Apport d'un "debugger" de communication au prototypage d'une mémoire partagée répartie

Laurent Rosenfeld — *CNAM / CEDRIC.* — *rosenfeld@cnam.fr*

Frédéric Ruget — *Chorus Systèmes* — *ruget@chorus.fr*

Frank Singhoff — *CNAM / CEDRIC* — *singhoff@cnam.fr*

Résumé

Les applications informatiques sont de plus en plus complexes, et les utilisateurs demandent aujourd'hui des applications aussi fiables que possible. Les systèmes d'exploitation subissent eux aussi cette escalade vers la complexité. Or, si le problème de la mise au point est en bonne partie maîtrisé en univers centralisé, il ne fait qu'émerger dans les systèmes répartis, entre autre à cause de l'indéterminisme qu'introduit la répartition. Les protocoles de gestion de la mémoire virtuelle répartie proposés ces dernières années n'échappent pas à cette complexité croissante, en particulier, les protocoles réalisant des cohérences causales ou relâchées. Cet article illustre l'utilisation qui pourrait être faite d'outils de mise au point spécifiques aux systèmes répartis pour le prototypage et le développement de protocoles de mémoire répartie partagée. Nous utilisons le "debugger" de communication CHORUS/CDB sur un protocole simple de gestion mémoire : l'algorithme distribué par diffusion de Kai Li et Paul Hudak.

Abstract

Computer programs are increasingly complicated, but what users are really looking for nowadays is reliability. Operating systems are also affected by the escalation towards complexity. Yet, whereas the debugging problem is, to a certain extent, solved in centralized systems, the issue is only emerging in distributed systems, because, notably, of the problem of non-determism. Distributed virtual memory managing algorithms are also affected by this growing complexity, especially those implementing causal or weakly coupled forms of coherence. This article exemplifies how debugging tools specifically designed for distributed systems can be used for the prototyping and the development of shared memory algorithms. We use the CHORUS/CDB communication debugger with a simple memory management algorithm: the broadcast distributed algorithm of Kai Li and Paul Hudak.

Mots clés

Mémoire partagée répartie, mise au point, réexécution, Kai Li, CHORUS, CDB

Key Words

Shared distributed virtual memory, debugging, replaying, Kai Li, CHORUS, CDB

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. S'il existe des outils relativement performants adaptés à la mise au point des programmes séquentiels traditionnels, sans concurrence, le problème reste 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 pas d'état global [Lam 78] [Cha 85].
- Le comportement d'ensemble devient partiellement indéterministe et généralement non-reproductible en raison des incertitudes liées à l'ordonnancement des événements sur chaque site et à la charge de chaque machine.
- Le seul fait d'observer le déroulement d'un algorithme risque de perturber son déroulement ; c'est l'effet de sonde.
- Il n'est pas possible de piloter l'exécution globale : par exemple, les délais de communication rendent difficile l'insertion de points d'arrêt.

Notre objectif était de valider une implantation sur le micro-noyau CHORUS de l'algorithme par diffusion proposé par Kai Li et Paul Hudak [Li 89] pour la gestion de la mémoire virtuelle répartie. A cette fin, nous avons suivi une démarche progressive de prototypage. Un premier prototype a été développé sur des stations de travail fonctionnant sous UNIX et communiquant entre elles via des *sockets* au moyen du protocole UDP. La mise au point s'est avérée longue et assez difficile. Ce prototype a ensuite été porté sur MiX SVR4 (le sous-système UNIX V.4 impanté sur CHORUS), puis réécrit sous la forme d'acteurs CHORUS "purs" (c'est-à-dire non-UNIX et n'effectuant que des appels système au micro-noyau).

Le premier prototype sous UNIX nous a amené à soupçonner un comportement anormal de cette implantation de l'algorithme dans certaines conditions. Or, l'outil de mise au point répartie sous CHORUS, CDB (*Communication Debugger*), décrit dans la thèse de doctorat de Frédéric Ruget [Rug 95], est particulièrement adapté aux applications communiquantes. Il nous a donc paru intéressant de tester le prototype final à l'aide de CDB. La principale force de cet outil de déverminage est de permettre une réexécution répartie à l'identique. L'utilisation de cet instrument a permis de reproduire et d'analyser l'anomalie de fonctionnement dont le prototype UNIX avait révélé l'existence, mais dont la mise en évidence était malaisée. CDB a également permis de valider la solution retenue pour résoudre ce problème, à savoir l'utilisation d'horloges vectorielles ([Fid 88][Mat 89]).

Nous rappellerons l'algorithme de Kai Li et Paul Hudak et expliquerons l’anomalie de fonctionnement mentionnée ci-dessus dans la section (2), puis décrirons brièvement l'outil CDB dans la section (3), et aborderons enfin l'implantation réalisée et les observations que nous avons pu en faire dans la section (4).

2. L’algorithme réparti par diffusion de Kai Li et Paul Hudak

2.1 Principe de fonctionnement de l’algorithme

Nous nous contenterons ici de résumer le principe de l’algorithme par diffusion de Kai Li et Paul Hudak [Li 89]. Une description en français peut être trouvée dans [Ray 92]. L'objectif premier de cet algorithme est d'assurer une cohérence forte des données : il faut faire en sorte qu'un site lisant une page lise bien les valeurs obtenues lors de la dernière écriture sur cette page.

Les hypothèses de base du fonctionnement de l'algorithme sont les suivantes :

- Plusieurs sites 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.
- Il n'y a pas de site gestionnaire centralisé fixe, mais pour chaque page, à un moment quelconque, un site propriétaire qui assure la gestion des accès à la page et sert les demandes des autres sites. Il s'agit du dernier site ayant effectué une écriture sur ladite page.
- En cas de demande de page en lecture, le site propriétaire envoie au site demandeur une copie de la page requise, mais reste propriétaire.
- Lors d'une demande de page en écriture, le site rédacteur reçoit la page demandée, qui "migre" donc de l'ancien site propriétaire sur le nouveau site ; le site rédacteur devient le nouveau propriétaire de la page.
- Le site rédacteur a au départ un accès en écriture sur sa page ; en cas de lecture par d'autres sites, il passe lui aussi en lecture.

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

- Le propriétaire (*Owner*) indique qui est propriétaire de la page (le dernier rédacteur).

- Le type d'accès (*Access*) du site sur la page peut prendre ces valeurs : lecture, écriture ou nil. Remarquons que si l'accès est à nil, cela signifie que 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).
- 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 rédacteur) d'annoncer aux sites lecteurs que leur copie de la page devient non-significative (procédure d'invalidation).
- Le verrou : il s'agit d'un sémaphore à la Dijkstra (c'est-à-dire, essentiellement, une file d'attente) assurant la synchronisation des accès aux pages [Dij 65][Tan 91].

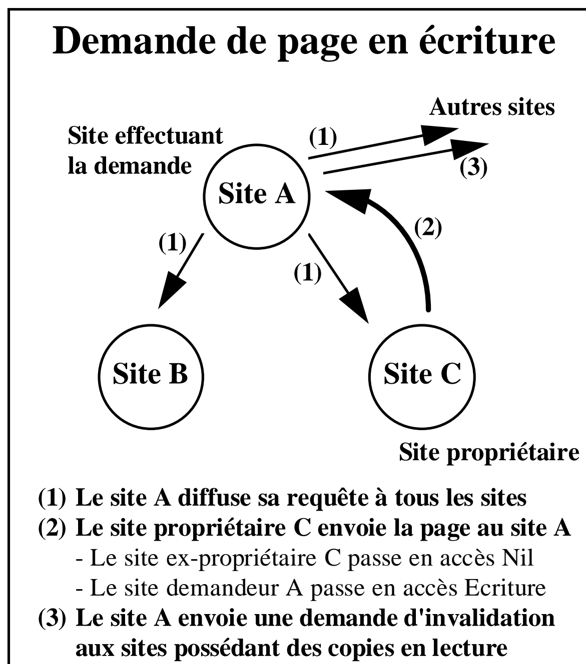
L'algorithme par diffusion fonctionne comme suit. Un site faisant un défaut de page diffuse sa requête à tous les sites ; tout site effectuant ou recevant un défaut de page pose un verrou sur cette page jusqu'à ce qu'il ait fini de traiter la demande. Toute demande reçue ultérieurement sur la même page est mise en attente jusqu'à la levée du verrou. Les sites non-propriétaires ignorent la demande. Le site propriétaire répond, en envoyant, selon le cas :

- En cas de demande d'accès en lecture, une copie du contenu de la page ; dans ce cas, le site propriétaire ajoute le site demandeur au *Copyset*.
- En cas de demande d'accès en écriture, la page, avec ses droits de propriétés et le *Copyset* ; dans ce dernier cas, le site demandeur devient le nouveau propriétaire de la page, et doit encore envoyer à tous les lecteurs (sites figurant dans le *Copyset* qu'il vient de recevoir) une invalidation, c'est-à-dire un avis que la page qu'ils détiennent en lecture n'est plus valable.

Le schéma de la *Figure 1a* 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.

usimultanément en écriture la même page appartenant à un troisième site C (*Figure 1b*). Par "simultanément", nous entendons que chacun des sites A et B a eu le temps de faire sa demande localement, de poser son verrou et de diffuser sa demande aux autres sites avant de recevoir la demande 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'après que la requête de B aura été 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, fait éventuellement une écriture, 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. Dans l'implantation que nous avions envisagée, il y avait également un problème de synchronisation (incrémentation supplémentaire du sémaphore qui verrouille la page demandée).



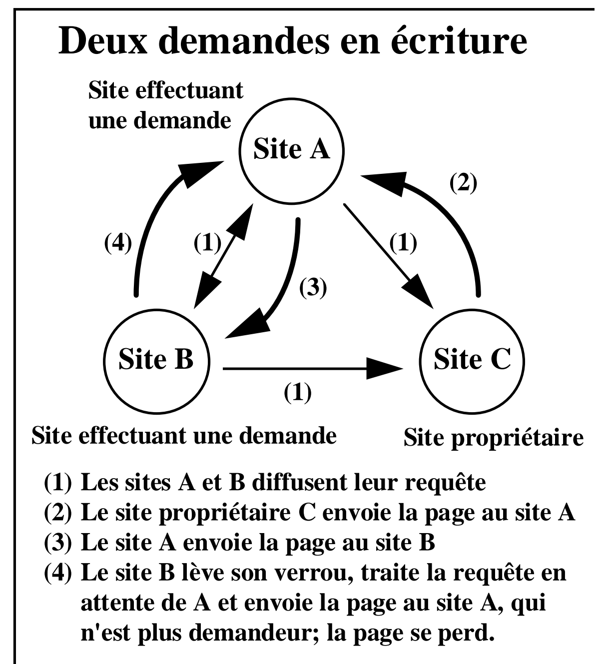


Figure 1a : l'algorithme de Li et Hudak de l'algorithme

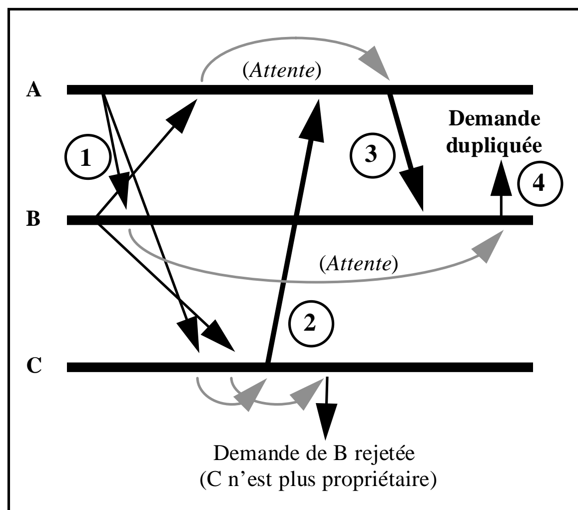
Figure 1b : l'anomalie

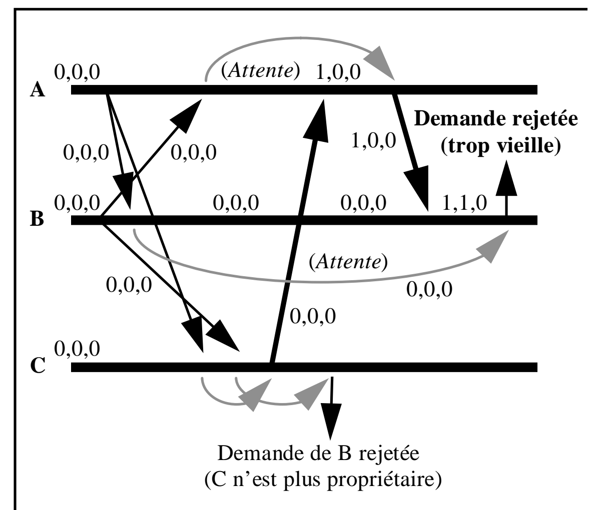
La solution retenue repose sur l'algorithme connu des horloges vectorielles de Fidge et Mattern ([Fid 88][Mat 89]) permettant d'ordonner causalement des

événements : une description détaillée de cet algorithme peut être trouvée dans [Ray 95].[1]

Nous nous contentons ici d'une version simplifiée de ces horloges vectorielles. En effet, comme les problèmes d'ordre causal et de datation ne se posent que lors des transferts de page, le seul événement du système pour lequel nous devons incrémenter l'horloge vectorielle est la réception d'une page en écriture. Ce marquage permet de détecter les demandes trop anciennes, et donc de les écarter.

Les *figures 2a et 2b* comparent, dans l'exemple décrit précédemment, le fonctionnement défectueux et celui de la version corrigée par horloges vectorielles. Dans la version corrigée (*Figure 2b*), 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 de la page (1,1,0). B constate que la demande est trop ancienne. L'ordre causal apporté par les horloges vectorielles (voir [Ray 95]) permet à B de savoir que la requête du site A est périmée et doit donc être rejetée.





[1] Cette solution par les horloges vectorielles a été proposée au cours d'un TP par un groupe d'étudiants du CNAM.

Figure 2a : le fonctionnement défectueux **Figure 2b : correction par horloges vectorielles**

Cette solution par horloges vectorielles a permis de faire fonctionner correctement le premier prototype réalisé sous UNIX. Il n'était toutefois pas facile de s'assurer que le comportement de l'application était strictement conforme aux objectifs. De plus, il n'était pas certain que le portage sur CHORUS se déroulerait sans problème. L'utilisation de CDB, en particulier la possibilité d'effectuer des réexécutions réparties pas à pas, devait nous permettre de répondre à ces questions.

3. Un outil de mise au point sur CHORUS : "Communication debugger"

3.1 Présentation de CDB

CDB est un outil CHORUS destiné à la mise au point d'applications réparties. Le système CHORUS est un système d'exploitation permettant de prendre en compte certains aspects de la répartition. Pour ce faire, il utilise un certain nombre d'abstractions que nous allons brièvement rappeler avant de décrire dans quelle mesure, CDB s'intègre dans CHORUS. Nous décrirons seulement succinctement le système CHORUS [Roz 91] [Tan 94].

Un système CHORUS 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, de porte et de région. 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 possèdent 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 (communications 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'application à base de communications asynchrones : CDB autorise la définition de points d'arrêt, la lecture et l'écriture dans l'espace d'adressage d'acteurs, et, surtout, CDB offre la possibilité d'observer une application grâce à son service de réexécution. En d'autres termes, il permet d'enregistrer une exécution, puis de la rejouer autant de fois que l'on souhaite, en observant 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 systèmes du micro-noyau CHORUS. La méthode utilisée pour réaliser ce service d'interposition est décrite dans [Rug 94e]. Enfin, un service d'observation avertit CDB de l'occurrence d'événements système [Rug 95] [Rug 94d].

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 [Rug 95]). Notre simulation décrite dans la partie 4 nous a permis de constater l'efficacité de cette fonction de CDB

3.2 Architecture de CDB :

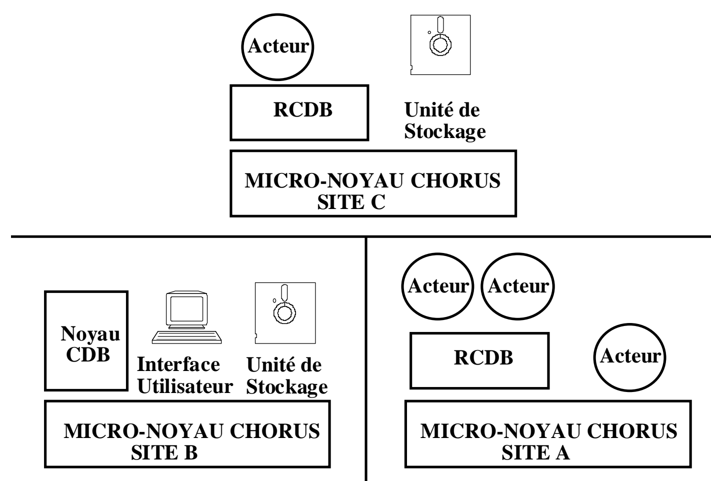


Figure 3 : Architecture de CDB dans un système CHORUS

CDB peut être découpé en quatre composantes :

(1) 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 ;

(2) le noyau CDB pilote les RCDB et interprète les commandes de l'interface utilisateur ;

(3) l'interface utilisateur permet d'interagir sur les objets CDB ; il ne peut y avoir qu'une seule interface et un seul noyau CDB dans un système CHORUS ;

(4) enfin, des unités de stockage permettent d'enregistrer les journaux.

4. Expérimentation et mise en évidence de l'anomalie du protocole de Kai Li

4.1 Description de notre mise en oeuvre

Notre expérimentation sous CHORUS simule les différents sites serveurs de page par un acteur 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 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 4* illustre le fonctionnement de notre expérimentation. Dans cet exemple, le serveur de tests diffuse à tous les sites une notification de défaut de page, qui devra dans ce cas être effectué par le site 1. Ce dernier diffuse ensuite la requête aux autres sites.

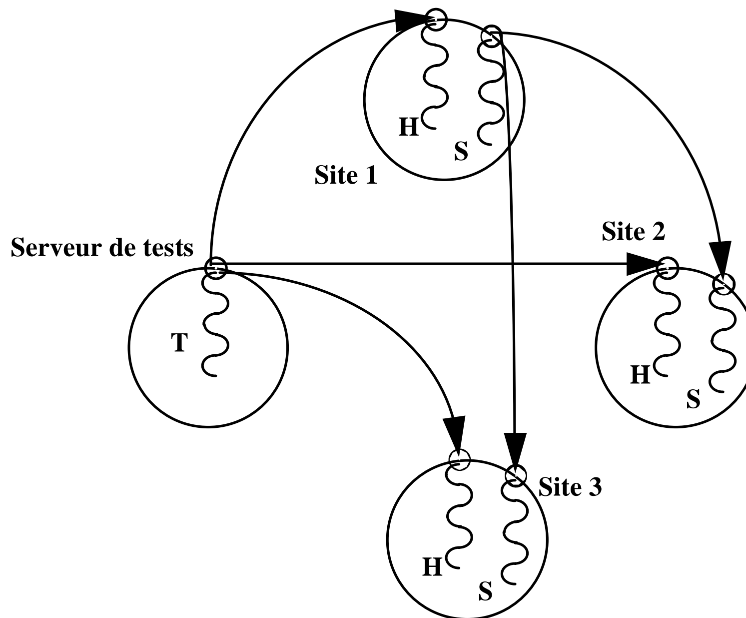


Figure 4 : Architecture de notre expérimentation de Kai Li.

4.2 Comment CDB permet de mettre au point un protocole réparti

Durant cette étude du protocole de Kai Li sur CHORUS, nous avons cherché à dégager les avantages que l'on pouvait tirer d'un outil comme CDB pour l'observation et la mise au point. Aussi, une fois le dernier prototype écrit, nous avons étudié son comportement sous CDB. 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 aidé à mettre clairement en évidence l'anomalie signalée du protocole de Kai Li, et à vérifier que la solution des horloges vectorielles éliminait définitivement le problème. CDB nous a aussi aidé à écarter nos dernières hésitations quant au fonctionnement de cet algorithme.

La prise en main de CDB est très rapide. La présence de deux exemples simples livrés avec l'outil permet de comprendre les quelques commandes de base à utiliser. Toutefois, il est bien évident que l'outil nécessite un minimum de culture CHORUS ; l'utilisateur se devra de connaître les abstractions usuelles du micro-noyau CHORUS. La manipulation des tables de symboles obtenues par compilation peut aussi être utile. En effet, si CDB ne permet pas la mise au point au niveau du source, on peut aisément manipuler les symboles d'un module objet.

La première étape consiste à enregistrer une exécution. 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 pas 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 l'effet de sonde de CDB est faible lors de l'enregistrement et qu'il n'a pas perturbé l'exécution de notre prototype.

Une fois la session d'enregistrement finie, il est possible soit de consulter les journaux, 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 systèmes effectués ainsi que leur résultat.

```
[  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]   188    64>   th_preempted: pc  804a639 bc  3ab
[ 34]   450    6d>   sc_ipcReceive: diag:8 rcvPort:-1 from outside [  5  ] Body:8 Annex:0
[  c]   a70    64>   sc_ipcSend: diag:0
[  8]   9874 7c>     pt_deleted:
[ 1c]   98b0 7f>     sc_ipcReceive: diag:-4
[  8]   98b8 d3>     ac_deleted:
```

Figure 5 : Un exemple d'historique produit par l'outil ASCII de CDB. La première colonne donne la taille de l'enregistrement de l'événement dans le journal, la deuxième colonne le déplacement sur le fichier journal et la troisième colonne l'identifiant CDB de l'activité. Viennent ensuite l'instruction source correspondant à l'événement et différents identifiants qui dépendent de l'événement enregistré (identifiants d'acteurs, compteur ordinal, code de retour d'appel système, taille de message, etc.)

Certains outils 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'arrêts au cours de la session de réexécution ; nous n'avons pas utilisé cette technique, car il nous a été assez facile d'évaluer le nombre de pas nécessaire pour s'arrêter un peu avant les passages intéressants. Il existe aussi la possibilité de lancer plusieurs sessions d'exécution simultanément et d'observer

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

chacune de ces sessions à des stades différents. Là aussi, nous connaissions suffisamment bien notre prototype pour ne pas avoir besoin de cette fonction. Cette fonction peut cependant se révéler très utile avec une application encore mal connue : si à un instant donné, on a besoin de savoir si telle ou telle partie de code a été exécutée au début de l'application, il peut être judicieux de lancer une autre session, de s'arrêter à cet endroit, puis de continuer sur la première session. Cette technique est bien plus rapide que d'arrêter totalement la session en cours pour recommencer.

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 nous a permis d'évaluer le 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 des outils de mise au point sur des applications distribuées

Nous avons aussi tenté de trouver les fonctionnalités supplémentaires que devrait offrir un outil de mise au point répartie. Sur ce point, CDB apporte certaines fonctions qui facilitent grandement le travail de mise au point. Citons en premier lieu une des fonctions essentielles : 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. Dans ce domaine, CDB est particulièrement efficace. En effet, si répéter une exécution en univers centralisé est facile, c'est beaucoup plus complexe en environnement réparti. Or, CDB est sur ce point très fiable et a supporté nos tests qui couvrent la majorité des appels système CHORUS courants. De plus, l'intégration sous CDB des rares appels systèmes non traités ne semblerait pas relever d'une difficulté insurmontable. La structure modulaire de CDB (et en particulier des RCDB) devrait permettre la réalisation de ce type de maintenance évolutive avec une relative simplicité.

Un outil de mise au point répartie doit offrir un moyen d'analyse post-mortem efficace. C'est le rôle des journaux. Ces journaux se doivent d'être peu volumineux et de contenir les informations les plus utiles. Les traces de CDB sont d'un volume raisonnable et, même après des tests assez importants, elles restent exploitables manuellement. De plus, elles sont concises et possèdent toutes les informations qui peuvent être nécessaires pour suivre tous 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 utilisés car ils demandent au départ de l'utilisateur un investissement en temps trop important. Or, l'interface de CDB comporte un nombre limité de commandes et quatre d'entre elles suffisent pour effectuer une réexécution. Cela fait de CDB un outil d'un abord facile.

Enfin, notons que le fonctionnement de CDB n'exige que peu de ressources matérielles. Nous avons expérimenté notre prototype sur des PC à base de i386 dotés de huit méga-octets de mémoire. Le faible taille du micro-noyau CHORUS nous a permis de faire fonctionner correctement CDB dans ces conditions. Il faut remarquer que les machines où se trouvaient les acteurs mis au point sous CDB ne possédaient que le micro-noyau CHORUS et un RCDB comme acteur superviseur. Une machine avec un processus MiX est cependant nécessaire pour faire fonctionner l'interface et le moniteur central de CDB.

Toutefois, dans sa version actuelle, l'outil CDB est soumis à un certain nombre de contraintes. En particulier, il a été conçu pour ne traiter que des acteurs CHORUS purs. Il n'est pas possible de l'utiliser sur des acteurs MiX (ce qui supprime toutes les applications avec des accès fichiers). Or, pour concevoir par exemple un système réparti comportant des machines hétérogènes, il serait utile de pouvoir mettre au point une application répartie sur plusieurs types d'environnements. De plus, il n'y a aucune raison que des développeurs d'applications sous MiX ou d'autres environnements n'aient pas besoin de ce type d'outil, dont la puissance leur serait d'une grande utilité.

On ne peut pas non plus effectuer une mise au point au niveau du source. L'utilisateur ne peut pas retrouver les abstractions qu'il a utilisées lors de l'écriture de son application. Même si les tables de symboles permettent de retrouver des informations suffisantes, elles ne sont pas d'utilisation aussi simple que la mise au point au niveau du source.

Un outil de mise au point répartie devrait offrir la possibilité d'obtenir une vision concise et globale de l'état du système. CDB manque quelque peu d'ergonomie sur ce point. En effet, bien que tous les outils d'observation existent dans l'interface de CDB, leur utilisation est un peu fastidieuse : il faut naviguer dans des hiérarchies d'objets pour les consulter, et il n'est pas toujours aisé de se faire une idée globale de l'état des acteurs et des activités. La consultation des informations sous CDB ressemble à la recherche d'un fichier dans un système de répertoires et de fichiers hiérarchiques. Il est à noter qu'une fonctionnalité intéressante existe sous CDB : la possibilité de donner aux acteurs et activités un nom, plus facilement manipulable que les identifiants numériques de CHORUS. Une interface graphique apporterait cependant une meilleure ergonomie que l'interface de type "ligne de commande" actuelle.

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, l'outil de mise au point du micro-noyau, et GDB, du 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).

5. Conclusion

L'utilisation de CDB nous a permis de valider notre prototype et d'analyser son comportement. Cette analyse a démontré que le fonctionnement de ce prototype suivait l'algorithme de Kai Li. Ce travail montre l'utilité d'outils de mise au point spécifiquement adaptés aux systèmes répartis.

La tendance en matière de mise au point répartie semble justement s'orienter vers ce type d'outil : ceux qui permettent la réexécution d'application. On peut ainsi trouver d'autres prototypes tels que Recap [Wit 88], Bugnet [Pan 89], etc. Toutefois, s'il existe un bon nombre d'outils, aucun de ceux que nous avons pu étudier n'avait résolu tous les problèmes qui nous semblent essentiels pour l'utilisateur : réexécution d'un ou plusieurs processus, et ce quel que soit le processus et à n'importe quel moment, sur des machines ou des systèmes hétérogènes, avec une notion de point de reprise et dans des conditions d'ergonomie aussi satisfaisantes que sur les outils centralisés, tout cela sans effet de sonde. De nombreuses recherches restent donc encore à faire dans ce domaine.

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 virtuelle répartie et des algorithmes de Kai Li et Paul Hudak, nous a aidés et soutenus assidûment, nous a prodigué ses conseils et fourni de la documentation et a bien voulu relire ce document, et Christian Santellani, doctorant au CEDRIC, qui nous a permis de travailler dans de bonnes conditions sur un réseau de machines fonctionnant sous CHORUS. Sophie Pichaureaux et Xavier Blondel ont participé à la réalisation du premier prototype sous UNIX.

6) Bibliographie

[Cha 85] "Distributed Snapshots: Determining Global States of Distributed Systems"

Chandy, K.M. & Lamport, L., ACM Trans. Comp. Syst. 3(1):63-75 (Feb. 1985)

[Chor 91] "Chorus Kernel V3 r5 : Specification and Interface"
Chorus Team
CS/TR-91-69.3, Chorus Systèmes, 1991

[Chor 92] "Chorus Kernel V3 r5 : Programmer's Reference Manual"
Chorus Team
CS/TR-92-26.3, Chorus Systèmes, 1992

[Chor 93] "Chorus Kernel V3.r5 : Network Architecture"
Chorus Team
CS/TR-93-35.2, mai 1994, Chorus Systèmes, 1993

[Dij 65] "Co-operating Sequential Processes"
E.W. Dijkstra, Programming Languages, Genuys F (éd.), Londres, 1965.

[Fid 88] "Timestamps in Message Passing Systems That Preserve the Partial Ordering"
Fidge, C.J., Proc. 11th Australian Computer Science Conference, pages 55-66, Feb. 1988

[Lam 78] "Time, Clocks and the Ordering of Events in a Destributed System"
Lamport, L., Communications of the ACM, 21(7):558-565, 1978

[Li 89] "Memory Coherence in Shared Virtual Memory Systems"
Li K. & Hudak P., ACM Transactions on Computer Systems, Vol. 7, No. 4, November 1989 (pp. 321-359)

[Mat 89] "Virtual Time and Global States of Distributed Systems"
Mattern, F. Proc. of Int. Workshop on Parallel and Distributed Algorithms, Bonas (France),
pp. 215-226, 1989.

[Pan 89] "Supporting Reverse Execution of Parallel Programs"
Douglas Z. Pan and Mark A. Linton, Stanford University
SIGPLAN Notices, Vol. 24, Number 1, 24 Jan. 1989

[Ray 95] "Logical Time : A Way to Capture Causality in Distributed Systems "
M. Raynal & M Singhal, Publication interne n°900, I.R.I.S.A.

[Ray 92] "Gestion des données réparties : problèmes et protocoles"
M. Raynal, Eyrolles, Paris, 1992

[Roz 91] "Overview of the Chorus Distributed Operating System"
CS-TR-90-25.1 sur ftp.chorus.fr

M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemeont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser, Février 1991, Chorus Systèmes

[Rug 95] "Mise au point de programmes répartis - Application au système CHORUS"
Thèse de doctorat, F. Ruget, janvier 1995

[Rug 94a] "Cheaper Matrix Clocks"
Fréderic Ruget
In Proc. of the 8th Int. Workshop on Distributed Algorithms
Terschelling, the Netherlands, September 1994. Springer Verlag
(CS-TR-94-63 (sur ftp.chorus.fr), juillet 1994, Chorus Systèmes)

[Rug 94b] "Time Travel for Chorus"
Fréderic Ruget, Septembre 1994

[Rug 94c] "A Distributed Execution Replay Facility for Chorus"
Fréderic Ruget, Chorus Systèmes, Septembre 1994

[Rug 94d] "Matching Operating Systems to Application Needs : A Case Study"
Fréderic Ruget et Martin Herdieckerhoff, Chorus Systèmes/SIEMENS, Septembre 1994

[Rug 94e] "Actor-Wide Trap Tables for Chorus"
CS/TN-94-XX, Chorus Systèmes

[Tan 91] "Les systèmes d'exploitation, conception et mise en oeuvre"
Andrew Tanenbaum (Traduction d'Alain Kaudé), InterEditions, 1991

[Tan 94] "Distributed Operating Systems"
Andrew Tanenbaum, Prentice Hall, 1994

[Wit 88] "Debugging Distributed C Programs by Real Time Replay"
Larry D. Wittie, State University of New York at Stony Brook
In Proc. of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging
May 5-6 1988, University of Wisconsin, Madison, Wisconsin 53706