
modèle FLP, 15

panne franche, 11
précision (perpétuellement) faible, 38
précision (perpétuellement) forte, 38
précision faible probabiliste, 58
précision finalement faible, 38
précision finalement forte, 38
précision forte probabiliste, 57
primitive décide, 19
primitive envoyer, 9
primitive propose, 19
primitive recevoir, 9
processus, 10
processus correct, 11
processus défaillant, 11
processus défectueux, 11

rcUFD, 59
round, 24

système, 7

terminaison, 19
terminaison probabiliste, 23

Annexe A : Preuve d'impossibilité FLP

Il est impossible de résoudre le problème du consensus à l'aide d'algorithmes déterministes dans un système asynchrone à partir du moment où un processus est susceptible d'être défectueux. La preuve de cette impossibilité est exposée dans cette annexe. Il s'agit d'une reformulation de la preuve originale [FLP85], avec les définitions faites au chapitre 2.

Définitions préliminaires

On appelle *protocole de consensus* P tout système asynchrone de n processus ($n > 1$) qui résout le problème du consensus.

Lors d'un pas atomique un processus p peut recevoir un message m (qui est éventuellement $\emptyset$), puis effectuer des calculs internes suivant la valeur de m, puis envoyer un ensemble fini de messages. Comme les processus sont déterministes, un pas est complètement déterminé par un couple $(p, m) = e$, qu'on appelle un *événement*.

On appelle *configuration* du système l'état de ce dernier à un instant donné. Une *configuration initiale* est une configuration correspondant à un état initial du système.

On note $C' = e(C)$ le fait de passer d'une configuration C à une configuration C' par l'application d'un événement e.

On dit que deux configurations sont *voisines* si et seulement si on peut passer de l'une à l'autre par l'application d'un seul événement. Quand $C' = e(C)$, on dit que l'événement e est *applicable* à la configuration C et que C' est *accessible* depuis C (la propriété d'accessibilité est clairement transitive). De plus, on dira que C est accessible (sans plus de précision) si elle est accessible depuis une certaine configuration initiale. Notons que l'événement $(p, \emptyset)$ peut toujours être appliqué, donc un processus a toujours la possibilité d'effectuer un pas.

On appelle un *programme* une séquence d'événements, et une *exécution* la séquence de pas effectuée lors d'un programme. On étend la notation "$e(C)$" vue pour les événements aux séquences d'événements de manière naturelle :
si $\sigma = e_1.e_2.\ldots.e_k$ alors $\sigma(C) = e_k(\ldots(e_2(e_1(C))))$.

Une exécution est *admissible* si au plus un processus est défectueux et si tous les processus corrects reçoivent tout message qui leur a été envoyé.

Une configuration C possède un *valeur de décision* v si un processus correct p est dans un état de décision avec $y_p = v$.

Un protocole de consensus P est *correct* si et seulement si :

- il vérifie les propriétés de **terminaison**, d'**intégrité uniforme** et d'**accord** énoncées précédemment dans le modèle,
- à la fois 0 et 1 sont des valeurs de décisions possibles (ce qui élimine les solutions triviales "0 est toujours choisi" ou "1 est toujours choisi").

Etant donnée une configuration C, soit V l'ensemble des valeurs de décision accessibles depuis C.
Si $|V| = 2$, on dit que C est *bivalente.*
Si $|V| = 1$, on dit que C est *univalente*, et pour $V = \{v\}$, on dit que C est *v-valente.*
Notons que si on suppose que le protocole de consensus considéré P est correct, on a toujours $V \neq \emptyset$.

Preuve d'impossibilité

Lemme 12
(exprime une propriété de "commutativité" des séquences d'événements)
Supposons que, depuis une certaine configuration C, les programmes σ_1 et σ_2 aboutissent respectivement aux configurations C_1 et C_2. Si les ensembles des processus effectuant des pas respectivement durant σ_1 et σ_2 sont disjoints, alors σ_2 peut être appliqué à C_1 et σ_1 peut être appliqué à C_2, et dans les deux cas cela conduit à une même configuration C'.

(voir Figure 9.)

Preuve
Ce résultat est une conséquence immédiate de la définition du système considéré.
Soit $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) l'ensemble des processus effectuant au moins un pas durant σ_1 (resp. σ_2).
Comme σ_1 et σ_2 n'interfèrent pas, c'est-à-dire que $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$, il est clair que les messages présents dans M en C pour chacun des processus de $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) y sont toujours en $C_2 = \sigma_2(C)$ (resp. $C_1 = \sigma_1(C)$) [2] , puisqu'ils n'ont pas pu être reçus entre temps (les processus concernés n'ayant pas fait de pas).
De plus, l'état des processus de $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) n'a pas pu varier entre C et C_2 (resp. C_1) (un processus n'effectuant aucun pas ne pouvant pas changer d'état).
Alors il est clair que $\sigma_2(C_1) = \sigma_1(C2)$.
(En fait, tout se passe comme si l'exécution des processus de $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) avait été suspendue, pendant un temps fini, jusqu'en C_2 (resp. C_1).) $\diamond$

[2] Notons qu'il peut y en avoir de nouveaux, émis lors de l'exécution de σ_2 (resp. σ_1) ...

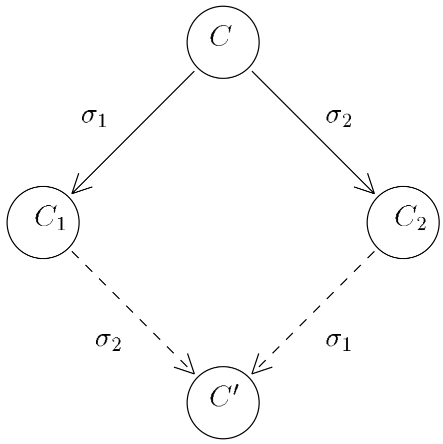

Figure 9: *Commutativité des séquences indépendantes*

Théorème ([FLP85])
Aucun protocole de consensus n'est correct si un des processus peut être défectueux.

Preuve
Raisonnons par l'absurde, en prenant comme hypothèse qu'il existe un protocole de consensus qui est correct en dépit d'une faute, soit P. Par conséquent, P est capable d'aboutir à une valeur de décision dans un temps fini, même si un seul processus p est défaillant.

Nous allons alors prouver un séquence de lemmes aboutissant à une contradiction.
L'idée de base est d'exhiber des circonstances dans lesquelles le protocole P ne décide jamais. Pour cela, nous raisonnerons par récurrence en montrant que :

- il existe une configuration initiale bivalente (i.e. pour laquelle la décision n'est pas déjà prédéterminée),
- depuis toute configuration bivalente C, on peut accéder à une autre configuration bivalente C' de manière parfaitement acceptable.

Lemme 13
P possède au moins une configuration initiale bivalente.

Preuve
Raisonnons par l'absurde : supposons qu'il n'existe pas de configuration initiale bivalente pour P.
Alors P doit posséder à la fois des configurations initiales 0-valentes et 1-valentes, puisqu'il est supposé correct. Appelons deux configurations initiales *adjacentes* si elles diffèrent uniquement par la valeur initiales x_p d'un certain processus p. Pour tout couple de configurations initiales, il existe donc une chaîne de configurations initiales, chacune adjacente à la suivante, les joignant.

(En effet, d'après le modèle, la seule chose qui diffère entre les configurations initiales est le mot $x_1 \ldots . x_n$ (composé des registres d'entrée x_p des n processus). On peut passer d'un mot à l'autre en changeant 1 seul bit, et donc passer d'une configuration initiale à la suivante comme dans le cas d'un réseau hypercubique, en constituant ainsi la chaîne annoncée.)
Par conséquent, il existe nécessairement une configuration initiale 0-valente C_0 adjacente à une configuration initiale 1-valente C_1. C_0 et C_1 diffèrent donc par l'état initial d'un seul processus, soit p.
Considérons maintenant une exécution admissible partant de C_0 durant laquelle le processus p n'effectue aucun pas, et soit σ le programme associé.
Alors σ peut être également appliqué à C_1 (puisque C_0 et C_1 ne diffèrent que par x_p, et p n'intervient pas dans σ), et il est clair que les deux configurations obtenues $\sigma(C_0)$ et $\sigma(C_1)$ sont identiques modulo l'état interne du processus p.
Par conséquent, si P peut décider en dépit d'une défaillance de processus, puisque p est le seul processus dont on bloque l'exécution, les deux configurations $\sigma(C_0)$ et $\sigma(C_1)$ vont finalement choisir la même valeur de décision.
Si cette valeur est 1, alors nécessairement C_0 était bivalente, et sinon cette valeur est 0 et nécessairement C_1 était bivalente. Dans les deux cas, cela contredit l'hypothèse de non-existence d'une configuration initiale bivalente. $\diamond$

Lemme 14
Soit C une configuration bivalente de P, et soit $e = (p, m)$ un événement applicable à C. Soit $\mathcal{C}$ l'ensemble de toutes les configurations accessibles depuis C sans appliquer e, et soit $\mathcal{D} = \{e(E)/E \in \mathcal{C}$ et "e est applicable à E"$\}$. Alors $\mathcal{D}$ contient une configuration bivalente.

Preuve

Remarque préliminaire
Comme e est applicable à C, il est applicable à toute configuration de $\mathcal{C}$ puisque tout message peut être retardé arbitrairement pendant un intervalle de temps fini. Par conséquent, $\mathcal{D} = e(\mathcal{C})$.

Raisonnons alors par l'absurde : supposons que $\mathcal{D}$ ne contienne aucune configuration bivalente. Dans ce cas, toute configuration de $\mathcal{D}$ est univalente.
Soit E_0 (resp. E_1) une configuration 0-valente (resp. 1-valente) accessible depuis C (E_0 et E_1 existent bien puisque C est bivalente).
Pour $i = 0, 1$, si $E_i \in \mathcal{C}$ posons $F_i = e(E_i)$ (alors $F_i \in \mathcal{D}$ d'après la remarque préliminaire). Sinon, c'est que e a été appliqué pour accéder à E_i, et donc il existe une configuration $F_i \in \mathcal{D}$ depuis laquelle E_i est accessible.
Dans tous les cas, on dispose de deux configurations F_0 et F_1 dans $\mathcal{D}$, respectivement 0-valente et 1-valente. Donc $\mathcal{D}$ contient à la fois des configurations 0-valentes et 1-valentes.

Considérons alors le graphe non orienté G défini de la manière suivante :

- les sommets de G sont les configurations de $\mathcal{C}$ (et donc C en fait partie),
- les arêtes de G sont les paires de configurations $\{E, E'\}$ telles que E' est voisine de E.

Soit G' le graphe obtenu à partir de G en appliquant e à chaque sommet de G, ainsi :

- les sommets de G' sont les configurations de $\mathcal{D}$,
- il y a une arête $\{F, F'\}$ dans G' si et seulement si il y a une arête $\{E, E'\}$ dans G (et $F = e(E)$, $F' = e(E')$) [3].

Comme $\mathcal{D}$ contient à la fois des configurations 0-valentes et 1-valentes, il est clair qu'il existe dans G' une arête joignant une configuration 0-valente (soit D_0) et une configuration 1-valente (soit D_1) [4]. Considérons alors l'arête correspondante dans G : elle joint deux configurations voisines C_0 et C_1 telles que $D_0 = e(C_0)$ et $D_1 = e(C_1)$.

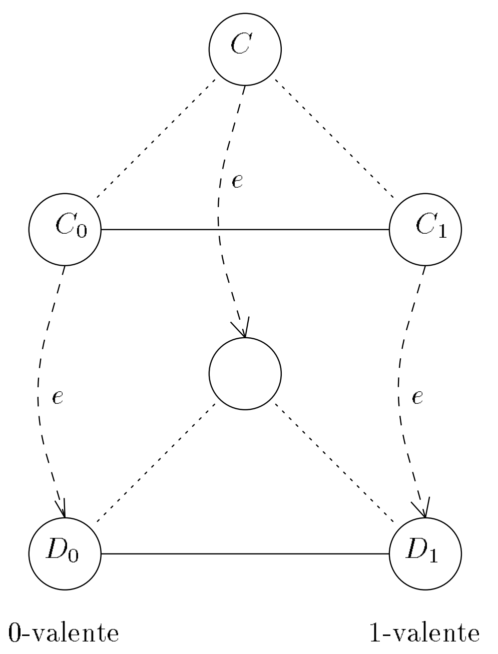

Figure 10: *Configurations voisines*

Il y a donc dans $\mathcal{C}$ deux configurations voisines C_0 et C_1, respectivement 0-valente et 1-valente, telles que pour $i = 0, 1$, $D_i = e(C_i)$ est i-valente.

Comme C_0 et C_1 sont voisines, considérons, sans perte de généralité, que par exemple $C_1 = e'(C_0)$, avec $e' = (p', m')$ (le cas $C_0 = e'(C_1)$ se traiterait de la même manière, par symètrie).

Rappel : pour $i = 0, 1$, $D_i = e(C_i)$ où $e = (p, m)$, et $C_1 = e'(C_0)$ où $e' = (p', m')$.

2 cas sont alors possibles :

[3] Bien valide d'après le lemme 12, puisque $\mathcal{D} = e(\mathcal{C})$ et que les configurations de $\mathcal{C}$ ont été obtenues à partir de C sans appliquer e.

[4] Rappel : on a supposé que $\mathcal{D}$ ne contenait pas de configuration bivalente.

- cas $p' \neq p$
  Alors, d'après le lemme 12, $D1 = e'(D_0)$, et donc D_1 est un successeur de D_0. C'est impossible, car D_0 est 0-valente et D_1 est 1-valente, or tout successeur d'une configuration 0-valente est toujours 0-valente.

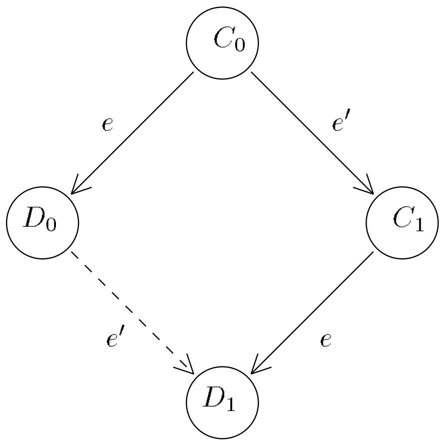

Figure 11: *cas $p \neq p'$*

- cas $p' = p$
  Considérons alors une exécution finie à partir de C_0 durant laquelle p n'effectue aucun pas. Soit σ le programme correspondant, et soit $A = \sigma(C_0)$.
  Comme $e = (p, m)$ et que p n'intervient pas dans σ, e et σ sont disjoints. De même, comme $e' = (p', m') = (p, m')$ (puisque $p' = p$), e' et σ sont également disjoints.
  D'après le lemme 12, σ est applicable à $D_i = e(C_i)$ pour $i = 0, 1$, et conduit à une configuration i-valente $E_i = \sigma(D_i)$.
  Encore par le lemme 12, e est applicable à $A = \sigma(C_0)$ et $e(A) = e(\sigma(C_0)) = \sigma(e(C_0)) = \sigma(D_0) = E_0$.
  Toujours par le lemme 12, $e(e'(A)) = e(e'(\sigma(C_0))) = e(\sigma(e'(C_0))) =$
  $e(\sigma(C_1)) = \sigma(e(C_1)) = \sigma(D_1) = E_1$.
  Par conséquent, A est bivalente, ce qui est impossible puisque l'exécution conduisant à A considérée décide, donc A doit être univalente.

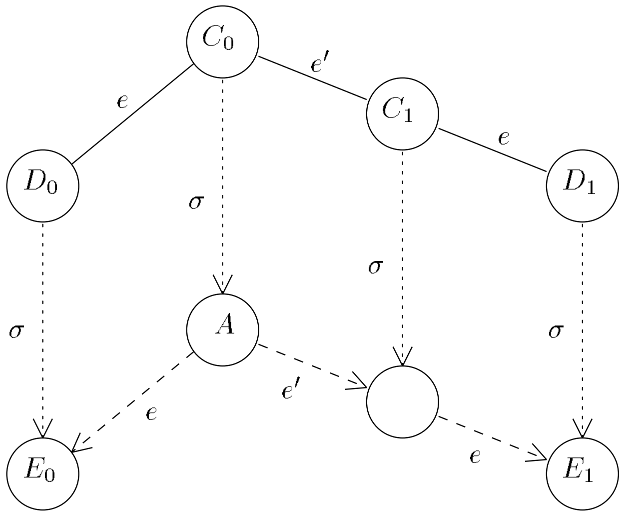

Figure 12: *cas $p = p'$*

Dans les deux cas, on aboutit à une contradiction.
Donc $\mathcal{D}$ contient une configuration bivalente. $\diamond$

Corps de la preuve

Toute exécution qui décide à partir d'une configuration bivalente doit nécessairement aboutir à une configuration univalente. Par conséquent, il doit y avoir passage d'une configuration bivalente à une configuration univalente lors d'un seul pas. Un tel pas détermine l'éventuelle valeur de décision. Nous allons maintenant montrer qu'il est toujours possible de faire évoluer le système de manière valide de telle sorte que de semblables pas soient tous évités, ce qui conduit à une exécution admissible mais qui ne décide pas.

L'exécution se compose d'étapes, partant d'une configuration initiale. Chaque étape consiste en un ou plusieurs pas d'un même processus p, et se termine par l'application d'un événement (p, m).

De plus, on maintient une liste ordonnée de processus, initialement composée des n processus dans un ordre quelconque. A chaque étape on considère le processus en tête de la liste, que l'on déplace à la fin avant de passer à l'étape suivante suivante (système de barillet).

Nous allons maintenant construire inductivement une exécution admissible qui ne décide pas :

- (base)
  Soit C_0 une configuration initiale bivalente, dont l'existence est assurée par le lemme 13. L'exécution commence en C_0.

- (étape inductive)
  Supposons que l'on ait effectué k étapes, et que l'on se trouve dans une configuration bivalente C_k.

> Soit p le processus en tête de la liste des processus, et soit m un message de M à destination de p (ou $\emptyset$ si il n'y en a pas). Soit $e = (p, m)$ l'événement correspondant.
> D'après le lemme 14, il existe une configuration bivalente, soit C_{k+1}, accessible depuis C_k par une séquence d'événements se terminant par e.
> La séquence de pas correspondante constitue la (k+1)-ième étape.
> Pour finir, on déplace le processus p à la fin de la liste.

L'exécution est bien admissible, car il est clair qu'en procédant ainsi tous les processus effectuent une infinité de pas et une infinité de demandes de réception de message.

On a donc exhibé une exécution parfaitement admissible, mais qui ne décide jamais (elle ne se termine pas). Par conséquent P n'est pas correct, ce qui contredit l'hypothèse initiale. $\diamond$

Conclusion
Le problème du consensus ne peut pas être résolu dès qu'un des processus peut être défectueux.

Annexe B :
Ben-Or $\rightarrow$ Bracha et Toueg

Ben-Or :

$x \leftarrow x_0$ `/* valeur initialement proposée */`

$r \leftarrow 1$ `/* numéro du` *round* `*/`

répéter
{
`/* Phase 1: Rapports */`

envoyer à tous $(\mathtt{R}, r, x)$
attendre $(n - f)$ messages de la forme $(\mathtt{R}, r, \cdots)$
si on a reçu plus de $(n/2)$ messages $(\mathtt{R}, r, v)$ pour un même v
$proposition \leftarrow v$
sinon
$proposition \leftarrow ?$

`/* Phase 2: Propositions */`

envoyer à tous $(\mathtt{P}, r, proposition)$
attendre $(n - f)$ messages de la forme $(\mathtt{P}, r, \cdots)$
si on a reçu au moins $(f + 1)$ messages $(\mathtt{P}, r, v)$ pour un même $v \neq ?$
décide(v)

`/* Phase supplémentaire: Ajustement */`

si il y avait un $(\mathtt{P}, r, v)$ où $v \neq ?$
$x \leftarrow v$
sinon
$x \leftarrow 0$ ou 1, au hasard (probabilité $\frac{1}{2}$ pour chaque valeur)

incrémenter r
}

Hybride 1 :

$x \leftarrow x_0$ `/* valeur initialement proposée */`

$r \leftarrow 1$ `/* numéro du` *round* `*/`

répéter

{

`/* Phase 1:  Rapports */`

envoyer à tous $(\mathtt{R}, r, x)$
attendre $(n - f)$ messages de la forme $(\mathtt{R}, r, \cdots)$
si on a reçu plus de $(n/2)$ messages $(\mathtt{R}, r, v)$ pour un même v
  $proposition \leftarrow v$
sinon
  $proposition \leftarrow ?$

`/* Phase 2:  Propositions */`

envoyer à tous $(\mathtt{P}, r, proposition)$
attendre $(n - f)$ messages de la forme $(\mathtt{P}, r, \cdots)$
si on a reçu au moins $(f + 1)$ messages $(\mathtt{P}, r, v)$ pour un même $v \neq ?$

{

<u>**envoyer à tous** $(\mathtt{R}, r + 1, v)$</u>
<u>**envoyer à tous** $(\mathtt{P}, r + 1, v)$</u>

décide(v)

<u>**terminer l'exécution**</u>

}

`/* Phase supplémentaire:  Ajustement */`

si il y avait un $(\mathtt{P}, r, v)$ où $v \neq ?$
  $x \leftarrow v$
sinon
  $x \leftarrow 0$ ou 1, au hasard (probabilité $\frac{1}{2}$ pour chaque valeur)

incrémenter r

}

Hybride 2 :

$x \leftarrow x_0$ `/* valeur initialement proposée */`

$r \leftarrow 1$ `/* numéro du` *round* `*/`

$nb\_rapports[2] \leftarrow \{0,0\}$ `/* nombre de rapports reçus */`
$nb\_propositions[2] \leftarrow \{0,0\}$ `/* nombre de propositions reçues */`

répéter
 {
  `/* Phase 1: Rapports */`

  $proposition \leftarrow ?$
  $nb\_rapports \leftarrow \{0,0\}$
  envoyer à tous $(\mathtt{R}, r, x)$
  tant que $(nb\_rapports[0] + nb\_rapports[1]) < (n - f)$
   {
    attendre un message de la forme $(\mathtt{R}, k, v)$

    si $k = r$
     {
      incrémenter $nb\_rapports[v]$
      si $nb\_rapports[v] > (n/2)$
       $proposition \leftarrow v$
     }
    sinon si $k > r$
     {
      `/* message pour un` *round* `futur: à conserver */`
      envoyer$(p, (\mathtt{R}, k, v))$ `/* remettre le message dans` M `*/`
     }
    `/* si` $k < r$`, c´est un message périmé... */`
   }

```
/* Phase 2: Propositions */

 nb_propositions ← {0, 0}
 envoyer à tous (P, r, proposition)
 tant que (nb_propositions[0] + nb_propositions[1]) < (n − f)
   {
     attendre un message de la forme (P, k, v)

     si k = r
       {
         incrémenter nb_propositions[v]
       }
     sinon si k > r
       {
         /* message pour un round futur: à conserver */
         envoyer(p,(P, k, v)) /* remettre le message dans M */
       }
     /* si k < r, c´est un message périmé... */
   }

 si il existe v tel que nb_propositions[v] ≥ (f + 1)
   {
     envoyer à tous (R, r + 1, v)
     envoyer à tous (P, r + 1, v)

     décide(v)

     terminer l'exécution
   }

 /* Phase supplémentaire: Ajustement */

 si il existe v tel que nb_propositions[v] > 0
   x ← v
 sinon
   x ← 0 ou 1, au hasard (probabilité 1/2 pour chaque valeur)

 incrémenter r
}
```

Hybride 3 :

$x \leftarrow x_0$ `/* valeur initialement proposée */`

$s \leftarrow 1$ `/* numéro de l´étape */`

$cardinalité \leftarrow 1$

$nb\_messages[2] \leftarrow \{0,0\}$ `/* nombre de messages reçus */`
$nb\_propositions[2] \leftarrow \{0,0\}$ `/* nombre de propositions reçues */`

répéter
{
  $nb\_messages \leftarrow \{0,0\}$
  $nb\_propositions \leftarrow \{0,0\}$
  envoyer à tous $(cardinalité, s, x)$
  tant que $(nb\_messages[0] + nb\_messages[1]) < (n - f)$
    {
      attendre un message de la forme (c, k, v)

      si $k = s$
        {
          incrémenter $nb\_messages[v]$
          si $c > (n/2)$ `/* super-rapport ?  */`
            incrémenter $nb\_propositions[v]$
        }
      sinon si $k > s$
        {
          `/* message pour une étape future:  à conserver */`
          envoyer$(p,(c, k, v))$ `/* remettre le message dans` M `*/`
        }
      `/* si` $k < s$`, c´est un message périmé...  */`
    }

```
    cardinalité ← max(nb_messages[0], nb_messages[1])

    si il existe v tel que nb_propositions[v] ≥ (f + 1)
      {
        envoyer à tous (cardinalité, s + 1, v)
        envoyer à tous (cardinalité, s + 2, v)

        décide(v)

        terminer l'exécution
      }

    /* Ajustement */

    si il existe v tel que nb_propositions[v] > 0
      x ← v
    sinon
      {
        x ← 0 ou 1, au hasard (probabilité 1/2 pour chaque valeur)

        incrémenter s
      }
}
```

Hybride 4 :

$x \leftarrow x_0$ `/* valeur initialement proposée */`

$s \leftarrow 1$ `/* numéro de l´étape */`

$cardinalit\acute{e} \leftarrow 1$

$nb\_messages[2] \leftarrow \{0, 0\}$ `/* nombre de messages reçus */`
$nb\_t\acute{e}moins[2] \leftarrow \{0, 0\}$ `/* nombre de témoins reçus */`

tant que $nb\_t\acute{e}moins[0] < (f + 1)$ **et** $nb\_t\acute{e}moins[1] < (f + 1)$
  {
    $nb\_messages \leftarrow \{0, 0\}$
    $nb\_t\acute{e}moins \leftarrow \{0, 0\}$
    envoyer à tous $(cardinalit\acute{e}, s, x)$
    tant que $(nb\_messages[0] + nb\_messages[1]) < (n - f)$
      {
        attendre un message de la forme (c, k, v)

        si $k = s$
          {
            incrémenter $nb\_messages[v]$
            si $c > (n/2)$ `/* super-rapport ?  */`
              incrémenter $nb\_t\acute{e}moins[v]$
          }
        sinon si $k > s$
          {
            `/* message pour une étape future:  à conserver */`
            envoyer$(p,(c, k, v))$ `/* remettre le message dans` M `*/`
          }
        `/* si` $k < s$`, c´est un message périmé...  */`
      }

`/* Ajustement */`

$cardinalit\acute{e} \leftarrow \max(nb\_messages[0], nb\_messages[1])$

si il existe v tel que $nb\_t\acute{e}moins[v] > 0$
  $x \leftarrow v$
sinon
  {
    $x \leftarrow 0$ ou 1, au hasard (probabilité $\frac{1}{2}$ pour chaque valeur)

    incrémenter s
  }
}

soit v tel que $nb\_t\acute{e}moins[v] \geq (f + 1)$

envoyer à tous $(n - f, s, v)$
envoyer à tous $(n - f, s + 1, v)$

décide(v)

Bracha et Toueg :

$x \leftarrow x_0$ `/* valeur initialement proposée */`

$s \leftarrow 1$ `/* numéro de l´étape */`

$cardinalité \leftarrow 1$

$nb\_messages[2] \leftarrow \{0, 0\}$ `/* nombre de messages reçus */`
$nb\_témoins[2] \leftarrow \{0, 0\}$ `/* nombre de témoins reçus */`

tant que $nb\_témoins[0] < (f + 1)$ **et** $nb\_témoins[1] < (f + 1)$
{
  $nb\_messages \leftarrow \{0, 0\}$
  $nb\_témoins \leftarrow \{0, 0\}$

  envoyer à tous $(cardinalité, s, x)$
  tant que $(nb\_messages[0] + nb\_messages[1]) < (n - f)$
  {
    attendre un message de la forme (c, k, v)

    si $k = s$
    {
      incrémenter $nb\_messages[v]$
      si $c > (n/2)$ `/* super-rapport ? */`
        incrémenter $nb\_témoins[v]$
    }
    sinon si $k > s$
    {
      `/* message pour une étape future: à conserver */`
      envoyer$(p,(c, k, v))$ `/* remettre le message dans` M `*/`
    }
    `/* si` $k < s$`, c´est un message périmé... */`
  }

```
/* Ajustement */
```

si il existe v tel que $nb\_témoins[v] > 0$
$x \leftarrow v$
sinon
{
si $nb\_messages[1] > nb\_messages[1]$
$x \leftarrow 1$
sinon
$x \leftarrow 0$
$cardinalité \leftarrow nb\_messages[x]$

incrémenter s
}
}

soit v tel que $nb\_témoins[v] \geq (f+1)$

envoyer à tous $(n-f, s, v)$
envoyer à tous $(n-f, s+1, v)$

décide(v)

Annexe C : Résultats des tests

Un processus participant au consensus est de la forme :

`consensus` n f p x

où n est le nombre total de processus, f est un majorant du nombre de processus défectueux, p est le numéro du processus et x la valeur qu'il propose.

Le jeu de tests est le suivant :

- aucun processus défectueux

  - test 1.1
  - test 1.2
  - test 1.3

test 1.1	test 1.2	test 1.3
`consensus 8 3 1 1`	`consensus 8 3 1 0`	`consensus 16 7 1 1`
`consensus 8 3 2 1`	`consensus 8 3 2 1`	`consensus 16 7 2 0`
`consensus 8 3 3 1`	`consensus 8 3 3 1`	`consensus 16 7 3 1`
`consensus 8 3 4 1`	`consensus 8 3 4 0`	`consensus 16 7 4 0`
`consensus 8 3 5 1`	`consensus 8 3 5 0`	`consensus 16 7 5 1`
`consensus 8 3 6 1`	`consensus 8 3 6 0`	`consensus 16 7 6 0`
`consensus 8 3 7 1`	`consensus 8 3 7 1`	`consensus 16 7 7 1`
`consensus 8 3 8 1`	`consensus 8 3 8 1`	`consensus 16 7 8 0`
		`consensus 16 7 9 1`
		`consensus 16 7 10 0`
		`consensus 16 7 11 1`
		`consensus 16 7 12 0`
		`consensus 16 7 13 1`
		`consensus 16 7 14 0`
		`consensus 16 7 15 1`
		`consensus 16 7 16 0`

- $\lfloor \frac{n}{2} \rfloor$ processus défectueux

• test 2.1	• test 2.2	• test 2.3
`consensus 8 3 4 1`	`consensus 8 3 4 0`	`consensus 16 7  8 0`
`consensus 8 3 5 1`	`consensus 8 3 5 0`	`consensus 16 7  9 1`
`consensus 8 3 6 1`	`consensus 8 3 6 0`	`consensus 16 7 10 0`
`consensus 8 3 7 1`	`consensus 8 3 7 1`	`consensus 16 7 11 1`
`consensus 8 3 8 1`	`consensus 8 3 8 1`	`consensus 16 7 12 0`
		`consensus 16 7 13 1`
		`consensus 16 7 14 0`
		`consensus 16 7 15 1`
		`consensus 16 7 16 0`

Pour l'algorithme de Bracha et Toueg, on obtient le plus souvent les résultats suivants :

- test 1.1

process 1 has decided value 1. stats - messages: 32 sent (4 self), 13 received, time: 1 s
process 4 has decided value 1. stats - messages: 32 sent (4 self), 13 received, time: 1 s
process 2 has decided value 1. stats - messages: 32 sent (4 self), 13 received, time: 1 s
process 5 has decided value 1. stats - messages: 32 sent (4 self), 13 received, time: 1 s
process 7 has decided value 1. stats - messages: 32 sent (4 self), 13 received, time: 1 s
process 8 has decided value 1. stats - messages: 32 sent (4 self), 13 received, time: 1 s
process 6 has decided value 1. stats - messages: 32 sent (4 self), 13 received, time: 0 s
process 3 has decided value 1. stats - messages: 32 sent (4 self), 13 received, time: 1 s

- test 1.2

process 1 has decided value 0. stats - messages: 40 sent (5 self), 19 received, time: 1 s
process 3 has decided value 0. stats - messages: 40 sent (5 self), 19 received, time: 2 s
process 2 has decided value 0. stats - messages: 40 sent (5 self), 19 received, time: 2 s
process 6 has decided value 0. stats - messages: 40 sent (5 self), 19 received, time: 1 s
process 7 has decided value 0. stats - messages: 40 sent (5 self), 19 received, time: 1 s
process 4 has decided value 0. stats - messages: 40 sent (5 self), 19 received, time: 1 s
process 5 has decided value 0. stats - messages: 40 sent (5 self), 19 received, time: 2 s
process 8 has decided value 0. stats - messages: 40 sent (5 self), 19 received, time: 1 s

- test 1.3

process 1 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 2 s
process 3 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 2 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 4 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 5 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 6 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 7 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 8 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 11 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 12 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 13 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 14 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 15 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 16 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 3 s
process 9 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 2 s
process 10 has decided value 1. stats - messages: 80 sent (5 self), 37 received, time: 2 s

- test 2.1

process 4 has decided value 1. stats - messages: 32 sent (4 self), 10 received, time: 1 s
process 5 has decided value 1. stats - messages: 32 sent (4 self), 10 received, time: 1 s
process 8 has decided value 1. stats - messages: 32 sent (4 self), 10 received, time: 1 s
process 6 has decided value 1. stats - messages: 32 sent (4 self), 10 received, time: 1 s
process 7 has decided value 1. stats - messages: 32 sent (4 self), 10 received, time: 1 s

- test 2.2

process 7 has decided value 0. stats - messages: 40 sent (5 self), 15 received, time: 1 s
process 6 has decided value 0. stats - messages: 40 sent (5 self), 15 received, time: 1 s
process 8 has decided value 0. stats - messages: 40 sent (5 self), 15 received, time: 1 s
process 5 has decided value 0. stats - messages: 40 sent (5 self), 15 received, time: 1 s
process 4 has decided value 0. stats - messages: 40 sent (5 self), 15 received, time: 2 s

- test 2.3

process 10 has decided value 0. stats - messages: 80 sent (5 self), 27 received, time: 2 s
process 8 has decided value 0. stats - messages: 80 sent (5 self), 27 received, time: 3 s
process 12 has decided value 0. stats - messages: 80 sent (5 self), 27 received, time: 3 s
process 9 has decided value 0. stats - messages: 80 sent (5 self), 27 received, time: 3 s
process 11 has decided value 0. stats - messages: 80 sent (5 self), 27 received, time: 3 s
process 15 has decided value 0. stats - messages: 80 sent (5 self), 27 received, time: 3 s
process 16 has decided value 0. stats - messages: 80 sent (5 self), 27 received, time: 3 s
process 13 has decided value 0. stats - messages: 80 sent (5 self), 27 received, time: 3 s
process 14 has decided value 0. stats - messages: 80 sent (5 self), 27 received, time: 3 s

Pour l'algorithme du "coordinateur tournant", on obtient le plus souvent les résultats suivants :

(le coordinateur final figure en gras)

- test 1.1

process 1 has decided value 1. stats - messages: 14 sent (1 self), 10 received, time: 3 s
process 2 has decided value 1. stats - messages: 25 sent (7 self), 29 received, time: 3 s
process 3 has decided value 1. stats - messages: 30 sent (7 self), 19 received, time: 3 s
process 5 has decided value 1. stats - messages: 14 sent (1 self), 10 received, time: 3 s
process 6 has decided value 1. stats - messages: 14 sent (1 self), 10 received, time: 3 s
process 7 has decided value 1. stats - messages: 14 sent (1 self), 10 received, time: 4 s
process 8 has decided value 1. stats - messages: 14 sent (1 self), 10 received, time: 4 s
process 4 has decided value 1. stats - messages: 39 sent (7 self), 25 received, time: 4 s

- test 1.2

process 1 has decided value 1. stats - messages: 12 sent (1 self), 7 received, time: 2 s
process 5 has decided value 1. stats - messages: 12 sent (1 self), 7 received, time: 2 s
process 6 has decided value 1. stats - messages: 12 sent (1 self), 7 received, time: 2 s
process 7 has decided value 1. stats - messages: 12 sent (1 self), 7 received, time: 2 s
process 8 has decided value 1. stats - messages: 12 sent (1 self), 7 received, time: 2 s
process 3 has decided value 1. stats - messages: 31 sent (8 self), 26 received, time: 3 s
process 2 has decided value 1. stats - messages: 23 sent (7 self), 26 received, time: 3 s
process 4 has decided value 1. stats - messages: 12 sent (1 self), 7 received, time: 3 s

- test 1.3

process 5 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 4 s
process 6 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 4 s
process 3 has decided value 1. stats - messages: 54 sent (20 self), 59 received, time: 4 s
process 7 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 4 s
process 14 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 3 s
process 8 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 4 s
process 9 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 4 s
process 15 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 3 s
process 16 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 3 s
process 1 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 4 s
process 12 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 2 s
process 11 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 4 s
process 13 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 2 s
process 10 has decided value 1. stats - messages: 22 sent (1 self), 11 received, time: 2 s
process 2 has decided value 1. stats - messages: 41 sent (7 self), 46 received, time: 5 s
process 4 has decided value 1. stats - messages: 61 sent (12 self), 48 received, time: 5 s

- test 2.1

process 6 has decided value 1. stats - messages: 16 sent (1 self), 4 received, time: 2 s
process 7 has decided value 1. stats - messages: 16 sent (1 self), 4 received, time: 2 s
process 8 has decided value 1. stats - messages: 16 sent (1 self), 4 received, time: 2 s
process 4 has decided value 1. stats - messages: 24 sent (4 self), 14 received, time: 3 s
process 5 has decided value 1. stats - messages: 32 sent (5 self), 17 received, time: 3 s

- test 2.2

process 4 has decided value 0. stats - messages: 24 sent (4 self), 17 received, time: 4 s
process 6 has decided value 0. stats - messages: 16 sent (1 self), 7 received, time: 4 s
process 7 has decided value 0. stats - messages: 16 sent (1 self), 7 received, time: 4 s
process 8 has decided value 0. stats - messages: 16 sent (1 self), 7 received, time: 4 s
process 5 has decided value 0. stats - messages: 32 sent (5 self), 17 received, time: 4 s

- test 2.3

process 8 has decided value 0. stats - messages: 48 sent (4 self), 29 received, time: 4 s
process 10 has decided value 0. stats - messages: 32 sent (1 self), 11 received, time: 4 s
process 12 has decided value 0. stats - messages: 32 sent (1 self), 11 received, time: 4 s
process 14 has decided value 0. stats - messages: 32 sent (1 self), 11 received, time: 4 s
process 15 has decided value 0. stats - messages: 32 sent (1 self), 11 received, time: 4 s
process 11 has decided value 0. stats - messages: 32 sent (1 self), 11 received, time: 4 s
process 16 has decided value 0. stats - messages: 32 sent (1 self), 11 received, time: 4 s
process 13 has decided value 0. stats - messages: 32 sent (1 self), 11 received, time: 4 s
process 9 has decided value 0. stats - messages: 64 sent (5 self), 29 received, time: 5 s

Annexe D : Sources de l'implémentation

Ne sont présentes ici que les sources des algorithmes de Bracha et Toueg (vu au chapitre 4) et du “coordinateur tournant” (vu au chapitre 6), ainsi que le fichier d'en-tête donnant l'interface du système rcUFD décrit au chapitre 7. Les sources complètes du système rcUFD, trop volumineuses, ne sont pas fournies dans le présent document mais sont toutefois disponibles auprès de l'auteur.