Observation du contrôle dans les applications distribuées

Daniel Prun, Patrick Barril
Laboratoire MASI - CNRS URA 818 - Université Paris 6
4, Place Jussieu 75252 Paris Cedex 05
E-mail: {Daniel.Prun, Patrick.Barril}@masi.ibp.fr

1. Introduction

Durant un acte de déboguage de programmes informatiques, l'action principale consiste à retrouver la cause d'une erreur observée à l'exécution. Elle est souvent due à des idées préconçues sur le comportement de l'application et n'a pas toujours de lien apparent avec son symptôme. Elle est découverte et expliquée à l'issue d'une confrontation d'une représentation mentale de l'application avec l'observation de son exécution ; la modification (correction ou ajout) peut alors être entreprise. Le programmeur d'application passe en moyenne [3] plus de 50% de ses efforts dans cette étape de compréhension lors d'un acte de maintenance corrective. Cette étape est encore plus difficile quand il s'agit de maintenir une application distribuée. Nous entendons par application distribuée une application composée d'un ensemble de processus répartis sur plusieurs sites, chacun ne pouvant accéder qu'à sa mémoire locale, communiquant par envois de messages. Le comportement de telles applications est particulièrement délicat à interpréter :

- Les applications distribuées ont des comportements temporels imprévisibles. Ceci est dû aux charges aléatoires du réseau de communication et des noeuds qui rendent respectivement les délais de transmission des messages et les temps de traitement imprévisibles.

- Les applications distribuées doivent incorporer des mécanismes supplémentaires de contrôle de la concurrence et de synchronisation (tels que la détection de terminaison ou l'élection par exemple). Ceci rend leurs comportements plus complexes à appréhender.

- Enfin, une forme d'indéterminisme peut être due à la conception. Si les processus sont vus comme des composants enfichables et ré-utilisables, ils se comportent comme des sous-systèmes ouverts plongés dans un environnement auquel ils doivent s'adapter. C'est le cas des applications conçues selon le paradigme multi-agents et utilisant des concepts de négociation, d'interaction et de coopération.

L'indéterminisme complique beaucoup l'application de techniques classiques de déboguage : il n'est pas possible de reproduire exactement un comportement dans un même contexte. Un enregistrement complet des traces est coûteux et difficile à interpréter. Le déboguage d'applications distribuées nécessite donc des outils et des méthodes particuliers. En effet, une erreur peut aussi bien être causée par une spécification incorrecte que par une mauvaise structuration du code ou encore à une erreur de codage proprement dite. Il s'agit de confronter une spécification et ses comportements attendus avec le comportement réel, et des outils interactifs sont précieux.

Nous présentons ici les mécanismes d'observations de l'exécution d'applications distribuées dans notre plate-forme Mathilda. Nous illustrons d'abord notre approche sur des exemples puis nous présentons les principes du système de communication ainsi que ceux de l'observation et de la ré-exécution.

2. L'observation

Le rôle d'un observateur est de vérifier la validité de certaines assertions sur l'application. Ces assertions impliquent parfois des états répartis sur plusieurs sites, compliquant le travail d'un observateur unique. Ce choix d'un observateur unique est assez contraignant car il implique des mécanismes dédiés (estampilles vectorielles [5]) reposant sur des propriétés formelles du calcul (par exemple la délivrance causale [4]). Il est lourd à mettre en oeuvre dans des milieux dynamiques (ajout/retrait de processus). En fait, ce choix ne permet pas de tirer parti du parallélisme potentiel : l'observateur reconstruit le treillis global des événements à partir d'estampilles (par exemple [8]). De plus, dans une plate-forme évolutive, permettant de définir de nouveaux mécanismes de communication et de contrôle, les modèles de calculs et d'observations doivent être de plus en plus abstraits ce qui est difficile à traiter par un observateur unique. Aussi, nous choisissons plusieurs observateurs.

Les différents observateurs, qui observent chacun une portée limitée, doivent coopérer afin de constituer un état global consistant : le résultat de l'observation. Dans notre réalisation, la communication utilise des diffusions asynchrones, l'observateur est local à un site et observe un processus. Le résultat d'une observation locale est une trace, c'est à dire une suite d'événements coorespondant à des réceptions de messages.

En plus d'enregistrer l'événement "réception d'un message" un observateur peut effectuer des tests. Lors d'un test, un observateur passif est automatiquement notifié du changement de valeur des variables pertinentes. Nos observateurs sont ici passifs et n'utilisent pas, par exemple, d'attente active. Nos observateurs ont des aspects actifs nécessitant des échanges de messages quand ils calculent des états globaux consistants ou combinent des tests pour vérifier une assertion.

Cette observation est non intrusive : elle ne nécessite pas de modification du code observé. En effet on n'observe que les activations des structures de contrôle : on peut donc embarquer l'observation sur le moteur assurant ce contrôle, dans notre cas la couche de communications.

Pour l'instant, la sémantique de la concurrence est en terme d'entrelacement et est incapable de rendre compte de la simultanéité d'événements : ceux-ci sont totalement ordonnés. Il est impossible à un observateur de faire la distinction entre l'exclusion mutuelle (ab+ba) et l'indépendance de deux événements (a//b). En l'absence d'une primitive de rendez-vous, on ne peut programmer l'occurrence simultanée de deux événements : son observation n'est donc pas pertinente pour la mise au point et une sémantique de vrai parallélisme n'est pas encore nécessaire. Par contre, notre approche consiste à prendre en compte dès l'intégration d'un composant de contrôle (par exemple un rendez-vous), des mécanismes d'observation correspondants ainsi qu'une sémantique adéquate pour la ré-exécution. Ainsi l'implantation d'un rendez-vous nécessite d'adapter le modèle de trace (voir par exemple [1]) afin de prendre en compte cette synchronisation. Plus généralement, nous rajoutons à chaque composant un mécanisme de notification pour les observateurs de chaque processus impliqué ainsi que le format de trace pour la ré-exécution. Enfin, il est nécessaire d'adapter localement la ré-exécution à ces nouvelles traces. De nombreux formalismes rendent compte de certains aspects du comportement parallèle (Pomset [10], structures d'événements [7], espaces de Chu [11]) et sont plus ou moins adaptés à une structure de contrôle particulière. Le choix du formalisme doit permettre de saisir des observations de haut niveau et plus proche de la vision du programmeur lorsqu'il utilise ces composants.

3. Réalisation

Mathilda est un projet de plate-forme distribuée pour l'intégration d'applications coopératives. Elles incorpore des mécanismes de placement de tâches, de répartition de charge, de tolérance aux fautes ainsi que pour la maintenance du système (arrêt du système, suivi d'application...).

Elle repose sur un serveur de message (le **NMS**: **N**etwork **M**essage **S**erver [6]) assurant des communications par diffusion asynchrone. Celui-ci est fiable, accessible d'une manière uniforme (tous les processus ont la même interface de communication), anonyme (les processus peuvent communiquer sans connaître l'identité de leurs interlocuteurs) et extensible (l'ajout de processus ou le retrait ne perturbe pas l'application). Il utilise les principes d'**abonnement** (subscription) et de **diffusion sélective** (multicast) de messages typés. Il respecte la norme ECMA à laquelle il ajoute un quatrième type de message (donnée) aux trois autres (requête - demande de service, notification - réponse à une requête, erreur - événement exceptionnel comme panne, échec d'une requête). Ces messages sont de plus sous-typés par des noms de services. Un abonnement est une demande explicite que fait un processus au NMS indiquant le type de messages qu'il désire recevoir et donc les services qu'il offre. Une requète pour un service non implanté (pas d'abonné) est ignorée.

Une instance du NMS est répliquée sur chaque noeud du réseau. Lorsqu'un processus émet un message, l'instance locale du NMS le délivre localement et le relaie aux autres instances ayant des abonnés locaux pour ce type de message.

Les propriétés spécifiant le NMS sont les suivantes:

1. Le NMS est sans perte ni altération de message. Il repose sur des protocoles de communications fiables (TCP/IP) et des codages des données indépendants de l'architecture (XDR).

2. Les messages envoyés par un émetteur donné sont dans le même ordre chez tous les processus receveurs (les messages d'un même émetteur restent ordonnés).

3. Les messages d'émetteurs différents peuvent s'entrelacer mais tous les messages sont reçus dans le même ordre par tous les récepteurs sur un même noeud.

4. Implantation de l'observation et de la ré-exécution

Par souci de protection, les processus sont organisés en niveaux de priorité et ne peuvent recevoir des message émis que par des processus de niveau supérieur. Un processus de niveau le plus bas est capable de recevoir tous les messages émis gràce à une valeur générique permettant de s'abonner à tous les types de message. Un observateur est un processus dont le niveau de priorité est le plus élevé donc capable de recevoir tous les messages routés par le NMS [2]. Un observateur unique, bien que limité sert pour l'administration du système (statistiques, accounting).

Un observateur par site est suffisant pour rejouer une application car tous les processus d'un même site reçoivent les messages dans le même ordre (propriété 3). Le principe de la ré-exécution est adapté de l'algorithme "Instant Replay" distribué [9] qui assure le même ordre de réception des messages que celui enregistré dans la trace de l'exécution originale : à l'exécution, chaque observateur note la réception des messages sur chaque noeud puis à la ré-exécution, les messages reçus sont ré-ordonnés en fonction de la trace enregistrée. Ceci assure le même ordre des réceptions de messages lors de l'exécution et de la ré-exécution. La répartition est pleinement exploitée (distribution de l'observation et du contrôle de la ré-exécution). Cette ré-exécution peut être partielle, n'impliquant que certains processus ou certains types de messages. Elle permet un contrôle interactif par l'utilisateur afin de modifier l'ordre de délivrance des messages (à des fins de test ou d'injection de fautes).

Plusieurs observateurs distribués permettent l'adaptation des algorithmes de calcul d'états globaux consistants, la vérification d'assertions réparties et la détection des interblocages. Nous avons entrepris l'implantation de ces algorithmes, mais en mettant l'accent sur leurs adaptations à de nouveau mécanismes de contrôle dans le cadre des applications coopératives.

5. Conclusion

Les mécanismes d'abonnement et de diffusion sélective de notre serveur de messages (le NMS) permettent une observation non intrusive, dynamique, distribuée et filtrée des communications. Des prototypes d'observateurs centralisés (pour les statistiques et la comptabilité (accounting) et l'analyse du contenu des messages) ont déjà permis de nombreuses opérations de déboguage, notamment en ce qui concerne les protocoles de communication. L'implantation d'un observateur distribué et la définition d'une méthode d'extension à de nouvelles structures de contrôle est en cours.

Mathilda est un projet de plate-forme distribuée pour l'intégration d'applications coopératives et nécessite une définition incrémentale de composants de contrôle. Leur implantation va de paire avec leur prise en compte par l'observation et la ré-exécution distribuée : dès qu'une structure de contrôle est certifiée, elle doit fournir une observation plus simple qu'une trace complète mais suffisante pour une ré-exécution. Ceci permet une interprétation plus abstraite des comportements par le programmeur et donc de faciliter son travail de compréhension lors d'un acte de maintenance corrective.

Bibliographie

1. Nagarajan R. Abramsky S., Gay S. Specification structures and propositio-as-types for concurrency. *To appear in Logics for Concurrency: Structure vs. Automata, Proc. of the 8th Banff Higher Order Workshop, Springer-Verlag, LNCS*, 1995.
2. Prun D. Baggio A., Bonnaire X. Intrusion free monitoring: An observation engine for message server based applications. *Proc. of the Tenth International Symposium on Computer and Information Sciences (ISCIS X)*, pages 541–548, 1995.
3. McClure C. The three r's of software automation: Re-engineering, repository, and reusability. *Prentice Hall*, 1992.
4. Tel G. Charron-Bost B., Mattern F. Synchronous and asyncronous communication in distributed computations. *Rapport technique no 91.55 du LITP*, pages 91–108, 1991.
5. Mattern F. Virtual time and global states of distributed systems. *In Parallel and Distributed Algorithms,North-Holland*, pages 215–226, 1988.
6. Folliot B. Foughali K. Topic-se: a tool-based open platform for integration of control in software environments. *Information and Software Technology, Vol. 36, no 7*, pages 427–433, 1994.
7. Winskel G. Event structures. *In LNCS 255, Springer Verlag*, pages 325–392, 1987.
8. Rampon J.X. Jard C., Jourdan G-V. On-line computations of the ideal lattice of posets. *Theoretical Informatics and Applications, Vol 29, no 3*, pages 227–244, 1995.
9. Mellor-Crummey J-M. Leblanc J. Debugging parallel programs with instant replay. *IEEE Transactions on Computers. C-36(4)7*, pages 471–482, 1987.
10. Pratt V. R. Modeling concurrency with partial orders. *International Journal of Parralel Programming, Vol 15, no 1*, pages 33–71, 1986.
11. Gupta V. Chu spaces: A model of concurrency. *Thèse de doctorat, Stanford University. Rapport technique disponible à: ftp://boole.stanford.edu/pub/gupthes.ps.Z*, 1994.