Détections de défaillance, de connectivité et de déconnexion

Lynda Temal et Denis Conan

GET / INT, CNRS UMR SAMOVAR
9 rue Charles Fourier
91011 Évry, France
Denis.Conan@int-evry.fr

02 avril 2004

Résumé

En environnement mobile, le terminal mobile est sujet à des déconnexions, par exemple, lors du passage de l'utilisateur dans une zone d'ombre radio. En outre, l'utilisateur exécute des applications réparties dont la défaillance d'un des processus fait défaillir l'ensemble. Ces deux propriétés extrafonctionnelles (gestion des déconnexions et tolérance aux fautes) reposent chacune sur des mécanismes de détection (de connectivité et de déconnexion, et de défaillance).

Dans ce papier, nous définissons les concepts de détecteur de connectivité et de détecteur de déconnexion, par comparaison avec le détecteur de défaillance non fiable. Ces trois types de détecteurs sont ensuite utilisés conjointement pour étendre la gestion de déconnexion et améliorer la tolérance aux fautes.

1 Introduction

La communication sans fil, le traitement d'informations personnelles et les services d'informations réparties auront une importance stratégique dans l'avenir proche. Il existe déjà des terminaux mobiles de type PDA (*Personal Digital Assistant)* commercialement disponibles fournissant ces services.

Malheureusement, le terminal mobile est sujet à des déconnexions. Nous considérons deux types de déconnexions : les déconnexions volontaires et les déconnexions involontaires. Les premières, décidées par l'utilisateur depuis son terminal mobile, sont justifiées par les bénéfices attendus sur le coût financier des communications, l'énergie, la disponibilité du service applicatif, et la minimisation des désagréments induits par des déconnexions inopinées. Les secondes sont le résultat de coupures intempestives des connexions physiques du réseau, par exemple, lors du passage de l'utilisateur dans une zone d'ombre

radio. Les travaux de recherche précédents [7] ont mis en exergue un algorithme de détection de connectivité permettant de décider si la requête de l'utilisateur peut être transmise ou non sur le réseau sans fil. Un mécanisme de chargement de certaines entités sur le terminal mobile permet de continuer à travailler localement lors de la déconnexion. Un autre mécanisme intervient lors de la reconnexion pour effectuer une réconciliation.

En outre, l'utilisateur exécute des applications réparties dont la défaillance d'un des processus fait défaillir l'ensemble. Dans un système (complètement) asynchrone, il est impossible d'obtenir un consensus entre processus répartis dès qu'un seul d'entre eux est défaillant (par exemple, par un *crash*) [9]. Cette impossibilité résulte de la difficulté de dire si le processus distant est juste lent ou s'il est défaillant. Pourtant, beaucoup de mécanismes de tolérance aux fautes nécessitent de telles décisions, ne serait-ce que pour déterminer de façon cohérente l'ensemble des processus corrects (non défaillants) à un instant donné. Pourtant, de nombreuses solutions de tolérance aux fautes existent déjà ; elles font l'hypothèse d'un modèle de système réparti moins faible appelé partiellement synchrone [8]. Dans ce modèle, la plupart des solutions de tolérance aux fautes utilisent des détecteurs de défaillance non fiables [6].

Ces deux propriétés extrafonctionnelles (gestion des déconnexions et tolérance aux fautes) reposent chacune sur des mécanismes de détection de situations la plupart du temps « anormales » (détections de connectivité et de déconnexion, et détection de défaillance, respectivement). L'objectif de ce papier est d'initier la définition et la comparaison des objectifs, des types de détection en terme de propriétés et d'algorithmes, et des architectures logicielles réparties. Par exemple, les déconnexions peuvent être volontaires ; un processus ne participant pas à un consensus pourrait être considéré comme défaillant et donc exclu des prochaines diffusions ; un processus pourrait préparer sa déconnexion pour avertir les autres ou pour déposer sur un nœud du réseau fixe un mandataire accessible lors de la déconnexion.

La suite du papier est organisée comme suit. La section 2 définit le modèle de système réparti avant la présentation des trois types de détecteurs : de défaillance dans la section 3, de connectivité dans la section 4 et de déconnexion dans la section 5. Ensuite, la section 6 mixe les trois détecteurs, montrant ainsi leur complémentarité et ouvrant de nombreuses perspectives. Enfin, la section 7 conclue avec une présentation des travaux connexes et des perspectives. Les preuves des algorithmes présentés sont rassemblées dans l'annexe.

2 Modèle de système réparti

Dans le modèle de système réparti partiellement synchrone considéré, les bornes sur les délais de transfert d'un message et sur la durée nécessaire pour l'exécution d'une étape d'un processus existent, mais ne sont valables qu'après une durée de stabilisation globale (GTS, *Global Stabilisation Time*) [6], elle-même inconnue. Intuitivement, dans ce modèle, il est fait l'hypothèse que l'exécution est divisée en la succession de phases complètement asynchrones suivies de phases partiellement synchrones pendant lesquelles le système est dit stabilisé, permettant alors par exemple de prendre une décision par consensus.

Le système consiste en un ensemble de n processus $\Pi = \{p_1, p_2 ..., p_n\}$ où chaque

paire de processus est connectée par un canal de communication (ou lien) fiable. Dans Internet, la fiabilité des liens est assurée par exemple par le protocole `TCP` : retransmission du message jusqu'à la réception d'un acquittement. En outre, nous supposons que nous disposons de primitives réalisant une diffusion fiable [10]. Enfin, pour plus de clarté dans la présentation, l'existence d'une horloge globale virtuelle $\mathcal{T}$ est supposée. $\mathcal{T}$ est inaccessible par les processus, elle prend ses valeurs dans l'ensemble des entiers naturels.

Un processus peut être défaillant à cause d'un *crash*, c.-à-d. un arrêt prématuré. Le modèle de défaillance F est une fonction de $\mathcal{T}$ vers 2^{Π}, où $F(t)$ est l'ensemble des processus défaillants jusqu'à l'instant t. $crashed(F) = \bigcup_{t \in \mathcal{T}} F(t)$ représente l'ensemble des processus défaillants dans F et $correct(F) = \Pi \setminus crashed(F)$ représente l'ensemble des processus corrects dans F. En outre, nous supposons qu'il existe toujours au moins un processus correct dans F. Enfin, un processus défaillant n'est pas sujet à reprise.

Un processus peut se déconnecter volontairement ou involontairement. Le modèle de déconnexion DC est une fonction de $\mathcal{T}$ vers 2^{Π}, où $DC(t)$ est l'ensemble des processus déconnectés à l'instant t. $disconnected(DC)$ représente l'ensemble des processus déconnectés dans DC, et $connected(DC) = \Pi \setminus disconnected(DC)$ représente les processus connectés. Si un processus se déconnecte alors il le reste pour l'instance du consensus courant. Une fois le processus déconnecté, il peut se reconnecter mais il n'est pas enlevé de $DC(t)$ pendant l'exécution du consensus courant. En outre, si $p \in disconnected(DC)$, nous disons que p est vu déconnecté dans DC et si $p \in connected(DC)$, que p est vu connecté dans DC. Nous supposons aussi que tous les processus de l'application répartie sont démarrés et terminés alors que la connectivité est bonne (mode « connecté » défini dans la section 4).

Enfin, le niveau de disponibilité r d'une ressource locale (bande passante du réseau sans fil...) peut varié très brutalement ou plus lentement dans des proportions plus ou moins grandes. Par analogie avec le synchronisme des communications, nous faisons l'hypothèse que les bornes sur les variations de disponibilité des ressources (tant en valeur que dans le temps) existent, mais ne sont pas connues. Intuitivement, le système est stable sans phase de grande variabilité et permet d'adapter l'architecture de l'application répartie au mode de connectivité.

3 Détection de défaillance

Dans les systèmes répartis partiellement synchrones, l'absence de réponse de la part d'un processus ne signifie pas nécessairement qu'il est défaillant. C'est pour cela que les solutions les plus courantes utilisent des détecteurs de défaillance non fiables. La section 3.1 définit leur rôle et leurs propriétés, puis la section 3.2 présente à titre d'exemple l'algorithme « battements de cœur » noté $\mathcal{HB}$ [1].

3.1 Rôle et définition

Constatant que la détection des défaillances dans un système partiellement synchrone est obligatoirement approximative, Chandra et Toueg ont introduit la notion de détec-

teurs de défaillance non fiables [6]. Chaque processus a accès à un module de détection de défaillance local qui maintient une liste de processus suspectés d'être défaillants. Le module de détection de défaillance peut faire de fausses suspicions en ajoutant par erreur un processus non défaillant à sa liste de suspects. Si ce module découvre plus tard qu'il a fait une erreur, il enlève le processus faussement suspecté de la liste des suspects. Ainsi, le détecteur de défaillance peut ajouter et enlever les processus de sa liste des suspects plusieurs fois pendant son exécution. En outre, à un instant donné, deux modules dans deux hôtes différents peuvent avoir des listes de suspects différentes. Chandra et Toueg ont montré qu'il est possible de résoudre le problème du consensus dans des systèmes répartis asynchrones munis de détecteurs de défaillance non fiables satisfaisant certaines propriétés de *précision* (relative à la *vivacité*), limitant les fausses suspicions, et de *complétude* (relative à la *sûreté*), assurant la détection effective des processus défaillants.

Formellement, l'historique H d'un détecteur de défaillance est une fonction de l'ensemble $\Pi \times \mathcal{T}$ vers l'ensemble 2^{Π} où $H(p,t)$ est la valeur du détecteur de défaillance du processus p à l'instant t dans H. Si $q \in H(p,t)$ alors p *suspecte* q à l'instant t dans H. Si $p \neq q$ alors $H(p,t) \neq H(q,t)$ est possible. Un détecteur de défaillance $\mathcal{D}$ est une fonction qui associe à chaque modèle de défaillance F un ensemble d'historiques de détecteurs de défaillance $\mathcal{D}(F)$. $\mathcal{D}(F)$ est l'ensemble de tous les historiques pouvant exister durant les exécutions avec l'ensemble des défaillances F et le détecteur de défaillance $\mathcal{D}$. Chandra et Toueg définissent deux propriétés de complétude et quatre de précisions donnant, une fois combinées, huit classes de détecteurs de défaillance [6]. Dans le modèle de système réparti présenté en section 2, l'existence de périodes complètement asynchrones avant celles de synchronie partielle implique que la classe de détection de défaillance la plus « faible » est $\diamond W$ [5] : complétude faible et précision faible finale. Par ailleurs, comme il est aisé de passer de la complétude faible à la complétude forte par un algorithme de réduction $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ [6], l'algorithme choisi appartient à la classe $\diamond S$: complétude forte et précision faible finale.

Complétude forte : il existe un instant après lequel tout processus défaillant est suspecté d'une manière permanente par tout processus correct. Formellement :

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

Précision faible finale : il existe un instant après lequel au moins un processus correct n'est suspecté par aucun processus correct. Formellement :

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

3.2 Algorithme

Le choix de l'algorithme « battements de cœur » noté $\mathcal{HB}$ [1] repose sur deux raisons principales qui apparaîtront naturelles dans la section 6. Tout d'abord, $\mathcal{HB}$ n'utilise pas de délai de garde. Ensuite, il manipule un tableau de compteurs $D = ((p_1,\ n_1),\ (p_2,\ n_2),\ ...,\ (p_k,\ n_k))$ où $p_1,\ p_2,\ ...\ p_k$ sont les voisins de p et chaque n_j est un entier positif, au lieu d'une liste de suspects. $\mathcal{HB}$ compte juste le nombre total de battements de cœur reçus des autres processus et retourne ce tableau de compteurs sans

traitement ou interprétation. D'une manière intuitive, n_j augmente tant que p_j n'est pas défaillant. On dit que n_j est « la valeur de battements de cœur du processus p_j dans p ». L'historique de D dans p à l'instant t, $H(p, t)$, est un vecteur d'index $\{p_1, p_2, ..., p_k\}$. Ainsi, $H(p, t)[p_j]$ est la valeur n_j du compteur de p_j dans p à l'instant t. La séquence de battements de cœur de p_j dans p est la séquence des valeurs des battements de coeur de p_j dans p en fonction du temps.

Le détecteur de défaillance présenté dans l'algorithme 1 est une implantation simple de $\mathcal{HB}$. Tout processus p exécute deux tâches concurrentes. Dans la première tâche, p émet périodiquement un message HEARTBEAT à tous ses voisins. La seconde tâche manipule la réception des messages HEARTBEAT. Quand un tel message est reçu de q, p incrémente la valeur du battement de cœur de q. Bien sûr, pour construire l'ensemble des processus suspects, $\mathcal{HB}$ doit être appelé par un algorithme introduisant la notion de temps : le nombre de battements de cœur reçus pendant une période est normalisé et l'ensemble des processus suspectés est construit.

Algorithme 1: $\mathcal{HB}$: détection de défaillance non fiable

```
1  for every process p :
2   initialisation :
3    for all q ∈ neighbour(p)
4     D_p[q] ← 0                          {vector of received heartbeats}
5   cobegin :
6    || task 1 : repeat forever
7     for all q ∈ neighbour(p)
8      send(HEARTBEAT) to q
9    || task 2 : upon receive(HEARTBEAT) from q
10    D_p[q] ← D_p[q] + 1
11  coend
```

4 Détection de connectivité

Dans les environnements mobiles, les terminaux mobiles sont en général d'assez faible capacité pour que la qualité de la communication fluctue beaucoup dans le temps. C'est pour cela que les ressources locales sont contrôlées localement par un détecteur de connectivité afin que l'intergiciel autorise ou non les applications à communiquer. La section 4.1 définit le rôle et la propriété du détecteur de connectivité, puis la section 4.2 présente l'algorithme $\mathcal{CD}$.

4.1 Rôle et définition

Le détecteur de connectivité est l'entité donnant les informations sur une ressource tel que, si le niveau de disponibilité de cette ressource est insuffisant, le processus doit être déconnecté : il évalue le taux d'utilisabilité des ressources locales. Des exemples de ressources à contrôler pour la connectivité sont le niveau de la batterie du terminal mobile

et le pourcentage de la bande passante du réseau sans fil. Les détecteurs de connectivité peuvent être déployés selon plusieurs granularités possibles : de la plus grosse à la plus fine, un détecteur de connectivité par hôte, par application, par processus ou par lien logique, la dernière solution étant la granularité la plus fine. Il est facile d'imaginer des politiques hybrides.

Le détecteur de connectivité a été conçu pour la plate-forme Domint de support de la mobilité des terminaux. Dans une application répartie « classique » fonctionnant avec une bonne connectivité entre les différentes composantes, les clients peuvent s'exécuter sur des terminaux légers et n'être constitués que d'une interface graphique. Pour y ajouter la continuité de service lors de déconnexions, des mandataires des serveurs doivent être instanciés sur les hôtes des clients. Bien sûr, tous les serveurs n'ont pas de mandataires : certains ne sont pas « déconnectables », d'autres ne sont pas « nécessaires » [12]. Pratiquement, dans le mode connecté, le processus dispose d'un accès normal aux autres processus distants. Les messages sont directement envoyées aux processus distants. Dans le mode déconnecté, les messages sont uniquement transmis aux mandataires et journalisés pour être retransmis à la reconnexion pour la réconciliation. Par conséquent, lors d'une déconnexion, un transfert d'état s'opère du processus distant vers le mandataire. Réciproquement, la reconnexion implique un transfert du journal des messages et une réconciliation.

L'objectif du détecteur de connectivité est de stabiliser l'application répartie afin de minimiser le nombre de transferts d'états et de journaux. L'effet « ping-pong » intervient si le niveau de disponibilité de la ressource oscillant autour d'une valeur frontière (seuil) entre deux modes de connectivité, provoque des changements de modes répétés. Dans [11], Harbus note un effet similaire qu'il appelle « effet boomerang » dans le contexte de la migration de processus. Formellement, soit r le niveau de disponibilité normalisé entre 0 (aucune disponibilité) et 1 (disponibilité complète), (r_1, r_2, ..., r_n) la séquence infinie de valeurs de niveau de disponibilité échantillonnés au cours de l'exécution, $m \in \mathbb{N}$ le mode de connectivité (dans la suite représenté par une lettre) et M la fonction de calcul du mode au cours de l'exécution ($m_i = M(r_i)$). Soit ϵ_r l'intervalle de niveau de disponibilité dans lequel l'effet ping-pong est non désiré, exprimant ainsi le degré de stabilité supposé.

Prévention contre l'effet « ping-pong » : un niveau de disponibilité de ressource qui oscille autour d'un seuil provoque un seul changement de mode. Formellement :
$\forall r \in [0,1], \forall i \in \{i \mid m_i \neq m_{i+1} \wedge (r_i \leq r < r_{i+1} \cup r_i \geq r > r_{i+1})\}$:
$(j = i{+}2 \cup (j > i{+}2 \wedge m_{j-1} = m_{i+1})) \wedge r_j \in [max(0, r{-}\epsilon_r),\ min(1, r{+}\epsilon_r)] \implies m_j = m_{i+1}$

4.2 Algorithme

La solution classique au problème de l'effet ping-pong est d'introduire un double seuil. Dans notre cas, puisque l'effet intervient dans deux situations (déconnexion et reconnexion), un mécanisme d'hystérésis comprenant deux doubles seuils est nécessaire. Le détecteur de connectivité utilise un mécanisme d'hystérésis pour lisser les variations de niveau de disponibilité d'une ressource. En conséquence, est introduit le mode partiellement connecté, ajouté aux modes connecté et déconnecté. Pratiquement, dans le mode partiellement connecté, les messages sont transmis aux mandataires présents localement,

puis aux processus distants.

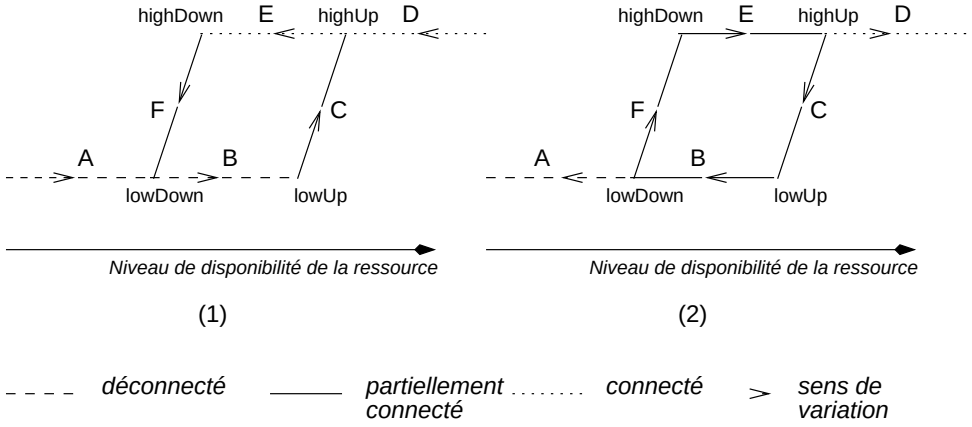

FIG. 1 – Hystérésis du gestionnaire de connectivité

Dans la figure 1-1, tant que le niveau de disponibilité croît et reste inférieur à `lowUp` (*resp.* `highUp`), le terminal mobile est déconnecté (*resp.* partiellement connecté). Lorsque le niveau de disponibilité décroît et est supérieur à `highDown` (*resp.* `lowDown`), le terminal mobile est connecté (*resp.* partiellement connecté). Sans la figure 1-2, il existe toujours un risque d'effet ping-pong autour des valeurs `highDown` et `lowUp`. Donc, quand le niveau de disponibilité arrive dans l'état `E` à partir de l'état `F` (*resp.* de l'état `B` à partir de l'état `C`) et que le niveau de disponibilité redevient supérieur à `highDown` (*resp.* inférieur à `lowUp`), le mode reste partiellement connecté jusqu'à la valeur `highUp` (*resp.* `lowDown`). Pour simplifier l'écriture de l'algorithme 2, les noms des variables représentant les seuils sont écourtées : `lU` pour `lowUp`...

Au plus proche des pilotes des ressources locales, un détecteur de connectivité surveille une seule ressource ; la valeur obtenue est soit une valeur brute soit une moyenne. Il est aisé de concevoir que d'autres détecteurs de connectivité sont créés au-dessus des premiers, organisés alors en hiérarchies. Les méthodes classiques de prise de décision sont alors applicables [14], allant des simples agrégations multicritères additives ou mutiplicatives aux décisions logiques.

5 Détection de déconnexion

Bien que le détecteur de connectivité fournisse des informations de déconnexions et de reconnexions, celles-ci restent locales. Une information de déconnexion globale est souhaitable afin d'en tenir compte dans les algorithmes répartis tels que le consensus. Pour cela, nous introduisons un détecteur de déconnexion sur chaque hôte qui surveille la connectivité locale et échange des informations avec les autres détecteurs de déconnexion. La section 5.1 définit le rôle et les propriétés du détecteur de déconnexion, puis la section 5.2 présente l'algorithme $\mathcal{DD}$.

Algorithme 2: $\mathcal{CD}$: détection de connectivité

```
1  Assumptions :
2  (1 > hU > lU > lD > 0) ∧ (hU > hD > lD)
3  lD, lU − lD, hU − lU, 1 − hU, hU − hD, hD − lD > ε_r
4  for every process p :
5   initialisation :
6    r ← MAXVALUE                                   {resource level value}
7    s ← 'D'                                                      {state}
8    m ← 'c'                                                      {mode}
9    voluntary ← false
10  cobegin :
11   || task 1 : repeat forever or upon request
12    r ← computeVivacity()
13    oldS ← s
14    switch oldS
15    case 'A'
16     if r ≥ lD then s ← 'B'
17    case 'B'
18     if r > lU then s ← 'C' ; m ← 'p'
19               else if r < lD then s ← 'A' ; m ← 'd'
20    case 'C'
21     if r > hU then s ← 'D' ; m ← 'c'
22               else if r ≤ lU then s ← 'B'
23    case 'D'
24     if r ≤ hU then s ← 'E'
25    case 'E'
26     if r < hD then s ← 'F' ; m ← 'p'
27               else if r > hU then s ← 'D' ; m ← 'c'
28    case 'F'
29     if r < lD then s ← 'A' ; m ← 'd'
30               else if r ≥ hD then s ← 'E'
31   || task 2 : upon voluntary disc./reconnection
32    voluntary ← ¬voluntary
33  coend
```

5.1 Rôle et définition

Un détecteur de déconnexion réparti est un module qui fournit une information sur les hôtes déconnectés. Le principe est qu'un processus sur le point de se déconnecter avertit les autres processus connectés de sa déconnexion et un processus qui se reconnecte avertit tous les autres.

Similairement aux détecteurs de défaillance, les détecteurs de déconnexion ne sont pas définis pour une implantation spécifique. Pour cela, nous définissons des propriétés abstraites de complétude et de précision : « complétude de déconnexion » et « précision de déconnexion ».

Formellement, l'historique H_{DC} d'un détecteur de déconnexion est une fonction de l'ensemble $\Pi \times \mathcal{T}$ vers l'ensemble 2^{Π} où $H_{DC}(p,t)$ est la valeur du détecteur de déconnexion du processus p à l'instant t dans H_{DC}. Si $q \in H(p,t)$ alors p voit q déconnecté à l'instant t. Un détecteur de déconnexion $\mathcal{DD}$ est une fonction qui associe à chaque modèle de déconnexion DC un ensemble d'historiques de détecteur de déconnexions $\mathcal{DD}(DC)$. $\mathcal{DD}(DC)$ est l'ensemble de tous les historiques pouvant exister durant les exécutions avec l'ensemble des déconnexions DC et le détecteur de déconnexion $\mathcal{DD}$.

Complétude de déconnexion forte : il existe un instant après lequel tout processus qui se déconnecte est vu déconnecté par tous les processus connectés. Formellement :
$\forall DC, \forall H_{DC} \in \mathcal{DD}(DC), \exists t \in \mathcal{T}, \forall p \in disconnected(DC), \forall q \in connected(DC), \forall t' \geq t :$
$p \in H_{DC}(q,\ t')$

Complétude de déconnexion faible : il existe un instant après lequel tout processus qui se déconnecte est vu déconnecté par au moins un processus connecté. Formellement :
$\forall DC,\ \forall H_{DC} \in \mathcal{DD}(DC),\ \exists t \in \mathcal{T}, \forall p \in disconnect(DC),\ \exists q \in connected(DC),\ \forall t' \geq t :$
$p \in H_{DC}(q,\ t')$

Précision de déconnexion forte : aucun processus n'est vu déconnecté avant qu'il ne soit déconnecté. Formellement :
$\forall DC,\ \forall H_{DC} \in \mathcal{DD}(DC),\ \forall t \in \mathcal{T},\ \forall p,\ q \in \Pi - DC(t) :$
$p \notin H_{DC}(q,\ t)$

La complétude faible est assurée dès qu'un processus q reçoit un message de déconnexion d'un processus p. Le passage de la complétude faible à la complétude forte est obtenu par un algorithme de réduction $T_{\mathcal{DD} \longrightarrow \mathcal{DD}'}$ transformant un détecteur de déconnexion $\mathcal{DD}$ en un autre détecteur de déconnexion $\mathcal{DD}'$ à la manière de l'algorithme de réduction pour les détecteurs de défaillance non fiables présenté en figure 3 de [6].

Contrairement aux propriétés du détecteur de défaillance, les propriétés de complétude et de précision du détecteur de déconnexion sont plus facilement satisfaisables. Grâce au détecteur de connectivité, à l'inverse d'une défaillance qui est subite, il existe un certain laps de temps entre la prévision d'une déconnexion et son effet. Un détecteur de déconnexion peut donc utiliser ce laps de temps pour prévenir les autres de sa déconnexion. Pour une déconnexion volontaire, il est clair que le détecteur de déconnexion peut « prendre le temps » de diffuser le message informant de sa déconnexion imminente. Pour une déconnexion involontaire, la période d'échantillonnage de la détection de connectivité doit être strictement inférieure au minimum de la durée nécessaire pour « traverser » le mode partiellement connecté. En outre, cette période d'échantillonnage doit permettre la

diffusion d'un message. Bien sûr, cela n'empêche pas qu'une déconnexion très soudaine soit assimilée à une défaillance : d'ailleurs, en un certain sens, c'est une défaillance de la détection de déconnexion.

5.2 Algorithme

L'algorithme 3 est composé de trois tâches parallèles. La première tâche s'exécute suite à la notification d'un changement de mode émise par le détecteur de connectivité. *mode* = 'd' indique le mode déconnecté ; un message de déconnexion est diffusé à tous les processus connectés. *mode* ='c' indique le mode connecté ; un message de reconnexion est diffusé à tous les processus de Π. La deuxième (*resp.* troisième) tâche est en écoute des messages de déconnexion (*resp.* reconnexion) envoyés par des processus qui transmettent leurs avis de déconnexion (*resp.* reconnexion). Le processus ajoute l'émetteur à (*resp.* enlève l'émetteur de) son ensemble des processus vus déconnectés.

Algorithme 3: $\mathcal{DD}$: détection de déconnexion

```
1  for every process p :
2   initialisation :
3    disc_p ← ∅                                   {set of processes seen disconnected}
4    for all q ∈ Π
5     N_p[q] ← 0                                  {vector of received disc./rec. numbers}
6   cobegin :
7    || task 1 : upon change mode notification
8     if getMode() = 'd'
9      for all q ∈ Π \ disc_p
10      send(DISCONNECT, N_p[p]) to q
11     N_p[p] ← N_p[p] + 1
12     disc_p ← ∅
13    else if getMode() = 'c'
14     for all q ∈ Π
15      send(RECONNECT, N_p[p]) to q
16     N_p[p] ← N_p[p] + 1
17   || task 2 : upon receive(DISCONNECT, n_q) from q
18    if q ∉ disc_p ∧ N_p[q] < n_q then disc_p ← disc_p ∪ {q}
19    N_p[q] ← max(N_p[q] + 1, n_q)
20   || task 3 : upon receive(RECONNECT, n_q) from q
21    if q ∈ disc_p ∧ N_p[q] < n_q then disc_p ← disc_p \ {q}
22    N_p[q] ← max(N_p[q] + 1, n_q)
23  coend
```

Pour tolérer les messages qui se doublent, les déconnexions et les reconnexions sont numérotées et appariées. Enfin, lorsqu'un processus se reconnecte, il ne trouve pas la même configuration des processus : d'autres processus peuvent s'être déconnectés ou reconnectés. Pour cela, $disc_p$ est vidé dès la déconnexion et l'algorithme de réduction de la complétude faible à la complétude forte de déconnexion assure la mise à jour de $disc_p$.

6 Couplage et sémantique

Nous suggérons trois approches pour associer et relier les trois types de détecteurs. Dans la première approche (cf. section 6.1), nous mettons le détecteur de défaillance à la disposition de la gestion de déconnexion, plus particulièrement du détecteur de connectivité. Cette approche complète la vue locale des détecteurs de connectivité d'une vue sur la qualité de service des communications bout en bout. La deuxième approche (cf. section 6.3) est l'utilisation des détecteurs de déconnexion et de connectivité dans le contexte de la tolérance aux fautes, particulièrement dans le consensus. Le but de cette approche est d'adapter la sémantique du consensus afin de tenir compte des déconnexions. Enfin, la troisième approche (cf. section 6.2) est la construction d'un détecteur de défaillances et de déconnexion, de manière directe à partir des algorithmes initiaux.

6.1 Détecteur de défaillance pour la détection de connectivité

Le détecteur de connectivité présenté dans la section 4 manipule une connexion logique entre un processus sur le terminal mobile et un processus distant. Les informations fournies ne concernent que les ressources locales. Elles n'offrent donc qu'une visibilité locale au terminal mobile. Dans cette section, nous ne donnons que quelques principes de solutions paraissant intéressantes à développer ultérieurement, l'objectif global étant de montrer la complémentarité des détecteurs.

L'ajout d'un module de détection de défaillance permet de connaître la qualité de service de bout en bout de la connexion logique à travers l'échange de battements de cœur. Le principe est alors de projeter sur un hystérésis soit le nombre de battements de cœur reçus pendant un intervalle de temps par l'algorithme $\mathcal{HB}$, soit la durée de transmission estimée telle que calculée par exemple dans [3].

La fréquence des battements de cœur peut aussi être configurée suivant la disponibilité des ressources locales. Si un processus dispose d'une bonne largeur de bande passante et d'une charge de batterie suffisante, il configure la périodicité de ses battements de cœur en des émissions relativement fréquents. Si ce même processus ne dispose pas d'une bonne connectivité « locale », il émet des battements de cœur plus espacés dans le temps. Ainsi, le détecteur de connectivité configure le détecteur de défaillance suivant son mode de connectivité. La fréquence d'émission de battements de cœur peut être en plus négociées connexion logique par connexion logique.

6.2 Consensus et détecteurs

Le consensus est la faculté d'atteindre un accord entre processus corrects pour prendre une décision [10]. Dans [6], le consensus utilisant un détecteur de défaillance non fiable de la classe $\diamond\mathcal{S}$ suppose que le nombre maximum de processus défaillants est strictement inférieur à $\lceil \frac{\Pi}{2} \rceil$. Un processus correct n'attend pas les acquittements des processus suspectés de défaillances. Sans modification, une déconnexion est considérée comme une défaillance. Notre nouvel algorithme de consensus $\mathcal{CWD}$ (« *Consensus With Disconnection* ») arrête de diffuser aux processus déconnectés. En outre, les propriétés du consensus

subissent une légère modification pour correspondre à un modèle de système réparti où les déconnexions et les reconnexions sont possibles en plus des défaillances. Avant de présenter les modifications et les nouvelles propriétés pour résoudre le consensus, rappelons d'abord ses propriétés d'origine :

Terminaison : il existe un instant après lequel tout processus correct décide une valeur.
Validité : si un processus correct décide une valeur v alors v a été proposée par un processus.
Intégrité : chaque processus correct décide au plus une fois.
Accord : deux processus corrects ne décident pas deux valeurs différentes.

Sans modification, la propriété de terminaison n'est pas vérifiée. En effet, un processus qui se déconnecte avant la fin du consensus n'arrive jamais à atteindre une décision, restant en attente de recevoir la décision. Aussi, une condition est ajoutée à l'attente de réception du message de décision : l'attente ne se fait que si le processus est connecté pendant toute la durée du consensus. Si un processus est déconnecté, il n'attend plus le message de décision et termine sans décider. La propriété de terminaison modifiée ne concerne que les processus corrects connectés. En outre, puisqu'une décision ne concerne maintenant que les processus connectés, la propriété de l'accord est également modifiée.

Terminaison connectée : il existe un instant après lequel tout processus connecté et correct décide une valeur.
Accord connecté : deux processus connectés et corrects ne décident pas deux valeurs différentes.

Le consensus ne peut être résolu que s'il est supposé que le nombre maximum de processus défaillants ou déconnectés est strictement inférieur à $\lceil \frac{\Pi}{2} \rceil$. L'algorithme 4, adapté de la figure 6 de [6], se compose de quatre tâches parallèles dont les deux premières sont identiques à celles de l'algorithme d'origine, excepté les instructions de contrôle du mode de connectivité et de l'ensemble des processus vus déconnectés dans les lignes 10, 18, 23, 24, 28 et 30 avant chaque émission, ainsi que dans les lignes 13, 20 et 27 dans les « *wait* ». La troisième et la dernière tâches permettent d'exclure du consensus courant les processus qui se déconnectent.

6.3 Détection de défaillance et de déconnexion

Bien que l'algorithme du consensus $\mathcal{CWD}$ supprime les émissions inutiles aux processus déconnectés, d'autres existent encore dans le détecteur de défaillance puisque le détecteur de défaillance ne peut pas savoir si un processus est déconnecté. Un détecteur de défaillance intégrant la détection de déconnexion optimise le nombre de messages envoyés en éliminant les processus déconnectés de la liste des diffusions. Nous nommons $\mathcal{HBD}$ (« *HeartBeat with Disconnection* ») l'algorithme de « *battements de cœur* » gérant en plus des défaillances, les déconnexions des processus. Comme pour l'algorithme de détection de déconnexion $\mathcal{DD}$, $\mathcal{HBD}$ doit être associé à un algorithme de réduction permettant de passer de la complétude faible de déconnexion à la complétude forte de déconnexion.

L'algorithme 5 est constitué de cinq tâches pour gérer tous les types de messages et

Algorithme 4: $\mathcal{CWD}$: consensus avec détections de déconnexion et de connectivité

1 **for** every process p :
2 **cobegin** :
3 $\|$ **task 1** : to execute propose(v_p)
4 $estimate_p \leftarrow v_p$; $state_p \leftarrow undecided$; $r_p \leftarrow 0$; $ts_p \leftarrow 0$; $connected_p \leftarrow getMode() =$'d'$\vee getVoluntary()$; $disc_p \leftarrow getDisc_p()$
5 **while** $connected_p \wedge state_p = undecided$
6 **repeat**
7 $r_p \leftarrow r_p + 1$; $c_p \leftarrow (r_p \bmod n) + 1$
8 **until** $c_p \notin disc_p$
9 **phase 1** :
10 **if** $connected_p$ **then** send(p, r_p, $estimate_p$, ts_p) to c_p
11 **phase 2** :
12 **if** $p = c_p \wedge |disc_p| < \lceil \frac{(n+1)}{2} \rceil$
13 **wait until** [(**for** $\lceil \frac{(n+1)}{2} \rceil$ processes q : received(q, r_p, $estimate_q$, ts_q)) $\vee$ $\neg connected_p$]
14 **if** [for $\lceil \frac{(n+1)}{2} \rceil$ processes q : received(q, r_p, $estimate_q$, ts_q)]
15 $msgs[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$
16 $t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
17 $estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
18 **if** $connected_p$ **then** send(p, r_p, $estimate_p$) to $\Pi \setminus disc_p$
19 **phase 3** :
20 **wait until** [received(c_p, r_p, $estimate_{c_p}$)$\vee c_p \in disc_p \vee c_p$ *suspected*$\vee \neg connected_p$]
21 **if** received(c_p, r_p, $estimate_{c_p}$) from c_p
22 $estimate_p \leftarrow estimate_{c_p}$; $ts_p \leftarrow r_p$
23 **if** $connected_p$ **then** send(p, r_p, ack)
24 **else if** $c_p \notin disc_p \wedge connected_p$ **then** send(p, r_p, $nack$)to c_p
25 **phase 4** :
26 **if** $p = c_p$
27 **wait until** [(**for** $\lceil \frac{(n+1)}{2} \rceil$ processes q : received(q, r_p, $ack/nack$)) $\vee \neg connected_p$]
28 **if** [**for** $\lceil \frac{(n+1)}{2} \rceil$ processes q : received $(q, r_p, ack) \wedge connected_p$ **then** Rbroadcast(q, r_p, $estimate_p$, $decide$)
29 $\|$ **task 2** : **upon** Rdeliver(q, r_p, $estimate_q$, $decide$)
30 **if** $state_p = undecided \wedge connected_p$ **then** decide($estimate_q$) ; $state_p \leftarrow decided$
31 $\|$ **task 3** : **upon** change mode notification **or upon** voluntary disconnection
32 **if** $getMode()$ = 'd' $\vee\, getVoluntary()$ **then** $connected_p \leftarrow false$
33 $\|$ **task 4** : **upon** disconnection notification
34 $disc_p \leftarrow disc_p \cup getDisc_p()$
35 **coend**

Algorithme 5: $\mathcal{HBD}$: détection de défaillance et de déconnexion

```
1  for every process p :
2   initialisation :
3    for all q ∈ neighbour(p) do D_p[q] ← 0
4    disc_p ← ∅
5    for all q ∈ Π do N_p[q] ← 0
6   cobegin :
7    ‖ task 1 : upon change mode notification
8     if getMode() = 'd'
9      for all q ∈ Π \ disc_p do send(DISCCONNECT, N_p[p]) to q
10     N_p[p] ← N_p[p] + 1 ; disc_p ← ∅
11    else if getMode() = 'c'
12     for all q ∈ Π do send(RECONNECT, N_p[p]) to q
13     N_p[p] ← N_p[p] + 1
14   ‖ task 2 : repeat periodically
15    if getMode() ≠ 'd' ∧ ¬getVoluntary() then for all q ∈ Π \ disc_p do
      send(HEARTBEAT) to q
16   ‖ task 3 : upon receive(HEARTBEAT) from q
17    D_p[q] ← D_p[q] + 1
18   ‖ task 4 : upon receive(DISCONNECT, n_q) from q
19    if q ∉ disc_p ∧ N_p[q] < n_q then disc_p ← disc_p ∨ {q}
20    N_p[q] ← max(N_p[q] + 1, n_q)
21   ‖ task 5 : upon receive(RECONNECT, n_q) from q
22    if q ∈ disc_p ∧ N_p[q] < n_q then disc_p ← disc_p \ {q}
23    N_p[q] ← max(N_p[q] + 1, n_q)
24  coend
```

de notifications. La première tâche est déclenchée par la notification d'un changement de mode de connectivité. Dans la deuxième tâche, périodiquement, le processus émet un battement de cœur à tous les processus connectés. L'émission est subordonné à la vérification de la condition $mode \neq$ 'd' $\wedge \neg voluntary$, indiquant que la connectivité permet les communications et qu'aucune déconnexion volontaire n'est en cours. La troisième tâche gère la réception des messages de battements de cœur. Enfin, la quatrième (*resp.* cinquième) tâche gère la réception des messages DISCONNECT (*resp.* RECONNECT).

7 Travaux connexes, conclusion et perspectives

Le concept de détecteur de défaillance non fiable a suscité une littérature abondante. D'une part, certains travaux d'aspect théorique étudient les classes de détecteurs de défaillance et leur possibilité d'implantation dans des environnements complètement asynchrones ou partiellement synchrones, et déterminent les classes les plus faibles pour résoudre certains paradigmes. D'autre part, d'autres travaux d'aspect plus pratique portent sur la qualité de service des détecteurs de défaillance et certaines architectures ou protocoles. Nous ne citons ici que quelques travaux de la seconde catégorie.

Dans ce papier, nous définissons par comparaison avec ces travaux la notion de connectivité (locale) et les deux détecteurs afférents, le premier, local, appelé détecteur de connectivité, et l'autre, réparti, appelé détecteur de déconnexion. Ensuite, ces deux nouveaux détecteurs sont couplés avec le détecteur de défaillance, et à titre d'exemple de paradigme pour la tolérance aux fautes, avec le consensus. Le résultat est une première tentative d'intégration de la tolérance aux fautes dans la mobilité des terminaux, plus particulièrement la gestion des déconnexions, ou *vice versa*.

Dans la littérature, [2] mettent en œuvre un service de détection de défaillance avec une gestion de groupes et de consensus sur une plate-forme CORBA. Ces architectures sont intéressantes pour ajouter la détection de défaillance à la plateforme Domint [7] qui possède déjà la détection de connectivité. Dans le cadre du protocole `TCP`, [3, 4] montrent comment implanter et adapter dynamiquement le détecteur de défaillance selon la vitesse de détection, la durée entre deux erreurs consécutives et la durée de correction des erreurs. Le détecteur de défaillance est structuré en deux couches : la première fournit une estimation des temps d'arrivée des messages pour optimiser le temps de détection et la seconde adapte le service de détection fourni par la première aux besoins de l'application. Cette structuration est à comparer et à agencer avec la hiérarchie de détecteur de connectivité. [13] modélise et évalue le fonctionnement des détecteurs de défaillances en environnement mobile. Les auteurs insistent sur l'asymétrie des communications sans fil en terme de performance : un PDA en réception est un goulot d'étranglement. [1] montre comment résoudre le consensus dans un réseau partitionnable avec $\diamond\mathcal{S}$ et un détecteur de défaillance $\mathcal{HB}$. La définition de la classe $\diamond\mathcal{S}$ est modifiée en $\diamond\mathcal{S}_{LP}$ pour une résolution du consensus dans la partition primaire contenant la majorité des processus corrects. C'est une autre raison du choix d'une version simple de $\mathcal{HB}$. De nouvelles perspectives s'ouvrent alors avec la comparaison entre ces trois événements indésirables : défaillance, partitionnement et déconnexion. Enfin, d'autres modèles de défaillances existent, plus spécialement, le modèle *crash*/reprise qui n'a pas encore été beaucoup étudié [15] et qui

montre que le sujet est encore très ouvert.

Références

[1] M. Aguilera, W. Chen, and S. Toueg. Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoretical Computer Science, special issue on distributed algorithms*, July 1999.

[2] Z. F. Baldoni, R. Designing a Service of Failure Detection in Asynchronous Distributed Systems. *Proc. 4th International Symposium on Object-Oriented Realt-Time Distributed Computing*, 2001.

[3] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proc. International Conference on Dependable Systems and Networks*, Washington DC, USA, June 2002.

[4] M. Bertier, O. Marin, and P. Sens. Performance analysis of a Hierarchical Failure Detector. In *Proc. International Conference on Dependable Systems and Networks*, San Francisco, USA, June 2003.

[5] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(2), Feb. 1996.

[6] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2) :225–267, Mar. 1996.

[7] D. Conan, S. Chabridon, O. Villin, and G. Bernard. Domint : une plate-forme pour la faible connectivité et la déconnexion d'objets CORBA en environnement mobile. Rapport de recherche, Institut National des Télécommunications, Évry, France, May 2003.

[8] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3), July 1986.

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

[10] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical Report TR 94-1425, Department of Computer Science, Cornell University, Ithaca, New-York (USA), May 1994.

[11] R. Harbus. Dynamic process migration : To migrate or not to migrate. Technical report csri-42, University of Toronto, Toronto, Canada, July 1986.

[12] N. Kouici, D. Conan, and G. Bernard. Disconnection Metadata for Distributed Applications in Mobile Environments. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada (USA), June 2003.

[13] C. Marchand and J.-M. Vincent. Détecteurs de défaillances et qualité de service dans un réseau ad-hoc hétérogène. In *Proc. 3è Conférence Française sur les Systèmes d'Exploitation*, pages 525–536, La Colle sur Loup (France), Oct. 2003.

[14] V. Portigliatti. *Contribution à l'allocation dynamique de ressources pour les composants expressifs dans les systèmes répartis.* PhD thesis, Universisté de Franche-Comté, France, Dec. 2003.

[15] L. Rodrigues and M. Raynal. Atomic Broadcast in Asynchronous Crash Recovery Distributed Systems and Its Use in Quorum Based Replication. *IEEE Transactions on Knowledge and Data Engineering*, 15(5) :1206–1217, Sept. 2003.

A Annexe

Nous rassemblons dans les sections qui suivent les preuves des algorithmes présentés tout au long de ce papier : détecteur de connectivité $\mathcal{CD}$ dans la section A.1, détecteur de déconnexion $\mathcal{DD}$ dans la section A.2, détecteur de défaillance et de déconnexions $\mathcal{HBD}$ dans la section A.4 et consensus avec détection de déconnexion et de connectivité $\mathcal{CWD}$ dans la section A.3.

A.1 Détecteur de connectivité $\mathcal{CD}$

La validité de l'algorithme du détecteur de connectivité $\mathcal{CD}$ repose sur la vérification du maintien de la propriété de prévention de l'effet « ping-pong ». Dans toutes les preuves de cette section, par hypothèse de l'algorithme $\mathcal{CD}$, chaque oscillation est confinée autour d'un seuil sans inclure les autres seuils.

Lemme 1 *L'algorithme de détection de connectivité $\mathcal{CD}$ n'est pas sujet à l'effet « ping-pong » autour du seuil* `lowDown`.

Preuve. Trois transitions d'états sont possibles autour de `lowDown`. La première transition est `A` $\rightarrow$ `B` (ligne 16) et ne provoque pas de changement de mode. La deuxième `B` $\rightarrow$ `A` (ligne 19) fixe le mode à 'd'. La troisième transition `F` $\rightarrow$ `A` (ligne 29) fixe aussi le mode à 'd'.

Soit $r =$ `lowDown`, la condition $r_i \geq$ `lowDown` $> r_{i+1} \wedge (state =$ 'F' $\vee\ state =$ 'B') provoque un changement d'état et de mode : le mode devient 'd'. $r_{i+2} \geq$ `lowDown` $> r_{i+1}$ active la transition `A` $\rightarrow$ `B` sans changer le mode : le mode reste à 'd'. $r_{i+3} <$ `lowdown` $\leq r_{i+2}$ active la transition `B` $\rightarrow$ `A` en gardant le mode à 'd'. Il est évident de montrer que, par récurrence, seules les deux dernières transitions sont actives pendant l'oscillation, le mode restant toujours à 'd'.

Lemme 2 *L'algorithme de détection de connectivité $\mathcal{CD}$ n'est pas sujet à l'effet « ping-pong » autour du seuil* `highUp`.

Preuve. Trois transitions d'états sont possibles autour de `highUp`. La première transition est `C` $\rightarrow$ `D` (ligne 21) et fixe le mode à 'c'. La deuxième `D` $\rightarrow$ `E` (ligne 24) ne change pas le mode. La troisième transition `E` $\rightarrow$ `D` (ligne 27) fixe aussi le mode à 'c'.

Soit $r =$ `highUp`, la condition $r_i \leq$ `highUp` $< r_{i+1} \wedge (state =$ 'F' $\vee\ state =$ 'C') provoque un changement d'état et de mode : le mode devient 'c'. $r_{i+2} \leq$ `highUp` $< r_{i+1}$ active la transition `D` $\rightarrow$ `E` sans changer le mode : le mode reste à 'c'. $r_{i+3} >$ `highUp` $\geq r_{i+2}$ active la transition `E` $\rightarrow$ `D` en gardant le mode à 'c'. Il est évident de montrer que, par récurrence, seules les deux dernières transitions sont actives pendant l'oscillation, le mode restant toujours à 'c'.

Lemme 3 *L'algorithme de détection de connectivité $\mathcal{CD}$ n'est pas sujet à l'effet « ping-pong » autour du seuil* `lowUp`.

PREUVE. Deux transitions d'états sont possibles autour de `lowUp`. La première transition est `B` $\rightarrow$ `C` (ligne 18) et fixe le mode à 'p'. La seconde `C` $\rightarrow$ `B` (ligne 22) ne change pas le mode.

Soit $r =$ `lowUp`, la condition $r_i \leq$ `lowUp` $< r_{i+1}$ provoque un changement d'état et de mode : le mode devient 'p'. $r_{i+2} \leq$ `lowUp` $< r_{i+1}$ active la transition `C` $\rightarrow$ `B` sans changer le mode : le mode reste à 'p'. $r_{i+3} >$ `lowUp` $\geq r_{i+2}$ active à nouveau la transition `B` $\rightarrow$ `C` en gardant toujours le mode à 'p'. Il est évident de montrer que, par récurrence, seules ces deux transitions sont actives pendant l'oscillation, le mode restant toujours à 'p'.

Lemme 4 *L'algorithme de détection de connectivité $\mathcal{CD}$ n'est pas sujet à l'effet « ping-pong » autour du seuil* `highDown`*.*

PREUVE. Deux transitions d'états sont possibles autour de `higDown`. La première transition est `E` $\rightarrow$ `F` (ligne 26) et fixe le mode à 'p'. La deuxième `F` $\rightarrow$ `E` (ligne 30) ne change pas le mode.

Soit $r =$ `highDown`, la condition $r_i \geq$ `highDown` $> r_{i+1}$ provoque un changement d'état et de mode : le mode devient 'p'. $r_{i+2} \geq$ `highDown` $> r_{i+1}$ active la transition `F` $\rightarrow$ `E` sans changer le mode : le mode reste à 'p'. Il est évident de montrer que, par récurrence, seules ces deux transitions sont actives pendant l'oscillation, le mode restant toujours à 'p'.

Théorème 1 *L'algorithme de détection de connectivité $\mathcal{CD}$ n'est pas sujet à l'effet « ping-pong ».*

PREUVE. Des lemmes 1, 2, 3 et 4.

A.2 Détecteur de déconnexion $\mathcal{DD}$

La validité de l'algorithme du détecteur de déconnexion $\mathcal{DD}$ repose sur la vérification du maintien des propriétés de complétude faible de déconnexion et de précision forte de déconnexion.

Lemme 5 (Complétude faible de déconnexion) *Il existe un instant après lequel tout processus qui se déconnecte est vu déconnecté par au moins un processus connecté.*

PREUVE. Supposons que p veut se déconnecter (*resp.* se reconnecter). Il émet un message de type DISCONNECT (*resp.* RECONNECT) à tous les processus qu'il voit connectés à l'instant t (*resp.* à tous les processus de Π).

Montrons que les propriétés de l'hystérésis garantissent que toutes les déconnexions sont vues et que l'utilisation d'un compteur de déconnexions permet que les messages de déconnexion et reconnexion se doublent dans les canaux de communications.

Pour la déconnexion, trois cas sont possibles. Dans le premier cas, p réussit à envoyer son message de déconnexion à tous les processus qu'il voit connectés. D'après le modèle de déconnexion (cf. section 2), au moins un processus n'est pas sujet à des déconnexions.

Ce processus n'est jamais vu déconnecté par p. D'où, au moins un processus reçoit le message de déconnexion de p. Ces derniers l'insèrent alors dans $disc_p$. Dans le second cas, p réussit à envoyer au moins un message de déconnexion à un processus qu'il voit connecté q. Si q est un processus qui n'est pas sujet à déconnexion, au moins un processus reçoit le message de déconnexion de p. Si q est sujet à déconnexion, nous sommes dans le dernier et troisième cas qui suit. p est déconnecté avant de pouvoir envoyer un message de déconnexion à destination d'un processus non sujet à déconnexion. Puisqu'il est supposé que les processus de l'application répartie sont terminés alors que la connectivité est bonne (cf. section 2), le message de déconnexion sera reçu par tous les processus connectés lors de la reconnexion. L'estampille ajoutée dans les messages de déconnexion et de reconnexion permet d'apparier les déconnexions et les reconnexions de telle façon qu'un message de déconnexion (*resp.* reconnexion) ancien non encore reçu ne provoque pas de déconnexion (*resp.* reconnexion).

Lemme 6 (Précision forte de déconnexion) *Aucun processus n'est vu déconnecté avant qu'il ne soit déconnecté.*

Preuve. Il est clair d'après l'algorithme 3 qu'un processus q n'est vu déconnecté par p que lorsqu'il reçoit un message de déconnexion de q. La ligne 18 est la seule instruction qui insère le processus q dans $disc_p$. Cette instruction n'est exécutée qu'à la réception d'un message de déconnexion de type DISCONNECT provenant du processus q et plus récent que la dernière déconnexion vue (ligne 10). L'autre possibilité d'insertion de q dans $disc_p$ est via l'algorithme de réduction $T_{\mathcal{DD}\longrightarrow\mathcal{DD}'}$.

Théorème 2 *L'algorithme de détection de déconnexion répartie $\mathcal{DD}$ satisfait la précision forte de déconnexion et la complétude faible de déconnexion.*

Preuve. Le théorème 2 se déduit directement des lemmes 5 et 6.

A.3 Consensus avec détection de déconnexion et de connectivité $\mathcal{CWD}$

La validité de l'algorithme du consensus $\mathcal{CWD}$ repose sur la vérification du maintien des propriétés de terminaison connectée, validité uniforme, accord connecté et intégrité. Dans le cas général, la preuve de l'algorithme est identique à celle présentée dans [6]. Elle reste inchangée pour les processus qui sont connectés en permanence pendant toute la durée d'une instance d'un consensus. Cependant, la nouveauté dans cette configuration provient du fait qu'un processus a la possibilité de se déconnecter volontairement ou involontairement. Un processus qui est déconnecté ne peut plus émettre de message, il n'est donc plus en mesure de participer au consensus. Avant de prouver les propriétés du consensus, nous commençons par énoncer quelque propriétés concernant les processus déconnectés lors du consensus.

Lemme 7 *Un processus se considère déconnecté si $\neg$connected.*

PREUVE. Par les propriétés du détecteur de déconnexion, la détection d'une déconnexion est notifiée au consensus dans la tâche 3. La variable *connected* est aussitôt mise à $false$ dans la ligne 32, indiquant à p qu'il est désormais déconnecté. Une fois *connected* à $false$, elle le reste pendant tout le consensus.

Lemme 8 *Un processus déconnecté ne participe plus au consensus.*

PREUVE. Selon le lemme 7, la variable *connected* de tout processus déconnecté est à $false$. Dans l'algorithme, tous les messages qu'un processus p émet aux autres processus ne sont émis que si p est connecté (*connected*). Ainsi, lorsque $\neg$*connect*, p n'émet aucun message aux autres processus, il arrête de participer au consensus.

Lemme 9 *Un processus déconnecté termine sans décider.*

PREUVE. Selon le lemme 8, dès qu'un processus se voit déconnecté, il ne participe plus au consensus. En outre, aucun processus déconnecté ne se bloque dans les « *wait* » des phases 2, 3 et 4. D'après la ligne 5, aucun processus déconnecté n'entame un nouveau tour, il termine l'exécution de la tâche 1, et par conséquence, le consensus sans décider.

Lemme 10 *Un processus qui est vu déconnecté demeure vu déconnecté par les autres processus pendant tout le consensus.*

PREUVE. L'ensemble $disc_p$ des processus vus déconnectés par p n'est modifié que dans la ligne 34. Un processus qui est inséré dans l'ensemble $disc_p$ n'est pas enlevé pendant l'instance de ce consensus. Il est donc clair que si un processus q est vu déconnecté par p, il le reste pendant tous le consensus.

Lemme 11 (Accord connecté) *Deux processus connectés et corrects ne décident pas deux valeurs différentes.*

PREUVE. Les processus connectés se comportent exactement de la même manière que dans l'algorithme d'origine [6]. Par conséquent, pour les processus connectés, le consensus satisfait la propriété de l'accord prouvé dans [6]. Selon les lemmes 8 et 9, puisqu'un processus déconnecté ne participe plus au consensus et qu'un processus déconnecté termine sans décider, deux décisions différentes ne peuvent exister.

Lemme 12 (Terminaison connectée) *Il existe un instant après lequel tout processus connecté et correct décide une valeur.*

PREUVE. Deux cas sont possibles :

1. Un processus correct et connecté décide. Il doit avoir livré d'une manière fiable (ligne 29) un message de type $(-, -, -, decide)$ diffusé de manière fiable (ligne 28). Par la propriété d'accord de la diffusion fiable, tous les processus connectés et corrects délivrent d'une manière fiable ce message et décident.

2. Aucun processus correct et connecté ne décide. Nous affirmons qu'aucun processus correct et connecté ne se bloque dans l'un des « *wait* ». La preuve est par contradiction. Soit r le plus petit numéro de tour dans lequel un processus connecté et correct se bloque indéfiniment dans l'un des « *wait* ». Ainsi, tous les processus connectés et corrects terminent la phase 1 du premier tour r : ils émettent tous un message de type $(-, r, estimate, -)$ au coordinateur $c = (r \bmod n) + 1$. Puisque la majorité des processus sont corrects et connectés, il y a au moins $\lceil \frac{n+1}{2} \rceil$ de ces messages envoyés à c. Trois cas sont à considérer :

   (a) c reçoit ces messages et répond en envoyant $(c, r, estimate_c)$. Il ne se bloque donc pas indéfiniment dans les blocs « *wait* » de la phase 2.

   (b) c subit un *crash*.

   (c) c se déconnecte.

   Dans le cas (a), tous les processus corrects et connectés reçoivent un message de type $(c, r, estimate_c)$. Dans le cas (b), puisque $\mathcal{HB}$ avec l'algorithme $T_{\mathcal{D} \longrightarrow \mathcal{D}'}$ présenté en figure 3 de [6] satisfont la complétude forte, pour tout processus correct p, il existe un instant après lequel c est suspecté d'une manière permanente par p. D'où, il n'existe pas de processus correct et connecté qui se bloque dans le second « *wait* » de la phase 3. Dans le cas (c), puisque $\mathcal{DD}$ satisfait la complétude forte de déconnexion, pour tout processus correct et connecté p, il existe un instant après lequel c est vu déconnecté d'une manière permanente par p, c.-à-d., $c \in disc_p$. De même, il n'existe pas de processus correct et connecté qui se bloque dans le second « *wait* » de la phase 3. De cette façon, chaque processus correct et connecté émet à c, durant la troisième phase, un message de type$(-, r, ack)$ s'il est dans le cas (a), un message de type $(-, r, nack)$ s'il est dans le cas (b), ou enfin n'émet aucun message s'il est dans le cas (c). Dans ce dernier cas où c est déconnecté, il n'est pas bloqué dans le «*wait*» de la phase 4 puisque $\neg connected$ (ligne 27). Dans le cas où c est connecté, puisque il y a au moins $\lceil \frac{n+1}{2} \rceil$ processus corrects et connectés, alors c ne peut pas être bloqué dans le bloc « *wait* » de la phase 4. Ceci démontre que tous les processus connectés et corrects complètent le tour r — contradiction qui complète la preuve de l'affirmation.

   Puisque $\mathcal{HB}$ satisfait la précision faible finale, il existe un processus connecté et correct q, et un instant t tel qu'aucun processus connecté et correct ne suspecte q après l'instant t. Soit $t' \geq t$ un instant tel que tous les processus défaillants subissent un *crash*. Notez qu'après l'instant t' aucun processus ne suspecte q. Par conséquent, il existe obligatoirement un tour r tel que :

   (a) Tous les processus connectés et corrects atteignent le tour r après l'instant t' (quand aucun processus ne suspecte q),

   (b) q est le coordinateur du tour r (c.-à-d., $q = (r \bmod n) + 1$).

   Durant la phase 1 du tour r, tous les processus connectés et corrects émettent leur estimation à q. Durant la phase 2, q reçoit $\lceil \frac{n+1}{2} \rceil$ estimations et émet $(q, r, estimate_q)$ à tous les processus connectés. Durant la phase 3, puisque q n'est suspecté par aucun autre processus connecté et correct après l'instant t, chaque processus connecté et correct attend l'estimation de q qu'il reçoit. Il répond alors par un message d'acquittement $(-, r, ack)$ à q. En outre, aucun processus ne répond avec un message

d'acquittement négatif $(-, r, nack)$ à q (ce qui n'arriverait que si un processus suspectait q). Par conséquent, durant la phase 4, q reçoit $\lceil \frac{n+1}{2} \rceil$ messages de type $(-, r, ack)$ et q diffuse d'une manière fiable $(q, r, estimate_q, decide)$. Par les propriétés de la validité et de l'accord de la diffusion fiable, tous les processus connectés et corrects délivrent d'une manière fiable le message de q et décident — contradiction. Il en résulte que le cas 2 est impossible, ce qui conclue la preuve du lemme.

Théorème 3 *L'algorithme $\mathcal{CWD}$ résout le consensus utilisant un détecteur de défaillance $\mathcal{HB}$, un détecteur de déconnexion $\mathcal{DD}$ et un détecteur de connectivité $\mathcal{CD}$ dans un système asynchrone avec $f(F, DC) < \lceil \frac{n}{2} \rceil$.*

Preuve. Les lemmes 11 et 12 montrent que $\mathcal{CWD}$ satisfait respectivement les propriétés d'accord connecté et de terminaison connectée. D'après la ligne 30 de l'algorithme, il est clair qu'aucun processus ne décide plus d'une fois. L'intégrité est donc vérifiée. En outre, il est clair aussi que toutes les estimations que le coordinateur reçoit durant la phase 2 sont des valeurs proposées. Ainsi, la valeur que le coordinateur choisit est une valeur proposée par au moins un processus. En conséquence, la validité est également satisfaite.

A.4 Détecteur de défaillance et de déconnexion $\mathcal{HBD}$

Nous allons prouver que $\mathcal{HBD}$ vérifie les propriétés de complétude et de précision de $\mathcal{HB}$ dans le mode connecté et les propriétés de complétude et de précision de déconnexion du détecteur de déconnexion réparti pour les processus corrects.

Pour rappel, $\mathcal{HB}$ satisfait les deux propriétés suivantes [1] :

$\mathcal{HB}-$*Complétude* : dans chaque processus correct, la séquence de battements de cœur de tout processus défaillant est borné. Formellement :
$\forall F,\ \forall H \in \mathcal{D}(F),\ \forall p \in correct(F), \forall q \in crashed(F) \cap neighbour(p), \exists K \in N,\ \forall t \in \mathcal{T}\ :$
$H(p,\ t)[q] \leq K$

$\mathcal{HB}-$*Précision* : dans chaque processus, la séquence de battements de cœur de chaque processus voisin ne diminue jamais. Formellement :
$\forall F,\ \forall H \in \mathcal{D}(F),\ \forall p \in \Pi,\ \forall q \in neighbour(p),\ \forall t \in \mathcal{T} : H(p,\ t)[q] \leq H(p,\ t+1)[q]$

Puisque nous utilisons le modèle de Chandra et Toueg où chaque paire de processus est connecté, les voisins de p sont tous les processus de Π.

Lemme 13 ($\mathcal{HB}$-Complétude) *Dans chaque processus correct, la séquence de battements de cœur de tout processus défaillant est bornée.*

Preuve. La tâche 2 de l'algorithme $\mathcal{HBD}$ est celle qui émet les battements de cœur. Cette tâche s'exécute jusqu'à la défaillance de l'hôte. L'émission du battement de cœur est inhibé lors de déconnexions involontaires ($getMode() =$ 'd') volontaires ($voluntary$), d'où le test à vérifier : $getMode() \neq$ 'd' $\wedge \neg getVoluntary()$. Il est donc évident que si un processus est défaillant ou déconnecté, il n'émet pas de battements de cœur. Tant que le processus est déconnecté, aucun battement de cœur n'est émis. Par conséquent, la valeur de son compteur dans un processus connecté et correct p est bornée pendant les déconnexions.

Lemme 14 *Dans chaque processus p, la séquence de battements de cœur de tout processus ne diminue jamais.*

PREUVE. Il est claire que $\mathcal{D}_p[q]$ ne diminue pas puisqu'il n'est changé que dans la tâche 3 à la ligne 17 qui contient une incrémentation.

Lemme 15 *Dans chaque processus p connecté et correct, la séquence de battements de cœur de tout processus connecté et correct q est non bornée.*

PREUVE. Soit p et q une paire de processus. Si q est correct, sa tâche 2 est exécutée un nombre infini de fois. Donc, si q est connecté, il émet un nombre infini de messages HEARTBEAT au processus connecté p. Par la propriété des communications fiables, si p est connecté et correct, il reçoit un nombre infini de messages HEARTBEAT de q. Lorsqu'un processus p connecté et correct reçoit un message HEARTBEAT de q, il incrémente $\mathcal{D}_p[q]$ à la ligne 17. Ainsi, p incrémente $\mathcal{D}_p[q]$ un nombre infini de fois. De plus, par le lemme 14, $\mathcal{D}_p[q]$ n'est pas décrémenté. Donc, le nombre de battements de cœur de q dans p est non bornée.

Corollaire 1 ($\mathcal{HBD}$-Précision) *Dans chaque processus, la séquence de battements de cœur de tous les processus de* Π *ne diminue pas. Et, dans chaque processus correct et connecté, la séquence de battements de cœur de tous les processus et corrects connectés est non bornée.*

PREUVE. Des lemmes 14 et 15.

Lemme 16 *L'algorithme $\mathcal{HBD}$ réalise la détection de défaillance de $\mathcal{HB}$ pour les processus connectés.*

PREUVE. Du lemme 13 et du corollaire 1.

Lemme 17 *L'algorithme $\mathcal{HBD}$ réalise la détection de déconnexion de $\mathcal{DD}$ pour les processus corrects.*

PREUVE. L'algorithme de détection de déconnexion $\mathcal{DD}$ est constitué d'une première tâche de notification de changement de mode, d'une seconde tâche de réception des messages de déconnexion et d'une troisième tâche de réception des messages de reconnexion. Ces tâches correspondent respectivement aux tâches 1, 4 et 5 de l'algorithme $\mathcal{HBD}$. Aucune des autres tâches n'entraver ces tâches, si ce n'est la défaillance de l'hôte.

Théorème 4 *L'algorithme $\mathcal{HBD}$ réalise la détection de défaillance de $\mathcal{HB}$ pour les processus connectés et la détection de déconnexion de $\mathcal{DD}$ pour les processus corrects.*

PREUVE. Des lemmes 16 et 17.