Chapitre 04 : Implémentations de détecteurs de défaillances :

1.Introduction :

Depuis que la notion de détecteur a été proposée par Chandra et Toueg [1996], un grand intérêt lui a été porté, d'abord au niveau théorique pour savoir quelles sont les propriétés minimales que ceux-ci devraient exhiber pour résoudre un problème donné, surtout les problèmes dits d'accord (et aussi quelles formes devrait prendre le détecteur de défaillances).

Un effort considérable a aussi été porté sur la conception de solutions algorithmiques fondées sur les détecteurs de défaillances qui ont des propriétés intéressantes quand à leur modularité de conception et la facilité d'établissement de preuve de correction.

Enfin, les problèmes d'accord sont réputés sans solution dans les systèmes répartis asynchrones. L'usage d'un détecteur de défaillances ne saurait les rendre solvables, à moins que le détecteur lui-même ne soit pas implémentable. Ceci dit, des implémentations de ceux-ci ont été proposées soit en supposant que le système jouit de propriétés de synchronie supplémentaires, soit que l'implémentation n'est censée être correcte qu'avec une certaine probabilité. Deux familles d'implémentation existent. Celles qui utilisent le temps physique et celles qui ne l'utilisent pas et en fin nous avons des solutions hybrides.

Dans ce chapitre, nous présentons en détails les trois familles de détecteurs de défaillances et les caractéristiques principales de chaque famille.

2.Implémentations de détecteurs de défaillances :

2.1.La famille qui utilise le temps physique :

Plusieurs travaux ont considéré l'implémentation de certaines ou de toutes les classes de détecteurs de défaillances de Chandra et Toueg. Fondamentalement, tous ces travaux considèrent que, inéluctablement, le système se comporte synchronement. Plus précisément, ils considèrent des modèles de système partiellement synchrone de Chandra et Toueg [1996] qui est une généralisation des modèles proposés par Dwork et al. [1988]. Un système partiellement synchrone suppose qu'il existe des bornes sur les vitesses relatives des processus et sur les délais de transmission des messages mais ses bornes ne sont pas connues et se détient seulement

après qu'un certain temps fini et non connu (appelé Temps Global de Stabilisation). Les protocoles implémentant les détecteurs de défaillances de tels systèmes obéissent au principe suivant: utiliser des approximations successives, tel que chaque processus détermine dynamiquement une valeur Δ qui devient finalement une borne supérieure sur les délais de transmission des messages.

Dans cette section, nous présentons la relation entre les détecteurs de défaillances et le synchronisme partiel, l'impossibilité de l'implémentation des détecteurs de défaillances, des stratégies de détection de défaillances, et deux protocole implémentant les détecteurs de défaillances.

2.1.1 Le modèle partiellement synchrone :

Entre les deux modèles d'extrémités d'un système réparti, purement synchrone et purement asynchrone, il existe une variété de modèles intermédiaires dits les modèles partiellement synchrones. En particulier, Dolev et al [1987] ont défini 32 modèles partiellement synchrones grâce à cinq principaux critères, dont chacun peut prendre deux valeurs soit vrai ''favorable'' ou faux ''défavorable''. Par exemple, un des paramètres est le délai maximum de transmission d'un message, qui peut être borné et connu (favorable) ou non borné (défavorable).

Dwork et al [1988] ont considéré les deux modèles partiellement synchrones suivants :

• **M1 :** dans chaque exécution, il existe des bornes sur les délais de transmission des messages et sur les vitesses relatives des processus, mais ces bornes ne sont pas connues.

• **M2 :** dans ce modèle, ils existent des bornes sur les vitesses relatives des processus et sur les délais de transmission des messages et sont connues, mais à partir d'un moment (non connu) appelé temps global de stabilisation2 (GST).

Un système conforme au modèle M2 peut être vu comme asynchrone jusqu'à GST, et comme synchrone après GST. Ainsi, M2 peut être vu comme un modèle inéluctablement synchrone.

Un troisième modèle partiellement synchrone le plus faible M3 a été proposé par Chandra et Toueg [1996], lequel généralise les deux modèles précédents M1 et M2.

• **M3 :** il existe un moment non connu (mais fini) appelé temps global de stabilisation, à partir duquel ils existent des bornes sur les délais de transmission des messages et sur les vitesses relatives des processus et ne sont pas connues.

Un système conforme au modèle M3 peut être vu comme asynchrone jusqu'au temps global de stabilisation GST et comme un système qui conforme au modèle M1 après le temps GST.

Un quatrième modèle a été proposé par Cristian et Fetzer [1999]. Il se caractérise principalement par l'existence d'une horloge physique et par des services temporisés.
Lequel généralise le modèle M2, en y ajoutant la notion de période.

• **M4** : infiniment souvent, ils existent des périodes assez longues sur lesquelles des bornes sur les vitesses relatives des processus et sur les délais transmission des messages existent et sont connues.

Cristian et Fetzer ont supposé l'existence d'une période de stabilité d'une longueur minimum après une période d'instabilité. La stabilité implique l'existence des services temporisés. Cette hypothèse nécessite qu'infiniment souvent, une majorité des communications soient synchrones.

Un système qui conforme au modèle M4 peut être vu comme asynchrone durant une période d'instabilité et comme synchrone durant une période de stabilité.

2.1.2. Les détecteurs de défaillances et le synchronisme partiel

2.1.2.1. Des stratégies de détection de défaillances

Dans cette section, nous présentons quatre mécanismes de détection de défaillances remplissant les propriétés de la classe $\Diamond S$. Nous allons en expliquer maintenant les principes de chaque mécanisme.

A. Le détecteur de défaillances Heart-Beat

Heart-Beat est une technique bien connue pour l'implémentation des détecteurs de défaillances. Le principe de ce détecteur de défaillances est très simple. Tout processus envoie périodiquement à tous les autres processus un message '' *I am alive*'' pour leur signaler qu'il est en vie. Si un processus n'a reçu aucun message d'un processus durant un intervalle de temps donné –timeout-, il le suspecte. S'il reçoit ultérieurement un message '' *I am alive*'' d'un processus suspecté, alors il le ôte de sa liste des processus suspectés.

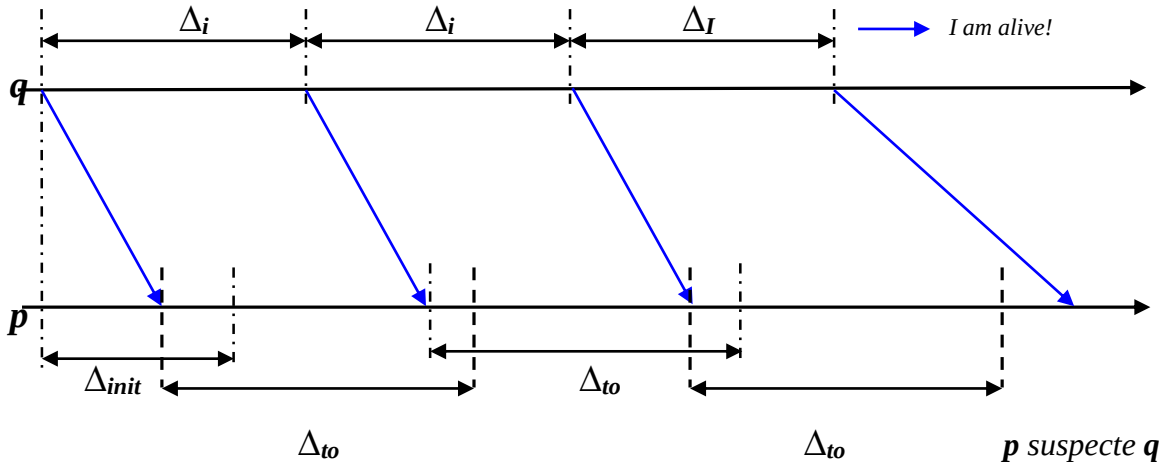

Figure 1. Détection de défaillances : l'implémentation Heart-Beat

B. Le détecteur de défaillances Interrogation

L'interrogation est une autre technique bien connue pour l'implémentation de détecteurs de défaillances Le principe du détecteur de défaillances Interrogation est, comme le principe de Heart-Beat, très simple. Il génère cependant un peu plus de messages que le mécanisme précédent. Un processus questionne périodiquement tous les autres processus pour savoir s'ils sont vivants en envoyant régulièrement des messages ‘’*Are you alive ?* ‘’. Il attend ensuite les message de réponses ‘ *’I am alive*’’. Le processus qui n'a pas répondu dans les délais est alors suspecté. Si la réponse parvient plus tard, le processus en question est réhabilité. La période d'envoi des interrogations doit être plus grande que la durée d'attente des réponses.

Avec cette implémentation, un détecteur de défaillances est également défini par deux paramètres : la période d'interrogation Δ_i et le délai de timeout Δ_{to}.

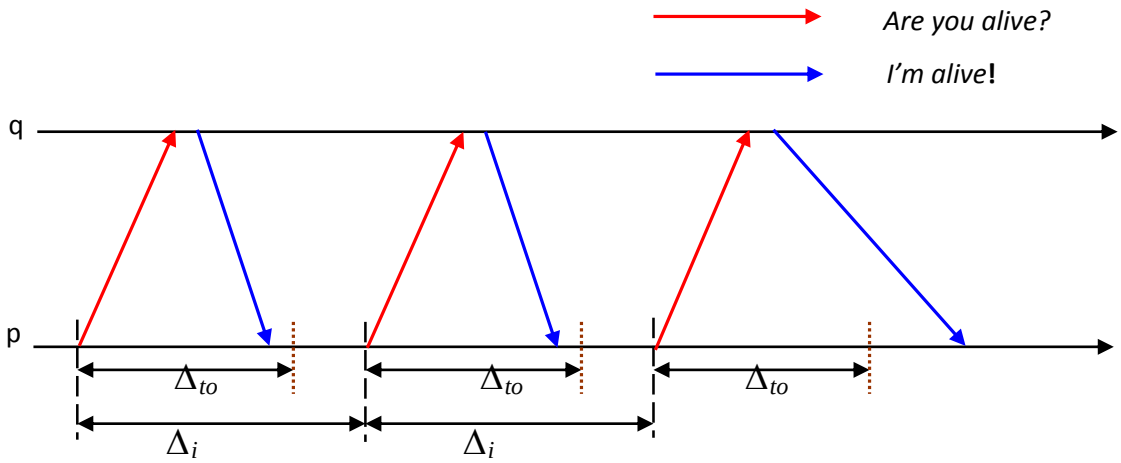

Figure 2. Détection de défaillances : l'implémentation Interrogation

C. Le détecteur de défaillances Ad hoc ‘’ no message’’

L’implémentation de ce mécanisme de détection de défaillances provient de l’observation que l’information fournit par un détecteur de défaillances est demandée par certain processus p_i seulement aux points très spécifique durant son exécution pour l’algorithme du consensus. Le concept de message critique, dans le contexte du consensus, a été défini. Un message critique est soit une estimation d'un processus, soit la proposition du coordonnateur envoyée en réponse. Le but recherché par ce principe est de diminuer le nombre de messages générés par le détecteur de défaillances. Nous pouvons résumer les détails de l’algorithme de consensus en introduisant la notion du *message critique* et de *réponse critique* (figure 3). Le message critique et la réponse critique sont des messages générés par l’algorithme exécuté par le processus *p* et le processus *q*. Le message critique m_p envoyé par *p* à *q* est tel que, après avoir envoyé m_p, le processus *p* attend le message m_q de *q*. Si *q* se crashe, alors *p* doit inéluctablement suspecter *q* et arrêter d’attendre le message m_q. Ce mécanisme peut être implémenté par un mécanisme simple de *timeout*, sans aucun message supplémentaire de ‘’détection de défaillances’’. Si *p* ne reçoit pas m_q après un délai Δ_{to}, alors *p* suspecte q. Cette implémentation de détecteur de défaillances est extrêmement bonne en termes de messages. Car, elle ne génère aucun message de ‘’détection de défaillances. L'inconvénient de cette implémentation de détecteurs de défaillances est la possibilité de produire des suspicions erronées.

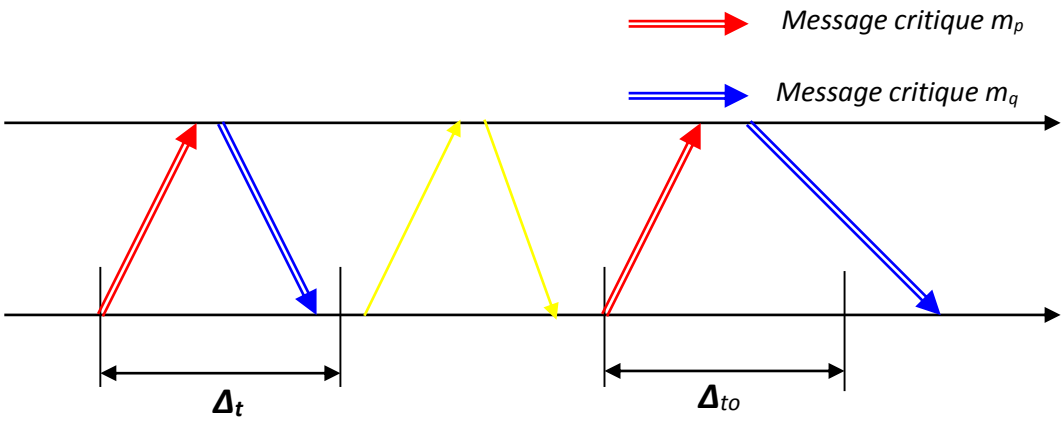

Figure 3. Détection de défaillances : l’implémentation Ad hoc ‘’*no message*’’

D. Le détecteur de défaillances Ad hoc Heart-Beat

Dans le cadre du consensus, le processus consulte son détecteur de fautes durant la période entre l'envoi et la réception de messages critiques. La première idée qui vient à l'esprit

est donc, après l'envoi d'un message critique, d'attendre la réponse critique durant un certain nombre de temps. Ensuite, si le processus n'a rien reçu, il suspecte le coordonnateur. Mais ce principe risque d'engendrer beaucoup de suspections incorrectes. En effet, avant d'envoyer la réponse critique, le coordonnateur doit attendre qu'au moins la moitié des processus lui aient envoyé leur estimation. Il risque donc de ne pas pouvoir répondre tout de suite, et d'être ainsi faussement suspecté par les processus qui attendent sa proposition.

Le moyen de remédier à cet inconvénient engendré par le mécanisme Ad hoc ‘’no message ‘’ est le mécanisme Ad hoc Heart-Beat.

Le mécanisme Ad hoc Heart-Beat fonctionne comme suit : à chaque fois le processus *q* reçoit message critique m_p d'un processus *p*, il lui envoie des messages de type Heart-Beat jusqu'à ce qu'il puisse lui envoyer son estimation m_q (figure 4).

Un détecteur de défaillances Ad hoc Heart-Beat est caractérisé par les trois paramètres Δ_{init}, Δ_i, et Δ_{to}. Le paramètre Δ_i détermine la fréquence des messages Heart-Beat, tandis que Δ_{init} et Δ_{to} définissent le délai de timeout initial et le délai de timeout périodique respectivement.

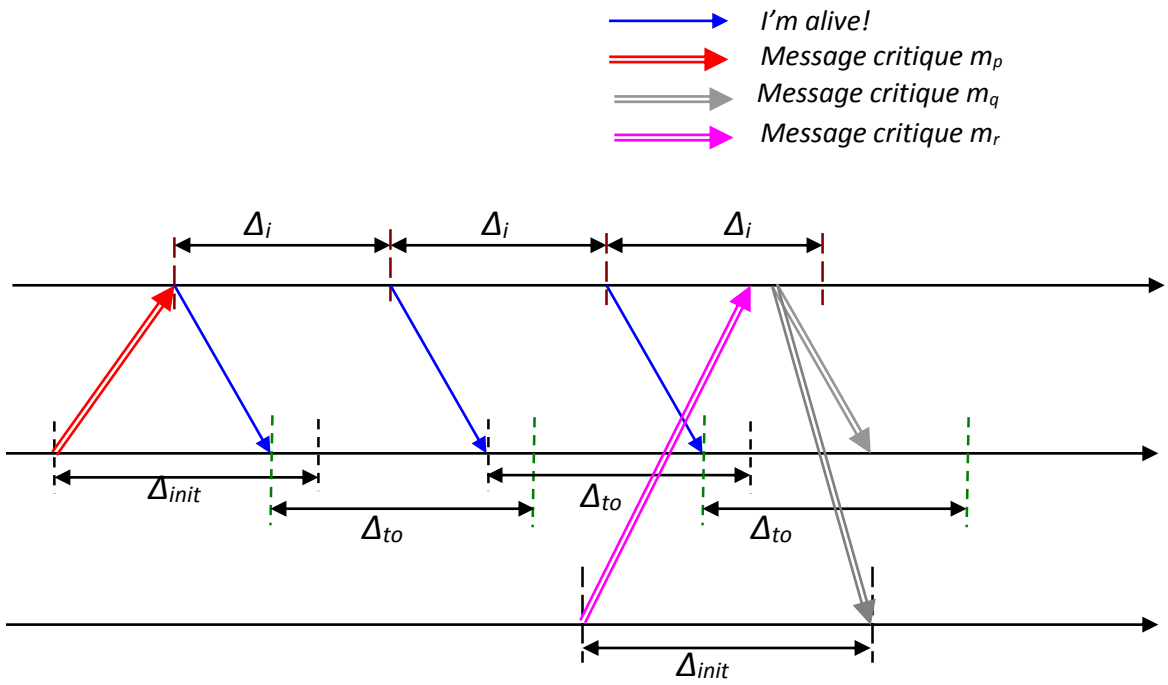

Figure 4 Détection de défaillances : l'implémentation Ad hoc Heart-Beat

2.1.2.2. Des protocoles implémentant les détecteurs de défaillances en utilisant le timeout

Dans cette section, nous allons présenter trois exemples de protocole implémentant les détecteurs de défaillances basés sur le mécanisme de timeout.

a)- Le protocole de Chandra et Toueg

En 1996, Chandra et Toueg ont proposés la première implémentation des détecteurs de défaillances pour la classe inéluctablement parfaite $\Diamond P$. cette classe des détecteurs de défaillances est caractérisée par deux propriété la complétude forte et la précision inéluctablement forte. Généralement, il est facile d'implémenter un détecteur de défaillances de cette classe dans l'un des deux modèles de système partiellement synchrone M_1 et M_2 définis par Dwork et al [1988]. En fait, nous pouvons implémenter un tel détecteur de défaillances dans un modèle partiellement synchrone plus faible M_3 [Chandra et Toueg 1996]. Avec cette implémentation, nous pouvons résoudre les problèmes de consensus et la diffusion atomique dans ces trois modèles de synchronisme partiel.

- **Principe du protocole :**

L'implémentation des détecteurs de défaillances de la classe $\Diamond P$ pour le modèle M_3 fonctionne comme suit (voir figure 5) :

Pour mesurer le temps écoulé, chaque processus *p* maintient une horloge locale. Le protocole se compose de trois tâches T_1, T_2 et T_3. Périodiquement, chaque processus *p* envoie un message *''p-is-alive''* à tous les processus du système (T_1). Si *p* n'a pas reçu un message *''q-is-alive''* de certain processus *q* pendant un timeout $\Delta_p(q)$, *p* ajoute *q* à sa liste des processus suspectés (T_2). Si *p* reçoit un message *''q-is-alive''* de certain processus *q* qui est actuellement, p sait que son timeout précédent sur *q* était prématuré. Dans ce cas, p enlève q de sa liste des processus suspectés et incrémente sa période de timeout $\Delta_p(q)$ (T_3).

```
Every process p executes the following:
output_p ← ∅
for all q ∈ Π                    {Δ_p (q) denotes the duration of p's time-out interval for q}
Δ_p (q) ← default time-out interval

cobegin
|| Task 1: repeat periodically
  send ``p-is-alive'' to all

|| Task 2: repeat periodically
 for all q ∈ Π
    if q ∉ output_p and
        p did not receive ``q-is-alive'' during the last Δ_p (q) ticks of p's clock
        output_p ← output_p ⋃ {q}     {p times-out on q: it now suspects q has crashed}

|| Task 3: when receive ``q-is-alive'' for some q
  if q ∈ output_p                    {p knows that it prematurely timed-out on q}
        output_p ← output_p - {q}     {1. p repents on q, and}
        Δ_p (q) ← Δ_p (q) + 1         {2. p increases its time-out period for q}
coend
```

Figure 5. L'implémentation basé sur le timeout de $\Diamond P$ dans le modèle M_3

b)- Le protocole de Larea et al

Larea et al ont proposé une implémentation optimale du détecteur de défaillances le plus faible pour résoudre le consensus [2000]. Cette implémentation est optimale par rapport autres implémentation existes en termes de nombre de messages et de quantité d'information totale envoyés périodiquement. Dans cette section, nous présentons un algorithme qui implémente les détecteurs de défaillances de la classe $\Diamond S$. Cet algorithme garantit qu'inéluctablement tous les processus corrects accordent sur un processus correct commun. Cette propriété permet trivialement de fournir les deux propriétés de la complétude forte et de la précision inéluctablement faible, exigées par $\Diamond S$.

Le modèle du système

Le système distribué considéré pour ce protocole se compose d'un ensemble fini de *n* processus, $\Pi = \{P_1, P2_{,\ldots,} Pn\}$, qui communiquent seulement par l'envoie et la réception de messages. Nous supposons que le réseau est complètement connecté et les canaux de communication sont fiables. Un processus peut être tombé en panne par arrêt définitif. Dans chaque exécution du système nous identifions deux sous-ensembles complémentaires de Π : le sous-ensemble des processus en panne, dénoté *crached, et* le sous-ensemble des processus qui ne sont pas en panne, dénoté *correct*. Nous utilisons *C* pour dénoter le nombre de processus corrects ($C = |correct| > 0$). Nous considérons que les processus sont ordonnés. Sans perte de généralité, le processus p_i est précédé par les processus $p_{1,\ldots,}\ p_{i-1}$, et suivi par les processus $p_{i+1,\ldots,}\ p_n$. Nous considérons aussi le modèle de système partiellement synchrone de Chandra et Toueg M_3.

Un algorithme implémentant les détecteurs de défaillances de la classe ◊S

Larea et al ont proposé un algorithme pour implémenter les détecteurs de défaillances de la classe. Cet algorithme garantit que inéluctablement tous les processus corrects accordent sur un processus correct commun, dénoté p_{leader}. Cette propriété permet trivialement de fournir la propriété de précision inéluctablement faible exigée par $\Diamond S$: Inéluctablement, p_{leader} n'est pas suspecté par aucun processus correct. La propriété de complétude $\Diamond S$ est atteinte en faisant simplement chaque processus p_i suspecte tous les processus dans le système à l'exception de p_{leader}.

Chaque processus p_i exécute une copie de l'algorithme de la figure 6, dans laquelle il y a une variable locale $trusted_i$. Pour chaque processus correct p_i, la valeur de $trusted_i$ finalement sera la même, et $trusted_i$ sera le processus correct p_{leader}. En fait, $trusted_i = leader$ sera l'indice du premier processus $\in correct$ dans le système (suivant l'ordre $p_{1,\ldots,}\ p_n$). À partir de la valeur de $trusted_i$, nous pouvons dériver l'ensemble $suspected_i$ qui satisfera les propriétés de la complétude et de la précision de $\Diamond S$. Maintenant, nous considérons deux possibilités, faire $suspected_i = \{trusted_i\}$ ou $suspected_i = \{trusted_i, p_i\}$, selon si nous voulons préserver la notion intuitive qu'un processus ne se suspecte jamais.

```
Every process p_i, i = 1,…, n executes:
trustedt_i ← 1
∀j ∈{1,…, i-1}: Δ_i,j ← default timeout
received_i ← false
cobegin
|| Task 1:
    loop
       if i = trusted_i then
          send I-AM-ALIVE to p_i+1,…, p_n
       end if
       delay Δ_HEARTBEAT
    end loop
|| Task 2:
    loop
       if trusted_i < i then
         delay Δ_i,trustedi
          if received_i then
             received_i ← false
          else
             trusted_i ← trusted_i + 1
          end if
       end if
    end loop
|| Task 3:
    loop
       receive I-AM-ALIVE from p_j
       if j = trusted_i then
         received_i ← true
       else if j < trusted_i then
            Δ_i,j ←Δ_i,j +1
          trusted_i ← j
          received_i ← true
       else
          discard message
       end if
    end loop
coend
```

Figure 6 : algorithme utilisé pour implémenter les détecteurs de défaillances de $\Diamond S$

-Principe du protocole :

L'algorithme de la figure 6 fonctionne comme suit : Initialement, chaque processus p_i commence avec $trusted_i = 1$, qui signifie que p_i sera leur premier candidat d'être le processus p_{leader}. Le processus p_1 commence à envoyer des messages ''I-AM-ALIVE'' périodiquement aux reste des processus $p_2, \ldots, p_n$. En général, un processus p_i enverra des messages ''I-AM-ALIVE'' périodiquement à ses successeurs $p_{i+1}, \ldots, p_n$ si $i = trusted_i$

(*Task 1*). Un processus p_i tel que $trusted_i \neq i$, attend juste des messages périodiques ''I-AM-ALIVE'' du processus $truseted_i$. S'il n'a pas reçu un message ''I-AM-ALIVE'' durant un timeout $\Delta_{i,trusted_i}$, alors il suspecte le processus $trusted_i$ et choisi le prochain candidat d'être le processus p_{leader} en incrémentant $trusted_i$ par un (*Task 2*). Si, plus tard, un processus p_i reçoit un message ''I-AM-ALIVE'' d'un processus p_j, tel que $j < trusted_i$, alors p_i sera arrêté de supposer que p_j est crashé, et trustera p_j encore (en faisant $trusted_i = j$). Afin d'empêcher ceci de se produisant un nombre infini de fois, p_i incrémente aussi la valeur de timeout $\Delta_{i,j}$ (*Task 3*). De plus, si p_i a envoyé périodiquement des messages ''I-AM-ALIVE'', il sera arrêté automatiquement de les envoyer, depuis que $i \neq trustedi$.

2.2. La famille qui n'utilise pas le temps physique (asynchrone) :

Contrairement à la famille de détecteurs de défaillances qui utilise le temps physique pour implémenter ceux-ci, Mostefaoui, Mourgaya, et Raynal [2003] ont proposé une deuxième famille des détecteurs de défaillances. Cette famille est une nouvelle approche en ce qui concerne l'implémentation asynchrone des détecteurs de défaillances.

Cette approche n'est pas fondée sur des hypothèses de synchronie, c.-à-d., n'utilise pas le temps physique dans l'implémentation. Elle base sur un mécanisme de questions et de réponses (Query-Response mechanism). L'implémentation asynchrone des détecteurs de défaillances nécessite certaines hypothèses pour être correcte.

2.2.1-Le modèle du système :

Nous considérons un système distribué comme composé d'un ensemble de n processus $(n \geq 3)$. Ces processus peuvent être sujets à des défaillances par crash. Le système de communication est fiable, c.-à-d., aucune défaillance sur les canaux de communication, et que le réseau est complètement connecté. Les processus communiquent entre eux par l'échange de messages. Il n'y a aucune borne sur les délais de transmission des messages, les vitesses relatives des processeurs, et la dérive de l'horloge. Les auteurs ont supposé également l'existence d'une horloge globale discrète, pour l'exploiter uniquement pour prouver les propriétés des protocoles.

2.2.2-Le mécanisme de questions et de réponses (Query-Response mechanism):

chaque processus a accès à deux primitives de communication *interrogation()* et *réponse().* Ainsi, tous les processus peuvent interroger les autres processus grâce à la première primitive, et répondre à une interrogation d'un processus quelconque grâce à la primitive réponse(). Un cycle d'interrogation commence par un appel à *interrogation()* et se termine par

la réception de $(n-f)$ premiers messages répondant à cette interrogation. Les cycles ne se chevauchent pas mais se succèdent jusqu'à ce que le processus soit défaillant. Ils ont supposé aussi que la réponse d'un processus à sa propre interrogation arrive toujours parmi les $(n-f)$ premières réponses.

2.2.3. Une implémentation asynchrone des détecteurs de la classe S et ◊S

Une propriété comportementale

Soit $t \in T$, les notations suivantes sont utilisées dans le suivant :

- q_j^t désigne la dernière interrogation, faite par le processus p_j, terminée avant la date t.

- $\mathrm{Re}c\_de_j^t$ désigne l'ensemble des processus desquels p_j a reçu un message. Ce message appartient aux $(n-f)$ premiers messages, en réponse à son interrogation q_j^t.

- r_i^t désigne l'ensemble des processus p_j, qui à l'instant *t*, ont reçu une réponse du processus p_i à l'interrogation q_j^t $(j \in [1,n])$. Nous avons donc $r^t = \{p_j, p_i \in \mathrm{Re}c\_de_j^t\}$, c.-à-d., p_i fait des $(n-f)$ processus les plus rapides du système pour l'interrogation courante (q^t).

- f^t désigne l'ensemble les processus qui ont subi une défaillance par crash avant l'instant *t*.

- $Ns_i^t = r_i^t \cup f^t$ contient des processus p_j qui ont reçu une réponse du processus p_i à l'interrogation q_j^t $(j \in [1,n])$, ou qui ont subi une défaillance par crash.

- $PR(\chi)$ désigne la propriété comportementale nécessaire pour que l'implémentation d'un détecteur de classe χ soit correct ($\chi \in \{S, \Diamond S\}$).

Formellement, les deux propriétés comportementale s'énoncent comme suit :

- $PR(S) \quad \underline{\underline{def}} \quad \exists p_i : \left| \bigcap_{t \geq 0} (r_i^t \cup f^t) \right| > f$,
- $PR(\Diamond S) \quad \underline{\underline{def}} \quad \exists u \in T : \exists p_i : \left| \bigcap_{t \geq u} (r_i^t \cup f^t) \right| > f$.

$PR(S)$: est la propriété comportementale qui caractérise un détecteur de défaillances de la classe S. Cette propriété énonce qu'il existe processus p_i et un ensemble Q contenant au moins $(f+1)$ processus tels que depuis le début de l'exécution $(t=0)$, chaque processus $p_j \in Q$ a reçu une réponse de p_i jusqu'à ce qu'éventuellement il ait subi une défaillance par crash.

$PR(\Diamond S)$: est la propriété comportementale qui caractérise un détecteur de défaillances de la classe $\Diamond S$. Cette propriété est similaire à la première mais plus faible, c.-à-d., depuis l'instant u $(u > 0)$, chaque processus $p_j \in Q$ a reçu une réponse de p_i jusqu'à ce qu'éventuellement il ait subi une défaillance par crash.

2.2.4. Un protocole générique implémentant les détecteurs de défaillances

Principe du protocole :

Le protocole décrit dans la figure 7 a été proposé par Mostefaoui et al [2003]. Ce protocole implémente le module FD_i du détecteur de défaillances qui fournit au processus p_i l'ensemble des processus suspectés $suspected_i$. Cet ensemble satisfait les propriétés de complétude et de précision exigées par S (resp. $\Diamond S$) quand la propriété $PR(S)$ (resp. $PR(\Diamond S)$) du système est satisfaite. Le protocole se compose de deux tâches T_1 et T_2.

Le protocole utilise et construise les ensembles suivants :

- $not\_rec\_from_i$: contient l'ensemble des processus pour lesquels le processus p_i n'a pas eu de réponse lors de l'interrogation courante.
- $suspected_i$: cet ensemble contient l'ensemble des processus suspectés par le module de détection de défaillances *FDi.*

```
(1) init  not_rec_from_i ← ∅ ; suspected_i ← ∅

task T1:
      repeat forever
(2)       foreach j do send QUERY() to p_j enddo;
(3)       wait until (RESPONSE(not_rec_from_i) received from (n - f) distinct processes);
(4)       let I= the set of the processes from which p_i received a RESPONSE message;
(5)       let X= the set of the not_rec_from sets received from the processes in I;
(6)       not_rec_from_i ← Π − I
(7)       suspected_i ← ⋂_{ls∈X} ls
      end repeat

task T2:
(8)  upon reception of QUERY() from p_j do send RESPONSE(not_rec_from_i) to pj
```

Figure 7.: Module FD_i du détecteur de défaillance associé à P_i

Le rôle de T_2 (la ligne 8) est simple : elle implémente le mécanisme de questions et de réponses (query-response mechanism) en retournant un message RESPONSE() à chaque fois qu'une QUERY() est reçue.

Le cœur du protocole est la tache T_1. Cette tâche se compose d'une boucle infinie. Chaque étape de la boucle consiste en une invocation query-response (les lignes 2-3) suivi d'un traitement local qui, selon l'ensemble de processus à partir lesquels les messages de RESPONSE() ont été reçus (la linge 4) et le contenu de ces messages (la ligne 5), met à jour la variable locale $not\_rec\_from_i$ (la linge 6) et la variable $suspected_i$, respectivment.

2.3.La famille hybride :

Mostefaoui,et al [2004] ont montré que les deux familles d'implémentation des détecteurs de défaillances précédentes, celle qui utilise le temps physique et celle qui ne l'utilise pas, ne sont pas antagonistes et peuvent être combinées pour construire un protocole hybride qui implémente les détecteurs de défaillances inéluctablement précis. Ce protocole peut être classifié dans une nouvelle famille des détecteurs de défaillances, appelée la famille hybride. Le protocole de cette famille bénéficie des deux meilleurs approches universelles, parce qu'il converge dés que soit le système se comporte synchronement, soit le modèle d'échange de message se produit.

Plus explicitement, si seulement une des hypothèses suivantes est satisfaite :

- Synchronisme partiel,
- Propriété comportementale sur le modèle d'échange de messages,

Alors, le protocole hybride proposé implémente un détecteur de défaillances qui est inéluctablement précis. Si les deux propriétés sont satisfaites, soit t_S l'instant après lequel le système se comporte synchronement, et soit t_M l'instant après lequel la propriété comportementale sur le modèle d'échange de message est satisfaite. Dans ce cas la précision inéluctable est obtenue à partir du $\min(t_S, t_M)$.

Dans cette section, en premier, nous présentons le modèle du système et les hypothèses supplémentaires considérées pour implémenter les détecteurs de défaillances des classes inéluctablement précises. Puis, nous présentons un protocole hybride qui implémente les détecteurs de défaillances de la classe $\Diamond S$.

2.3.1. Modèle de système et hypothèses

-Modèle de système

Nous considérons un système distribué comme composé d'un ensemble de n processus $(n \geq 3)$, c.-à-d., $\Pi = \{p_1, p_2, ..., p_n\}$. Ces processus peuvent être sujets à des défaillances par crash. Le nombre des processus qui peuvent être crashés est f $(1 \leq f < n)$. Les processus communiquent entre eux par l'envoie et la réception de messages via des canaux. Chaque paire de processus est connectée par un canal. Le système de communication est supposé fiable, c.-à-d., les canaux ne créent pas, ne modifient pas ou ne détruisent pas les messages. Il n'y a aucune borne sur les délais de transmission des messages, les vitesses relatives des processeurs, et la dérive de l'horloge. Les auteurs ont supposé également l'existence d'une horloge globale discrète, pour l'exploiter uniquement pour prouver les propriétés des protocoles. Le rang T de l'horloge est l'ensemble des entiers naturels.

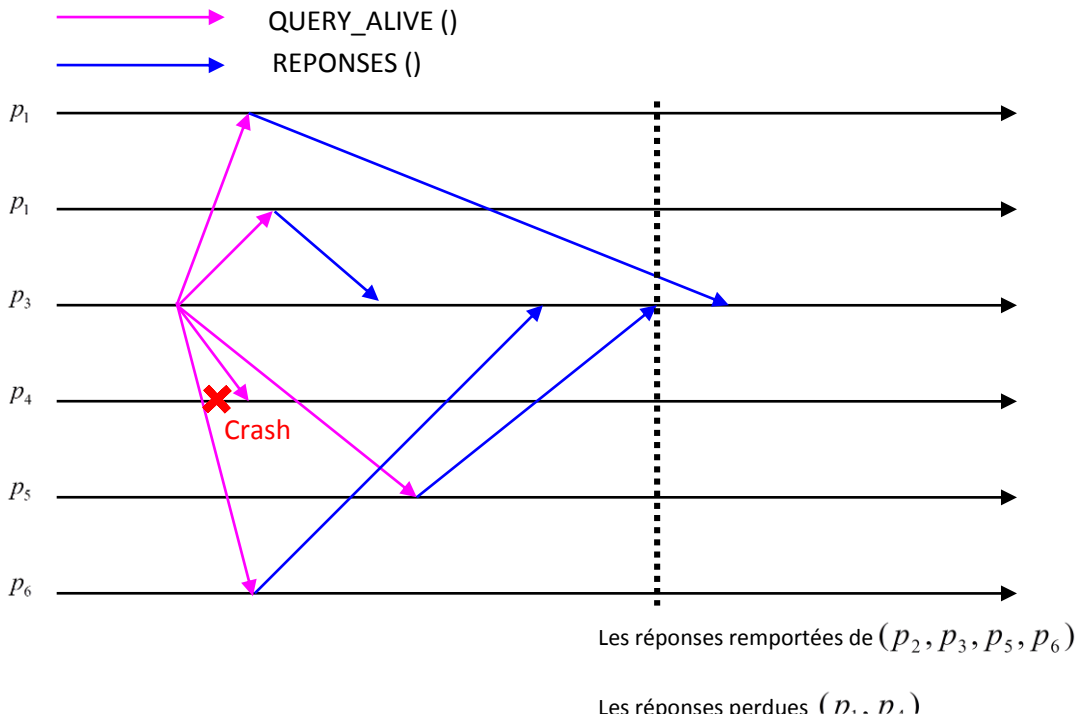

Figure 8. Le mécanisme Question/Réponse (Query/Response)

Plus spécifiquement, n'importe quel processus p_i peut diffuser un message QUERY_ALIVE() et puis attend les messages RESPONSE() correspondants de $(n-f)$ processus. Les autres messages RESPONSE() associés à une requête sont systématiquement rejetés.

Une requête invoquée par p_i est terminée si p_i a reçu les $(n-f)$ premières réponses correspondantes. Un processus invoque une nouvelle requête seulement si la précédente est

terminée. Sans perte de généralité, la réponse d'un processus à ses propres requêtes est supposée arriver parmi les $(n-f)$ premières réponses.

La figure 8 représente un mécanisme de query-response dans un système réparti qui se compose de $n=6$ processus, et $f=2$.

-Détecteurs inéluctables de défaillances

Dans cette section, nous nous intéressons à la classe de détecteurs de défaillances inéluctablement forts. Cette classe est dénotée $\Diamond S$.

-Hypothèses supplémentaires

Mostefaoui et al [2004] ont défini deux types d'hypothèses supplémentaires qui permettent l'implémentation des détecteurs de défaillances de la classe $\Diamond S$

A- Synchronisme partiel

L'hypothèse de synchronisme partiel est définie comme suit :

- La Propriété *PS* **:** il existe un instant t_S après lequel ils existent des bornes sur les vitesses relatives des processus et sur les délais de transmission des messages, mais ces bornes ne sont pas connues.

D'une façon générale, un système évolue par une séquence de périodes stables et longues, Séparées par (relativement courtes) des périodes instables. Dans une période stable, le comportement du système est prévisible, c.-à-d., ils existent des bornes sur les vitesses relatives des processus et sur le temps de traitement.

B- Propriétés comportementales sur le modèle d'échange de messages :

Deux propriétés comportementales ont été définies. Ces deux propriétés caractérisent le modèle d'échange de messages impliqué dans le mécanisme de query-response.

Les deux propriété sont définies comme suit :

- La propriété MP_W (Weak Message Pattern) : il existe un instant t_{Mw}, où un processus correct p_i et un ensemble Q de $(f+1)$ processus (p_i, t_{Mw}, et Q ne sont pas connus à l'avance), tel que, après t_{Mw}, chaque processus $p_j \in Q$ obtient une réponse remportée de p_i à chacune de ses requêtes (jusqu'à ce que p_j peut être crashé) .
- La propriété MP_S (Strong Message Pattern) : il existe un instant t_{Ms} après lequel, $\forall\ p_i$, il existe un ensemble Q_i de $(f+1)$ processus p_j tel que, tout processus

obtient une réponses remportée de p_i à chacune de ses réponses (jusqu'à ce que p_i ou p_j crashé). Comme précédemment, l'instant t_{Ms}, et les ensembles Q_i ne sont pas connus à l'avance.

Généralement, même si le système n'est pas stable, il est possible que son comportement ait certaine ''régularité'' qui peut être exploiter pour construire les détecteurs de défaillances. Cette régularité peut être vue comme certain ''synchronisme logique'' (par opposition au synchronisme ''physique'' capturé par la propriété *PS*).

2.3.2. Un protocole hybride pour ◊S.

Ce protocole se compose de trois tâches (figure 9). Le but des taches T_1 et T_2 est de gérer les deux ensembles des processus suspectés, appelés, $suspected\_MP_i$ (l'ensemble de processus suspectés en ce qui concerne l'hypothèse MP_W), et $suspected\_PS_i$ (l'ensemble de processus suspectés en ce qui concerne l'hypothèse *PS*). Le but de la tâche T_3 est de répondre sur les appels émis par la couche supérieure de l'application quand elle invoque la primitive SUSPECT() (la linge 14). Le demandeur est alors fourni avec l'ensemble de processus qui sont actuellement suspectés, dénoté, $suspected\_MP_i \cap suspectedPS_i$ (les lignes 15-16).

Les autres variables locales gérées par un processus p_i sont : un ensemble $not\_rec\_from_i$ et un tableau $\Delta_i[1..n]$. L'ensemble $not\_rec\_from_i$ est utilisé pour sauvegarder les processus dont les réponses de la dernière requête de p_i n'ont pas été reçues parmi les $(n-f)$ premières. $\Delta_i[j]$ est un compteur de temps utilisé par p_i pour détecter la crash possible de p_j quand considérer l'hypothèse *PS*.

Périodiquement, p_i émet des messages QUERY_ALIVE() (la ligne 3). La répétition de la période de la tache T_1 n'a aucune incidence sur la correction de protocole. Un message QUERY_ALIVE() a un double but : l'activation d'un mécanisme requête-réponse (ceci est liée à l'hypothèse MP_W), et l'indication aux autres processus que p_i est vivant (ceci liée à l'hypothèse *PS*). Après qu'il ait émis une requête, p_i attend jusqu'à ce qu'il ait reçu les $(n-f)$ réponses correspondantes (la ligne 4). Alors, il détermine l'ensemble de f processus dont il n'a pas reçu des réponses et met à jour $not\_rec\_from_i$ par conséquent.

La gestion de l'ensemble $suspected\_MP_i$ est plus subtile. Nous observons que chaque message RESPONSE() porte la valeur courante de l'ensemble $not\_rec\_from_i$ de son émetteur p_j (la

ligne 10). Après avoir reçu les $(n-f)$ premières réponses qu'elles sont attendues, p_i met à jour $suspected\_MP_i$ pour être égal à l'intersection des $(n-f)$ ensembles $not\_rec\_from_j$ (la ligne 8).

(1) **init**: $not\_rec\_from_i \leftarrow \emptyset$; $suspected\_MP_i \leftarrow \emptyset$; $suspected\_PS_i \leftarrow \emptyset$;
(2) **for each** *j* **do** $\Delta_i[j] \leftarrow$ default timeout value **end do**;
Task *T*1:
repeat periodically
(3) **foreach** *j* **do** send QUERY ALIVE() to p_j **enddo**;
(4) **wait until** (RESPONSE(not_rec_from) received from (*n* - *f*) processes)
(5) **let** *I*= the set of the processes from which p_i received a RESPONSE message;
(6) **let** *X*= the set of the not_rec_from sets received from the processes in *I*;
(7) $not\_rec\_from_i \leftarrow \Pi - I$
(8) $suspected\_MP_i \leftarrow \bigcap_{x \in X} x$
end repeat
task *T*2:
(9) **upon reception of** QUERY ALIVE() **from** p_j
(10) **do** send RESPONSE($not\_rec\_from_i$) **to** p_j ;
(11) **if** ($p_j \in suspected\_PS_i$)**then** $\Delta_i[j] \leftarrow \Delta_i[j] + 1$ **end if**;
(12) $suspected\_PS_i$ ={p_k /no QUERY ALIVE() has been received from p_k during the last $\Delta_i[k]$time units};
(13) **end do**

task *T*3:
(14) **when** SUSPECT() **is invoked by the upper layer**
(15) **do let** $suspected_i$ = $suspected\_MP_i \cap suspected\_PS_i$;
(16) **return** ($suspected_i$)
(17) **end do**

Figure 9. Module de détecteur de défaillances FD_i associé avec p_i

Maintenant considérons la gestion de l'ensemble $suspected\_PS_i$ **.** Chaque fois un processus p_i reçoit un message QUERY_ALIVE() de certain processus p_j , il incrémente la valeur du timeout associée avec p_i si $p \in suspected\_PS_i$ (la ligne 11). Alors, p_i recalcule la valeur de cet ensemble lequel, dont dorénavant, inclus les processus p_k de quel p_i n'a pas reçu un message QUERY_ALIVE() depuis $\Delta_i[k]$ unités de temps. Nous observons que la valeur recalculée de

$suspected\_MP_i$ (la linge 12) n'incluse pas p_j depuis p_i juste reçu un message QUERY_ALIVE() de p_j et $\Delta_i[j]>0$.