La tolérance aux fautes dans un système temps-réel à contraintes strictes

Maryline Silly

IRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France

Tél. (1) 39 63 55 11

Rapports de Recherche

N° 512

LA TOLÉRANCE AUX FAUTES DANS UN SYSTÈME TEMPS-RÉEL À CONTRAINTES STRICTES

Maryline SILLY

Mars 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042-RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Publication Interne n° 286

32 pages - Février 1986

LA TOLERANCE AUX FAUTES DANS UN SYSTEME TEMPS-REEL A CONTRAINTES STRICTES

Maryline SILLY

IRISA - Unité Associée au CNRS n° 227

Campus de Beaulieu - 35042 RENNES Cédex

Mots-clés : Temps-Réel, ordonnancement, échéance, tolérance aux fautes, mécanisme à échéance.

Résumé : Une grande partie des applications dites "temps réel à contraintes strictes" impose au système chargé de les conduire le respect de son cahier des charges exprimé en termes de temps de réponse imposés, sous peine d'engendrer un dommage des équipements voire même parfois la perte de vies humaines. Une mauvaise conception du matériel ou une implémentation incorrecte du logiciel peut être à l'origine d'une faute qui se manifeste pour de telles applications par le non respect des contraintes temporelles.

Ce papier décrit le mécanisme à échéance développé pour l'aide à la conception et à l'implémentation des systèmes temps-réel tolérant les fautes. Fondé sur le principe de la redondance, il permet de construire un système temps réel garantissant ses temps de réponse, même en présence de fautes.

Après avoir présenté ce mécanisme, on décrit les stratégies d'ordonnancement sur lesquelles il repose ainsi que les performances obtenues. On aborde ensuite les façons de l'implémenter et de l'utiliser pour le problème de la reconfiguration en cas de défaillance matérielle dans un système distribué.

Key-words : Real-time, scheduling, deadline, fault-tolerance, deadline mechanism

Summary : Many of real time applications are said hard real-time because the system in charge of their monitoring must insure compliance with severe timing constraints defined in the specifications.

Bad design or implementation of the system and its software can cause faults which result in failure to meet real-time constraints.

This paper describes the deadline mechanism proposed as a means to design and implementation of fault tolerant real-time systems. Based upon redundancy, it allows to construct a real-time system with all its guaranteed response times, even in presence of faults.

We present this mechanism and describe scheduling algorithms on which it is based upon. Then, we discuss an implementation in a a high level language and its use in a distributed system with failing components.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

S O M M A I R E

I. Introduction.

La caractéristique essentielle d'un système de contrôle temps-réel est sa sensibilité c'est-à-dire son temps de réponse aux diverses sollicitations émises par le procédé contrôlé. Or, dans les spécifications du cahier des charges d'une application, on peut généralement distinguer

- les sollicitations qui doivent être traitées par le système de contrôle dans un temps de réponse imposé
- de celles qui peuvent être traitées dans un temps de réponse espéré.

Dans le premier cas, la sensibilité du système de contrôle doit satisfaire impérativement les contraintes temporelles de l'environnement externe avec lequel il interragit. Les actions du logiciel de contrôle doivent s'exécuter dans un laps de temps strict à partir de leur activation sous peine de non respect du cahier des charges (en termes de précision ou de temps de réponse par exemple), voire même sous peine de dégradation du procédé (instabilité d'états internes, oscillations inacceptables des actionneurs, évolution en butée des grandeurs de commande, etc ...). De telles spécifications caractérisent les applications dites Temps-Réel à Contraintes Strictes : TRCS.

L'existence de telles contraintes peut engendrer des conséquences immédiates sur la conception même du système de contrôle, en particulier sa technologie, sa structuration (monoprocesseur, multiprocesseur à mémoire commune, multicalculateur), la répartition des fonctions de contrôle dans le système, etc ... et ce, de façon à éviter toute surcharge même temporaire du système conduisant au non respect du temps de réponse d'une ou de plusieurs sollicitations émises par l'environnement. La notion de temps critique caractérise donc une application TRCS.

Au contraire, lorsque le temps de réponse spécifié dans le cahier des charges désigne plutôt un "intervalle d'espérance" dans lequel on souhaite que l'action s'effectue, l'application est alors dite "Temps-Réel à Contraintes Relatives" : TRCR. Ce type de contraintes n'a alors aucune incidence sur le choix de la structure matérielle du système de contrôle puisque sa finalité n'est que de minimiser les retards dûs aux dépassements des temps de réponse espérés, sans que ces retards soient en fait bornés (la limitation quelconque de ces retards nous ramenant implicitement au cas TRCS) ; il peut apparaître même, dans ce cas, difficile de qualifier de "temps réel" ce type d'applications.

Le problème abordé ici est celui de la conception d'un système temps-réel sûr, par la mise en oeuvre de logiciels tolérant les fautes et permettant d'assurer dans toutes circonstances, le respect des contraintes temporelles déduites du cahier des charges d'une application TRCS.

Généralement, la mise en oeuvre de la commande en temps-réel d'un procédé physique passe par les phases préalables suivantes :

- à partir du cahier des charges de l'application, on recense tous les événements (signaux d'interruption) et états (mesures) qui traduisent l'évolution du comportement du procédé.
- on définit ensuite, sous forme d'algorithmes, les fonctions que doit remplir le système de contrôle en réponse à ces événements et états pour assurer le suivi du procédé.
- les programmes de ces fonctions appelés tâches sont alors implantés en mémoire du calculateur.

Au cours de son évolution, toute tâche peut être amenée à se bloquer en attente d'un événement interne ou externe suivant qu'il est émis par une autre tâche ou par l'environnement. De par le caractère aléatoire de ces événements, il est souvent difficile de disposer hors ligne d'une mesure précise de ces durées de blocage, celles-ci variant d'une activation à l'autre de la tâche. Dans ces conditions, il s'avère difficile de donner une borne supérieure sur la durée d'exécution des tâches et d'affirmer que toutes les échéances pourront être respectées, ce qui pour une application TRCS devrait être une certitude avant de la lancer.

Le mécanisme à échéance introduit par Campbell [CAMPBELL 79a] fournit une méthodologie pour construire des systèmes temps-réel sûrs c'est-à-dire respectant toujours leurs contraintes temporelles. ce mécanisme associe deux algorithmes au lieu d'un à chaque tâche. Chaque tâche met ainsi en oeuvre :

- un programme dit primaire supposé fournir un service de bonne qualité mais dans un temps non parfaitement connu.
- un programme dit secondaire, fournissant toujours un résultat acceptable dans un temps de longueur finie et connue.

La tolérance aux fautes est alors obtenue en s'assurant que pour toute tâche, soit son programme primaire soit son programme secondaire termine son exécution avant l'échéance.

II. Fonctionnement d'un système temps-réel.

1. Description.

L'une des caractéristiques principales d'un logiciel de contrôle de procédé industriel réside dans les contraintes temporelles de son exécution. Cette exécution doit en effet réaliser le suivi en "temps réel" du procédé, c'est-à-dire des multiples signaux et états qui reflètent l'évolution de son comportement.

Lorsque ce logiciel de contrôle réside sur un micro-ordinateur, il est alors nécessaire de le doter d'un système d'exploitation spécialisé appelé exécutif temps-réel [SILLY 84]. Son rôle est d'assurer la synchronisation du déroulement des tâches, avec l'évolution du procédé et de veiller au respect des contraintes temporelles définies dans le cahier des charges. Cette dernière fonction est alors réalisée par un logiciel spécifique de l'exécutif : l'ordonnanceur.

Dans un contexte TRCS, la politique d'ordonnancement doit assurer le respect absolu du temps de réponse des tâches dans un délai fixé et ce, à tout moment. Le choix de cette politique [COFFMAN 76] conditionne donc la mise en oeuvre de toute application temps-réel.

2. Modélisation.

La modélisation à la fois des caractéristiques temporelles des actions d'une application ainsi que des performances du système informatique chargé de son exécution, s'avère nécessaire dès lors qu'il apparaît impératif d'assurer dans un délai critique l'exécution de toute action. En effet, en raison du caractère a priori aléatoire des occurrences des événements émis par le procédé, il est souvent impossible d'analyser en simulation ou en phase d'essais sur site si tous ses délais critiques pourront toujours être respectés en exploitation. C'est pourquoi, il est essentiel de disposer d'un modèle susceptible de représenter les cas les plus contraignants de conflits temporels d'une application donnée pour examiner si le système de contrôle choisi (processeur et politique d'ordonnancement) pourra les résoudre, et ce même si la probabilité d'occurrence de ces conflits n'est qu'infime.

Toute tâche temps-réel qui doit s'exécuter en respectant des contraintes de temps est appelée tâche à date critique car si son signal d'initialisation arrive à l'instant r, elle doit impérativement être achevée avant la date d appelée date critique ou échéance [CONWAY 67]. La grandeur R = d - r correspond alors à son temps de réponse imposé appelé aussi délai critique. Il faut bien voir que l'on suppose le monde externe insensible à la date exacte de début et de fin d'exécution d'une tâche pourvu que celle-ci soit traitée entièrement entre l'instant de son initialisation et sa date critique.

Toute tâche à date critique $\mathcal{C}_i$ (fig. 1) est ainsi caractérisée par trois paramètres :

- sa date d'initialisation : r_i
- sa date critique : d_i

- sa durée d'exécution (durée de traitement sans interruption sur un processeur donné) : C_i.

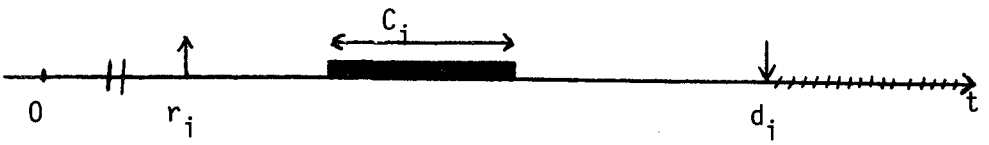

Fig. 1 : Description d'une tâche à date critique

Dans de nombreuses applications, il existe une classe de tâches dont le réveil est conditionné par l'occurrence d'un signal périodique provenant d'une horloge. En effet, dans une application de commande de processus où il est nécessaire d'effectuer un suivi régulier du comportement de l'environnement, l'horloge est utilisée pour lancer périodiquement des tâches chargées de scruter, voire modifier l'état de l'environnement via des voies de liaison qui lui sont spécialisement réservées.

De telles tâches dites périodiques, se décomposent généralement en trois phases :

- l'acquisition de données
- le calcul de la loi de commande
- l'émission de la commande.

Leur période est alors déterminée en fonction de la précision du contrôle exigée par les dynamiques du procédé à surveiller.

Le programme d'une tâche se présente comme une suite d'instructions exécutées séquentiellement dont certaines sont des appels aux procédures de l'exécutif.

On peut distinguer deux types d'appel :

- l'appel bloquant : l'exécution de la procédure appelée conduit à un blocage de la tâche ; celle-ci doit attendre l'événement dont seule, l'occurrence lui permettra de poursuivre son évolution
- l'appel non bloquant : celui-ci ne provoque pas le blocage de la tâche et n'engendre donc aucun retard dans son exécution.

Si on appelle module, une séquence d'instructions ne comportant aucun appel bloquant à l'exécutif, toute tâche peut être vue comme une suite de modules séparés par des apples bloquants à l'exécutif appelés points de synchronisation [SILLY 84]. Par la suite, toute tâche ne présentant aucun point de synchronisation sera appelée tâche-module.

Toute application temps-réel peut ainsi être modélisée par :
- un processeur $\mathcal{P}$ (ou plusieurs) de vitesse connue.
- un ensemble de n tâches $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ appelée configuration où chaque $\mathcal{C}_i$ est caractérisée par :
  - . sa période : T_i
  - . son délai critique : R_i
  - . un vecteur $\mathcal{C}_i = (C_{i1}, \ldots, C_{in_i})$ où l'on désigne par :
    - . C_{ij} : la durée d'exécution sur $\mathcal{P}$ du jème module
    - . n_i : le nombre de modules de $\mathcal{C}_i$.

La durée d'exécution sans préemption de $\mathcal{C}_i$ sur $\mathcal{P}$ est donnée par :

$$C_i = \sum_{j=1}^{n_i} C_{ij}$$

Par la suite, on appellera :
- facteur de charge de $\mathcal{C}$, la quantité $CH = \sum_{i=1}^{n} \frac{C_i}{R_i}$
- facteur d'utilisation de $\mathcal{C}$, la quantité $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$

Dans un tel modèle, toute tâche est assimilée à une tâche périodique. Certes, la majorité des tâches de contrôle doivent être traitées à une cadence spécifique des besoins de l'application et sont donc périodiques par nature. Mais une application peut aussi mettre en oeuvre des tâches dont le réveil s'effectue de façon irrégulière. C'est le cas des tâches d'alarme dont le réveil dû à une perturbation d'un composant du système a lieu inopinément. On convient donc d'associer à de telles tâches, une période fictive aussi appelée pseudo-période désignant l'intervalle de temps minimal pouvant séparer deux occurrences successives de l'événement qui conditionne leur exécution.

3. L'ordonnancement dans un système TRCS.

a. Rappels de concepts fondamentaux.

Un algorithme d'ordonnancement se définit comme une procédure capable de donner une description appelée séquence du travail effectué par le(s) processeur(s) sur les tâches d'une configuration [COFFMAN 76]. Lorsque toutes les tâches réveillées durant un intervalle de temps donné ont leur date critique respectée, la séquence est dite valide.

On dit qu'un algorithme d'ordonnancement est fiable pour une configuration (ou qu'une configuration est acceptée par cet algorithme) s'il produit une séquence valide sur une durée infinie et ce, qu'elles que soient les dates de réveil des tâches. Une configuration pour laquelle il existe au moins un algorithme fiable est dite ordonnançable.

La performance d'un algorithme d'ordonnancement se mesure en termes de pourcentages de configurations ordonnançables qu'il peut ordonnancer fiablement. Un algorithme optimal est donc celui capable d'ordonnancer fiablement toute configuration ordonnançable.

Le développement et l'analyse de politiques d'ordonnancement dans un tel contexte consistent en :

- la recherche de manière heuristique d'algorithmes optimaux
- la détermination de conditions d'acceptation des configurations par ces algorithmes.

Soit $\mathcal{C}$ une configuration. On appelle condition suffisante d'acceptation d'un algorithme P, l'ensemble des relations entre les variables temporelles de $\mathcal{C}$ suffisantes pour garantir l'acceptation de $\mathcal{C}$ par P.

Alors, dans le cas d'une application particulière, il suffit de donner à ces variables, les valeurs des paramètres des tâches de l'application et d'examiner si ces relations sont ou non respectées pour juger de la validité du système de contrôle.

Les algorithmes mis en oeuvre dans les systèmes temps-réel sont des algorithmes en ligne puisqu'à tout instant courant, ils doivent pouvoir générer une séquence sans connaissance des tâches réveillées postérieurement. Parmi ces algorithmes en ligne, certains utilisent le concept de priorité pour spécifier l'urgence d'une tâche. Selon que cette priorité est fixée à l'initialisation ou évolue au cours du temps, l'algorithme est dit conduit par une priorité fixe ou conduit par une priorité dynamique [GONZALEZ 77].

Un algorithme d'ordonnancement se définit aussi par la nature des événements qui conditionnent sa mise en oeuvre. Ainsi, un algorithme est dit préemptif quand l'exécution d'une tâche peut être interrompue au bénéfice d'une autre plus prioritaire. Au contraire, si une tâche activée doit poursuivre son exécution jusqu'à sa fin, et ce sans préemption, l'algorithme est dit non préemptif.

b. Présentation des algorithmes.

La majorité des travaux concernant l'ordonnancement monoprocesseur de tâches périodiques à date critique a été effectuée sous l'hypothèse de tâches-module. Une synthèse de ces travaux est présentée dans [SILLY 84].

On ne décrira ici que deux algorithmes connus pour leur bonne performance et leur simplicité d'implémentation.

1) Un algorithme optimal appelé RU (Relative Urgency) a été développé dans [FINEBERG 67], [LIU 73], [SERLIN 72] et [LABETOULLE 74]. RU est préemptif à priorité dynamique. A tout instant, la tâche la plus prioritaire est celle dont l'échéance est la plus imminente. Il fut montré dans [LIU 73] qu'une configuration $\mathcal{C}$ est acceptée par RU Si son facteur de charge est inférieur ou égale à 1, c'est-à-dire :

$$\sum_{i=1}^{n} \frac{C_i}{R_i} \leqslant 1$$

D'autre part, pour qu'une configuration $\mathcal{C}$ soit ordonnançable sur un processeur $\mathcal{P}$, il est nécessaire que son facteur d'utilisation soit inférieur à la capacité de traitement de $\mathcal{P}$. Il est donc nécessaire que :

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant 1$$

Si l'on suppose que toutes les tâches de $\mathcal{C}$ ont un délai critique égal à leur période, il suffit de calculer le facteur d'utilisation de $\mathcal{C}$ pour tester son acceptation par RU.

2) Si le choix d'une stratégie d'ordonnancement est restreinte à celles conduites par une priorité fixe (pour des commodités d'implémentation par exemple), on optera pour l'algorithme RM (Rate Monotonic) développé dans [SERLIN 72] et démontré optimal parmi ceux à priorité fixe. La politique d'allocation de priorité est telle que les tâches de faible délai critique ont une forte priorité. Etant donné l'absence d'une condition suffisante d'acceptation par RM, la détection d'un éventuel dépassement d'échéance doit passer par l'écriture d'une procédure qui par énumération doit vérifier la validité de la séquence produite par RM sur l'intervalle de longueur égale à la plus grande des périodes dans le cas où toutes les tâches sont réveillées simultanément.

Toutefois, sous l'hypothèse de tâches à délai critique égal à leur période, on dispose d'une condition d'acceptation simple à tester qui est :
$U \leqslant n(2^{1/n} - 1)$ où n désigne le nombre de tâches de la configuration [SERLIN 72].

Malheureusement celle-ci s'avère beaucoup trop restrictive et donc d'un intérêt pratique limité.

L'utilisation de ces deux algorithmes d'ordonnancement est largement répandue en raison de la facilité avec laquelle ils peuvent être implémentés et du faible overhead engendré par leur mise en oeuvre.

L'exécutif temps-réel RTEX développé au Laboratoire d'Automatique de Nantes [DEPLANCHE 85] de même que NESTOR développé à l'IRISA [BELMANS 85] propose à leurs utilisateurs un ordonnanceur sélectionnable offrant entre autres les politiques RU et RM mais aussi RUNP (Relative Urgency Non Preemptive) et RMNP (Rate Monotonic Non Preemptive) [KIM 80a].

c. Cas des tâches non module.

Dans la quasi-totalité, les tâches interragissent par des échanges de messages ou de simples signaux de synchronisation. Ces tâches présentent donc des points de synchronisation qui retardent leur exécution. Lorsqu'on essaie d'appliquer à ces tâches, les politiques précédemment citées, on constate que ces dernières ne présentent plus d'aussi bonnes performances que celles obtenues sous l'hypothèse de tâches-module [CHETTO 85]. Peu de travaux exeptés [SORENSON 74] et [LEINBAUGH 80] ont traité de l'ordonnancement des tâches non-module. De par l'absence de résultats significatifs dans ce domaine, et malgré une performance diminuée de RU et RM, ces derniers sont utilisés même pour des tâches non-module. Il reste cependant à rechercher pour de telles tâches, de nouvelles conditions d'acceptation, les tests simples décrits dans le paragraphe précédent ne permettant plus de garantir l'absence de violations d'échéance.

La recherche dans ce domaine est quasi-inexistante ; la seule idée retenue jusqu'à maintenant est celle qui consiste à déterminer la durée maximale de blocage d'une tâche et à modéliser celle-ci par une tâche-module dont la durée d'exécution aurait été majorée de sa durée de blocage [CHETTO 85]. Les tests habituels appliqués aux tâches-module sont alors valides.

Bien que séduisante, cette méthode s'avère très restrictive à cause du faible pourcentage de configurations de tâches dont elle peut garantir l'acceptation.

d. Cas d'un système multiprocesseur.

Le problème de l'ordonnancement dans un environnement multiprocesseur s'est révélé NP-hard [LENSTRA 80] montrant ainsi l'absence d'existence de tout algorithme optimal. La seule technique permettant d'échapper à cette incalculabilité consiste à considérer un système multiprocesseur sour une forme fictivement décentralisée. Cela suppose qu'à chaque processeur est associé un ordon-

nanceur local qui met en oeuvre un algorithme d'ordonnancement monoprocesseur performant tel que RU. Il n'en résulte cependant pas une optimalité globale du système. Sous l'hypothèse de tâches-module, plusieurs stratégies dites de "Bin Packing" décrites dans [DHALL 78] répartissent les tâches d'une configuration d'une part en assurant que chaque ensemble de tâches assignées à un processeur est acceptée et d'autre part conduisant à l'utilisation d'un nombre quasi-minimal de processeurs.

De même que précèdemment, cette méthode est difficilement applicable à des tâches non-module étant donné la difficulté à tester pour chaque processeur, l'acceptation des tâches affectées.

Dans la majorité des applications rencontrées, il s'avère donc complexe voire même impossible de vérifier hors ligne qu'un système est fiable ou non. Il est donc apparu primordial de disposer d'une méthodologie systématique pour concevoir un système temps-réel sûr c'est-à-dire exempt de toute violation de ses contraintes temporelles.

III. Le mécanisme à échéance.

1. Description.

Les définitions suivantes des concepts de base de la tolérance aux fautes sont issues de celles de [RANDELL 78] et [HECHT 76].

On dit qu'un système est défaillant lorsque celui-ci dévie de son comportement spécifié. Plus particulièrement, une défaillance temporelle se produit quand un système temps-réel viole une de ses contraintes temporelles.

On appelle faute temporelle, une faute qui provoque une erreur temporelle, c'est-à-dire une composante non désirée dans l'état du système telle que: une échéance imminente, le point d'exécution d'une tâche ... Les origines des fautes temporelles sont très diverses. Dans les systèmes temps-réel, la majorité des fautes temporelles sont des retards dans l'exécution des tâches dûs à des délais de blocage importants en leurs points de synchronisation. L'occurrence de telles fautes est aléatoire à cause de l'asynchronisme dans l'évolution des tâches et donc de l'indéterminisme qui caractérise la durée de franchissement de certains points de synchronisation.

Les fautes temporelles peuvent avoir d'autres origines telle qu'une malfaçon dans la programmation (une instruction erronée par exemple) qui a pour conséquence le retardement anormal de la fin d'exécution de la tâche, voire la délivrance de données erronées.

Un système temps-réel tolérant les fautes (TRTF) désigne un système qui contient un surplus de logiciel dont l'objet est d'assurer que l'occurrence d'un état erroné dú système n'engendre pas de défaillance temporelle.

Le mécanisme à échéance, présenté dans [CAMPBELL 79a] fournit une approche dans la construction de systèmes temps-réel tolérant les fautes. Il est inspiré du mécanisme de bloc de recouvrement [RANDELL 75]. La raison pour laquelle ce dernier ne peut être employé dans un système temps-réel est son insensibilité au passage du temps et donc à l'occurrence d'une échéance. Le test d'acceptation est ainsi remplacé dans le mécanisme à échéance, par un ordonnanceur spécialisé. Ces deux mécanismes peuvent toutefois être implémentés conjointement pour traiter des fautes d'origines logicielle et temporelle.

Le principe de base du mécanisme à échéance est d'associer à chaque tâche un algorithme dit primaire et un algorithme dit secondaire. La mise en oeuvre de ces deux algorithmes sera respectivement appelée processus primaire et processus secondaire.

Le processus primaire fournit le service le plus souhaité mais est sujet à des fautes temporelles alors que le processus secondaire fournit un service juste acceptable mais est exempt de toutes fautes temporelles.

Un algorithme d'ordonnancement doit alors assurer que sur une durée infinie, toutes les échéances soient satisfaites soit par le processus primaire soit par le secondaire. Si le processus primaire termine son exécution avant son échéance, ses résultats sont utilisés en priorité par rapport à ceux du processus secondaire.

Par contre, si le processus primaire ne parvient pas au terme de son exécution avant son échéance, les résultats du secondaire sont utiisés. Cela suppose que le processus primaire doit obligatoirement être exécuté avant la même échéance que celle du primaire échoué. Ordonnancer une configuration de tâches consiste donc en la spécification de l'identité du primaire ou du secondaire qui doit être exécuté à tout instant.

On dira qu'une séquence est TF-valide si toutes les tâches ont leur échéance respectée même si aucun processus primaire n'a terminé son exécution avec succès.

2. Algorithmes d'ordonnancement.

a. Généralités.

La principale exigence sur laquelle repose le mécanisme à échéance est

de connaître avec précision la durée d'exécution maximale des processus secondaires. Ce mécanisme a effectivement été conçu de telle sorte que l'ordonnanceur réserve suffisamment de temps pour l'exécution des secondaires quand une requête survient. Les primaires sont alors ordonnancés dans le temps restant appelé temps creux.

Un algorithme d'ordonnancement de processus primaires est alors utilisé pour séquencer ceux-ci dans le temps creux, la réservation du temps pour l'exécution des secondaires étant réalisée par un autre algorithme.

Le séquencement du primaire et du secondaire dans le délai critique peut être réalisé de différentes façons :

- le primaire avant le secondaire
- le secondaire avant le primaire
- primaire et secondaire en pseudo-parallélisme ...

D'une façon générale, on peut définir une stratégie d'ordonnancement tolérant les fautes comme la juxtaposition d'une stratégie d'ordonnancement des primaires et d'une stratégie d'ordonnancement des secondaires.

b. Conditions d'ordonnançabilité.

la mise en oeuvre detoute application temps-réel doit au préalable passer par une phase de test de la fiabilité du système. Il est effectivement nécessaire de s'assurer à l'initialisation que tous les processus secondaires pourront s'exécuter sans violer les contraintes temporelles.

On dira qu'une configuration de tâches est TF_ordonnançable si tous ses processus secondaires peuvent être fiablement ordonnancés.

La condition nécessaire de TF_ordonnançabilité est que la charge infligée au processeur par l'exécution de tous les processus secondaires soit inférieure ou égale à la capacité de traitement de ce processeur.

Soit $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ une configuration de tâches où chaque $\mathcal{C}_i$ est caractérisée par :

- T_i sa période
- C_i la durée d'exécution de son processus secondaire.

$\mathcal{C}$ est TF_ordonnançable si l'inégalité $\sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant 1$ est vérifiée.

3. Stratégies d'ordonnancement des secondaires.

Deux stratégies ont été présentées dans [CAMPBELL 79a]. Il s'agit de l'algorithme dit de première chance et de l'algorithme dit de dernière chance.

a. L'algorithme de première chance.

Les processus secondaires sont ordonnancés en faisant abstraction des processus primaires. Ces derniers s'exécuteront dans le temps creux après que leur secondaire se soit terminé. Les résultats de calcul du secondaire doivent être conservés jusqu'à ce que le primaire se termine correctement ou que l'échéance soit atteinte. Les résultats du secondaire peuvent aussi être utilisés comme un test de validité des résultats obtenus par le primaire.

L'ordonnancement des processus secondaires pourra faire appel à l'un des algorithmes décrits au § II.3.a. De par son optimalité, on pourra choisir RU. Si l'on est contraint d'utiliser un algorithme à priorité fixe, on optera pour RM.

Exemple :

Soit $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2\}$ avec $(C_1, R_1, T_1) = (0.5, 3, 3)$

$(C_2, R_2, T_2) = (1, 5, 6)$

$\mathcal{C}$ est ordonnançable puisque $\frac{C_1}{T_1} + \frac{C_2}{T_2} \leqslant 1$

D'autre part, $CH = \frac{C_1}{R_1} + \frac{C_2}{R_2} = \frac{5.5}{15}$. Donc RU produit une séquence TF-valide.

Si l'on suppose les primaires ordonnancés selon RU, la séquence produite sur [0,12] est la suivante :

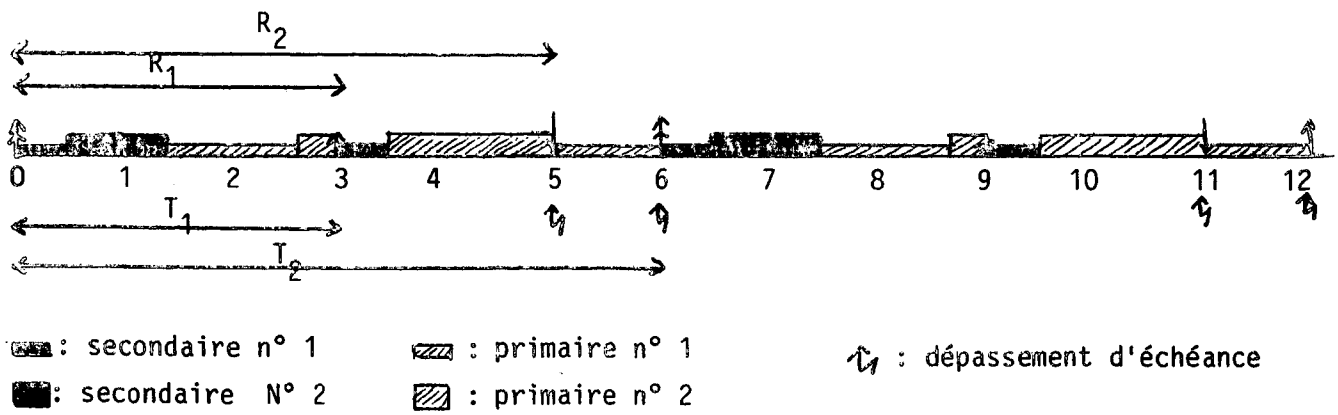

Fig. 2 : Description d'une séquence T F-valide

On constate que :

- sur les deux requêtes de $\mathcal{C}_2$, le primaire échoue deux fois, il faut alors utiliser les résultats du secondaire
- sur les quatre requêtes de $\mathcal{C}_1$, le primaire réussit deux fois. On utilisera ses résultats de préférence à ceux de son secondaire.

b. L'algorithme de dernière chance.

Quand une requête arrive, l'ordonnanceur chargé du séquencement des processus secondaires réserve du temps processeur pour l'exécution des secondaires. Cet intervalle de temps est choisi de telle sorte que le secondaire s'exécute au plus tard de manière à ce que sa fin d'exécution coïncide avec son échéance. On suppose, pour faciliter l'implémentation, que les secondaires ne peuvent se suspendre entre eux. Les processus primaires sont alors exécutés dans le temps creux avant leur secondaire. Un secondaire peut donc suspendre un primaire en cours d'exécution si l'instant courant correspond au début de l'intervalle d'activation du secondaire.

Le problème posé par un ordonnancement de dernière chance est celui du chevauchement des périodes d'exécution des secondaires. En effet, au fur et à mesure que des requêtes arrivent, il est nécessaire de redéfinir les intervalles d'exécution des secondaires. On conviendra d'utiliser l'algorithme RUNP 'au plus tard' [KIM 80a] ce qui revient à exécuter les secondaires de façon non préemptive et en fonction de leur urgence.

Exemple :

Soit $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ avec $(C_1, R_1) = (1, 5)$
$(C_2, R_2) = (2, 10)$
$(C_3, R_3) = (1, 8)$

à t = 0, $\mathcal{C}_1$ et $\mathcal{C}_2$ sont réveillées (Fig. 3a).

Les intervalles d'exécution des secondaires de $\mathcal{C}_1$ et $\mathcal{C}_2$ sont respectivement [4,5] et [8,10].

Fig. 3a

à t = 1, $\mathcal{C}_3$ est réveillée (Fig. 3b).

Les nouveaux intervalles d'exécution sont pour $\mathcal{C}_1$, $\mathcal{C}_2$ et $\mathcal{C}_3$ respectivement [4,5], [8,10] et [7,8].

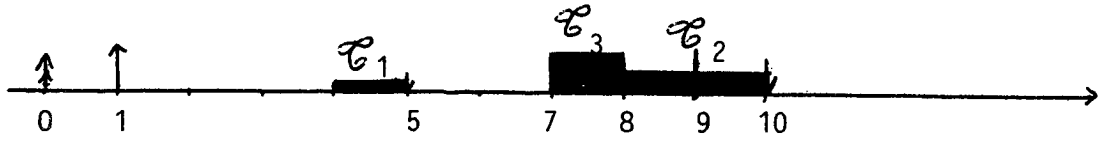

Fig. 3b

Il est clair qu'avec l'algorithme de dernière chance, les processus secondaires ne peuvent être ordonnancés que par une politique à priorité dynamique. Une politique à priorité fixe conduirait fréquemment à une séquence non TF_valide, ne prenant pas en compte l'imminence des échéances.

Comme la préemption des processus primaires est interdite, une contrainte sévère concernant les durées d'exécution des secondaires doit être respectée afin de garantir la fiabilité du système.

Cette condition est $\sum_{i=1}^{n} C_i \leqslant R_n$ où R_n désigne le plus petit délai critique.

Pour l'algorithme de première chance utilisant RU, pour assurer la fiabilité du système, il suffit que le facteur de charge des secondaires soit inférieure à un, c'est-à-dire

$$\sum_{i=1}^{n} \frac{C_i}{R_i} \leqslant 1.$$

Par contre, si RM est utilisé et si les tâches sont telles que $R_i = T_i$, alors la fiabilité du système est assurée si :

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant n(2^{1/n} - 1) \qquad [\text{SERLIN 72}]$$

ou encore si :

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant \log 2$$

L'avantage de la stratégie de dernière chance est qu'elle organise les exécutions de telle sorte que si le primaire réussit, le secours ne s'exécutera pas et le temps creux disponible pour l'exécution des primaires s'en trouvera d'autant augmenté.

4. Stratégies d'ordonnancement des primaires.

Le mécanisme à échéance vise à assurer le respect de toutes les échéances par l'exécution correcte des primaires ou de leur secondaire. Plusieurs stratégies d'ordonnancement des primaires peuvent être choisies selon le critère de performance que l'on désire optimiser.

* On désire maximiser le nombre de primaires respectant leur échéance. Dans ce cas, on dira qu'une séquence est optimale si elle est TF_valide et si le nombre de processus primaires ordonancés fiablement est maximum parmi toutes les séquences TF_valides.
Sachant que RU est l'algorithme qui permet à un nombre maximal de tâches à date critique de partager un processeur, la production d'une séquence optimale sera obtenue en ordonnançant par RU les processus primaires dans le temps creux.

* On désire privilégier certains primaires en raison de leur importance aux vues de l'application. En effet, dans bon nombre d'applications, le service fourni par certaines tâches est plus crucial au regard du cahier des charges que celui fourni par d'autres tâches. Par exemple, il est plus important pour un avion de tirer une rocket au bon moment plutôt que de renvoyer des informations de capteur à la terre et ce bien que la fréquence d'émission d'une rocket est largement plus faible que celle de la transmission d'informations. L'importance d'une tâche n'est donc pas forcément liée à sa fréquence d'activation ni à son délai critique mais plutôt à sa fonctionnalité.
On associe initialement à chaque primaire, un poids fonction de son importance vis-à-vis de l'application qui permet, pendant les périodes de surcharge du processeur, d'allouer à chaque tâche une fraction pondérée de sa capacité de traitement.

On dira alors qu'une séquence est optimale si elle est TF_valide et si la somme pondérée des processus primaires ordonnancés fiablement est maximum parmi toutes les séquences TF-valides.

D'évidence, l'algorithme d'ordonnancement qui associe à chaque primaire une priorité égale à son poids est celui qui conduit à une séquence optimale.

Pour des primaires de même priorité, on pourra ordonnancer ces derniers en pseudo-parallèle ou utiliser un deuxième critère de choix tel que l'urgence, la périodicité ou la date passée la plus proche de la dernière exécution réussie.

Exemple :

Soit $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ telle que $(C_1, R_1, T_1) = (0.5, 4, 4)$
$(C_2, R_2, T_2) = (0.5, 6, 6)$
$(C_3, R_3, T_3) = (1, 12, 12)$

$\mathcal{C}$ est TF ordonnançable puisque $\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C}{T_3} < 1$

Supposons $\mathcal{C}$ ordonnancée par la stratégie de première chance utilisant RU pour le séquencement des primaires et des secondaires.

La séquence produitesur l'intervalle [0,12] est la suivante :

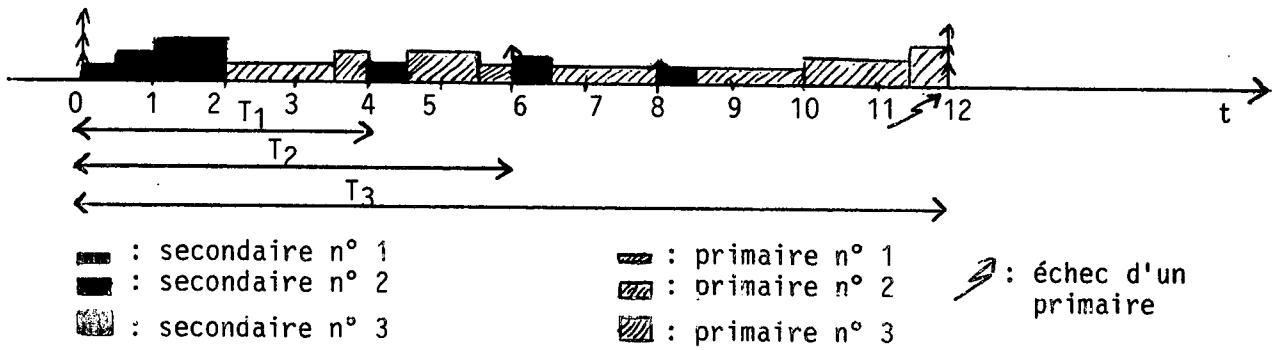

Fig. 4a

La figure 4a montre que sur six requêtes émises, un seul primaire a échoué et donc cinq secondaires ont été exécutés inutilement, représentant un taux d'utilisation inutile du processeur égal à 25 %.

Supposons maintenant que $\mathcal{C}$ est ordonnancée par la stratégie de dernière chance : le chronogramme de la fig. 4b représente les intervalles de temps réservés pour les secondaires alors que celui de la fig. 4c représente la séquence effectivement produite.

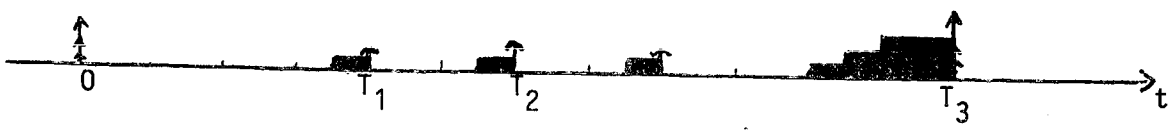

Fig. 4b

Fig. 4c

La fig. 4c montre qu'aucun primaire n'a échoué et donc qu'aucun secondaire n'a eu besoin de s'exécuter.

On constate sur cet exemple, où la durée d'exécution d'un primaire est supposée égale à trois fois celle de son secondaire, que la stratégie de dernière chance est plus performante que la stratégie de première chance.

5. Performance des stratégies.

a. Critères de performance.

Dans le paragraphe précédent, la performance d'une stratégie d'ordonnancement était perçue uniquement comme une mesure du pourcentage de primaires ordonnancés fiablement.

Cependant, cette mesure ne donne qu'une vue partielle de la performance car elle est effectuée parmi des classes particulières de stratégies que sont les stratégies de première et de dernière chance. En effet, rien ne laisse supposer la non existence d'une stratégie qui ordonnancerait selon un algorithme commun processus primaires et secondaires. D'autre part, n'oublions pas que le mécanisme à échéance a été introduit pour améliorer la sûreté de fonctionnement par une adjonction de logiciels redondants. Il faut donc s'interroger sur l'efficacité d'un tel surplus de charge, celle-ci risquant de dégrader la sensibilité du système plutôt que de l'améliorer.

La comparaison de différentes stratégies doit donc passer par la mesure de deux grandeurs :

- le pourcentage de temps processeur utilisé pour l'exécution des secondaires dont les résultats ne sont jamais utilisés
- le pourcentage de temps processeur utilisé pour l'exécution des primaires abandonnés car non terminés avant leur échéance.

L'ensemble de ces deux grandeurs constitue une mesure du coût de la tolérance aux fautes exprimée en terme de gaspillage de capacité de traitement. Il s'agit donc de trouver un compromis entre le taux de logiciel redondant et les performances obtenues, exprimées en taux d'utilisation utile du processeur.

On dira donc qu'une séquence est TF_optimale si elle est TF_valide, optimale et de coût minimal.

b. Etudes de performance.

Un modèle simple de simulation a été développé par Wei et all [WEI 80] afin d'obtenir une estimation du coût induit par les stratégies de première et de dernière chance, en supposant que RU est utilisé pour les primaires et les secondaires. Ces simulations ont permis de mettre en évidence les points suivants :

- l'utilisation de la stratégie de dernière chance conduit à un gaspillage du temps provoqué par des primaires qui n'arrivent pas à terme
- l'utilisation de la stratégie de première chance conduit à un gaspillage du temps provoqué à la fois par les primaires et les secondaires
- quand les longueurs d'exécution des secondaires sont petites comparées à celles des primaires (ce qui est généralement le cas), le temps utilisé par les secondaires sont beaucoup moins influents et les performances de la stratégie de première chance semblent plus compétitives.

IV. Application et implémentation du mécanisme.

1. Réalisations pratiques.

Le mécanisme à échéance a été introduit pour maintenir une dégradation élégante du système sans produire de pannes temporelles. Des systèmes tolérant les fautes présentant cette propriété sont requis dans la plupart des applications où une intervention humaine est impossible.

Dans des domaines très variés tels que les procédés chimiques, les centrales nucléaires, les télécommunications et l'aéronautique, la sophistication grandissante des systèmes nécessite des techniques élaborées d'amélioration de performances mais surtout de fiabilité voire de sécurité.

Le mécanisme à échéance fournit donc une des approches possibles pour la conception de systèmes sûrs de fonctionnement. Celui-ci a d'ailleurs été implanté sur l'ordinateur de bord du satellite MMS (Multimission Modula Spacecraft) conçu par la NASA pour les années 80 [NASA 77]. Les résutlats de cette application ont d'ailleurs démontré la fiabilité du mécanisme. Un exemple de tâche critique implantée était constitué d'un filtre de Kalman pour le processus primaire et d'une simple prédiction par matrice de transition pour le processus secondaire.

2. Implémentation.

Le mécanisme à échéance peut être spécifié dans un programme écrit dans un langage évolué en adjoignant à celui-ci une nouvelle construction. Celle-ci devrait permettre de programmer explicitement les divers paramètres temporels d'une tâche, utilisé à un niveau inférieur par l'ordonnanceur du système d'exploitation. Les langages bas niveau, encore fréquemment utilisés dans les systèmes temps réel peuvent difficilement spécifier des contraintes temporelles. D'autre part, des études de comportement en simulation ou en phase d'essais sur site ne sont pas des méthodes qui permettent de garantir à cent pour cent la fiabilité d'un système.

Cette fiabilité peut cependant être améliorée par le développement de langages de programmation haut niveau [WIRTH 77a]. Une simple extension de langages tels que Modula [WIRTH 77b], Concurrent Pascal [BRINCH HANSEN 77], Ada [WEGNER 79] ou Path Pascal [CAMPBELL 79b] permet alors de programmer de façon simple le mécanisme à échéance.

Avec de tels langages étendus, on peut alors décrire une tâche critique de la façon suivante :

```
tâche à date critique Calcul
      co déclarations locales fco
   procédure primaire
      début
        | co corps du processus primaire fco
      fin
   procédure secondaire
      début
        | co corps du processus secondaire fco
      fin
   début
     | co initialisation de variables locales fco
     | attendre(date de début)
     | répéter tous les x secondes
     |         dans y secondes faire
     |             processus primaire sinon secondaire ;
     | jqa terminé
   fin
```

L'interprétation d'une phrase telle que celle décrite ci-dessus peut être réalisée de façon simple à l'aide d'un exécutif temps-réel muni d'un ordonnanceur apte à gérer deux classes de processus : les primaires et les secondaires.

Prenons l'exemple de l'exécutif temps-réel RTEX [DEPLANCHE 85]. Toute tâche est déclarée à l'exécutif par une primitive qui initialise une zone de mémoire de taille finie appelée descripteur de tâche et adressable uniquement par l'exécutif. Il a la forme d'un vecteur comprenant d'une part, les informations caractéristiques de la tâche tels que son nom, sa priorité,... d'autre part des pointeurs qui assurent le chaînage du descripteur dans les différentes structures de données de l'exécutif. Ainsi, tel état d'une tâche (bloqué, réveillée, active) est caractérisée par l'appartenance de son descripteur à telle structure et toute transition d'état consiste en son transfert d'une structure à une autre par modification des pointeurs.

Lorsqu'une tâche est réveillée et désire s'exécuter, son descripteur est placé dans une liste appelée liste des tâches activables (LTA). La résolution des conflits dûs à la présence simultanée de plusieurs descripteurs dans la LTA est alors confiée à un logiciel spécifique, l'ordonnanceur qui met en oeuvre la règle de choix de la tâche à activer à tout instant. En fonction du choix émis par l'utilisateur, l'ordonnanceur utilisera une stratégie telle que : RU, RM, RUNP, RMNP, ...

Pour implémenter le mécanisme à échéance, cet exécutif doit alors être modifié de façon à pouvoir assurer la gestion de deux types de tâches. Chaque descripteur est ainsi muni d'un champ supplémentaire afin d'y stocker le type de la tâche, c'est-à-dire primaire ou secondaire.

Avant le lancement d'une application, l'utilisateur doit alors spécifier la nature de l'ordonnancement utilisé par le mécanisme à échéance, c'est-à-dire spécifier :

- la stratégie : première ou dernière chance
- l'algorithme d'ordonnancement des primaires
- l'algorithme d'ordonnancement des secondaires.

Quel que soit leur type, toutes les tâches (ou processus) réveillées sont regroupées dans la LTA. Supposons choisie la stratégie de première chance. Comme les primaires s'exécutent dans le temps creux après leur secondaire, tout secondaire sera plus prioritaire que tout primaire. A tout instant, la tâche active est donc soit le secondaire le plus prioritaire de la LTA, soit le primaire le plus prioritaire de la LTA s'il n'y a pas de secondaire réveillé.

L'implémentation de la stratégie de dernière chance est un peu plus compliquée car un secondaire s'exécute uniquement si son primaire a échoué. Quand une requête arrive, le primaire et le secondaire associés à cette requête passent dans la LTA. L'ordonnanceur doit alors réévaluer pour chaque secondaire présent dans la LTA, son instant d'activation. Les primaires s'exécutent jusqu'à l'oc-

currence de l'instant d'activation le plus imminent d'un secondaire. D'autre part, si un primaire termine correctement son exécution, un signal doit être envoyé à son secondaire lui spécifiant de ne pas s'exécuter. Cela peut être réalisé très simplement en terminant le code de chaque primaire par l'appel à une primitive de l'exécutif chargée de bloquer le secondaire jusqu'à l'occurrence de la requête suivante.

3. Extension aux systèmes répartis.

Des développements continus dans le domaine des techniques de transmission laisse entrevoir une augmentation des systèmes informatiques dans lesquels tout le logiciel est distribué sur plusieurs processeurs qui travaillent de façon asynchrone [STANKOVIC 84]. Les systèmes de ce type vont jouer dans un futur immédiat des rôles déterminants dans les applications temps réel à contraintes strictes à cause d'une réduction significative des coûts des processeurs et des mémoires mais sûrtout à cause de la disponibilité d'une plus grande gamme de réseaux de communication performants et sûrs. En outre, un des avantages les plus reconnus du traitement distribué, sur le traitement centralisé est la possibilité d'incorporer des méthodes de dégradation élégante en cas de défaillance d'un processeur. Dans un système distribué muni d'une fonction de reconfiguration, tout ou partie des tâches assignées à un calculateur défaillant peut être prise en charge par un ou plusieurs autres calculateurs. Toutefois, l'exigence fondamentale des applications TRCS est que cette reconfiguration s'effectue dans les limites de temps imposées par le cahier des charges.

Alors que le mécanisme à échéance est utilisé dans les systèmes centralisés mono ou multiprocesseurs afin d'assurer une robustesse vis-à-vis des fautes temporelles, ce mécanisme peut aussi être utilisé dans les systèmes distribués afin d'assurer une robustesse vis-à-vis des fautes matérielles.

A chaque processus primaire résidant sur un calculateur est associé un processus secondaire résidant sur un autre calculateur. Si le primaire échoue à cause de la défaillance d'un matériel tel qu'un module mémoire, un canal d'E/S , ... le secondaire correspondant devient actif et doit pouvoir terminer son exécution avant son échéance [KIM 80b].

Le problème de la conception de systèmes de traitement distribué sûrs consiste alors à s'assurer que dans tous les cas de pannes susceptibles de se produire, toutes les contraintes temporelles seront respectées.

On ne s'étendra pas davantage sur ce sujet encore mal dominé. On mettra seulement en évidence quelques éléments déterminants dans la difficulté du problème de la reconfiguration :

1- La répartition des processus secondaires.

Cette répartition faite initialement pourra être soit dictée par des considérations d'ordre technologique soit satisfaire à un critère d'optimisation (équilibre de charge, minimisation des coûts de communication ...). D'autre part, en cas de pannes simultanées de plusieurs calculateurs, il est nécessaire de disposer de plusieurs processus secondaires pour le même primaire. Le facteur de réplication est alors une donnée du problème de la répartition. De nombreux papiers portent sur ce thème, dont [CHOU 82], [GYLYS 76], [EFE 82], [KAR 83], [HUANG 85] et [BANNISTER 83].

2- La reconfiguration en cas de panne.

Le problème de la reconfiguration est celui du choix du processus secondaire parmi ceux répartis dans le réseau et associés au même primaire, qui devra prendre en charge les responsabilités de ce dernier en cas d'échec. Plusieurs stratégies de reconfiguration [BARIGAZZI 82], [KIM 79], [FERGUSON 85] peuvent être envisagées, statiques ou dynamiques selon que les décisions se prennent hors ligne ou en ligne.

3- La fiabilité.

Il s'agit de vérifier que sur chaque calculateur et dans toute situation ordinaire ou de panne, toutes les contraintes temporelles sont satisfaites même si ce n'est que par des processus secondaires. Cela doit donc être effectué au moment de la conception de l'application en utilisant les techniques évoquées précédemment pour la détection d'une faute temporelle. Ce problème est précédé de celui du réajustement des échéances des processus primaires, nécessaire pour prévenir un dépassement d'échéance lors de l'échec du processus primaire [KIM 80b]. La nouvelle échéance d'un primaire est égale à son échéance initiale minorée du temps de réponse maximal de son secondaire et du délai s'écoulant entre les instants de connaissance de la défaillance et d'occurrence de la panne.

En fait, c'est parce que ces trois points sont intimement liés que la mise en oeuvre du mécanisme à échéance dans un système distribué s'avère très complexe.

V. Conclusions.

On a décrit dans ce papier une méthodologie pour concevoir des systèmes temps-réel sûrs, c'est-à-dire respectant toutes leurs contraintes temporelles même en présence de fautes.

Les contraintes temporelles sont un aspect important de la fiabilité des systèmes temps-réel et sont rigoureusement spécifiées. Cependant vérifier que ces contraintes sont satisfaites s'avère être très difficile. Il est donc important de doter systématiquement les systèmes temps-réel du mécanisme à échéance afin de leur assurer une haute robustesse vis-à-vis des pannes temporelles.

Bien que semblables, le mécanisme à échéance assure une fonction différente de celle du schéma classique du bloc de recouvrement. Conçu pour faire face à des fautes temporelles, il assure des temps de réponse bornés grâce à des stratégies d'ordonnancement pertinentes.

Ce mécanisme peut être implémenté non seulement dans un système centralisé mais aussi dans un système distribué sujet à des défaillances matérielles. Par une réplication des processus secondaires et une reconfiguration du système lors d'une panne de calculateur, il assure une dégradation élégante du comportement du système tout en maintenant le respect des échéances.

Le comportement du mécanisme à échéance dans un système centralisé a été largement étudié [CAMPBELL 79], [WEI 80], [LIESTMAN 83]. Les études à venir devront donc se porter sur son utilisation dans les systèmes répartis. Bien qu'un nombre important de chercheurs se penchent depuis récemment sur les problèmes de répartition de charge et de reconfiguration dynamique, le domaine de faibilité d'un système temps-réel distribué tolérant les fautes n'est encore pas actuellement, totalement couvert.

BIBLIOGRAPHIE

[BANNISTER 83] : J.A. Bannister et K.S. Trivedi
"Task allocation in fault-tolerant distributed systems", Acta Informatica, 20, pp. 261-281 (1983).

[BARIGAZZI 82] : G. Barigazzi, A. Ciuffoletti et L. Strigini
"A distributed Algorithm for post-failure load redistribution", 3th Int. Conf. on Distributed Computing Systems, Miami, Floride, 1982.

[BELMANS 85] : P. Belmans, J.J. Borrelly, M. Silly et D. Simon
"NESTOR : Noyau d'Exécutif pour le Suivi en Temps-réel des applications Orientées Robotique", Publication interne n° 267, IRISA, Rennes, Sept. 1985.

[BRINCH HANSEN 77] : P. Brinch Hansen
"The Architecture of Concurrent Programs", Prentice Hall, Englewood Cliffs, New Jersey, 1977.

[CAMPBELL 79a] : R.H. Campbell, K.H. Horton et G.G. Belford
"Simulations of a fault-tolerant deadline mechanism", Proc. of the FTCS-9, Maddison, Wisonsin, pp. 95-101, Juin 1979.

[CAMPBELL 79b] : R.H. Campbell et R.B. Kolstad
"Path Expressions in Pascal", 4th Int. Conf. on Software Engineering, Munich, Allemagne, Sept. 1979.

[CHETTO 85] : H. Chetto, M. Silly et J.P. Elloy
"Politiques d'ordonnancement de tâches temps-réel dans un système mono-processeur", 3th Int. Symp. Applied Informatics, Grindelwald, Suisse, Fév. 1985.

[CHOU 82] : T.C.K. Chou et J.A. Abraham
"Load balancing in distributed systems", IEEE T.S.E., Vol.SE-8, N° 4, Juillet 1982.

[COFFMAN 76] : E.G. Coffman
"Computer and Job-shop scheduling theory", John Wiley Eds., 1976.

[CONWAY 67] : R.W. Conway, W.L. Maxwell et L.W. Miller
"Theory of scheduling", Addison-Wesley, Reading, Massachusetts, 1967.

[DEPLANCHE 85] : A.M. Deplanche et M. Silly
"Conception et implémentation d'un noyau d'exécutif temps-réel pour la commande de processus par microprocesseur", the 6th Int. Conf. on Control systems and Computer Science, Bucharest, Roumanie, Mai 1985.

[DHALL 78] : S.K. Dhall et C.L. Liu
"On a real-time scheduling problem", Operations Research, Vol. 26, N° 1, pp. 127-140, Jan/Fév. 1978.

[EFE 82] : K. Efe
"Heuristic Models of task assignment scheduling in distributed systems", IEEE Computer, Vol. 15, N° 6, Juin 1982.

[FERGUSON 85] : D.F. Ferguson et G. Leither
"Relocating processes in a distributed computer system", IBM Research Division, RC 11 190, Dept. of Computer Science, Columbia University, New-York, Nov. 1985.

[FINEBERG 67] : M.S. Fineberg et O. Serlin
"Multiprogramming for hybrid computation", Proc. AFIPS FJCC 1967.

[GONZALEZ 77] : M.J. Gonzalez
"Deterministic processor scheduling", Computing Surveys, Vol. 9, N° 3, Sept. 1977.

[GYLYS 76] : V.B. Gylys et J.A. Edwards
"Optimal partitioning of workload for distributed systems", Proc. Compcon Fall 76, pp. 353-357.

[HECHT 76] : H. Hecht
"Fault-tolerant software for real-time applications", Computing Surveys Vol. 8, N° 4, Déc. 1976.

[HUANG 85] : J.P. Huang
"Modeling of software partition for distributed real-time applications", IEEE T.S.E., Vol. SE-11, N° 10, Oct. 1985.

[KAR 83] : G. Kar, C.N. Nikolaou et J. Reif
"Assigning processes to processors : A fault-tolerant approach", Research Report RC 10538, IBM Research Division, Yorktwon Heights, Nov. 1983.

[KIM 79] : K.H. Kim
"Erro detection, reconfiguration and recovery in distributed processing systems", Proc. 1st Int. Conf. on distributed Computing systems, Oct. 1979.

[KIM 80a] : K.H. Kim et M. Naghibzadeh
"Prevention of tasks overruns in real-time non preemptive multiprogramming systems", ACM Sigmetrics, Vol. 9, N° 2, pp. 267-276, 1980.

[KIM 80b] : K.H. Kim
"An approach to secure design of distributed and reconfigurable real-time computer systems", Distributed Data Acquisition, Computing and Control Symposium, Miami, Floride, Déc. 1980.

[LABETOULLE 74] : J. Labetoulle
"Ordonnancement des processus temps-réel sur une ressource préemptive", Rapport de recherche IRIA N° 74, Mai 1974.

[LEINBAUGH 80] : D.W. Leinbaugh
"Guaranted response times in a hard real-time environment", IEEE T.S.E., Vol. SE-6, N° 1, Janv. 1980.

[LENSTRA 80] : J.K. Lenstra et A.H.G. Rinnoy Kan
"An introduction to multiprocessor scheduling", Rapport BW 121/80, Mathematisch Centrum, Amsterdam 1980.

[LIESTMAN 83] : A.L. Liestman et R.H. Campbell
"A fault-tolerant scheduling problem", Proc. of the FTCS-13, 1983.

[LIU 73] : C.L. Liu et J.W. Layland
"Scheduling algorithms for multiprogramming in a hard real-time environment", J.A.C.M. N° 20, pp. 46-61, Janv. 1973.

[NASA 77] : NASA GSFC
"Multimission Modular Spacecraft System Specification", S-700-10, Mai 1977.

[RANDELL 75] : B. Randell
"System Structure for software fault-tolerance", IEEE T.S.E., Vol. SE-1, N° 2, Juin 1979.

[RANDELL 78] : B. Randell, P.A. Lee et P.C. Treleaven
"Reliability issues in Computing System design", Computing Surveys, Vol. 10, N° 2, Juin 1978.

[SERLIN 72] : O. Serlin
"Scheduling of time critical processes", Proc. of the spring joint computers conference, pp. 925-932, 1972.

[SILLY 84] : M. Silly
"Contribution à l'ordonnancement de tâches temps-réel pour exécutifs centralisé et réparti", Thèse de doctorat de 3ème cycle, Université de Nantes, Sept. 1984.

[SORENSON 74] : P.G. Sorenson
"A methodology for real-time system development", Ph. D. Université de Toronto, Canada, 1974.

[STANKOVIC 84] : J.A. Stankovic
"A perpective on distributed computer systems", IEEE Transactions on Computers, Vol. C33, N° 12, Déc. 1984.

[WEGNER 79] : P. Wegner
"Programming with ADA : An Introduction by means of graduated examples", SIGPLAN NOTICES, Vol. 14, N° 12, Déc. 1979.

[WEI 80] : A.Y. Wei, K. Hiraishi, R. Cheng et R.H. Campbell
"Application of the fault-tolerant deadline mechanism to a satellite on board computer system", Proc. of the FTCS-10, Tokyo, pp. 107-109, 1980.

[WIRTH 77a] : N. Wirth
"Towards a discipline of real-time Programming", CACM, Vol. 20, N° 8, Août 1977.

[WIRTH 77b] : N. Wirth
"Modula : a language for modular multiprogramming", Software Practice and Experience, 7, 1977.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique