Expériences de mise au point d'un algorithme réparti

Xavier Blondel‡ — Laurent Rosenfeld‡
Frédéric Ruget* — Frank Singhoff‡

** Sun Microsystems, Embedded System Software Group*
6, avenue Gustave Eiffel
F-78182 Saint-Quentin-en-Yvelines

‡ CNAM-Laboratoire CEDRIC
292, Rue St Martin
75141 Paris Cedex 03

RÉSUMÉ. Mettre au point un algorithme réparti est toujours une tâche délicate. Plusieurs techniques sont disponibles pour ce faire. Nous avons cherché à vérifier un algorithme réparti suivant deux méthodes. La première approche consiste à utiliser un outil d'analyse dynamique, basé sur la réexécution d'une application. La seconde approche est de procéder à une analyse statique par le biais d'une validation automatisée d'un modèle décrit à l'aide d'une technique de description formelle. Nous dégageons de nos expérimentations une méthode permettant la vérification d'applications réparties accédant à des ressources critiques et communiquant par messages asynchrones.

ABSTRACT. Perfecting a distributed algorithm is never a straightforward task. Several techniques exist to verify a distributed algorithm. Among these, we have tried to use two different methods; the first one is based on a dynamic analysis tool, enabling us to replay an application. The second method is a static analysis: an automatical verification of a model described with a formal description technique. Our work leads to a method to verify distributed applications that share critical resources and communicate by means of asynchronous messages.

MOTS-CLÉS : Mise au point, analyses statique et dynamique, réexécution, CDB, méthodes formelles, validation, LDS KEY WORDS : Debugging, static and dynamic analyses, replaying, CDB, formal methods, validating, SDL

1. Introduction

La phase de mise au point et de validation a toujours été l'une des étapes les plus importantes et les plus coûteuses du développement des programmes informatiques. Elle est particulièrement délicate avec les systèmes répartis.

L'objet de cet article est de relater une expérience de mise au point d'un algorithme réparti sous CHORUS, en l'occurrence l'algorithme de mémoire partagée répartie par diffusion de Kai Li et Paul Hudak [LI 89], que nous appellerons plus simplement «l'algorithme de Li et Hudak». Le choix de cet algorithme a été motivé par la découverte d'une anomalie dans une première maquette que nous avions réalisée sous Unix, mais dont la mise en évidence était malaisée. De plus, nous disposions d'une solution pour cette anomalie, mais sans avoir de certitude sur son bon fonctionnement ni sur son innocuité.

Nous avons utilisé deux techniques complémentaires pour procéder à cette vérification. La première est basée sur une méthode d'analyse dite *dynamique*, qui consiste à observer le comportement d'une mise en œuvre réelle. La seconde utilise une technique de description formelle et un outil de validation pour effectuer une analyse dite *statique*, ne s'appuyant donc pas sur une mise en œuvre réelle.

L'outil d'analyse dynamique utilisé est CDB (*Communication Debugger*), qui permet la mise au point d'applications s'exécutant dans un système CHORUS. L'analyse statique est effectuée sur un modèle LDS (*Langage de Description et de Spécification*[1]) par le validateur de l'outil SDT 3.02 (*SDL Design Tool*) de Telelogic.

Nous présentons dans cet article la démarche que nous avons utilisée pour bénéficier de ces deux techniques. Nous en constatons ainsi de nombreux avantages, mais aussi quelques inconvénients, ce qui nous permet de mettre en évidence leur complémentarité.

Plus généralement, l'approche présentée dans ce document peut être appliquée à la vérification et à la validation d'applications réparties, partageant des ressources accédées par le biais de messages asynchrones.

La difficulté de mettre au point une application répartie et quelques solutions possibles font l'objet de la partie 2. Nous rappelons l'algorithme de Li et Hudak et expliquons l'anomalie de fonctionnement, mentionnée ci-dessus, dans la partie 3. L'utilisation de l'approche dynamique pour analyser cet algorithme est abordée dans la partie 4 et celle de l'approche statique dans la partie 5.

2. Mettre au point un algorithme réparti

Nous décrivons dans cette partie les difficultés inhérentes à la mise au point d'un algorithme réparti. Plusieurs techniques ont été proposées pour faciliter cette tâche ; nous en présentons quelques-unes.

[1] Également connu sous son nom anglais SDL (*Specification and Description Language*).

2.1. *Les difficultés de la mise au point en environnement réparti*

S'il existe des outils relativement performants adaptés à la mise au point de programmes séquentiels sans concurrence, le problème reste plus difficile dans le cas de processus s'exécutant en parallèle (concurrence locale) et devient particulièrement ardu lorsque l'exécution d'un programme est répartie sur plusieurs sites. En effet, les environnements répartis posent des difficultés nouvelles de plusieurs types :

- Les interactions entre différents processus sur différents sites deviennent beaucoup plus complexes et très difficiles à observer ; il n'y a ni état global, ni horloge globale aisés à mettre en œuvre [LAM 78, CHA 85].
- Les variables d'état globales ne sont précisément plus globales, car elles peuvent prendre à certains moments des valeurs différentes selon les sites.
- Le comportement d'ensemble devient partiellement indéterministe et généralement non-reproductible en raison des incertitudes liées à l'ordonnancement des événements sur chaque site, de la charge de chaque machine, ainsi que des délais de communications entre les sites.
- Le seul fait d'observer le déroulement d'un algorithme risque de perturber celui-ci ; c'est l'effet de sonde.
- Il n'est pas possible de piloter l'exécution globale : par exemple, les délais de communication compliquent l'acheminement d'ordres synchronisés, ce qui rend difficile l'insertion de points d'arrêt ou de points de reprise [AGA 93].

2.2. *Grandes méthodes de mise au point et répartition*

On peut discerner trois grandes méthodes de mise au point [VAL 90, LEU 91, MOU 90] :

2.2.1. *L'analyse statique*

L'analyse statique consiste en un examen détaillé, mais sans exécution, du code source. En dehors des vérifications syntaxiques plus approfondies que celles effectuées lors d'une compilation classique, les outils de mise au point statique dans un environnement réparti recourent généralement à la construction de différents types d'arbres ou de graphes (graphes d'états, de flot de données, d'événements, de communications, de dépendances transactionnelles, de causalité, de concurrence, réseaux de Petri, etc.). De nombreuses solutions ont été proposées, qui permettent de détecter les cas de concurrence d'accès, d'interblocage, de famine ou d'attente infinie [LEU 91, MCD 89]. Dans l'idéal, cette forme de mise au point est l'approche la plus satisfaisante, puisqu'elle ne dépend pas d'une exécution particulière et permet de certifier tous les comportements possibles d'un programme. Ces outils se heurtent cependant au problème de l'explosion combinatoire : les arbres ou graphes d'exécution deviennent vite trop complexes pour être exploités. Cependant, certaines techniques d'exécution symbolique et

d'interprétation abstraite permettent de réduire la dimension du problème. Nous verrons un exemple d'analyse statique basé sur la méthode LDS dans la partie 5.

2.2.2. *L'analyse dynamique*

L'analyse dynamique consiste à instrumenter le programme ou son environnement d'exécution de façon à observer et, éventuellement, à piloter son déroulement : insertion de points d'arrêt, exécution pas à pas, etc. En environnement réparti, on peut associer un débogueur séquentiel traditionnel à chaque site. Toutefois, l'absence d'état global limite singulièrement les possibilités de ces solutions : par exemple, dans un système où les communications sont effectuées de façon asynchrone, il est très délicat de définir un point d'arrêt global à plusieurs sites. En outre, toutes ces solutions entraînent une perturbation importante de l'exécution, très gênante dans le cas d'algorithmes non-déterministes. Elles permettent toutefois la détection d'erreurs non-liées au temps [APP 88, MCD 89].

2.2.3. *L'analyse post-mortem*

L'analyse post-mortem consiste à examiner le contenu de la mémoire et les traces d'exécution après une interruption, éventuellement imprévue, d'un programme. Le principal avantage est que l'on ne perturbe pas l'exécution du programme, à condition bien sûr qu'aucune trace utilisée par un outil de déverminage ne soit ajoutée à l'application. En effet, certains outils insèrent de manière transparente des traces qu'ils exploitent ultérieurement[BER 97]. La réexécution [JAM 91, JAM 93, PLA 94, NER 97] est un cas particulier : il s'agit d'enregistrer les paramètres importants, en particulier non-déterministes, d'une exécution, afin de pouvoir la *rejouer* et de détecter la cause d'une erreur. Une méthode complémentaire, utilisée par exemple dans *Bugnet* [JON 87, WIT 88], consiste à enregistrer périodiquement des points de reprise, c'est-à-dire suffisamment de renseignements sur l'état d'un système pour pouvoir reprendre l'exécution à un moment assez proche. Ces méthodes ne résolvent pas complètement le problème de la mise au point en environnement réparti, car l'enregistrement de données d'exécution induit un effet de sonde qu'il faut chercher à limiter au maximum. En outre, elles posent des problèmes de cohérence, de causalité et de globalité. Une taxinomie des outils de réexécution ainsi que des algorithmes utilisés peut être consultée dans [DIO 96].

Il n'est pas rare de trouver des outils combinant analyse dynamique et réexécution. C'est notamment le cas du débogueur CDB, que nous étudierons dans la partie 4.

3. L'algorithme réparti par diffusion de Kai Li et Paul Hudak

Nous nous contenterons ici de résumer le principe de l'algorithme réparti par diffusion de Kai Li et Paul Hudak [LI 89]. Une description en français peut en être trouvée dans [RAY 92].

3.1. *Principes de l'algorithme*

L'objectif de cet algorithme est de permettre à des processeurs faiblement couplés de partager des données par un système de *mémoire partagée répartie*, en s'assurant que cette mémoire est *cohérente*. Pour Li et Hudak, une mémoire est cohérente si une opération de lecture à une adresse retourne toujours la dernière valeur écrite à cette adresse, quels que soient les processeurs effectuant ces opérations de lecture et d'écriture.

Cet algorithme met en jeu un ensemble de processus qui s'exécutent sur des sites distincts, qui ne peuvent pas avoir de défaillances. Ces sites sont faiblements couplés, en ce sens qu'ils ne peuvent s'échanger des informations que sous forme de messages transmis par un canal de communication asynchrone.

Aucune hypothèse n'est faite dans l'article de Li et Hudak sur le canal de communication, mais il est évident, au vu des algorithmes, que celui-ci doit être fiable, c'est-à-dire que les messages ne sont ni perdus, ni altérés. En revanche, Li et Hudak ne précisent pas si ce canal doit être FIFO (*First In, First Out*) ; nous n'avons pas rencontré de cas où cette contrainte se révélerait nécessaire.

Cet algorithme s'appuie sur deux types de communication. Un message envoyé *en diffusion* est reçu par tous les sites impliqués dans l'algorithme. Un message envoyé en *point à point* a un émetteur unique et un destinataire unique.

Un ensemble de processus se partagent des pages de mémoire. Les accès aux pages peuvent se faire en lecture ou en écriture ; à tout moment, sur une page donnée, il peut y avoir plusieurs lecteurs ou exclusivement un seul rédacteur. Un défaut de page est donc effectué soit en lecture, soit en écriture.

Pour chaque page, à un moment quelconque, il existe un site propriétaire qui assure la gestion des accès à la page et sert les demandes des autres sites. Le propriétaire est défini comme le dernier site ayant effectué une écriture sur ladite page ; il n'y a donc pas de propriétaire fixe.

Chaque site possède une table des pages notée *Ptable* renseignant, pour chaque page, les valeurs suivantes :

- Le type d'accès (*Access*) du site sur la page peut prendre ces valeurs : lecture, écriture ou *nil*. Si l'accès est à *nil*, le site n'a aucun accès sur la page ; dans ce cas, la valeur des autres champs de la *Ptable* pour cette page sur ce site est non-significative.

- Le propriétaire (*Owner*) indique qui est propriétaire de la page (le dernier rédacteur). Il peut s'agir en fait d'un simple booléen permettant de déterminer si le site est ou non propriétaire de la page. Notons que dans [LI 89], l'algorithme omet la mise à jour de ce champ en cas de perte de la propriété de la page.

- Liste des lecteurs (*Copyset*) : c'est la liste des sites possédant une copie en lecture de la page ; lors d'une écriture, cette liste permet au site propriétaire (c'est-à-dire au dernier rédacteur) d'annoncer aux sites lecteurs que leur copie de la page devient invalide (procédure d'invalidation).

- Le verrou : il s'agit d'un sémaphore tel que décrit par Dijkstra assurant la synchronisation des accès aux pages [DIJ 65, TAN 91].

3.2. *Description de l'algorithme*

L'algorithme de Li et Hudak est structuré en cinq éléments. Chacun de ces éléments existe sur tous les sites. Les algorithmes ci-dessous présentent les différents modules de l'algorithme de Li et Hudak, tels qu'ils sont décrits dans [LI 89]. Ils exposent le cas de requêtes sur une page `p`. Ces requêtes sont éventuellement distantes ; elles sont dans ce cas émises par le site `s`. Le site local, où sont exécutés ces algorithmes, est désigné par `ego`. Les *gestionnaires de défaut en lecture et en écriture* gèrent les défauts de page effectués localement. De même, les *serveurs en lecture et en écriture* délivrent les pages demandées par d'autres sites via le réseau. Un *serveur d'invalidation* reçoit et applique les requêtes d'invalidation de pages.

```
01 Gestionnaire de défauts en lecture
02      Verrouiller ( PTable[p].verrou )
03      diffuser une demande en lecture pour p
04      attendre la réception de p
05      PTable[p].accès := lecture
06      Déverrouiller ( PTable[p].verrou )

10 Serveur en lecture
11      Verrouiller ( PTable[p].verrou )
12      Si je suis le propriétaire de p
13      Alors
14          PTable[p].copyset := PTable[p].copyset ⋃ {s}
15          PTable[p].accès := lecture
16          Envoyer p au site s
17      FSi
18      Déverrouiller ( PTable[p].verrou )

20 Gestionnaire de défauts en écriture
21      Verrouiller ( PTable[p].verrou )
22      diffuser une demande en écriture pour p
23      attendre la réception de p et de son copyset
24      Invalider ( p, PTable[p].copyset )
25      PTable[p].accès := écriture
26      PTable[p].copyset := ∅
27      PTable[p].propriétaire := ego
```

```
28      Déverrouiller ( PTable[p].verrou )

30 Serveur en écriture
31      Verrouiller ( PTable[p].verrou )
32      Si je suis le propriétaire de p
33      Alors
34          Envoyer p et PTable[p].copyset au site s
35          PTable[p].accès:= nil
36          PTable[p].propriétaire:= s
37      FSi
38      Déverrouiller ( PTable[p].verrou )

40 Fonction d'invalidation
41      Invalider (p, copyset)
42          Pour i dans copyset Faire
43              envoyer une requête d'invalidation de p au site i
44          FPour

50 Serveur d'invalidation
51      PTable[p].accès:= nil
```

Le schéma de la figure 1 décrit le déroulement de l'algorithme lors d'une demande de page en écriture. Ici, le site A demande une page appartenant au site C.

3.3. *Anomalie dans le fonctionnement de l'algorithme de Li et Hudak*

Notre première mise en œuvre de l'algorithme de Li et Hudak présentait un comportement anormal dans certains cas. Une étude de l'algorithme a permis d'y détecter une anomalie, que nous décrivons ci-après. Notons que le formalisme choisi par Li et Hudak pour spécifier leurs algorithmes dans [LI 89] introduit un certain nombre d'ambiguïtés, qui conduisent à plusieurs anomalies, dont certaines ont été relevées par ailleurs. Nous en présentons dans le paragraphe 5.5.2 une autre, qui n'était pas référencée dans la littérature, à notre connaissance. Cet article n'utilisant l'algorithme de Li et Hudak qu'à titre d'exemple, nous ne nous étendrons pas sur l'ensemble des anomalies.

3.3.1. *Description de l'anomalie*

Tel qu'il est sommairement décrit par Li et Hudak dans [LI 89], cet algorithme présente une première anomalie lorsque, par exemple, deux sites A et B demandent simultanément en écriture la même page appartenant à un troisième site C (figure 2). Par «simultanément», nous entendons que chacun des sites A et B a eu le temps de détecter un défaut de page localement, de poser

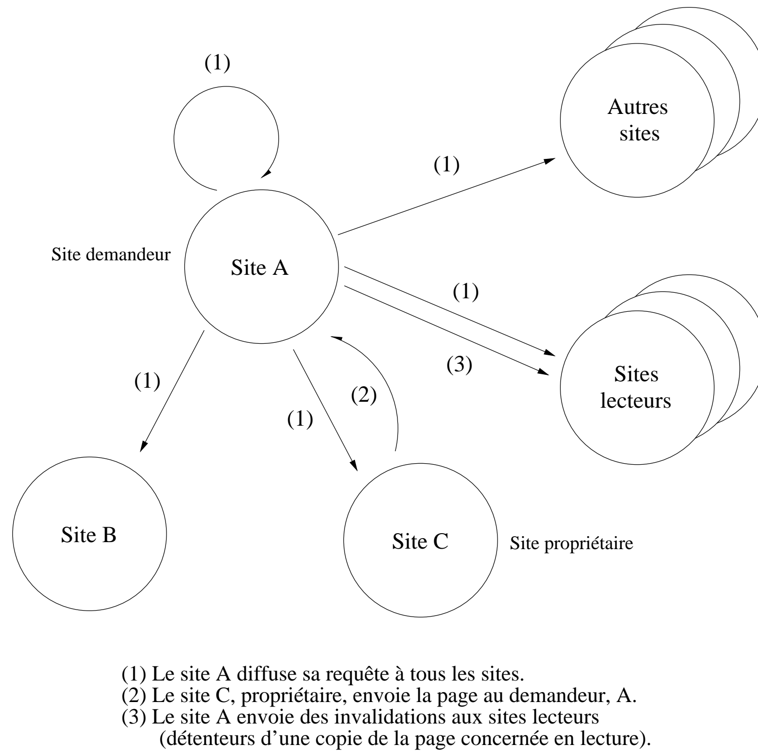
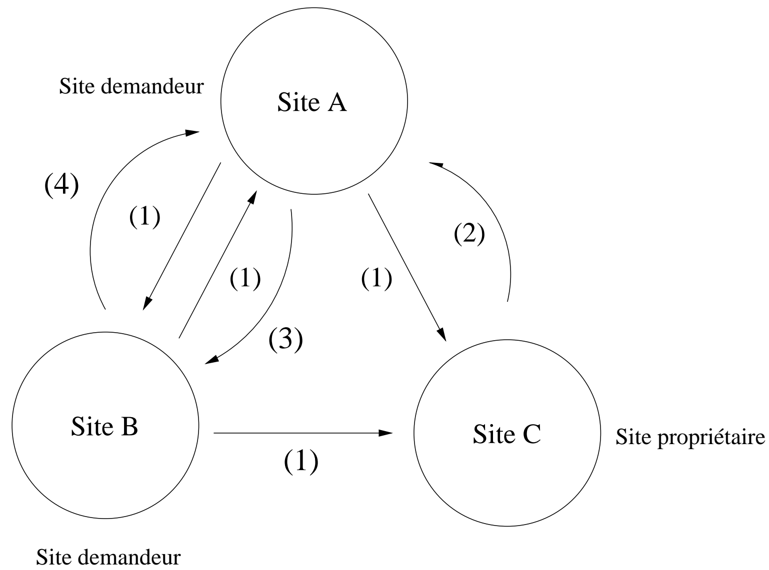


Figure 1. *L'algorithme par diffusion de Li et Hudak (demande en écriture)*

son verrou et de diffuser sa demande avant de recevoir celle de l'autre site. Les sites A et B sont alors bloqués en attente et, sur chacun des deux sites, la requête distante est placée en attente derrière la requête locale. Par exemple, sur le site B, la requête distante du site A est en attente derrière la requête locale de B, et ne sera traitée qu'une fois la requête de B satisfaite.

Supposons maintenant que C reçoive d'abord la demande de A, puis celle de B. Il envoie la page au site A puis, n'étant plus propriétaire, rejette la demande de B. Le site A reçoit la page, effectue l'écriture à l'origine du défaut de page, puis lève son verrou. Le site A traite alors la requête en attente de B et lui envoie la page. Quand B reçoit la page, il lève son verrou et doit alors traiter la requête en attente de A : il envoie la page au site A, *qui n'est en fait plus demandeur*. A ce stade, tout dépend du comportement de A dans cette situation. Li et Hudak n'ayant pas prévu dans leur article le comportement du serveur de page dans ce cas précis (réception d'une page non-demandée), nous considérons que la page se perd, car elle n'a plus de propriétaire. Cette anomalie est mentionnée dans [COU 94].

Un problème similaire se pose dans le cas des invalidations. En effet, en appliquant le même scénario que pour les défauts en écriture, il est aisé d'exhiber



(1) Les sites A et B diffusent leur requête.
(2) Le site C, propriétaire de la page, envoie la page au site A.
(3) Le site A envoie la page au site B.
(4) Le site B lève son verrou, traite la requête en attente de A
et envoie la page au site A, qui n'est plus demandeur.

Figure 2. *Le fonctionnement défectueux*

un cas où un site traite une invalidation avec retard, car la page est verrouillée. Nous invitons le lecteur à vérifier ce cas dans l'exemple décrit ci-dessus.

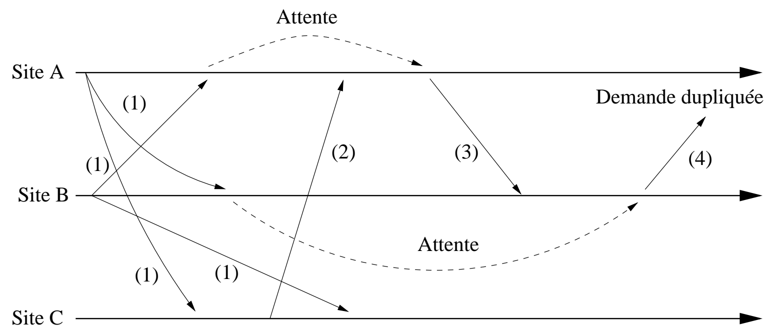


Figure 3. *L'anomalie de fonctionnement de l'algorithme de Li et Hudak*

Figure 4. *Correction de l'anomalie par horloges vectorielles*

3.3.2. *Une solution de l'anomalie*

La solution retenue s'inspire de l'algorithme connu des horloges vectorielles de Fidge et Mattern [FID 88, MAT 89] permettant d'ordonner causalement des événements, un événement étant une émission ou une réception de message, ou un action interne à un site. Une description détaillée de cet algorithme peut être trouvée dans [RAY 95]. Rappelons qu'une horloge vectorielle est un vecteur d'entiers $HV[1 \ldots N]$ où N est le nombre de sites dans le système. Chaque site i maintient une horloge vectorielle, que nous noterons HV_i. Dans la version classique des horloges vectorielles, lorsqu'un site i émet un message, il incrémente la composante de son horloge vectorielle correspondant à son rang, c'est-à-dire qu'il effectue l'opération :

$$HV_i[i] = HV_i[i] + 1$$

L'horloge vectorielle du site émetteur est jointe au message. Le site j recevant un message du site i incrémente l'élément correspondant à son rang :

$$HV_j[j] = HV_j[j] + 1$$

puis compare les éléments de l'horloge vectorielle reçue (notée req_i) à leurs éléments homologues de l'horloge vectorielle locale HV_j, et affecte le maximum obtenu pour chaque rang à l'élément correspondant de son vecteur local :

$$\forall k \leq N, HV_j[k] = max(req_i[k], HV_j[k])$$

L'horloge vectorielle d'un site i résume donc la connaissance que possède i de l'évolution du temps logique sur l'ensemble des sites. La comparaison des horloges vectorielles est effectuée, dans le cas des horloges de Fidge et Mattern, élément par élément. Elle permet ainsi de caractériser ou de vérifier les relations de dépendances causales entre des événements répartis [BAL 91].

Nous nous contentons ici d'une version simplifiée de ces horloges vectorielles. Une horloge vectorielle est attachée à chaque page sur chaque site, même si la

page n'est pas présente sur ce site. Comme les problèmes de datation ne se posent que lors des transferts de page et des invalidations, les seuls événements du système pour lesquels nous devons incrémenter l'horloge vectorielle sont la réception d'une page en écriture et la réception d'une invalidation; dans ces deux cas, le site destinataire reçoit également l'horloge vectorielle liée à la page concernée sur le site émetteur. Ce marquage permet de détecter les demandes trop anciennes, et donc de les écarter. Ici, le site j détecte qu'une demande du site i est trop ancienne si :

$$HV_j[i] > req_i[i]$$

Dans ce cas, la demande est périmée et doit donc être rejetée. Notons que nous ne comparons qu'un seul élément de chaque vecteur, et non pas tous les éléments. *En effet, il suffit d'obtenir un ordre total des requêtes sur une page pour un site donné, l'ordre causal déduit de la comparaison de tous les éléments d'horloges de Mattern ne nous étant pas utile dans notre cas.*

Les figures 3 et 4 comparent, dans l'exemple décrit précédemment, le fonctionnement défectueux et celui de la version corrigée par horloges vectorielles. Dans la version corrigée (figure 4), les demandes de page des sites A et B sont datées par le vecteur (0,0,0). Le site C, propriétaire de la page, la fournit au site A, dont la demande est arrivée en premier. Le site A devient donc propriétaire de cette page. En même temps que la page migre vers A, on transmet aussi le vecteur d'horloge associé, à savoir (0,0,0). A la réception de la page, A calcule le nouveau vecteur en suivant l'algorithme des horloges vectorielles, ce qui donne, dans notre cas, (1,0,0). Quand la page migre de A vers B, le même principe est appliqué : A transmet le vecteur (1,0,0), puis B y applique l'algorithme des horloges vectorielles et obtient (1,1,0). Le site B traite alors la requête (obsolète) du site A ; B commence par comparer le vecteur de la requête émise par A (0,0,0) à celui obtenu lors de la réception de la page (1,1,0). B constate que la demande est trop ancienne. L'ordre total apporté par les horloges vectorielles permet à B de savoir que la requête du site A est périmée et doit donc être rejetée.

Une telle solution est praticable d'un point de vue théorique, mais son coût dans une mise en œuvre réelle est assez élevé. En effet, chaque vecteur est composé de n éléments, n étant le nombre de sites, donc la taille des messages est en $O(n)$. De surcroît, la place mémoire nécessaire au stockage des vecteurs d'horloge pour toutes les pages, et ce sur chaque site peut être importante, de l'ordre de $O(n * m)$, avec n le nombre de sites et m le nombre de pages. Dans certains cas, ce coût peut être prohibitif.

Une autre solution est à la vérité envisageable : tout site recevant une page en écriture en devient propriétaire, même s'il ne l'a pas demandée. Cette solution permet d'éviter la perte de page, mais elle ne règle pas l'anomalie, mentionnée à la fin du paragraphe 3.3.1, des invalidations mises en attente et arrivant à un moment inopportun, alors que notre solution des horloges vectorielles permet de résoudre ce problème.

Cette solution par horloges vectorielles a permis de faire fonctionner correctement un premier prototype. Il n'était toutefois pas facile de s'assurer que le comportement de l'application était strictement conforme aux objectifs. La suite de cet article s'intéresse à deux approches pour vérifier la présence effective de l'anomalie dans l'algorithme, et l'adéquation de la solution proposée.

4. Expériences d'un outil d'analyse dynamique

Nous étudions ici l'intérêt d'un outil d'analyse dynamique, basé sur la réexécution d'une application, afin de vérifier le fonctionnement d'un algorithme distribué. Pour ce faire, nous nous basons sur une expérimentation effectuée à l'aide de l'outil CDB sur CHORUS, qui combine l'utilisation de techniques d'analyse dynamique et post-mortem. Nous cherchons ainsi à vérifier une mise en œuvre de l'algorithme de Li et Hudak.

4.1. *Présentation de CDB*

CDB est un outil destiné à la mise au point d'applications réparties dans l'environnement CHORUS. Le système CHORUS est un système d'exploitation construit autour d'un micro-noyau permettant de prendre en compte certains aspects de la répartition. Pour ce faire, il utilise plusieurs abstractions que nous allons brièvement rappeler avant de décrire dans quelle mesure CDB s'intègre dans CHORUS.

Un système CHORUS [ROZ 91, TAN 95] est un ensemble de sites connectés par un réseau. Sur chaque site, une copie du micro-noyau implante principalement les notions d'acteur, d'activité et de porte. L'acteur est l'unité d'allocation des ressources du système (mémoire, sémaphores, etc.). La deuxième abstraction est l'activité, qui représente l'unité séquentielle d'ordonnancement. Une activité est associée à un acteur unique. Si un acteur possède plusieurs activités, celles-ci s'exécutent en parallèle et en partageant complètement les ressources de l'acteur. Enfin, les acteurs utilisent des portes pour communiquer entre eux par le biais d'IPC (*Inter-Process Communication*), qui permet la communication par messages asynchrones ou par appels de procédure distante.

CDB permet la mise au point des acteurs et des activités d'applications réparties sur plusieurs sites. La version actuelle, décrite dans [RUG 95], offre un certain nombre de services essentiels pour la mise au point d'applications à base de communications asynchrones ou d'appels de procédure distante : elle autorise la définition de points d'arrêt, la lecture et l'écriture dans l'espace d'adressage d'acteurs, et, surtout, offre la possibilité d'observer une application grâce à son service de réexécution. En d'autres termes, CDB permet d'enregistrer une exécution, puis de la rejouer autant de fois qu'on le souhaite, en suivant précisément les communications et le comportement des activités.

Les techniques utilisées pour réaliser cette réexécution sont essentiellement

de deux types. Tout d'abord, CDB effectue une interception des appels système du micro-noyau CHORUS. La méthode utilisée pour réaliser ce service d'interposition est décrite dans [RUG 94]. Enfin, un service d'observation avertit CDB de l'occurrence d'événements système [RUG 95, HER 94]. Ces techniques permettent d'enregistrer dans un journal, puis de rejouer très précisément l'ordonnancement des activités ainsi que tous les appels système apportant de l'indéterminisme lors de l'exécution. A la réexécution, CDB exécute les instructions déterministes, mais simule grâce au journal d'enregistrement les événements indéterministes. Nous avons ici simplifié le fonctionnement de la réexécution, qui est en fait un peu plus complexe, comme décrit dans [RUG 95].

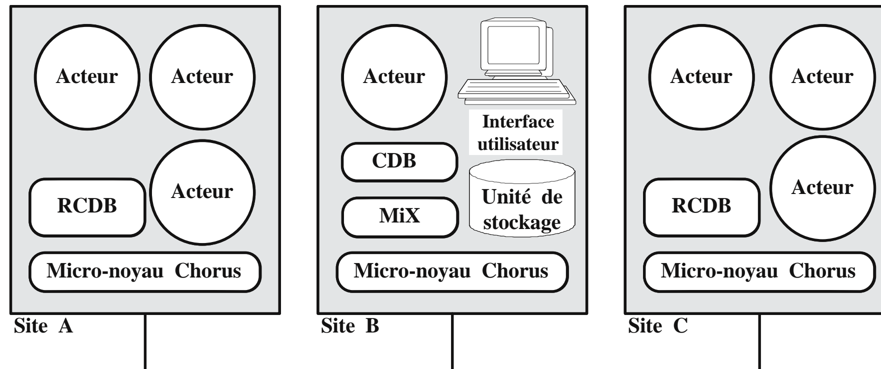


Figure 5. *Architecture de CDB*

CDB peut être découpé en quatre composantes (voir figure 5) :

- Un moniteur réparti, RCDB (*Remote Communication Debugger*) : il s'agit d'un acteur superviseur, qui capte tous les événements indéterministes du site sur lequel il s'exécute ; on trouve un RCDB par site observé.

- Le noyau CDB qui pilote les RCDB et interprète les commandes de l'interface utilisateur. Il ne peut y avoir qu'un seul noyau CDB dans un système CHORUS.

- L'interface utilisateur qui permet d'interagir sur les objets CDB ; il ne peut y avoir qu'une seule interface dans un système CHORUS.

- Enfin, des unités de stockage permettent d'enregistrer les journaux.

4.2. *Expérimentation de l'algorithme de Li et Hudak*

Nous avons vu que l'algorithme de Li et Hudak comporte une anomalie. Nous décrivons dans cette partie une mise en œuvre sous CHORUS de cet algorithme, et comment l'utilisation de CDB nous a permis de reproduire l'anomalie et de vérifier notre solution.

4.2.1. *Description de notre mise en œuvre*

Notre expérimentation sous Chorus simule les différents sites serveurs de pages par des acteurs Chorus. Chaque acteur possède à tout moment au moins deux activités. La première sert les pages qui sont demandées par les autres sites (nous l'appellerons «serveur de pages») et traite également les invalidations ; la seconde génère les demandes de pages vers les autres sites (nous l'appellerons «gestionnaire de page»). Remarquons que s'il y a toujours un seul gestionnaire de pages par acteur, plusieurs serveurs de pages peuvent cohabiter dans un même acteur. En fait, à chaque réception d'une page, d'une demande de page ou d'une invalidation, on crée une activité dont la durée de vie correspond au temps nécessaire pour le traitement de la requête. Enfin, signalons l'existence d'un acteur un peu particulier : le serveur de tests qui est chargé d'envoyer à chaque site une notification des demandes qu'il doit faire. Cet acteur, qui nous permet de conduire le déroulement des jeux d'essais, n'entre pas en ligne de compte dans le fonctionnement de l'algorithme de Li et Hudak.

La figure 6 illustre le fonctionnement de notre expérimentation. Dans cet exemple, le serveur de tests diffuse à tous les sites une notification de défaut de page (flèches 1), qui devra dans ce cas être effectué par le site 1. Ce dernier diffuse ensuite la requête aux autres sites (flèches 2), conformément à l'algorithme de Li et Hudak.

4.2.2. *Comment CDB permet de mettre au point un protocole réparti*

Cette étude de l'algorithme de Li et Hudak sur Chorus nous a amenés à évaluer CDB, et à déterminer les intérêts de ce type d'outil pour l'observation et la mise au point. Aussi, une fois le dernier prototype écrit, nous avons étudié son comportement avec cet outil. A ce stade, les prototypes successifs que nous avions réalisés nous avaient permis d'arriver à une solution relativement stable. Aussi, dans notre cas, CDB nous a surtout aidés à mettre clairement en évidence l'anomalie signalée de l'algorithme de Li et Hudak, et à vérifier que la solution des horloges vectorielles éliminait réellement le problème.

Après une rapide prise en main de CDB, nous pouvons l'utiliser pour vérifier notre mise en œuvre de l'algorithme de Li et Hudak.

La première étape pour vérifier notre mise en œuvre consiste à enregistrer une exécution de celle-ci. Dans notre cas, il a été facile d'enregistrer une exécution fautive, car nous connaissions déjà bien les conditions d'apparition de l'anomalie. On pourra remarquer que les outils de réexécution ne suffisent pas toujours à simplifier la tâche de l'utilisateur à ce niveau : il faut produire au moins une fois l'anomalie avant de pouvoir la réexécuter pour l'étudier, ce qui peut poser des difficultés dans le cas d'événements rares. Notons également que, dans notre cas, nous n'avons pas détecté d'effet de sonde.

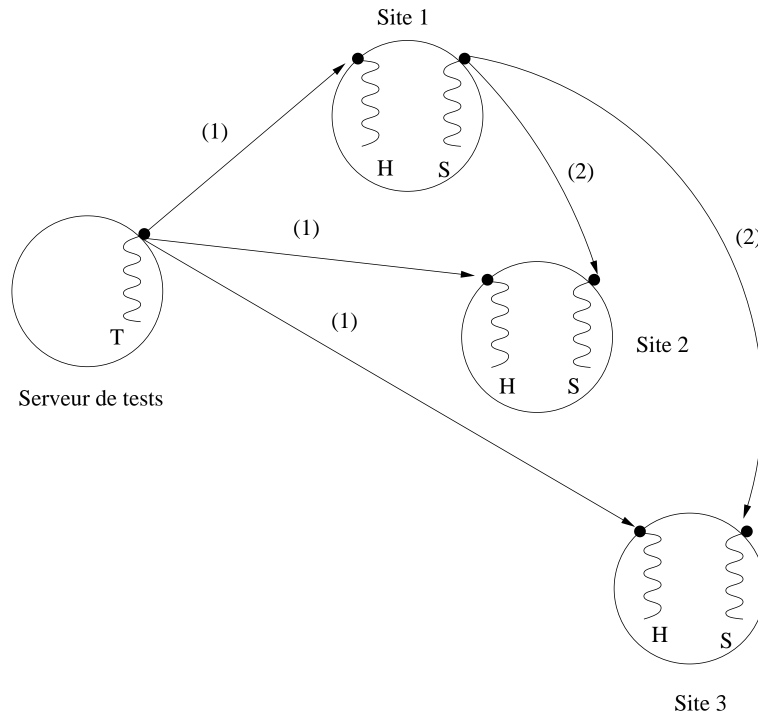


Figure 6. *Architecture de notre expérimentation de l'algorithme de Li et Hudak. Les sites utilisateurs déroulent au moins deux activités : un serveur de pages (S), un gestionnaire ou handler (H). Un acteur particulier pilote le déroulement des tests : le serveur de tests, qui ne comprend qu'une seule activité (T).*

Une fois la session d'enregistrement finie, il est possible soit de consulter les journaux (voir figure 7), soit de lancer une réexécution pas à pas [2]. L'utilisation simultanée des deux fonctions est aussi très intéressante : cette technique permet de visualiser précisément les appels système effectués ainsi que leurs résultats. Malheureusement, CDB ne permet pas la mise au point au niveau du source, ce qui oblige l'utilisateur à manipuler les tables de symboles obtenues par compilation.

Il existe des outils qui permettent de paramétrer ou d'enrichir la réexécution. Certains d'entre eux ne nous ont pas semblé utiles *dans notre cas*. Ainsi, il est possible de poser des points d'arrêt au cours de la session de réexécution ; nous n'avons pas utilisé cette technique, car il nous a été assez facile d'évaluer le

[2] Un pas étant un changement de contexte. On peut réexécuter l'application, soit entièrement, soit pas à pas.

[8]	8	49	> *th-run*:
[28]	30	73	> *pt-created*: cap 50130053 13f 320 320 li 41 ui 50120053 1e inkey 75
[14]	44	49	> *th-beUnReady*: pc 8049ca7 bc 42d status 10
[10]	54	64	> *th-preempted*: pc 804a639 bc 3ab
[34]	88	6d	> *sc-ipcReceive*: diag:8 rcvPort:-1 from outside [5] Body:8 Annex:0
[c]	94	64	> *sc-ipcSend*: diag:0
[8]	9C	7c	> *pt-deleted*:
[1c]	B8	7f	> *sc-ipcReceive*: diag:-4
[8]	C0	d3	> *ac-deleted*:

Figure 7. *Un exemple de journal produit par CDB. La première colonne donne la taille de l'enregistrement de l'événement dans le journal, la deuxième colonne le déplacement dans le fichier journal et la troisième colonne l'identifiant CDB de l'activité. Viennent ensuite l'événement et différents identifiants qui dépendent de l'événement enregistré (identifiant d'acteurs, compteur ordinal, code retour de l'appel système, taille de message, etc.)*

nombre de pas nécessaires pour s'arrêter un peu avant les passages intéressants. Il existe aussi la possibilité de lancer simultanément plusieurs réexécutions identiques et d'observer chacune d'entre elles à des stades différents. Là aussi, nous connaissions suffisamment bien notre prototype pour ne pas avoir besoin de cette fonction.

La consultation des acteurs, activités et messages fut en revanche pour nous d'une grande utilité : la création d'une activité à chaque traitement d'une requête fournit une évaluation du nombre de requêtes qui se traitaient «en parallèle». Enfin, la consultation des messages nous a permis de vérifier quand ils étaient envoyés, et surtout par qui. En conclusion, il est certain que si nous avions disposé d'un tel outil pour les prototypes intermédiaires, le temps de mise au point de ceux-ci aurait été réduit dans une importante proportion.

4.3. *Utilisation d'outils de réexécution sur des applications réparties*

Nous présentons ici les fonctionnalités qui nous ont parue nécessaires à un outil de mise au point répartie. En premier lieu, le développeur doit pouvoir employer une méthode «cyclique», c'est-à-dire une méthode qui lui permette de répéter l'exécution d'un programme jusqu'à ce qu'il en comprenne bien le comportement. En effet, si répéter une exécution en univers centralisé est facile, c'est beaucoup plus complexe en environnement réparti. Ceci nécessite, dans notre cas, de pouvoir couvrir la majorité des appels système CHORUS courants.

Un outil de mise au point répartie doit offrir un moyen d'analyse post-

mortem efficace. C'est le rôle des journaux qui se doivent d'être peu volumineux et de contenir les informations les plus utiles pour suivre les appels système que peut faire une application.

Il faut qu'un outil de mise au point soit simple d'emploi. En effet, nombreux sont les outils puissants qui ne sont guère ou sont mal utilisés, car ils demandent au départ un investissement en temps trop important de la part de l'utilisateur. Il serait souhaitable que le développeur puisse manipuler les mêmes abstractions que celles utilisées au cours de la réalisation de l'application. Par exemple, le programmeur souhaite retrouver la notion d'objet si celle-ci est à la base de la conception de son application.

Dans l'idéal, un tel outil devrait fournir une vision globale et concise du système, même si ce dernier est hétérogène. Notamment, une application complexe doit rester aisée à examiner.

Toutefois, le point essentiel de tout outil de réexécution est de réduire le plus possible l'effet de sonde. Autrement dit, l'instrumentation doit modifier aussi peu que possible le comportement de l'application.

L'outil CDB satisfait à la grande majorité de ces contraintes. En effet, il permet d'effectuer une mise au point cyclique par le biais d'une interface simple, en consommant peu de ressources ; notamment, les journaux restent de taille raisonnable, et l'espace mémoire consommé par l'outil est peu élevé. Par ailleurs, nous n'avons pas relevé d'effet de sonde préjudiciable. Les résultats obtenus par l'outil, dont les principaux sont les journaux, sont concis mais suffisamment complets pour en permettre une bonne exploitation.

Toutefois, l'usage de CDB ne se fait pas sans difficulté dans certains cas. Notons, par exemple, l'absence de mise au point au niveau source, qui prive l'utilisateur des abstractions utilisées pendant l'écriture du programme. Plus généralement, CDB peut rapidement devenir fastidieux à utiliser dès que nous cherchons à avoir une vision globale du système. Finalement, nous regrettons également l'impossibilité d'utiliser cet outil sur des acteurs MiX, ce qui interdit son emploi pour le déverminage de toutes les applications avec des accès fichiers.

Ces contraintes n'empêchent pas à CDB d'être un outil puissant et efficace et de constituer un complément de choix aux autres outils de mise au point disponibles sous CHORUS, tels que KDB (*Kernel Debugger*), l'outil de mise au point du micro-noyau, et GDB, le débogueur de GNU. Il serait d'ailleurs intéressant de réaliser une bonne intégration de CDB avec un débogueur traditionnel comme GDB. En conclusion, nous pouvons dire que CDB est un outil d'observation très bien adapté aux applications pour lesquelles il a été conçu, à savoir des applications à base de communications asynchrones ou d'appels de procédure distante (RPC).

4.4. *Conclusion sur l'utilisation d'outils de réexécution*

Nous avons pu constater que la réexécution est très adaptée à la mise au point d'applications réparties. Toutefois, si ce type d'outils peut aider à la dé-

tection d'anomalies de fonctionnement et permet leur résolution, il ne garantit pas la validité d'une application. En effet, l'outil valide des exécutions particulières, mais n'assure pas que l'application ne sera pas fautive pour d'autres exécutions non examinées.

5. Expériences d'une méthode d'analyse statique

Nous avons, jusqu'ici, décrit des techniques de vérification *dynamiques* permettant de mettre au point des algorithmes distribués. Dans la présente partie, nous nous concentrons sur un outil de vérification *statique*. Plus précisément, nous cherchons à utiliser une technique de description formelle, telle que définie par Quemada *et al.* dans [QUE 91], afin de valider un algorithme distribué. Nous nous appuyons sur l'exemple de l'algorithme de Li et Hudak, déjà présenté dans cet article, pour vérifier nos concepts, aussi bien pour la détection des anomalies que pour la validation de solutions.

5.1. *Présentation de l'outil*

Nous nous intéressons ici à LDS, normalisé par l'UIT (*Union Internationale des Télécommunications*) dans [ITU 94].

Cette méthode permet de décrire, sous forme graphique, un système de processus communicants, représentés par des machines à états finies étendues s'échangeant des signaux. LDS a été conçu pour être utilisé dans le domaine des télécommunications; toutefois, ses abstractions sont bien adaptées à la modélisation d'algorithmes répartis.

La mise en œuvre de LDS utilisée est SDT 3.02 (*SDL Design Tool*), de la société Telelogic [TEL 95], qui permet d'écrire, de simuler et de valider un modèle LDS. Cet outil permet également de manipuler des MSC (*Message Sequence Charts*), qui sont des traces graphiques d'exécution d'un système de processus communicants, également normalisées par l'UIT dans [ITU 92]. Ce type de trace est parfaitement adapté à la description de l'exécution d'un modèle LDS.

SDT peut être considéré comme un outil de déverminage statique, pour reprendre la classification du paragraphe 2.2.1. En effet, nous ne nous intéressons pas ici à une exécution d'une mise en œuvre de l'algorithme, mais au comportement de ce dernier, déduit de l'arbre de comportement (*Behaviour Tree*) de ses exécutions possibles. Nous avons appliqué cette méthode à l'algorithme de Li et Hudak, ce qui nous a permis de détecter des anomalies et d'en valider des solutions.

5.2. *Modélisation de l'algorithme*

Modéliser un algorithme distribué relativement simple, tel que celui de Li et Hudak, est une tâche assez aisée pour qui maîtrise un tant soit peu LDS et SDT. Notons que la modélisation d'algorithmes plus complexes est traitée dans [BLO 97]. Nous ne nous étendrons pas plus sur le problème de la modélisation d'algorithmes distribués en LDS, et nous considérons donc que, dans la suite de cet article, nous disposons d'un modèle LDS de l'algorithme de Li et Hudak.

5.3. *Approche générale*

Un grand nombre d'outils de validation sont basés sur l'exploration de l'espace d'états du modèle. Une contrainte forte est la finitude de celui-ci ; Quemener présente dans [QUE 96] des méthodes pour passer outre cette contrainte. Toutefois, dans le cas de l'algorithme de Li et Hudak, ce problème ne s'est pas posé, car le nombre d'états est borné par le dimensionnement du modèle, en l'occurrence le nombre de pages et le nombre de sites, ce qui rend fini l'arbre de comportement.

Sans être infini, l'espace d'états du modèle de l'algorithme de Li et Hudak est suffisamment important pour exclure toute exploration manuelle de celui-ci. Nous nous appuyons donc sur les possibilités de l'outil SDT pour réaliser automatiquement ce parcours.

Par ailleurs, la symétrie, inhérente à la conception de l'algorithme de Li et Hudak, permet de généraliser les résultats obtenus pour des valeurs données du nombre de pages et de sites. Les paragraphes suivants proposent simplement le squelette de la preuve de cette affirmation, sa rédaction formelle étant en effet assez longue.

Objectif de la preuve

Nous cherchons à prouver que des résultats, obtenus par une quelconque méthode de validation d'un modèle de l'algorithme de Li et Hudak, ne sont pas dépendants des nombres de pages et de sites définis dans le modèle. Nous approfondissons, dans les paragraphes suivants, la preuve portant sur l'indépendance vis-à-vis du nombre de sites.

Hypothèses

Nous partitionnons $Sites$, l'ensemble des sites, en trois sous-ensembles pour chaque requête i. L'ensemble $Propriétaire_i$ est le singleton contenant le propriétaire de la page demandée. L'ensemble $Demandeur_i$ est le singleton contenant le site émetteur de la requête[3]. L'ensemble $Neutres$ contient tous les autres sites. Nous avons :

[3] Notons qu'il est possible d'avoir $Propriétaire_i \equiv Demandeur_i$.

$$Propriétaire_i \cap Neutres_i \equiv \emptyset$$

$$Demandeur_i \cap Neutres_i \equiv \emptyset$$

$$Propriétaire_i \cup Demandeur_i \cup Neutres_i \equiv Sites$$

$$Card(Propriétaire_i) = 1$$

$$Card(Demandeur_i) = 1$$

Nous disons qu'un site est *impliqué* dans une requête s'il est soit demandeur, soit propriétaire de la page. Tout autre site recevant la requête n'en tient pas compte, en ce sens qu'il ne modifie pas ses données, selon le principe de l'algorithme de Li et Hudak.

Propriété de symétrie des comportements

Pour commencer, nous cherchons à prouver la propriété suivante, dite de symétrie des comportements : *le comportement d'un site donné ne dépend que de l'ensemble auquel appartient ce site.*

Pour ce faire, nous constatons que tous les sites exécutent le même code. Plaçons-nous dans le cas d'une seule requête sur l'ensemble du système. Nous constatons que la distribution des pages dans le système n'a d'influence que sur l'appartenance d'un site à l'un ou l'autre ensemble. Le nombre de sites neutres est également sans effet : le comportement d'un site donné n'est pas dépendant du comportement des sites neutres.

Étendons maintenant cette propriété au cas où plusieurs requêtes sont effectuées simultanément dans le système. Nous devons étudier deux cas :

- Si aucun site n'est impliqué dans plusieurs requêtes, il est aisé de constater que, pour deux requêtes i et j simultanées quelconques ($i \neq j$), nous avons :

$$Propriétaire_i \bigcap Propriétaire_j \equiv \emptyset$$

$$Demandeur_i \bigcap Demandeur_j \equiv \emptyset$$

Il est donc trivial de voir que la propriété de symétrie des comportements est vérifiée dans ce cas.

- S'il existe des sites impliqués dans plusieurs requêtes, nous constatons que plusieurs requêtes concernant la même page sur le même site sont traitées séquentiellement, grâce au verrou. Par ailleurs, plusieurs requêtes concernant plusieurs pages distinctes ne partagent pas les mêmes données (elles accèdent des éléments différents de la *PTable*) ; elles peuvent donc être traitées en parallèle sans effet de bord. Par conséquent, nous pouvons intuitivement dire que la propriété de symétrie des comportements est vérifiée dans ce cas.

Utilisons maintenant cette propriété de symétrie des comportements pour prouver qu'il est possible de généraliser des résultats, concernant l'algorithme de Li et Hudak, fournis par un validateur.

Preuve par récurrence

Supposons que l'algorithme a été vérifié pour un nombre n quelconque de sites, $n \geq 3$. Nous cherchons à prouver que, dans ce cas, l'algorithme s'applique également à $n+1$ sites.

Notons x le nouveau site. Trois cas se présentent : x est soit le demandeur, soit le propriétaire, soit neutre. Or, ces trois cas ont été traités pour n sites. En effet, il est évident que toute requête implique un propriétaire et un demandeur. De plus, nous avons posé $n \geq 3$, d'où nous concluons qu'un site au moins est neutre à chaque requête pour n sites. En d'autres termes, tous les cas possibles ont été prouvés pour n sites. L'ajout d'un site supplémentaire ne met pas en cause le bon fonctionnement de l'algorithme, grâce à la propriété de symétrie des comportements.

Un problème voisin se pose avec le nombre de pages, qui est fixé dans le modèle. Une preuve très similaire à celle portant sur les sites peut être construite, afin de vérifier que notre validation n'est pas dépendante du nombre de pages. Le lecteur intéressé peut rédiger cette preuve en s'appuyant sur le squelette de celle portant sur les sites.

Conclusion

Une fois assurés du bon fonctionnement de l'algorithme de Li et Hudak pour un nombre quelconque n de sites, $n \geq 3$, nous sommes également sûrs de ce bon fonctionnement pour des nombres supérieurs de sites et de pages.

SDT va nous permettre de vérifier l'algorithme de Li et Hudak pour un nombre donné de sites. La preuve ci-dessus généralisera ces résultats.

Nous décrivons ici deux approches différentes d'utilisation de SDT, montrant la preuve de l'existence de deux anomalies, ainsi que la résolution de celles-ci, l'une de ces dernières nous étant préalablement connue, au contraire de l'autre.

5.4. *Connaissance préalable d'une anomalie*

5.4.1. *Approche*

Connaître une anomalie nous permet d'aisément exhiber une exécution où celle-ci se produit, condition suffisante pour prouver son existence. Une fois le modèle modifié pour y intégrer une solution, la difficulté est de montrer qu'il n'existe aucune exécution qui présente l'anomalie, condition nécessaire et suffisante à la preuve de ladite solution.

Avec l'outil SDT, il est possible d'exécuter un modèle, en ayant la maîtrise des entrées qui lui sont fournies. Il nous est donc possible d'exhiber un MSC, trace d'une exécution où l'anomalie se produit. Nous prouvons ainsi le besoin d'une solution.

Mettre en œuvre une solution dans le modèle LDS est un pur problème de

modélisation, sur lequel nous ne nous étendrons pas. Prouver que cette solution résout notre problème peut s'effectuer en deux étapes.

La première étape consiste à utiliser le MSC, que nous avons construit auparavant, et qui présente l'anomalie qui nous concerne. Nous pouvons demander au validateur SDT de rechercher dans l'arbre de comportement une exécution correspondant au MSC, celui-ci décrivant une branche de cet arbre depuis la racine. S'il ne la trouve pas, nous prouvons simplement que nous avons supprimé cette occurrence de l'anomalie, mais nous n'obtenons aucun résultat général : il serait plus qu'audacieux de prétendre que le problème ne se présente pas dans une autre exécution. Ceci vient d'une limitation de la recherche d'un MSC dans l'arbre de comportement : le MSC à valider doit décrire une exécution depuis la racine de l'arbre pour être vérifié. Imaginons une exécution qui amène le système dans un état à partir duquel l'anomalie se produit. Il nous faudrait connaître cet état et l'exécution partant de la racine pour y parvenir, afin de construire un MSC permettant de vérifier si l'anomalie a bien été supprimée. Eu égard à la complexité d'un arbre de comportement, nous considérons cette tâche comme impossible. Par conséquent, nous devons aller au-delà d'une simple validation par l'usage de MSC.

La seconde étape vise donc à obtenir un résultat général. Une approche possible consiste à identifier un comportement[4] particulier de l'algorithme en présence de l'anomalie, par exemple la réception de plusieurs réponses à une unique requête. Une étude approfondie du problème, s'appuyant sur des simulations, nous permet d'identifier un tel comportement. Par contre, une connaissance précise de l'algorithme est indispensable pour pouvoir prouver qu'un tel comportement est toujours le symptôme d'une anomalie.

Il est alors possible d'effectuer un parcours de l'arbre de comportement du modèle de l'algorithme muni de notre solution, en demandant au validateur de rechercher ce comportement symptomatique. Toutefois, afin de faciliter le travail du validateur, nous pouvons modifier le modèle LDS afin de lui faire émettre un signal précis si le comportement se présente. En effet, demander au validateur de rechercher une émission de signal consiste à lui demander de vérifier un prédicat sur l'ensemble des exécutions de l'algorithme. Ceci est, naturellement, plus simple que de mettre le validateur en quête de la suite d'événements révélant l'anomalie, puisqu'il faudrait installer une mémoire des comportements antérieurs dans un prédicat à vérifier, ce qui n'est jamais chose facile, alors que placer cette information dans le modèle est aisé. Par contre, il faut prendre garde, lors de la modification du modèle, à ne pas dévier de l'algorithme initial.

Si le validateur ne repère pas une telle émission de signal, nous pouvons affirmer que l'anomalie est résolue par notre solution. Toutefois, nous devons exprimer les réserves d'usage quant à la conformité absolue du modèle à l'algorithme. Par ailleurs, prouver l'exhaustivité du parcours de l'arbre de com-

[4]Un *comportement* peut être vu comme une partie de l'exécution, c'est-à-dire une suite d'états du modèle correspondant à l'algorithme, et les différentes transitions correspondant aux changements entre ces états.

portement, effectué par l'outil de preuve, est impossible à notre niveau ; nous devons nous en remettre aux garanties fournies par le concepteur du produit.

5.4.2. *Exemple : la perte du propriétaire*

Cette anomalie de l'algorithme de Li et Hudak est celle décrite au paragraphe 3.3.1, où une page perd son propriétaire. L'exécution d'une simulation de notre modèle LDS, grâce à l'outil SDT, nous permet de retrouver aisément ce problème.

Une première approche est de déduire une trace MSC d'une exécution du modèle qui conduit à l'anomalie. Le modèle est alors modifié comme il est décrit dans le paragraphe 3.3.2, en y ajoutant des horloges vectorielles. A partir de ce modèle modifié, nous demandons au validateur de construire un arbre de comportement, qui représente une couverture de toutes les exécutions possibles du modèle. L'outil peut alors rechercher dans cet arbre une exécution qui correspond à celle erronée que nous avons trouvée plus haut, et qui est donc décrite par un MSC. Il ne trouve aucune exécution correspondante. Nous pouvons alors affirmer que l'anomalie ne se produit plus, *mais seulement dans le cadre de cette exécution particulière*. En effet, comme il est dit plus haut, un MSC décrit une branche de l'arbre de comportement en partant de la racine ; si l'anomalie se présente dans une autre branche ou à partir d'un état qui n'est pas la racine, nous ne détectons pas cette exécution par cette méthode. Il nous est donc impossible, par cette technique, de généraliser cette certitude à l'ensemble des exécutions possibles.

Prenons une approche différente, pour vérifier exhaustivement notre solution. Nous introduisons un signal `Bug`, émis par un site lorsque celui-ci reçoit une page en écriture qu'il n'attend pas, ce qui est un symptôme de l'incident. En effet, si cette page n'est pas attendue, elle n'est pas acceptée par le site, donc elle est rejetée et perdue. Si nous utilisons la version du modèle qui produit l'anomalie, le validateur nous informe que le signal `Bug` est envoyé dans plusieurs exécutions, ce qui représente autant de cas d'erreur. Une fois le modèle modifié, avec la solution proposée plus haut, le validateur ne détecte plus aucune émission du signal `Bug`. Nous pouvons donc en conclure qu'il n'a trouvé aucune exécution comportant l'anomalie. Or, le validateur a pour objet l'exploration de l'arbre de comportement, qui est censé contenir toutes les exécutions possibles du modèle. La non-détection du signal `Bug` dans la totalité des exécutions prouve que notre proposition de solution résout bien cette erreur. Il est toutefois essentiel de noter qu'une telle conclusion s'appuie sur une confiance absolue dans les résultats fournis par l'outil, notamment qu'il parcourt bien l'intégralité des exécutions possibles. Nous ne pouvons que nous en remettre aux garanties fournies par le concepteur de SDT.

5.5. *Anomalie inconnue*

5.5.1. *Approche*

Il est extrêmement délicat d'identifier une anomalie qui n'a jamais été constatée auparavant. Par conséquent, il est impossible d'affirmer que l'algorithme est dénué d'anomalie par des techniques générales.

Dans les systèmes répartis, et plus généralement dans les algorithmes mettant en jeu de la communication, plusieurs problèmes sont bien connus. Pour l'exemple, citons les cas d'interblocage ou d'absence du vivacité du système. Un outil de preuve prévu pour être appliqué à de tels algorithmes doit être en mesure de détecter ce genre de comportement.

Le validateur SDT repère ces situations lors d'un parcours de l'arbre de comportement. Toutefois, ses diagnostics sont assez généraux ; ainsi, par le terme générique d'*interblocage*, le validateur désigne toute absence de vivacité du modèle, qu'il y ait ou non des requêtes en attente [5]. L'outil nous propose alors une trace complète, sous forme de MSC, qui peut être arpentée, l'utilisateur pouvant examiner les variables, ainsi que les signaux en attente. Il est ainsi possible de se faire une opinion précise sur l'éventuelle anomalie soulignée par le validateur SDT.

5.5.2. *Exemple : Un cas d'interblocage*

Lors de la construction de l'arbre de comportement par le validateur, ce dernier nous a signalé un cas d'interblocage, qui se révèle être une autre anomalie de l'algorithme de Li et Hudak.

L'anomalie se produit dans le cas suivant : supposons qu'un site est propriétaire d'une page dont il existe des copies ; il n'a donc qu'un accès en lecture. Si ce site fait un défaut en écriture sur cette page, il verrouille celle-ci et diffuse sa requête sur le réseau. Il reçoit alors sa propre requête, par le réseau, mais ne peut pas la traiter, car la page est verrouillée. Or, il est le seul à pouvoir répondre à sa propre requête ; il est donc bloqué. Ceci n'est pas dépendant de la mise en œuvre, comme un simple parcours de l'algorithme des gestionnaires de défaut de page (lignes 20 et suivantes, page 6) permet de le constater. En effet, ces modules commencent par un verrouillage de la page dans la *PTable*, et sont suivis d'une diffusion de la requête, sans vérification de l'éventuelle détention locale de la page.

La solution consiste à empêcher un propriétaire d'émettre une requête sur une page qu'il détient. Une fois le modèle modifié, le validateur ne détecte plus l'anomalie, ce qui nous assure de la correction de notre solution (sous réserve, comme indiqué au paragraphe 5.4.2, de l'exhaustivité des exécutions examinées par SDT). Notons que le problème ne s'est pas posé dans le prototype décrit dans le paragraphe 4.2.1, car nous avions apporté la même modification à l'algorithme, dans un but d'efficacité, ce qui a empêché l'erreur de survenir.

---

[5] Généralement, un interblocage peut être défini par des requêtes en attente dans un système non-vivant.

5.6. *Conclusion sur l'utilisation de LDS*

Comme nous le voyons, l'utilisation de la méthode LDS et de sa mise en œuvre SDT nous a permis de valider aisément l'algorithme de Li et Hudak, de détecter des anomalies et de valider les solutions proposées. Nous nous sommes servi de cet outil en cherchant à utiliser une approche générale, afin qu'elle puisse être reproduite.

Même si certaines faiblesses de l'outil rendent celui-ci parfois fastidieux à utiliser, notamment au niveau de la manipulation des MSC, qui doivent être retravaillés manuellement avant d'être traités par le validateur, il est néanmoins très adapté à notre problème.

6. Conclusion

Différentes techniques nous ont permis de vérifier le bon fonctionnement d'un algorithme réparti, celui de Li et Hudak en l'occurrence. D'une part, nous avons utilisé un outil d'analyse dynamique basé sur l'analyse post-mortem et la réexécution et, d'autre part, un outil d'analyse statique, se présentant sous la forme d'une technique de description formelle et du validateur associé.

La réexécution permet d'effectuer la mise au point d'applications réparties. Les techniques utilisées à cet effet offrent une première solution pour permettre aux utilisateurs de bénéficier des avantages de la mise au point cyclique. Toutefois, nous avons vu qu'un outil comme CDB (et il en est de même pour d'autres outils tels que *Recap* [PAN 89] ou *Bugnet* [WIT 88]) est cruellement limité par son incapacité à pouvoir systématiquement détecter des anomalies. Il existe néanmoins des travaux qui tentent d'apporter des solutions à ce problème [KIL 97].

Enfin, s'il existe aujourd'hui bon nombre d'outils de réexécution (voir [DIO 96]), aucun de ceux que nous avons pu étudier n'avait résolu tous les problèmes qui nous semblent essentiels pour l'utilisateur : réexécution de processus quelconques, et à n'importe quel moment, sur des machines ou des systèmes hétérogènes, avec une notion de point d'arrêt et dans des conditions d'ergonomie aussi satisfaisantes que sur les outils centralisés, tout cela avec un effet de sonde aussi faible que possible. De nombreuses études restent donc encore à faire dans ce domaine, et des travaux tels que ceux de Jard et al. [JAR 94] et Garg [GAR 97] permettront très certainement l'amélioration de l'observation et du pilotage des applications réparties durant la phase de mise au point.

Les méthodes de spécification formelle et les outils de preuve qui leur sont associés permettent de s'assurer du bon fonctionnement d'un algorithme dans tous les cas de figure possibles. Il faut toutefois rester prudent sur cette affirmation, qui s'appuie uniquement sur les garanties de fiabilité fournies par le concepteur de l'outil de validation. Par ailleurs, une telle méthode ne permet pas de vérifier l'implantation pratique réalisée. En effet, un modèle n'est pas une implantation : il n'est pas à exclure que d'importantes différences existent

entre le comportement du premier et celui de la seconde, notamment en raison des contraintes de l'environnement de mise en œuvre.

Nos expériences sont basées sur un algorithme de mémoire partagée répartie. Toutefois, nous pouvons constater que les techniques présentées ici sont généralisables à toute application distribuée utilisant des ressources critiques utilisées par un système de communication asynchrone. Il apparaît que les techniques de déverminage dynamique et de validation formelle, loin d'être rivales, sont complémentaires pour le développement de ce type d'application.

Nous tenons à remercier toutes les personnes qui nous ont permis de réaliser ce travail, en particulier Claude Kaiser, professeur au CNAM, qui nous a proposé de travailler avec CDB sous CHORUS*, Eric Gressier, maître de conférence au CNAM, qui nous a aiguillés sur le sujet de la mémoire partagée répartie et des algorithmes de Kai Li et Paul Hudak, nous a prodigué ses conseils et fourni de la documentation, Christian Santellani, doctorant au CEDRIC, qui nous a permis de travailler dans de bonnes conditions ainsi que Christophe Ménival, Thierry Naquin et Cédric Billaud qui ont découvert la première anomalie citée dans ce document, et Sophie Pichaureaux, qui a participé à notre première implantation sur un réseau de stations Unix.*

7. Bibliographie

[AGA 93] A. AGARWAL et K. SALEH. « Efficient and Fault Tolerant Checkpointing and Recovery Procedures for Distributed Systems ». *Réseaux et Informatique Répartie*, (2), 1993.

[APP 88] W. APPELBE et C. MCDOWELL. « Developing Multitasking Programs ». Dans *Proc. of Hawaii International Conf. on System Sciences*, pages 94–102, January 1988.

[BAL 91] R. BALTER, J.-P. BANÂTRE, et S. KRAKOWIAK, éditeurs. « *Principe et mécanismes de base de la répartition* », Chapitre 4. INRIA, Rocquencourt, avril 1991.

[BER 97] F. BERGDOLT. « De la spécification au déverminage : vérification de propriétés dans un univers à objets répartis ». Mémoire d'ingénieur C.N.A.M., Centre de Paris, février 1997.

[BLO 97] Xavier BLONDEL, Eric FARGES, et Lise MASSIMELLI. « Utilisation de LDS pour la Validation de Protocoles OSEK pour Réseaux Automobiles Embarqués ». Dans *RTS'97*, Paris, 1997.

[CHA 85] K. M. CHANDY et L. LAMPORT. « Distributed Snapshots: Determining Global States of Distributed Systems ». *ACM Trans. Comput. Syst.*, (1):63–75, February 1985.

[COU 94] G. COULOURIS, J. DOLLIMORE, et T. KINDBERG. *Distributed Systems—Concepts and Design, 2nd Ed.* Addison-Wesley Publishers Ltd., 1994.

[DIJ 65] E.W. DIJKSTRA. « Co-operating Sequential Processes ». Programming Languages, Genuys F (éd), Londres, 1965.

[DIO 96] C. DIONNE, M. FEELEY, et J. DESBIENS. « A Taxonomy of Distributed Debuggers Based on Execution Replay ». Dans *Proceedings of the International*

Conference on Parallel and distributed Processing Techniques and Applications (PDPTA'96), California, August 1996.

[FID 88] J. Fidge. « Timestamps in message passing systems that preserve the partial ordering ». Dans *Proc. 11th Australian Computer Science Conference*, pages 55–66, February 1988.

[GAR 97] V. K. Garg. « Observation and Control for Debugging Distributed Computations ». Dans *AADEBUG'97. Proceedings of the Third International Workshop on Automatic Debugging*, May 1997.

[HER 94] M. Herdieckerhoff et F. Ruget. « Matching operating systems to application needs – A case study. ». Dans *Proc. of SIGOPS'94*, 1994.

[ITU 92] ITU. « Message Sequence Charts MSC ». ITU Recommendation Z.120, 1992.

[ITU 94] ITU. « Specification and Description Language SDL ». ITU Recommendation Z.100, 1994.

[JAM 91] H. J. Jamrozik, C. Roisin, et M. Santana. « A graphical Debugger for Object-Oriented Distributed Programs ». Dans *Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 117–128, Jully, 1991.

[JAM 93] H. Jamrozik. « *Aide à la Mise au Point des Applications Parallèles et Réparties à base d'Objets Persistants* ». Thèse de doctorat, Université Joseph Fourier, Grenoble, mai 1993.

[JAR 94] C. Jard, T. Jéron, J. Guy-Vincent, et J.-X. Rampon. « A general approach to trace-checking in distributed computing systems ». Rapport Technique 811, IRISA, 1994.

[JON 87] S. H. Jones, R. H. Barkan, et L. D. Wittie. « Bugnet: a real time distributed debugging system ». Dans *Proc. of the 6th Symp. on Reliability in Distributed Software and Database Systems*, pages 56–65, March 1987.

[KIL 97] R. Kilgore et C. Chase. « Re-execution of Distributed Programs to Detect Bugs Hidden by Racing messages ». Dans *Proceedings of the International Conference on System Sciences, Hawaii*, January 1997.

[LAM 78] L. Lamport. « Time, Clocks, and the Ordering of Events in a Distributed System ». *Communications of the ACM*, 21(7):558–565, Jully 1978.

[LEU 91] E. Leu et A. Schiper. « Techniques de déverminage pour programmes parallèles ». *Technique et Science Informatiques*, 10(1):5–21, 1991.

[LI 89] K. Li et P. Hudak. « Memory Coherence in Shared Virtual Memory Systems ». *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[MAT 89] F. Mattern. « Virtual time and global states of distributed systems ». Dans *Proc. of Int. Workshop on Parallel and Distributed Algorithms, Bonas*, pages 215–226. Cosnard, Quinton, Raynal and Robert editors, 1989.

[MCD 89] C. E. McDowell et D. P. Helmbold. « Debugging Concurrent Programs ». *ACM Computing Surveys*, 21(4):593–622, December 1989.

[MOU 90] P. Moukeli. « Les Principales Approches de Conception des Débogueurs pour les Langages Parallèles ». Rapport Technique 90–05, Laboratoire LIP Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, janvier 1990.

[NER 97] N. Neri, L. Pautet, et S. Tardieu. « Debugging distributed applications with replay capabilities ». Dans *ACM TRI-ADA'97*, pages 189–195, November 1997.

[PAN 89] D. Pan et M. Linton. « Supporting Reverse Execution for Parallel Programs ». *ACM SIGPLAN*, 24, January 1989.

[PLA 94] Pascal Placide. « Mise au point d'applications réparties ». Rapport de DEA, Conservatoire National des Arts et Métiers (CEDRIC), 1994.

[QUE 91] J. Quemada, J.Manas, et E.Vasquez, éditeurs. *FORTE/International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, Sydney, Australia,*, 1991.

[QUE 96] Yves-Marie Quemener. « *Vérification de protocoles à espace d'états infinis représentable par une grammaire de graphes* ». Thèse de doctorat, Université de Rennes 1, 1996.

[RAY 92] M. Raynal. « *Gestion des données réparties: problèmes et protocoles* », Chapitre 2, pages 6–26. Eyrolles, Paris, 1992.

[RAY 95] M. Raynal et M. Singhal. « Logical Time : A Way to capture Causality in Distributed Systems ». Publication interne numéro 900, I.R.I.S.A., 1995.

[ROZ 91] M. Rozier, V. Abrassimov, et F. Armand. « Overview of the CHORUS Distributed Operating Systems ». Chorus Systèmes, February 1991. CS/TR-90-25.1.

[RUG 94] F. Ruget. « Actor-Wide Trap Tables for CHORUS ». Technical note, Chorus Systems, 1994.

[RUG 95] F. Ruget. « *Mise au point de programmes répartis - Application au système* Chorus ». Thèse de doctorat, Université Joseh Fourier, Grenoble, janvier 1995.

[TAN 91] Andrew Tanenbaum. « Les systèmes d'exploitation, conception et mise en oeuvre ». Traduction d'Alain Kaudé, Interéditions, 1991.

[TAN 95] A. S. Tanenbaum. « *Distributed Operating Systems* », Chapitre 6, pages 289–375. Prentice-Hall, Inc., 1995.

[TEL 95] Telelogic. « *Getting Started with SDT 3.02* », November 1995.

[VAL 90] C. Valot. « A Survey of distributed debugging techniques ». Note technique SOR–94, INRIA Rocquencourt, Septembre 1990.

[WIT 88] L. D. Wittie. « Debugging Distributed C Programs by Real Time Replay ». Dans *Proc. of ACM SIGPLAN and SIGOPS Worshop on Parallel and Distributed Debugging*, May 1988.

Xavier Blondel *a obtenu le diplôme d'ingénieur en informatique du Conservatoire National des Arts et Métiers en 1996 ainsi que le DEA de Systèmes Informatiques de l'université Pierre et Marie Curie. Il prépare actuellement une thèse de doctorat à l'INRIA, en collaboration avec le laboratoire CEDRIC du CNAM, portant sur les ramasse-miettes dans une mémoire répartie persistante.*

Laurent Rosenfeld *a obtenu le diplôme d'ingénieur en informatique du Conservatoire National des Arts et Métiers en 1996 ainsi que le DEA de Systèmes Informatiques de l'université Pierre et Marie Curie.*

Frédéric Ruget *est chef de produit chez Sun Microsystems. Ancien élève de l'École Polytechnique, il a obtenu son titre de doctorat de l'université de Grenoble en 1994, en informatique (systèmes distribués).*

Frank Singhoff *a obtenu le diplôme d'ingénieur en informatique du Conservatoire National des Arts et Métiers en 1996 ainsi que le DEA de Systèmes Informatiques de l'université Pierre et Marie Curie. Il prépare actuellement une thèse de doctorat à l'Ecole Nationale Supérieure des Télécommunications dans le domaine de l'ordonnancement pour les applications multimédias.*