Intergiciels pour la tolérance aux fautes

Etat de l'art et défis

François Taïani* — **Marc-Olivier Killijian**** — **Jean-Charles Fabre****

** Computing Department, Lancaster University*
LANCASTER LA1 4WA, Grande Bretagne
francois.taiani@comp.lancs.ac.uk

*** LAAS-CNRS*
7, avenue du Colonel Roche
31077 Toulouse Cedex 4
{marco.killijian, jean-charles.fabre}@laas.fr

RÉSUMÉ. *La tolérance aux fautes informatique peut aujourd'hui s'appuyer sur presque trente ans de résultats théoriques et expérimentaux. Depuis le début des années quatre-vingt, un effort important a en particulier été porté sur l'intégration de mécanismes génériques pour la tolérance aux fautes dans des intergiciels répartis. Dans cet article, nous revenons sur l'évolution de ces plates-formes, en illustrant notamment comment les notions de transparence et de flexibilité ont progressivement déterminé l'organisation interne de ces intergiciels, pour culminer dans l'utilisation d'architectures réflexives. L'article s'ouvre tout d'abord sur une introduction à la tolérance aux fautes, pour se poursuivre par un historique de la notion d'intergiciel dans les systèmes tolérant les fautes, et finir sur une perspective des avancées à venir.*

ABSTRACT. *Fault tolerance in computer systems can look back at nearly thirty years of theoretical and experimental results. Since the 80's an important effort has been brought to integrate generic fault-tolerance mechanisms into reusable distributed middleware. In this paper, we offer a retrospective on those fault-tolerant platforms, from the early-day pioneers, to the most recent ones. This presentation highlights how transparency and flexibility concerns have increasingly shaped the internal organisation of fault-tolerance middleware, and eventually resulted in reflexive fault-tolerance platforms. The article is organised as follow: we first introduce the basics of fault-tolerance, before presenting the evolution of fault-tolerance middleware. We then conclude on the remaining challenges and perspectives.*

MOTS-CLÉS : *tolérance aux fautes, intergiciels, réflexivité.*

KEYWORDS: *fault-tolerance, middleware, reflection.*

1. Introduction

La tolérance aux fautes est aujourd'hui un domaine informatique relativement mûr, qui peut s'appuyer sur un imposant corpus de résultats théoriques et expérimentaux. Comprendre et réaliser des mécanismes de tolérance aux fautes reste cependant un domaine d'expert, et depuis vingt-cinq ans de nombreux efforts ont été faits pour chercher à factoriser cette expertise dans des plates-formes de développement prêtes à l'emploi. Dans cet article, nous proposons un tour d'horizon de ces différents projets, en montrant comment les premières tentatives visant à proposer des primitives de programmation adaptées au développement de mécanismes de tolérance aux fautes ont progressivement débouché sur des architectures élaborées centrées sur la transparence et la flexibilité. La problématique des intergiciels pour la tolérance aux fautes a aujourd'hui rejoint celle, plus vaste, de la séparation des préoccupations (*separation of concerns* (Hürsch *et al.,* 1995)), des architectures réflexives, et des langages orientés aspects. C'est ce parcours que nous proposons de retracer avec le lecteur, dans une rétrospective historique de vingt-cinq ans de recherche à la frontière entre l'algorithmique répartie et les architectures logicielles.

Pour situer le contexte de cette rétrospective, l'article commence par un rappel des principes fondamentaux de la tolérance aux fautes (section 2), ainsi qu'aux mécanismes de tolérance aux fautes dans les systèmes répartis (section 3). Nous poursuivons par une discussion générale des propriétés recherchées par les intergiciels pour la tolérance aux fautes, et des grands types d'architecture rencontrés (section 4). La section 5 illustre, sur la base des projets les plus représentatifs, chaque grand type d'architecture, en soulignant son intérêt et ses limites. Nous concluons enfin sur les limites identifiées dans les plates-formes les plus récentes et donnons une perspective sur les défis à venir.

2. La tolérance aux fautes

Dans cette partie, nous présentons les notions de base de la *sûreté de fonctionnement* qui nous permettront de cerner plus précisément la tolérance aux fautes. Nous reprenons pour cela la taxonomie développée, entre autres, par Jean-Claude Laprie, Brian Randell et Algirdas Avizienis, qui fait largement référence.

2.1. *Notions de base : fautes, erreurs et défaillances*

La *sûreté de fonctionnement* d'un système est définie comme : *[L]a propriété qui permet à ses utilisateurs de placer une confiance justifiée dans la qualité du service qu'il leur délivre*. (Laprie, 1996)

Un système *défaille* lorsque le service qu'il délivre diverge du service attendu[1]. La sûreté de fonctionnement cherche donc à éviter les défaillances, ou au moins à prévenir les plus catastrophiques. Ainsi, on préférera faire arriver les voyageurs d'un train en retard (ce qui est en soi une « défaillance » du système), plutôt que de leur faire risquer une collision mortelle (qui est aussi une défaillance, mais catastrophique).

Prévenir les défaillances requiert d'introduire deux nouvelles notions : *faute* et *erreur*. Une *faute* est la cause d'une défaillance[2]. La notion de causalité étant récursive, l'identification d'une faute impose toujours une part de subjectivité. Par exemple, si un ion lourd traverse le circuit d'un satellite, entraînant sa perte, doit-on incriminer l'ion lourd (faute physique, externe au système, opérationnelle), ou doit-on chercher du côté du blindage du circuit qui serait mal conçu (faute humaine, « interne » au système, de conception) ? Une défaillance peut aussi résulter d'une combinaison de fautes multiples, mais faute de place, nous ne rentrerons pas dans ces subtilités.

Les fautes étant les causes des défaillances, la sûreté de fonctionnement cherche à les combattre, si possible en évitant qu'elles se produisent (*prévention*, c'est le premier moyen de la sûreté de fonctionnement), ou en les éliminant (*élimination*, c'est le second moyen de la sûreté de fonctionnement). Ces deux moyens ne sont cependant pas suffisants du fait de l'usure inévitable des systèmes (pour leur partie matérielle), des environnements souvent agressifs dans lesquels ils évoluent, et, particulièrement pour les logiciels, de l'impossibilité de concevoir des systèmes totalement exempts de fautes. Ces constatations conduisent à introduire une troisième notion, celle d'*erreur*, qui fait le lien entre fautes et défaillances. Un état erroné désigne l'état anormal d'un système, résultant de l'activation d'une faute, qui peut potentiellement amener à une défaillance. La notion d'erreur permet de désigner la chaîne de causalité qui conduit de la faute à la défaillance perçue par l'utilisateur (figure 1).

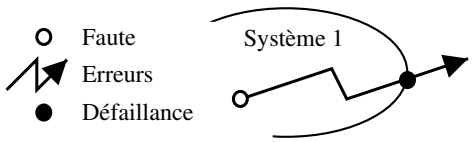

Figure 1. *Faute, erreurs et défaillance*

1. On évite en général de relier nécessairement la notion de *comportement attendu* à celle de *spécification*. Un système peut en effet défaillir tout en respectant parfaitement une spécification erronée. L'expression d'un comportement attendu pourra par conséquent selon les contextes prendre différentes formes : formelles (automate, formule logique), semi-formelles (contrat, langage de conception tel UML ou SDL), ou informelles (lois, réglementations, engagements contractuels).
2. On rencontre parfois le mot *panne* à la place de celui de *faute*. La notion de *panne* a cependant une connotation matérielle très forte, et nous ne l'emploierons pas pour cette raison. Il est par exemple très peu intuitif de parler de panne « logicielle » pour un « bogue » informatique.

En cassant cette chaîne de causalité, c'est-à-dire en empêchant une erreur de se propager jusqu'à l'utilisateur, on améliore la sûreté de fonctionnement d'un système tout en tolérant ses fautes : c'est le troisième moyen de la sûreté de fonctionnement, la *tolérance aux fautes*, que nous détaillons plus avant dans les sections suivantes.

2.2. *La tolérance aux fautes*

Un système est usuellement composé d'autres systèmes, plus petits, qui lui délivrent leurs services. La défaillance d'un de ses composants devient alors une faute pour le système global (figure 2). La tolérance aux fautes consiste à protéger l'utilisateur externe des conséquences des défaillances des composants internes du système.

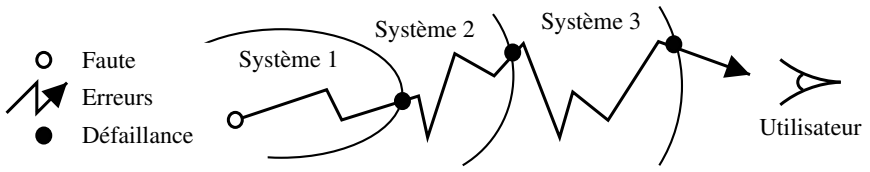

Figure 2. *Récursivité des fautes, erreurs et défaillances*

Cette protection peut prendre plusieurs formes, selon que l'on cherche à masquer complètement les fautes à l'utilisateur (par exemple dans un réseau local, en basculant de manière transparente d'un serveur DNS – résolveur de nom – primaire sur un serveur secondaire), à assurer une continuité de service en acceptant une dégradation temporaire de sa qualité, ou à prévenir les conséquences les plus catastrophiques d'une défaillance en mettant le système dans un état sûr (comme d'arrêter un train dès que l'on n'est pas certain que le tronçon de voie suivant soit libre). Ici se révèle le caractère polymorphique de la sûreté de fonctionnement, qui se décline selon de nombreux sous-aspects tels que disponibilité, fiabilité, robustesse, testabilité, maintenabilité, etc., ces aspects pouvant parfois être antinomiques. Le réseau ferroviaire le plus fiable est, par exemple, celui où tous les trains sont à l'arrêt (état sûr), mais c'est aussi celui ayant la plus faible disponibilité.

2.3. *Redondance et diversité*

En dépit de cette grande hétérogénéité de situations, deux grandes idées traversent la tolérance aux fautes : celles de *redondance* et de *diversité*. La redondance consiste à construire dans un système une sorte de « surcapacité » de service ou d'information, qui peut être partielle (un code correcteur d'erreur, la vérification en ligne d'une condition de validité), ou totale (embarquer deux calculateurs là où un seul suffit en fonctionnement normal). La diversité consiste à s'assurer que la « surcapacité »

ainsi ajoutée défaille indépendamment du service non redondant, ce que Laprie *et al.* dénomment comme « l'indépendance des redondances par rapport aux processus de création et d'activation des fautes » (Laprie, 1996). Le concept de diversité est essentiel, et est étroitement lié aux fautes que l'on cherche à tolérer, ceci en vertu d'une constatation très simple : aucun élément redondant ne peut protéger des fautes qui le conduisent à défaillir de la même manière et au même moment que le système qu'il double (notion de défaillance de mode commun). C'est ainsi que le vol 501 d'Ariane 5 a échoué en 1996 suite à une défaillance de son système de référence inertielle, en dépit de ses deux calculateurs embarqués (ESA 1996). La cause de la défaillance était une faute du logiciel et impactait les deux calculateurs de la même manière, ceux-ci étant rigoureusement identiques, et faisant tourner le même programme. S'il s'était agi d'une avarie matérielle, les conséquences auraient été tout autres, le second calculateur prenant la main, et masquant ainsi la faute. Dans un tout autre contexte, on observe le même phénomène lors des épidémies virales qui secouent régulièrement le réseau Internet. La réplication à l'identique des serveurs, si elle facilite grandement la maintenabilité des systèmes et permet de bénéficier d'effets d'échelle en termes de coûts, est impuissante à contrer ce genre de menaces puisque les mêmes vulnérabilités existent dans les machines répliquées. Cet état de fait explique l'intérêt accru apporté à la notion d'« infodiversité » dans les grands réseaux informatiques (Deswarte, 2003). On voit ici qu'à chaque type de redondance (matérielle dans le cas d'Ariane 501, de conception dans l'avionique civile, d'information avec les codes correcteurs d'erreur), correspond une classe de fautes, ou *modèle de fautes*, qui peuvent être tolérées.

2.4. *Modèles de fautes*

Nous venons de le voir avec l'exemple d'Ariane 501, il est essentiel de bien cerner la nature des fautes que l'on cherche à tolérer afin de construire des mécanismes de tolérance qui contiennent la diversité appropriée pour être efficaces. De nombreuses classifications des fautes existent : on peut par exemple les classer en fonction de la phase du cycle de vie du système au cours duquel elles apparaissent (fautes de conception — le cas du vol Ariane 501, faute opérationnelle), de leur origine par rapport aux frontières du système (internes, externes), de leur nature phénoménologique (fautes physiques, fautes dues à l'homme), de leur persistance (fautes transitoires, fautes permanentes), etc. Chacune de ces classes de fautes appelle un traitement spécifique de tolérance aux fautes. Ainsi les fautes de conception ne peuvent être tolérées par une simple réplication matérielle à l'identique, c'est pourquoi en avionique civile des programmes répondant aux mêmes spécifications sont développés en plusieurs versions par des équipes différentes, ce qu'on appelle en anglais le *N-version programming* (Avizienis *et al.,* 1984).

Dans un système composite, dans lequel les défaillances des composants sont considérées comme les fautes à combattre, on distingue plusieurs types de défaillances (des fautes pour le système global) selon le comportement du composant défaillant :

– *Silence sur défaillance :* un composant à *silence sur défaillance*[3] se comporte soit conformément à sa spécification, soit cesse toute interaction avec les autres composants du système (ce qu'est une interaction dépendra de la nature du système considéré : envoyer des messages dans un système de communication par messages, interagir par mémoire partagée dans un système à mémoire partagée) ;

– *Défaillance incontrôlée :* un composant qui suit un modèle de *défaillances incontrôlées* peut se comporter de manière totalement arbitraire. Dans un système de communication par messages, un tel composant peut envoyer des messages avec un contenu incorrect (défaillance en valeur), des messages trop tôt ou trop tard (défaillance temporelle), voir même des messages non prévus (Powell, 1991). Ces défaillances sont parfois appelées des défaillances *byzantines*.

2.5. *Les mécanismes de tolérance aux fautes*

Les mécanismes de tolérance aux fautes font typiquement intervenir les deux primitives suivantes :

– *La détection d'erreur* permet au système d'identifier un état erroné au cours de son fonctionnement. La détection s'accompagne éventuellement d'un diagnostic pour évaluer le degré de propagation de l'erreur, et l'importance des dommages causés ;

– *Le recouvrement d'erreur* consiste à ramener le système vers un fonctionnement sûr (mais éventuellement dégradé) en remplaçant l'état erroné et donc potentiellement dangereux, par un état sans erreur.

On distingue trois formes de recouvrement :

– *Recouvrement par reprise :* le système est ramené dans un état antérieur à partir duquel il reprend son fonctionnement (un exemple de la vie quotidienne consiste à rallumer son ordinateur de bureau après qu'il se soit bloqué, et à travailler à partir de la dernière version sauvegardée de son travail) ;

– *Recouvrement par poursuite :* le système est amené dans un nouvel état, connu à priori, à partir duquel il peut continuer son fonctionnement (souvent de manière dégradée). Ce mode de recouvrement est souvent très dépendant de l'application ;

– *Recouvrement par compensation :* la redondance contenue dans l'état erroné suffit à éliminer l'erreur et à poursuivre l'exécution.

Si l'on estime que la faute est encore présente après le recouvrement (par exemple dans le cas d'une faute *persistante*), et que la probabilité d'une nouvelle activation

3. Un distinction est parfois introduite dans les systèmes distribués entre *silence sur défaillance* (*fail-silent*) et *défaillance par arrêt* (*fail-stop*). Nous ne pouvons faute de place rentrer dans ces détails mais nous renvoyons le lecteur à par exemple (Guerraoui *et al.,* 2006) pour une discussion approfondie de ces aspects.

n'est pas suffisamment faible, un traitement de faute constitué d'un *diagnostic* suivi d'une *passivation* s'impose en complément.

La redondance est souvent utilisée pour réaliser détection d'erreur et recouvrement : des erreurs peuvent ainsi être détectées en comparant les résultats fournis par plusieurs répliques, ou en vérifiant la conformité d'un résultat fourni par un composant par rapport à des contraintes exprimées sur son fonctionnement (on parle alors d'un composant *autotestable*). La programmation par contrat (Meyer, 1997) peut par exemple être vue comme une forme de redondance dans lequel le comportement d'un module logiciel est exprimé deux fois de manières différentes : par son implémentation d'une part, et par les pré- et post-conditions de son contrat d'autre part. Le principe de l'*extreme programming* (Beck, 1999) qui consiste à faire coder à un même programmeur des tests unitaires avant l'implémentation d'un module relève de la même stratégie. Là encore, le programmeur exprime de deux manières différentes son intention, d'abord sous forme de test, puis dans une implémentation. L'activation du test révèle toute incohérence entre les deux. Enfin la redondance joue aussi un rôle essentiel dans le recouvrement par compensation (par exemple plusieurs répliques d'un même service votent pour décider du résultat).

2.6. *Capture d'état*

Dans une architecture répliquée, le recouvrement arrière nécessite le plus souvent soit de transférer l'état d'une réplique à une autre, soit de capturer l'état d'une réplique sur support stable, pour pouvoir le restaurer ultérieurement. La capture d'état consiste alors à obtenir suffisamment d'information sur le système concerné (entrées en attente, processus internes, ressources utilisées, valeurs des données traitées, etc.) à un instant t de son fonctionnement pour pouvoir reprendre ultérieurement l'exécution du système de manière cohérente avec le passé déjà observé. Considérons par exemple la figure 3. Un état s est capturé sur une réplique **X** après que le comportement A ait été

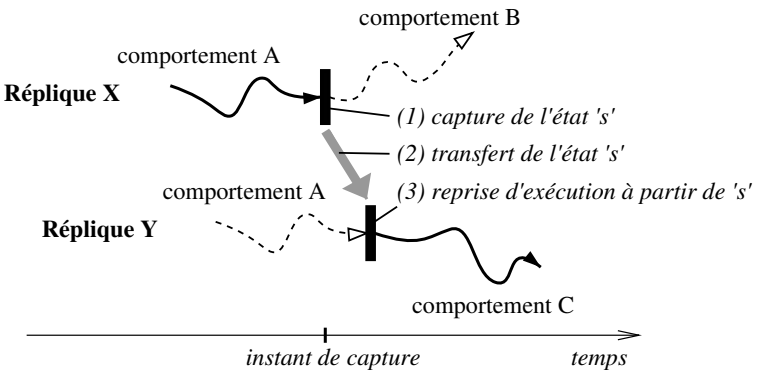

Figure 3. *Capture d'état : $A + B$ et $A + C$ doivent être des comportements valides*

observé. Suite à la défaillance par arrêt de **X**, supposons que le système soit ensuite redémarré sur une réplique **Y** en utilisant cet état s. C est le comportement observé après la reprise. Pour exclure toute incohérence du comportement observé, la séquence de comportement $A+C$ doit être un comportement attendu[4] du système. Si B est le comportement *qui aurait été observé* si la réplique **X** avait poursuivi son exécution après la capture de s, cet impératif de cohérence n'implique pas que C doive être identique à B. Dit de manière figurée, s doit contenir la promesse d'un futur compatible (ici C) avec le passé déjà observé (A).

On notera dans ces quelques remarques l'importance de la position de l'observateur des différents comportements. C'est ce qu'attend l'observateur du système qui détermine si $A + B$ est un comportement acceptable, et donc si l'état s contient suffisamment d'information pour une reprise cohérente. L'idée qu'a un observateur d'un comportement cohérent est aussi intimement liée à ses capacités d'observation. Si celui-ci est « aveugle » à certains aspects du comportement du système, ceux-ci n'influenceront pas sa perception de la cohérence, et pourront donc être ignorés dans s.

3. Systèmes répartis et tolérance aux fautes

Nous venons de présenter les grands principes de la tolérance aux fautes. Nous allons maintenant nous intéresser à l'utilisation de ces principes dans le cas particulier des systèmes informatiques répartis. Répartition physique et tolérance aux fautes sont en effet très souvent associées, la réplication d'un même service de manière répartie permettant d'assurer redondance et indépendance des modes de défaillance. L'indépendance des modes de défaillance dépend alors de la nature de la distribution. Deux répliques installées physiquement dans le même bâtiment ne seront pas, par exemple, tolérantes à un incendie qui ravagerait le bâtiment.

La distribution impose cependant de coordonner l'activité des répliques, à la fois d'un point de vue fonctionnel, pour garantir la cohérence du service fourni, et du point de vue de la tolérance aux fautes, pour assurer une détection et un recouvrement d'erreur répartis (voir la section 2.5). Dans cette section, nous mettons en lumière deux aspects particuliers de cette coordination répartie : la synchronisation des communications, et la coordination des captures d'état.

3.1. *Synchronisation des communications*

La coordination de l'activité des répliques requiert le plus souvent l'utilisation de primitives de synchronisation des communications qui soient à la fois tolérantes aux fautes, et garantissent un certain nombre de propriétés (fiabilité de communication, ordre d'arrivée des messages, exclusion mutuelle vis-à-vis d'une ressource, etc.). Ces primitives diffèrent par le degré de contrôle imposé aux répliques, leur granularité,

4. Voir la note 1.

leur robustesse, leur prédictibilité. Deux d'entre elles ont reçu une attention particulièrement soutenue du fait de leur caractère fondamental, et de leur difficulté de réalisation : la *diffusion atomique* et le *consensus* (Lamport *et al.,* 1982, Or, 1983, Fischer *et al.,* 1985, Fetzer *et al.,* 1995, Chandra *et al.,* 1996, Guerraoui *et al.,* 1997b, Herlihy *et al.,* 1999, Guerraoui *et al.,* 2004). La diffusion atomique permet à des entités réparties d'échanger des messages par diffusion (*broadcast*), tout en garantissant le même ordre d'arrivée à toutes les destinations. Elle joue très souvent un rôle essentiel dans la mise en œuvre des stratégies de réplication répartie (Powell, 1991). Le consensus permet à ces mêmes entités de s'accorder sur une même valeur de sortie, à partir de l'ensemble des valeurs proposées par chacune des entités. Notons que diffusion atomique et consensus ont la même « force de synchronisation » (ce sont des primitives équivalentes), l'une permettant d'implémenter l'autre et vice-versa.

La possibilité de réaliser de façon tolérante aux fautes telle ou telle primitive de synchronisation dans un système réparti est en grande partie conditionnée par la nature des communications entre les répliques. Les architectures rencontrées vont ainsi de processeurs synchronisés sur la même horloge (Taylor *et al.,* 1989) (*lockstep processors*), dont les sorties sont comparées, à des systèmes utilisant des réseaux locaux à jetons (Powell, 1991), en passant par les liaisons point à point rencontrées dans l'avionique civile (norme ARINC 429 (AEE n.d.)).

Ces différentes architectures se répartissent en deux grands types : les systèmes répartis *synchrones*, et ceux dits *asynchrones*. Un système *synchrone* est un système dans lequel les latences de communication et les vitesses respectives d'exécution des entités réparties sont connues. Un système asynchrone est un système dans lequel l'une de ces informations au moins n'est pas garantie. Systèmes synchrones et asynchrones sont en fait les deux extrêmes d'un spectre continu de propriétés. Il existe ainsi plusieurs modèles de synchronie partielle, comme l'asynchronie temporisée (*Timed Asynchronous Systems*) (Fetzer *et al.,* 1995).

Les systèmes asynchrones, parce qu'ils imposent moins de contraintes sur la plateforme de communication, sont plus flexibles, moins lourds à déployer, plus faciles à reconfigurer et à faire évoluer. Ils souffrent cependant d'un handicap majeur en termes de tolérance aux fautes : ils ne permettent pas de résoudre des problèmes essentiels comme le consensus ou la diffusion atomique dès lors qu'au moins un des processus répartis est autorisé à défaillir par arrêt[5] (Fischer *et al.,* 1985). Intuitivement, cette impossibilité résulte du fait que dans un système asynchrone, il n'est pas possible de distinguer un système extrêmement lent d'un système qui a défailli.

En pratique ce verrou est contourné en ajoutant des hypothèses de synchronie partielle (comme des détecteurs d'erreur imparfaits (Chandra *et al.,* 1996)), qui ne sont alors pas vérifiées avec une probabilité de 1, mais lèvent le résultat d'impossibilité

5. Notons que ce résultat d'impossibilité vaut pour les algorithmes déterministes. Il est possible de résoudre le problème du consensus dans un environnement asynchrone de façon *probabiliste* dès lors que l'on autorise les processus à générer des valeurs aléatoires (Or, 1983).

en factorisant dans un composant particulier du système la quantité de synchronisme nécessaire à la résolution de la tolérance aux fautes.

3.2. *Coordination des captures d'état*

Dans un système réparti, la capture d'état de chacune des entités réparties du système (souvent alors appelées *nœuds*) revêt un degré supplémentaire de complexité parce qu'elle doit assurer la cohérence globale du système, en plus de la cohérence locale dont nous avons déjà parlé (section 2.6). Il n'est pas acceptable, par exemple, de redémarrer un nœud dans un état qui *annule* pour ce nœud certaines des interactions qu'il a pu avoir avec d'autres nœuds, si les effets de ces interactions continuent d'exister pour le reste du système. C'est le cas par exemple lorsque des messages ont été reçus d'un expéditeur qui considère ne les avoir jamais envoyés. Ces messages sont alors dits *orphelins*. Sur la figure 4, deux processus répartis échangent les messages m_1 et m_2, tout en capturant leurs états aux points a, b, c et d. Du fait du message m_1 l'état global (a, c) n'est pas un état cohérent. Dans cet état, le processus P_2 « considère » avoir reçu le message m_1, alors que P_1 « considère » ne pas l'avoir envoyé, ce qui ne peut pas se produire dans une exécution valide du système. L'état (b, d) en revanche est cohérent : m_1 a été envoyé et reçu ; m_2 a été envoyé, sans avoir encore été reçu (m_2 est alors dit *en transit*). La présence de messages en transit est inhérente au fonctionnement normal d'un système réparti, et ne remet donc pas en cause la cohérence de l'état (b, d).

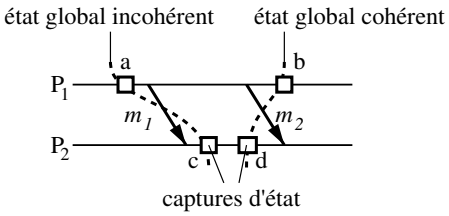

Figure 4. *Cohérence d'un état réparti*

Plus généralement, un état global réparti est cohérent si c'est un état qui *peut* se produire au cours d'une exécution sans défaillance du système. Ce problème de cohérence des états répartis a été reconnu très tôt comme un point essentiel de la tolérance aux fautes des systèmes répartis (Chandy *et al.,* 1985, Koo *et al.,* 1987). En effet, lorsque des processus répartis ne coordonnent pas leurs captures d'état, il peut arriver qu'aucune combinaison des captures locales ainsi obtenues ne permette de reconstruire un état global cohérent. Dans ce cas, la défaillance d'un seul processus entraîne le redémarrage, dans son état initial, du système tout entier. Ce phénomène est appelé « effet domino » (Randell, 1975). Eviter l'effet domino nécessite de coordonner (explicitement ou implicitement) les différentes opérations de capture et de reprise, et

touche donc aux problématiques de synchronisation. Il existe de nombreux protocoles de capture d'état réparti (*checkpointing protocols*) qui abordent ce problème (Johnson *et al.,* 1987, Strom *et al.,* 1988, Netzer *et al.,* 1995, Baldoni *et al.,* 1997, Elnozahy *et al.,* 2002).

4. Pourquoi des intergiciels pour la tolérance aux fautes ?

Les mécanismes de tolérance aux fautes qui permettent de synchroniser les communications de répliques entre elles, ou d'opérer la capture d'état d'un ou plusieurs nœuds s'expriment pour la plupart en termes génériques, indépendants de la nature de l'application à laquelle ils s'appliquent (avec une exception notable de la *reprise avant*, très dépendante du domaine d'application). Ainsi des algorithmes de capture d'état réparti comme ceux présentés dans (Chandy *et al.,* 1985) ou (Koo *et al.,* 1987) sont décrits en termes de *processus*, *messages*, *canaux*, *machines à états finis*, *transitions*, etc. Dès lors que les éléments d'une application concrète qui correspondent à ces concepts abstraits ont été identifiés et rendus observables et contrôlables, il est possible de factoriser la tolérance aux fautes dans des briques logicielles réutilisables. Selon la nature et la richesse des « briques » fournies, on pourra parler de « kit de développement », de « canevas » (*framework*), d' « intergiciel » ou de « plate-forme ».

Pour ne pas alourdir notre présentation, nous parlerons dans cet article de plate-forme ou d'intergiciel, en gardant en mémoire le caractère plus ou moins étendu que peuvent revêtir ces notions. La notion la plus ancienne, celle de plate-forme, désigne un ensemble de logiciels dont l'objectif est de servir de fondation au développement et au fonctionnement de nouvelles applications. La notion d'intergiciel (*middleware* en anglais) est, elle, beaucoup plus récente et désigne plus particulièrement un ensemble de services réalisés entre le système d'exploitation et l'application. Parmi les intergiciels les plus populaires ces dernières années, l'on peut citer les intergiciels orientés communication, notamment ceux basés sur la norme CORBA (OMG 2002a). Les intergiciels de ce type servent de « glu » à des morceaux d'application s'exécutant sur des machines distantes. Ils masquent une partie de la complexité de la programmation répartie en permettant à un développeur de manipuler de la même manière des entités locales et distantes.

4.1. *Quels problèmes ? Quels objectifs ?*

Les différents aspects qu'ont cherché à aborder les intergiciels pour la tolérance aux fautes peuvent être approximativement classifiés de la manière suivante :

– *Synchronisation :* un intergiciel de communication permet de factoriser dans un composant unique les primitives de synchronisation répartie qui sont nécessaires à la réalisation de mécanismes de tolérance aux fautes. Ces primitives de synchronisation garantissent le plus souvent des propriétés d'ordre (diffusion atomique, diffusion causale) et d'intégrité sur les messages (communication de groupes) ;

– *Capture d'état :* la capture de l'état d'une réplique présente le plus souvent un aspect *données* (quels attributs, quels objets sont gérés par l'application) et un aspect *contrôle* (quels traitements sont exécutés). Lorsqu'une application revient régulièrement dans des états « d'attente » dans lesquels aucun traitement n'est en cours (c'est le cas par exemple d'une application client/serveur qui sérialise le traitement des requêtes qu'elle reçoit), il est possible de limiter l'état capturé à sa partie donnée, en ne prenant de captures qu'entre deux traitements. Lorsqu'en revanche une application parallélise le traitement des requêtes reçues, il n'est plus possible d'ignorer la partie *contrôle* de son état[6]. Il est alors nécessaire de sauvegarder l'état d'avancement des traitements internes en même temps que les données traitées ;

– *Détection d'erreur :* la détection d'erreur consiste à identifier au plus tôt des situations anormales de fonctionnement pour pouvoir y réagir de manière adaptée. Ce type de fonctionnalité existe au moins de manière minimale dans la plupart des substrats d'exécution sous la forme de *codes de retour d'erreur* ou d'*exceptions*. Des approches plus élaborées ont été proposées, notamment à base d'architectures réflexives, pour faciliter le durcissement d'un système. Nous en dirons quelques mots dans la section 5.3.

Du fait de leur grande complexité, ces trois aspects ont tout d'abord été abordés indépendamment les uns des autres, un intergiciel s'intéressant plus spécifiquement à la synchronisation, un autre à la capture d'état. Au fur et à mesure des avancées des recherches, les propriétés tenant à la *transparence* et à la *flexibilité* des intergiciels proposés sont apparues comme essentielles pour permettre une mise en pratique aisée de ces mécanismes.

Dans le contexte de la tolérance aux fautes, la notion de transparence renvoie à la possibilité pour un développeur d'application et un développeur de mécanismes de tolérance aux fautes de travailler indépendamment l'un de l'autre. Par exemple, la problématique de l'obtention de l'état peut-elle être abordée de manière orthogonale au développement de l'application ? Idéalement, le développeur d'application doit pouvoir tout ignorer de la problématique de la tolérance aux fautes (ici la capture d'état), qui lui est donc « transparente ». De même, le développeur de tolérance aux fautes doit pouvoir concevoir ses mécanismes sans avoir à assimiler la logique applicative du système auquel ils s'appliqueront. La transparence est donc intimement liée à la notion de *découplage* (pouvoir ajouter la tolérance aux fautes indépendamment du développement de l'application voire *a posteriori*), et de *séparation des préoccupations* (*separation of concerns*).

La flexibilité désigne la facilité à adapter des mécanismes de tolérance aux fautes à de nouveaux besoins, à un nouveau contexte d'exécution, à permettre dans sa forme la plus avancée leur modification durant l'exécution du système. Par exemple, pour

6. Les instants sans traitement peuvent d'une part ne plus se produire du tout dans le cas d'une charge élevée. D'autre part, même lorsqu'ils se produisent (spontanément ou de manière provoquée en bloquant les requêtes entrantes), leur régularité n'est alors plus assez fine pour empêcher l'effet « domino », dont nous avons parlé dans la section 3.2.

un mécanisme de capture d'état, dans quel format l'état est-il sauvegardé ? le système peut-il être redémarré dans un contexte d'exécution différent de celui de la prise d'état ? sur un nouvel OS ? dans une implémentation écrite dans un nouveau langage ?

Notons que ces problèmes, qui concernent la *réutilisabilité*, ou la *portabilité*, restent aujourd'hui des domaines de recherche très actifs et rejoignent des sujets plus généraux comme la séparation des préoccupations (*separation of concerns*), ou les problèmes de composabilité en génie logiciel.

4.2. *Types d'architectures pour la tolérance aux fautes*

Plusieurs architectures d'intergiciel ont successivement été proposées pour aborder les aspects que nous venons de présenter. En s'inspirant de la classification proposée par Briot et Guerraoui (Briot *et al.,* 1996) pour l'intégration de la distribution et du parallélisme à l'orienté objet, l'on peut schématiquement distinguer quatre grands types d'approches : *l'approche explicite*, *l'approche intégrée*, *l'approche par interception*, et *l'approche réflexive*. Notons que cette classification s'étend au-delà de la simple tolérance aux fautes, et peut s'appliquer de façon générale à toute préoccupation transversale (*crosscutting concerns* en anglais).

– *Approche explicite :* les mécanismes de tolérance aux fautes sont présentés sous la forme d'une API (*Application Programming Interface*, interface de programmation applicative) spécifiquement conçue pour la tolérance aux fautes (Birman, 1985, Peterson *et al.,* 1989, Amir *et al.,* 1992). Cette API peut par exemple être fournie par une bibliothèque, par un système opératoire spécifique, ou au travers d'un service réparti (comme les services CORBA). Les intergiciels de ce type n'offrent quasiment aucune transparence, et leur utilisation n'est pas aisée. Ils fournissent en effet des interfaces qui leur sont propres, plutôt de bas niveau, qui se présentent typiquement comme un ensemble de fonctions en C pour manipuler des brins d'exécution, des messages génériques, des groupes de processus, des primitives de communication, de multidiffusion, etc. Ils nécessitent une très bonne connaissance des protocoles mis en jeu, et ne peuvent pas facilement être utilisés par des non spécialistes de la tolérance aux fautes. En revanche, certains d'entre eux (les plus récents) sont extrêmement flexibles, en offrant de nombreux mécanismes d'extension et de reconfiguration (Mishra *et al.,* 1993, van Renesse *et al.,* 1996, Hiltunen *et al.,* 1998).

– *Approche par intégration :* avec l'approche par intégration, une plate-forme de développement « générique » non tolérante aux fautes (un bus à objets CORBA, un système opératoire POSIX) est durcie en construisant à l'intérieur de la plate-forme des mécanismes de tolérance aux fautes (par exemple de diffusion atomique, ou de capture d'état). L'interface originale de la plate-forme n'est pas modifiée (à l'exception de quelques extensions de configuration), ce qui permet d'atteindre un haut degré de transparence. En interne les parties de la plate-forme qui traitent explicitement de la tolérance aux fautes et celles qui n'en traitent pas sont fusionnées et ne peuvent être facilement séparées les unes des autres, ce qui nuit à leur flexibilité. Les plates-formes

DELTA-4 (Powell, 1991) et ELECTRA (Maffeis, 1995) sont des exemples de ce second type d'intergiciel.

– *Approche par interception :* l'approche par interception consiste, comme l'approche par intégration, à étendre une plate-forme non tolérante aux fautes, mais en utilisant des mécanismes spécifiques à la plate-forme pour intercepter certains de ses traitements et événements internes : appels système, émission/réception de requêtes en utilisant les mécanismes d'invocation dynamique d'un bus à objets (dans CORBA le *Dynamic Invocation Interface* ou DII et le *Dynamic Skeleton Interface* ou DSI) (Felber *et al.,* 1996), etc. Ce type d'architectures conserve la transparence apportée par l'intégration, et ré-introduit en même temps la flexibilité de l'approche explicite, en rendant accessibles les éléments qui, à l'intérieur d'une plate-forme, implémentent la tolérance aux fautes. L'utilisation d'intercepteurs permet de modifier plus facilement les mécanismes inclus dans la plate-forme ou de les réutiliser sur d'autres implémentations qui proposent la même interface d'interception.

– *Approche réflexive :* l'approche réflexive étend l'approche par interception en la formalisant conceptuellement et en en augmentant les capacités. Nous nous intéresserons ici plus particulièrement à la *réflexivité calculatoire* (*computational reflection*), définie comme la capacité pour un système d'appliquer à lui même ses propres capacités de calcul (Smith, 1984, Maes, 1987). D'abord anecdotique, la réflexivité est progressivement apparue comme une solution élégante à des problèmes récurrents d'adaptabilité et de réutilisation, notamment avec les recherches sur le langage de programmation LISP. Les architectures réflexives permettent de clairement séparer l'activité fonctionnelle d'un système, réalisée dans ce qu'on appelle alors le *niveau de base*, de ses mécanismes non fonctionnels, implémentés au *méta-niveau* (figure 5).

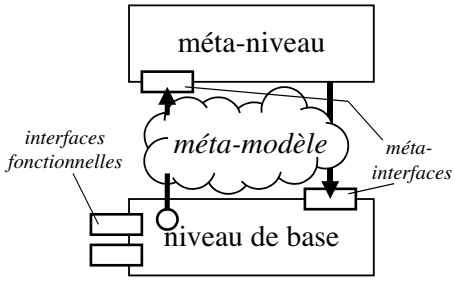

Figure 5. *Architecture d'un système réflexif*

L'interaction entre les deux niveaux s'opère au travers d'une représentation du système, appelée *méta-modèle*, qui rend visible et permet de modifier au méta-niveau les éléments constitutifs du système considéré, à la fois en termes de représentation (statique) et d'exécution (dynamique) (Briot *et al.,* 1996). Pour un système orienté-objet, le *méta-modèle* pourra donner par exemple accès aux classes, objets, et associations

du système, et permettra de les observer *et* de les modifier au méta-niveau. L'approche réflexive peut être vue comme une formalisation et un enrichissement de l'approche par interception, en proposant une interface de programmation (usuellement appelée *méta-interface*) qui permet de manipuler de manière disciplinée et contrôlée les éléments liés au fonctionnement d'une plate-forme généraliste (appels de méthodes distantes, cycle de vie des objets, état de la plate-forme et des objets qui vivent au-dessus d'elle).

L'approche réflexive pour la tolérance aux fautes consiste à ajouter les mécanismes de tolérance aux fautes sous forme de *méta-programmes*, exprimés en utilisant les éléments génériques du modèle de programmation de la plate-forme originale. Un méta-programme extrêmement simple serait par exemple : « A chaque invocation distante de méthode, sauvegarder l'état de l'objet appelant ». Les concepts *d'invocation distante*, *d'objet appelant* ou *d'état d'un objet* se retrouvent dans tout modèle de programmation orienté objet distribué, mais sont *génériques* en ce qu'ils sont indépendants de tout domaine d'application. En s'appuyant sur cette généricité un méta-programme peut s'exprimer indépendamment de toute application particulière, et s'appliquer *a posteriori* de façon cohérente à toute une famille de systèmes. C'est là la grande force de la réflexivité qui lui permet d'assurer à la fois transparence et flexibilité des mécanismes, tout en proposant un modèle de programmation clair, cohérent, et robuste.

5. Exemples d'intergiciels pour la tolérance aux fautes

Pour chacun des grands types d'architecture que nous venons de décrire (*explicite*, *intégrée*, *par interception*, *réflexive*), nous présentons maintenant des exemples d'intergiciel pour la tolérance aux fautes. L'objectif est ici de montrer comment chaque grand type a évolué à partir du précédent, et comment les propriétés de *transparence* et de *flexibilité* ont progressivement gagné une place prépondérante dans l'organisation de ces plates-formes.

5.1. *Les pionniers : les intergiciels explicites*

5.1.1. *Intergiciels de synchronisation :* ISIS, PSYNC, *etc.*

ISIS, PSYNC, TRANSIS sont des kits de développement d'applications réparties tolérant les fautes, basés sur des protocoles de multidiffusion qui permettent l'ordonnancement cohérent d'évènements distribués.

Le premier d'entre eux, ISIS, a été développé autour de Kenneth Birman à l'Université de Cornell sur une période allant de 1981 (début du projet) à 1998 (arrêt de la commercialisation d'ISIS par la société *Stratus Computer, Inc.*). La première version d'ISIS fournit des services de multidiffusion tolérants aux défaillances par arrêt (Birman, 1985). Chacun de ces services impose des contraintes différentes sur l'ordonnancement respectif des messages à la réception (ordre causal ou total, sur la totalité ou seulement un sous-groupe de messages par exemple). Le modèle de développement

d'ISIS est basé sur la notion d'*objet résilient*, une entité logicielle répartie organisée en un certain nombre de répliques fonctionnant selon un mode de réplication passive.

Dans la foulée d'ISIS ont été développées d'autres infrastructures similaires, notamment PSYNC (Peterson *et al.,* 1989) ou TRANSIS (Amir *et al.,* 1992).

La limitation des primitives de multidiffusion proposées par ces différentes plates-formes, et l'impossibilité d'adapter leurs propriétés en fonction du contexte opérationnel ont été à l'origine de projets qui ont plus particulièrement recherché la flexibilité tout en gardant les capacités des plates-formes précédentes. CONSUL (Mishra *et al.,* 1993), basé sur PSYNC, CACTUS (Hiltunen *et al.,* 1998), successeur de CONSUL, HORUS (van Renesse *et al.,* 1996) et ENSEMBLE (van Renesse *et al.,* 1998), successeurs d'ISIS, sont de ceux-là.

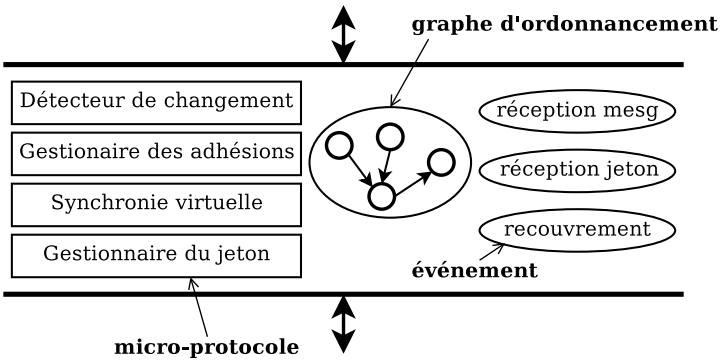

Figure 6. *Protocole réalisé par microprotocoles (adapté de (Hiltunen* et al.*, 1998))*

Ces différents projets introduisent la notion de *microprotocole* (figure 6). Les microprotocoles sont des modules logiciels qui transforment un flux d'événements en entrée en un flux d'événements en sortie en garantissant diverses propriétés d'ordre ou de robustesse. Ils peuvent être assemblés en de multiples combinaisons pour créer des piles de communication personnalisées selon les besoins de l'application. Deux questions en particulier sont abordées pour la réalisation de ces microprotocoles :

1) Quelle interface générique choisir pour pouvoir composer librement les différents microprotocoles ?

2) Comment prendre en compte les contraintes de dépendance entre les différents microprotocoles ?

Pour répondre à la première question, la plupart de ces projets choisissent une forme d'interaction par événements : réception, envoi de message, détection d'une défaillance, entrée d'un nouveau membre dans un groupe, etc. Le second problème est abordé en décrivant les dépendances entre protocoles sous forme soit d'un ordre

partiel (deux protocoles sont reliés si le fonctionnement correct de l'un assure le fonctionnement correct du second (Mishra *et al.,* 1993)), ou de manière plus élaborée sous forme de « contrats » conditions garanties (van Renesse *et al.,* 1998).

Les intergiciels pour la synchronisation et de la détection d'erreur ne sont pas les seuls intergiciels pertinents pour la tolérance aux fautes à avoir fait l'objet d'une approche explicite. L'on peut aussi distinguer des intergiciels explicites de capture d'état, que nous abordons dans la section suivante.

5.1.2. *Intergiciels pour la capture d'état*

Les approches explicites pour la capture d'état sont pour la plupart implémentée au niveau langage. Ils requièrent de la part du programmeur d'indiquer explicitement quelles données il souhaite gérer de manière persistante. Le *Persistent State Service* (PSS) de CORBA, utilise par exemple un langage de description de l'état à sauvegarder appelé PSDL (*Persistent State Description Language*) pour décrire les attributs de chaque objet réparti qui doivent pouvoir être rendus persistants (OMG 2002d). Un autre exemple est le langage PS-ALGOL, qui permet au programmeur de déclarer les objets qu'il souhaite stocker dans une base de donnée persistante (Atkinson *et al.,* 1983).

Parce qu'ils utilisent une description de l'état à sauvegarder de haut niveau (basée sur le système de types de CORBA pour PSS, ou celui d'Algol pour PS-ALGOL), et font appel à des formats de mise à plat indépendants des intergiciels sous-jacents, ces intergiciels permettent de transférer des états entre des contextes d'exécution (OS, matériel) hétérogènes. Le *Persistent State Service* de CORBA permet même d'échanger un état entre deux implémentations d'un même service écrites dans des langages différents. Parce qu'elles sont explicites, ces techniques nécessitent cependant de la part du concepteur de préciser en amont du développement quelles données devront être gérées de manière persistante pour permettre des reprises cohérentes. Cette contrainte rend l'adaptation des choix de persistance difficile en cours de projet. Elle ne permet pas non plus de gérer la capture d'état de manière totalement transparente vis-à-vis des aspects fonctionnels de l'application. Enfin, l'approche explicite à la capture d'état est mal adaptée à la capture de l'aspect « contrôle » de l'état d'une application (quels traitements sont en cours ? avec quels degrés d'avancement ?), qui est intrinsèquement lié au substrat d'exécution et se prête mal à toute approche déclarative.

5.1.3. *Conclusion sur l'approche explicite*

L'un des grandes forces de l'approche explicite et de permettre la réalisation de mécanismes spécifiquement adaptés à une application, ou à des types de fautes précisément identifiés, ceci de façon concise et efficace. Cette propriété est très intéressante dans des contextes opératoires particulièrement contraints (systèmes embarqués, par exemple) où adéquation à l'application et performance sont incontournables. Les considérations de transparence passent dans ces systèmes souvent au second plan, en particulier du fait de leur taille relativement réduite.

Le principal défaut de l'approche explicite est qu'elle n'est pas transparente au programmeur d'application. Elle impose donc une connaissance relativement fine des mécanismes non fonctionnels à réaliser, et tend à encourager des architectures dans lesquelles tolérance aux fautes et aspects fonctionnels sont inextricablement mêlés. Son emploi peut par conséquent fragiliser un système tout entier du fait d'une connaissance approximative des algorithmes sous-jacents par les équipes de développement. L'approche explicite peut aussi rendre le développement et la maintenance de grands systèmes particulièrement malaisés en n'introduisant pas de séparation claire entre préoccupations fonctionnelles et tolérance aux fautes.

5.2. *Intergiciels par intégration et interception*

Les limitations de l'approche explicite, aussi bien pour la synchronisation que pour la capture d'état, ont encouragé plusieurs projets visant a introduire de façon transparente des mécanismes de tolérance aux fautes dans des substrats d'exécution génériques (intergiciel de communication, systèmes d'exploitation). Ces projets, en particulier pour les aspects de synchronisation et de communication, sont en fait très souvent basés sur un kit explicite, mais en masquent la complexité. Ce sont les approches par *intégration* et par *interception*, que nous détaillons dans les deux sections suivantes.

5.2.1. *Synchronisation et communication de groupe*

Le développement d'intergiciels orientés objet pour la programmation répartie tels que CORBA (OMG 2002a) ou RMI (Sun 2003) a motivé plusieurs travaux cherchant à combiner la facilité de développement offerte par ces intergiciels (réutilisabilité, portabilité, interopérabilité, intégration des anciennes applications) avec les mécanismes de tolérance aux fautes basés sur les kits que nous venons de présenter.

Le projet ELECTRA (Maffeis, 1995, Maffeis, 1995), est un exemple d'approche par intégration. ELECTRA utilise la plate-forme de communication de groupe HORUS, citée précédemment, pour implémenter un bus à objets conforme à la norme CORBA qui permet d'envoyer de façon transparente une requête à un groupe d'objets répliqués. La réponse d'une telle requête peut alors être sélectionnée de plusieurs façons : après l'obtention d'une majorité de réponses, après l'obtention de toutes les réponses, etc.

Un des reproches faits à ELECTRA est son couplage trop fort entre communication de groupe et la plate-forme CORBA : l'interface de gestion des groupes d'objets (création, modification, génération de références pour un groupe), n'est en effet pas accessible au programmeur d'application CORBA. Il n'est donc pas possible de changer de kit de communication de groupe, ou d'ajouter une capacité de réplication à une implémentation CORBA préexistante. Ces limitations ont été à l'origine de travaux *par interception*, comme ETERNAL (Narasimhan *et al.,* 1997) ou OGS (*Object Group Service*) (Felber, 1998), qui, en déroutant certaines interactions du système, ajoutent des capacités de réplication de manière transparente, tout en conservant une

forte modularité entre communication répartie « monointerlocuteur » et communication de groupe.

Les deux projets se distinguent cependant par leur niveau d'intrusion : ETERNAL capture les appels faits par le bus à objets sur l'interface de communication de l'OS sous-jacent et les redirige sur un service de communication de groupe, ce qui permet de rendre la réplication transparente aux clients. OGS adopte une approche par *service CORBA* : les mécanismes de communication de groupe (création, gestion, invocation, reconfiguration) sont fournis explicitement sous la forme d'objets CORBA au travers de leurs interfaces IDL. Pour rendre transparent ce service, OGS utilise les mécanismes de réception et d'invocation génériques de CORBA (respectivement DSI, *Dynamic Skeleton Interface*, et DII, *Dynamic Invocation Interface*), qui permettent d'intercepter de manière transparente les requêtes faites par un client sur un serveur, et de rediriger ces requêtes sur le service de groupe.

Ces travaux ont, entre autres, été à l'origine du standard FT-CORBA (*Fault-Tolerant CORBA*), une extension du standard CORBA qui n'intégrait pas à l'origine de considérations de tolérance aux fautes (OMG 2002b). L'approche par interception a aussi motivé l'introduction d'intercepteurs standardisés dans CORBA (OMG 2002c), qui, même s'ils sont limités (Bennani *et al.,* 2004) (par exemple, impossibilité de modifier les paramètres des requêtes), introduisent une part de réflexivité dans l'ORB.

5.2.2. *Capture d'état intégrée*

Comme pour les considérations de synchronisation distribuée, l'intégration de capacités de capture d'état à des plates-formes d'exécution permet de découpler les considérations de persistance du reste des fonctionnalités d'une application. Les plus efficaces de ces approches tirent profit des mécanismes de gestion mémoire fournis par la plupart des systèmes d'exploitation.

Capture d'état et gestion mémoire sont en effet deux problématiques très proches. Le principe consiste à enregistrer en format binaire l'espace mémoire alloué à une application sur un support stable (un disque dur typiquement). La mise à profit des mécanismes de gestion mémoire permet notamment d'arrêter le moins longtemps possible l'application dont on capture l'état. Deux mécanismes sont essentiellement utilisés : le mécanisme de protection des pages mémoire d'un OS et celui de permutation entre mémoire et disque dur (*swapping*) :

– Le mécanisme de protection des pages mémoire permet à un OS de contrôler et de modifier les droits d'accès des pages allouées à une application. Cette fonctionnalité sert essentiellement des objectifs de confinement et de protection, et peut être mise à profit pour permettre des captures d'état « à chaud » sans avoir à geler l'application cible. Par exemple, si l'application essaye d'écrire dans une page qui n'a pas encore été capturée, le mécanisme de capture d'état peut en être notifié pour réagir de façon appropriée (par exemple, en déclenchant une opération de *copy on write* (Li *et al.,* 1994)).

– Le système de permutation permet de donner accès à un espace mémoire plus grand que la mémoire physique disponible en utilisant la mémoire vive de l'ordinateur comme un cache des blocs mémoire sauvegardés sur le disque. À un instant donné, la mémoire effectivement utilisée par une application se trouve donc déjà partiellement sérialisée sur l'espace de permutation du disque. Cette sérialisation partielle augmentée des mécanismes de mise en cohérence nécessaires permet de réaliser des captures d'état particulièrement performantes. (Koo *et al.,* 1987, Ousterhout *et al.,* 1988, Espen Skoglund, 2000).

La capture d'état binaire d'une application, que ce soit par l'un ou l'autre des mécanismes présentés, doit cependant surmonter plusieurs problèmes :

1) L'image binaire de l'espace mémoire d'un processus ne contient pas toutes les informations nécessaires à la reprise de l'application : le système d'exploitation gère en effet *au nom de l'application* un certain nombre d'entités (descripteurs de fichier, verrous, canaux de communication) dont l'état est interne à l'OS et n'est pas capturé par la seule image de l'espace d'adressage.

2) Pour référencer les entités gérées en son nom par l'OS, mais aussi pour désigner des objets qui lui sont extérieurs (fichiers, serveurs distants), une application utilise des identifiants qu'elle stocke en mémoire, et qui lui sont fournis par exemple lorsqu'elle ouvre un fichier, ou lorsqu'elle crée un verrou. Ces identifiants ne sont plus valables lors d'une reprise. Même lorsque de nouvelles entités sont recréées dans l'OS en remplacement des anciennes disparues, ces nouvelles entités ont souvent des identifiants différents, qui doivent être « réappris » par l'application.

3) Certaines entités « extérieures » à l'application, comme des fichiers, ou des ressources graphiques, peuvent être perdues en même temps que l'application et ne plus être disponibles lors d'une reprise, ou posséder un état incohérent avec celui du reste de l'application. Le mécanisme de capture doit être capable de restaurer leur état de manière cohérente avec celui de l'application. Ce problème de la partie « externe » de l'état d'une application rejoint celui de la cohérence des états répartis que nous avons mentionné dans la section 3.2.

Pour résoudre le premier problème, la plupart des approches intégrées de capture d'état utilisent une stratégie par journalisation qui consiste à intercepter les appels OS faits par l'application pour en inférer l'état des objets que l'OS gère au nom de l'application (Taylor *et al.,* 1986, Russinovich *et al.,* 1993, Dieter *et al.,* 2001, Karablieh *et al.,* 2002). Pour résoudre le second problème, deux stratégies ont été proposées : soit modifier dans l'image binaire les descripteurs dont on connaît la sémantique (c'est ce que fait (Dieter *et al.,* 2001) pour les identifiants des brins d'exécution, en tirant partie du fait que la gestion du multitraitement (*multithreading*) de GNU/LINUX jusqu'à la version 2.4 du noyau est implémentée dans une bibliothèque gérée en espace utilisateur) ; soit insérer une couche de traduction des identifiants entre le système d'exploitation et l'application (Karablieh *et al.,* 2002).

La capture binaire peut donc s'avérer une solution relativement transparente, une fois surmontés les problèmes que nous venons de mentionner. Elle s'avère, cepen-

dant, très pénalisante en termes de flexibilité. Les états ainsi capturés ne sont en effet pas portables d'un système d'exploitation à l'autre, et requièrent d'utiliser des architectures matérielles et logicielles rigoureusement identiques. Cette contrainte d'homogénéité logicielle et matérielle n'est pas facile à réaliser dans de grands systèmes complexes, et se trouve de plus être particulièrement handicapante du point de vue de la tolérance aux fautes (nous renvoyons le lecteur à la section 2.3 pour une discussion sur l'importance de la diversité pour la tolérance aux fautes).

Ce manque de flexibilité de l'approche par capture binaire se retrouve lorsqu'il s'agit de porter un même mécanisme d'un système d'exploitation à l'autre, la gestion de la mémoire utilisant soit des adaptations *ad hoc* au niveau de l'OS, soit des particularités non standard du système d'exploitation choisi (nous renvoyons à (Dieter *et al.,* 1999) et (Dieter *et al.,* 2001) pour une présentation d'un portage d'une bibliothèque de capture binaire d'applications à brins d'exécution multiples de SOLARIS vers GNU/LINUX).

5.2.3. *Conclusion sur les approches par intégration et interception*

L'approches par intégration et l'approche par interception permettent d'assurer une grande transparence aux mécanismes de tolérance aux fautes, au contraire de l'approche explicite. Les deux approches différent de par leur degré d'ouverture et de flexibilité. L'approche par intégration ne donne pas accès aux mécanismes sous-jacents. Elle est donc beaucoup moins flexible, et, selon le niveau où elle est implémentée, peut rendre difficile le maintien de la cohérence du système (notamment dans le cas de la capture d'état binaire).

L'approche par interception se caractérise au contraire par une plus grande adaptabilité en rendant accessible l'interface des intergiciels ou des systèmes opératoires auxquels elle s'applique. L'approche réflexive, que nous présentons dans la section qui suit, généralise cette stratégie d'ouverture en étendant l'observabilité d'une plateforme au-delà de la seule interception et en intégrant la maîtrise du contrôle du comportement des objets fonctionnels d'un système.

5.3. *Les solutions réflexives*

L'approche réflexive pour la tolérance aux fautes a essentiellement mis à profit la notion de *protocole à méta-objets* (*Meta-Object Protocol* en anglais). Les *protocoles à méta-objets* sont utilisés dans les systèmes à la fois *orientés objet* et *réflexifs* (Maes, 1987, Kiczales *et al.,* 1991). Le méta-niveau de ces systèmes est habituellement structuré sous forme d'objets particuliers, appelés *méta-objets*, qui encapsulent le comportement réflexif du système. Un *protocole à méta-objets* décrit l'ensemble de conventions qui régissent les interactions entre *méta-objets* et *objets de base*, notamment en termes d'association (comment les méta-objets sont-ils associés aux objets de base ?), et de cycles de vie (comment les méta-objets sont-ils créés ? détruits ? comment leur cycle de vie interagit-il avec celui des objets de base ?).

D'autres techniques réflexives que les protocoles à méta-objets ont aussi été utilisées pour fournir des mécanismes de tolérance aux fautes de façon transparente, comme par exemple celles à base de compilateurs ouverts (parfois appelés compilateurs réflexifs) (Tatsubori *et al.,* 2000, Chiba, 1995), ou de réécriture source à source.

Nous donnons ici différents exemples d'intergiciels réflexifs pour la tolérance aux fautes qui utilisent ces différentes techniques, ainsi que, rapidement, un exemple concernant la mise en place d'empaquetages (*wrapper*) de confinement d'erreur par une approche réflexive sur des micronoyaux temps-réel. Nous revenons ensuite dans la section 5.4 sur les forces et limitations de ces approches, et discutons de leur extension possible en ce que nous avons appelé *la réflexivité multiniveaux*.

5.3.1. *Réplication répartie :* MAUD, GARF

MAUD (Agha *et al.,* 1992) est une plate-forme de développement répartie qui utilise une approche réflexive pour proposer des mécanismes de tolérance aux fautes modulaires, composables, et transparents. MAUD est basée sur le langage multiagents HAL, et associe à chaque acteur trois méta-objets différents : un *dispatcher* qui intercepte les messages sortants, une *receive queue* (queue de réception), qui traite les messages arrivant, et un objet *acquaintance* qui supervise avec quels autres acteurs un acteur donné est connecté. Une stratégie *en pelure d'oignons* est proposée pour permettre la composition récursive de plusieurs méta-objets. Cette stratégie impose que les hiérarchies de méta-objets du côté d'un agent expéditeur et du côté d'un agent destinataire qui communiquent par messages soient les miroirs l'une de l'autre.

GARF (Guerraoui *et al.,* 1997a) est aussi une plate-forme répartie orientée objet. Elle est écrite en SMALLTALK et utilise les caractéristiques réflexives de ce langage pour intégrer de manière transparente à l'application des mécanismes de réplication répartie. Par cette approche réflexive, GARF sépare clairement les aspects d'un système qui sont liés au parallélisme, à la distribution et à la réplication, des considérations applicatives qui peuvent continuer à être abordées selon un point de vue séquentiel et centralisé. Pour réaliser cette séparation, GARF distingue deux sortes d'objets : les *data objects* (objets donnée) qui encapsulent la logique applicative indépendamment de toute considération de distribution et les *behavioral objects* (objets comportementaux) qui interceptent les interactions interobjets en traitant plus particulièrement des aspects ayant trait aux mécanismes d'invocation (*encapsulators*) et de distribution (*mailers*). GARF est construit au-dessus de la plate-forme de synchronisation ISIS qui est rendue disponible par l'intermédiaire d'*encapsulators* et de *mailers* spécifiques. Cependant un développeur de tolérance aux fautes a toute latitude pour adapter les objets de réplication proposés par défaut par GARF à ses propres besoins, éventuellement en court-circuitant complètement ISIS.

5.3.2. *Capture d'état symbolique par réécriture source à source*

Une autre utilisation de la réflexivité consiste à utiliser un compilateur source à source pour exprimer l'état d'un programme au travers d'une version modifiée de ce programme, qui lorsqu'elle est compilée et exécutée se comporte de manière équiva-

lente à l'originale mais redémarre au point de la prise d'état (Theimer *et al.,* 1991). Cette approche a inspiré plusieurs projets pour la capture d'état d'applications à brins d'exécution multiples. Ces projets sauvegardent une représentation *symbolique* (par opposition à « binaire ») de l'état des brins d'exécution d'une application, soit en utilisant les informations de débogage incluses dans l'exécutable par les compilateurs les plus courants[7] (Ferrari *et al.,* 1997, Balkrishna Ramkumar, 1997), soit en rajoutant du code qui *simule* symboliquement l'exécution de la pile d'un brin (Karablieh *et al.,* 2002). L'état des brins d'exécution ainsi capturé ne dépend que de la structure du code source du programme considéré, ce qui le rend extrêmement portable.

La restauration d'une pile à partir de sa représentation symbolique s'opère en rajoutant au début de chaque fonction (ou méthode) un bloc de code qui permet de restaurer les variables locales associées à celle-ci puis de *sauter* directement aux différents appels emboîtés contenus dans la fonction. La restauration commence alors au point d'entrée du programme, à la fonction *main*, le bloc de code ajouté se chargeant de « sauter » à l'appel de fonction indiqué par la pile symbolique. Ce processus se reproduit récursivement, jusqu'à restitution complète de la pile sauvegardée. Les entités de synchronisation gérées par le noyau pour l'application (sémaphores, verrous, etc.) sont quant à elles recréées en utilisant les informations collectées par l'interception des appels OS appropriés.

5.3.3. *Gestion de l'état et distribution en C++ :* FRIENDS

Le projet FRIENDS V.1 (Pérennou *et al.,* 1998) pousse plus loin l'utilisation de la réflexivité pour la tolérance aux fautes en adaptant les stratégies de GARF et MAUD à un langage de programmation courant, C++, par l'entremise du compilateur réflexif OPENC++ (Chiba, 1995). FRIENDS V.1 intègre en plus de la tolérance aux fautes des considérations de sécurité, d'authentification, et de capture d'état pour des objets ne contenant que des attributs ayant des types de base. Ce projet a été étendu à un environnement CORBA dans le projet FRIENDS V.2 (Killijian *et al.,* 1998, Killijian *et al.,* 1999, Killijian *et al.,* 2000), en intégrant une fonctionnalité de capture d'état généralisée, transparente à l'utilisateur. Le protocole à méta-objets de chacun de ces projets forme le cœur de leur modèle de programmation. FRIENDS V.2 intègre ainsi des mécanismes de réification du cycle de vie d'un objet (création, destruction), de l'invocation de ses méthodes, des mécanismes d'introspection et de modification (intercession) de son état etc.

L'idée particulièrement innovante de FRIENDS V.2 consiste à utiliser la représentation commune des données standardisées par CORBA (*Common Data Representation*, CDR (OMG 2002a)) pour mettre à plat les structures orientées objet qui sont réifiées par le compilateur ouvert OPENC++. Cette approche, qui peut être étendue à tout langage orienté objet pour lequel existe un *mapping* CORBA, a permis de réaliser des transferts d'état entre des programmes écrits dans des langages différents, ici C++ et Java (Killijian *et al.,* 2002). L'hétérogénéité des langages est souvent consi-

7. Ces informations permettent par exemple d'introspecter la pile d'appels d'un brin d'exécution, ou la valeur de ses variables locales, allouées sur sa pile.

dérée comme un obstacle rédhibitoire au transfert d'état, mais lorsqu'elle peut être maîtrisée, elle apporte un avantage considérable du point de vue de la tolérance des fautes logicielles de par la diversification des implémentations qu'elle autorise (voir section 2.3).

5.3.4. *Systèmes d'exploitation*

La réflexivité a aussi été utilisée pour permettre le durcissement de micronoyaux (*Chorus* (Armand, 2001), *LynxOS* (LYN 2002)) par l'utilisation d'encapsulateurs (*wrappers*), c'est-à-dire d'une couche de confinement logiciel qui, en associant observation, diagnostic et contrôle, permet d'arrêter la propagation d'erreurs au plus tôt (Rodriguez *et al.,* 2000, Arlat *et al.,* 2002). Cette approche réifie des événements internes à un micronoyau comme la création d'un brin d'exécution ou la prise d'une sémaphore vers un *méta-noyau*, et permet alors la spécification sous forme de formules de logique temporelle linéaire de contraintes de fonctionnement « normal » qui permettent de détecter les erreurs au plus tôt, et soit d'imposer un silence sur défaillance (confinement), ou de tenter d'entreprendre des mesures correctrices (Rodriguez, 2002).

5.4. *Discussion sur l'approche réflexive*

Les architectures réflexives représentent l'une des formes de structuration des intergiciels de tolérance de fautes les plus prometteuses en termes de transparence et de flexibilité.

Cependant, ces approches se situent toutes à un seul niveau d'abstraction, ce qui limite grandement la nature et l'efficacité des mécanismes de tolérance aux fautes qu'elles permettent de développer. Elle utilisent le plus souvent des méta-modèles de haut niveau, basés sur les informations rendues disponibles par un langage de programmation (C++ et CORBA IDL pour FRIENDS V.2, SMALLTALK pour GARF, HAL pour MAUD). Or de nombreux aspects de « bas niveau » du fonctionnement d'un système, tels la politique d'ordonnancement des brins d'exécution, ou la gestion détaillée de la mémoire virtuelle, échappent à ces langages. Ces aspects ne peuvent donc être rendus accessible par une approche réflexive se limitant à leur seul modèle de programmation.

Un exemple caractéristique illustrant cette faiblesse est la capture de l'état d'un objet sur un système multiniveaux. (cf. section 5.2.2). En effet, l'état d'un objet inclut ses structures de données au niveau langage, les ressources qu'il a acquises (descripteurs de fichier, segments mémoire, verrous, etc.) ou qui lui sont implicitement associées par l'intergiciel ou le système opératoire (files de messages, brins d'exécution, etc.).

Ces limitations contraignent la tolérance aux fautes, car cette dernière est une propriété transversale. Elle nécessite de prendre en compte l'intégralité des couches en présence pour maximiser la couverture des fautes potentielles, et fournir aux équipes de tolérance aux fautes un accès le plus complet possible aux différents niveaux logiques du système. Une application serveur à brins d'exécution multiples ne peut par

exemple être répliquée de manière active que si elle est préalablement rendue déterministe. Ceci suppose – entre autres – de pouvoir modifier les mécanismes de synchronisation (verrous, sémaphores) utilisés par l'application et dans les librairies qu'elle utilise afin d'assurer une cohérence de comportement entre les répliques (Basile *et al.,* 2003, Napper *et al.,* 2003). Or ce type de modification n'est pas réalisable par une approche réflexive si le méta-modèle utilisé est de trop haut niveau et n'intègre pas de considérations de synchronisation (Taïani *et al.,* 2003).

Si la limitation aux seuls hauts niveaux d'un système contraint la tolérance aux fautes qui peut être réalisée de façon réflexive, il en va de même des seuls bas niveaux (le système d'exploitation, les librairies système). En ouvrant ces bas niveaux, on obtient des capacités d'observation et d'intervention extrêmement complètes et très puissantes. Mais si l'information accessible est plus complète, elle a perdu en sémantique, et ne peut plus être interprétée par rapport l'activité des couches supérieures du système. Certains mécanismes de tolérance aux fautes, comme la migration de programmes entre plates-formes hétérogènes (Smith *et al.,* 1998, Killijian *et al.,* 2002), ne sont alors tous simplement plus réalisables. D'autres deviennent extrêmement coûteux. Napper *et al.* indiquent par exemple que l'interception indifférenciée de *l'ensemble* des verrous systèmes utilisés par une machine virtuelle Java pour la rendre déterministe peut induire des surcoûts allant jusqu'à 375 % du temps d'exécution normal (Napper *et al.,* 2003). Or certains verrous système (tels que ceux qui gèrent l'accès à la mémoire par exemple) n'ont pas d'influence observable sur le déterminisme d'une machine virtuelle, et n'ont donc pas besoin d'être interceptés. Cependant, sans plus d'informations de « haut niveau », il est impossible de distinguer les verrous qui sont « neutres » vis-à-vis du déterminisme de ceux qui ne le sont pas.

La réalisation de mécanismes de tolérance aux fautes a donc beaucoup à gagner de l'ouverture combinée, par la réflexivité, des hauts et des bas niveaux d'une architecture complexe. L'approche par réécriture source à source proposée dans (Karablieh *et al.,* 2002), parce qu'elle intègre réflexivité à la compilation (sémantique langage), et interception des appels POSIX (sémantique OS) représente un premier pas dans cette direction. Nous avons pour notre part poursuivi cette intuition en proposant une extension de la réflexivité, la *réflexivité multiniveaux*, pour prendre en compte l'ensemble des niveaux d'un système organisé en couches lors de la réalisation de la tolérance aux fautes. L'idée de base de la réflexivité multiniveaux est d'intégrer dans une même méta-interface des capacités réflexives de haut et de bas niveaux. Cette approche permet alors d'utiliser la puissance d'inférence sémantique des hautes couches pour guider la force de contrôle brute des bas niveaux selon une perception globale du système (Taïani *et al.,* 2002, Taïani *et al.,* 2003).

La réflexivité multiniveau relève un nouveau défi : la mise en cohérence de la manipulation de différents méta-modèles. En effet, si l'on suppose un système formé de trois niveaux *réflexifs* de type système opératoire, intergiciel et applicatif, le développement d'un mécanisme de tolérance aux fautes doit agir de manière cohérente sur leurs méta-modèles pour maîtriser la capture de l'état des objets applicatifs et le non déterminisme. Des travaux récents (Taïani *et al.,* 2005) corroborent ce point de vue.

6. Conclusion et perspectives

Nous avons proposé dans cet article un tour d'horizon de l'effort accompli ces trente dernières années pour intégrer la tolérance aux fautes dans des plates-formes d'exécution génériques. Nous avons rappelé les principes fondamentaux de la tolérance aux fautes et de ses mécanismes, pour ensuite nous intéresser à la manière dont ces mécanismes avaient pu être intégrés dans des intergiciels de plus en plus élaborés.

Cette rétrospective nous a permis d'observer comment les différentes approches proposées ont évolué d'une logique exclusivement centrée sur les aspects de la tolérance aux fautes (synchronisation, capture d'état, détection d'erreur) vers des architectures guidées par des objectifs de *transparence* et *d'adaptabilité*.

Ces dernières préoccupations revêtent aujourd'hui une importance essentielle.

Les systèmes informatiques modernes se caractérisent en effet par une complexité et une ouverture jusqu'alors inégalées, tout en couvrant des échelles de grandeurs extrêmement diverses, de nos machines à laver à nos centrales nucléaires. Cette révolution majeure, parfois appelée informatique omniprésente, ou encore *ubiquitous and pervasive computing*, lorsque qu'elle touche des domaines d'application à très fortes exigences en matière de sécurité, de disponibilité, et de temps-réel (réseaux de télécommunication, commerce électronique, chaînes logistiques, gestion d'entreprise, transport, énergie etc.), ne permet plus d'aborder la tolérance aux fautes sans prendre en considération les contraintes de développement auxquelles ces systèmes sont soumis. Dans un environnement hautement volatile, ou les technologies et les menaces évoluent en permanence, les systèmes informatiques doivent évoluer à chaud, pour prendre en compte leur nouveau contexte d'exécution sans discontinuité de service. L'étude des problèmes d'adaptabilité des aspects fonctionnels d'un système peut s'appuyer sur de nombreuses années de recherche en génie logiciel. L'adaptabilité de mécanismes transversaux comme la tolérance aux fautes est en revanche une problématique beaucoup plus récente, qui soulève encore de nombreux défis.

Parmi les architectures que nous avons présentées dans cet article, l'approche réflexive apparaît comme la plus prometteuse en termes de transparence et d'adaptativité, mais un travail important reste cependant à faire pour lui permettre de passer à l'échelle des systèmes industriels. La réalisation de systèmes informatiques de grande taille, intégrant de nombreuses technologies hétérogènes, introduit en effet des problèmes de gestion de la complexité dont les dimensions techniques et organisationnelles sont intimement mêlées. La réalisation de plates-formes tolérantes aux fautes dans un tel environnement protéiforme requiert de prendre en compte l'aspect « compartimenté » et hétérogène de ces systèmes, pour permettre un véritable découplage entre intégration fonctionnelle et préoccupations non fonctionnelles. Cette recherche d'équilibre entre le *fédéralisme* traditionnel des architectures informatiques (basé sur des notions telles que celles d'encapsulation, de couches, de modularité), et la nécessité d'une *vision globale* d'un système pour réaliser des aspects transversaux comme la tolérance aux fautes, donne aujourd'hui lieu à de nombreux efforts de recherche,

autour par exemple de la programmation par aspects (Kiczales *et al.,* 2001), de la réflexivité, ou encore des filtres de composition (Aksit *et al.,* 1994).

Ces efforts nous semblent d'autant plus significatifs qu'ils ont déjà commencé à impacter des technologies couramment utilisées dans l'industrie : JAVA et C# avec leurs interfaces de programmation réflexives, les *Entreprise Java Beans* (EJB) avec leur modèle de programmation à base de *containers*, ou encore la plate-forme open source JBOSS avec son canevas orienté aspects (le *JBoss AOP Framework*).

Nous sommes convaincus que ces nouvelles approches auront une forte influence sur les nouvelles formes de développement et d'architectures qui apparaîtront. Les enjeux économiques mettent en effet au premier plan des préoccupations des intégrateurs de systèmes les aspects de sûreté de fonctionnement dans des architectures hautement évolutives. Les technologies traditionnelles de construction de systèmes (à base de composants) et de validation ne permettent pas de maîtriser l'évolution dynamique des systèmes tout en préservant des propriétés de sûreté de fonctionnement. Ce constat pose donc de nouveaux défis scientifiques et technologiques, dont les solutions s'appuieront probablement sur certaines des approches que nous avons présentées dans cet article.

Remerciements

Ce travail a été partiellement financé par le project Européen IST "Dependable Systems of Systems" n.1999-11585.

7. Bibliographie

AEE, *ARINC-429 Digital Information Transfer System Parts 1, 2, 3*, n.d.

Agha G., Frolund S., Panwar R., Sturman D., « A Linguistic Framework for Dynamic Composition of Dependability Protocols », *in* , I. Transactions (ed.), *the IFIP Conf. on Dependable Computing for Critical Applications (DCCA-3)*, Elsevier, Palermo (Sicily), Italy, p. 197-207, 1992.

Aksit M., Bosch J., van der Sterren W., Bergmans L., « Real-Time Specification Inheritance Anomalies and Real-Time Filters », *8th European Conf. on Object-Oriented Programming (ECOOP)*, vol. 821 of *Lecture Notes in Computer Science (LNCS)*, Springer Verlag, Bologna, Italy, p. 386-407, 1994.

Amir Y., Dolev D., Kramer S., Malki D., « Transis : A Communication Subsystem for High Availability », *FTCS-22 : 22nd Int. Symp. on Fault Tolerant Computing*, IEEE Computer Society Press, Boston, Massachusetts, p. 76-84, 1992.

Arlat J., Fabre J.-C., Rodriguez M., Salles F., « Dependability of COTS microkernel-based Systems », *IEEE Transactions on Computers*, vol. 51, n° 2, p. 138-163, February, 2002.

Armand F., ChorusOS 5.0 Features and Architecture overview, Sun Technical Report n° 806-6897-10, Sun Microsystems, Palo Alto, CA, USA, 2001.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., Morrison, R., « PS-algol : A Language for Persistent Programming », *Proc. 10th Australian National Computer Conf., Melbourne, Australia*, p. 70-79, 1983.

Avizienis A., Kelly J. P. J., « Fault Tolerance by Design Diversity : Concepts and Experiments », *IEEE Computer*, vol. 17, n° 8, p. 67-80, August, 1984.

Baldoni R., Hélary J.-M., Mostefaoui A., Raynal M., « A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability », *Int. Symp. on Faut-Tolerant Computing FTCS-27*, Seattle, Washington, p. 68-77, 1997.

Balkrishna Ramkumar V. S., « Portable Checkpointing for Heterogeneous Architectures », *27th Int. Symp. on Fault-Tolerant Computing (FTCS'97) - Digest of Papers*, Seattle, WA (USA), p. 58-67, June, 1997.

Basile C., Kalbarczyk Z., Iyer R., « A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas », *The Int. Conf. on Dependable Systems and Networks (DSN-2003)*, IEEE Computer Society, San Francisco, CA, p. 149-158, 2003.

Beck K., *Extreme Programming Explained : Embrace Change*, Addison-Wesley, 1999.

Bennani M. T., Blain L., Courtès L., Fabre J.-C., Killijian M.-O., Taïani F., « Implementing Simple Replication Protocols Using CORBA Portable Interceptors and Java Serialization », *Proceedings of the Int. Conf. on Dependable Systems and Networks (DSN 2004), DCC Symp.*, Pisa, Italy, p. 549-554, June, 2004.

Birman K., « Replication and Fault-Tolerance in the ISIS System », *10th ACM Symp. on Operating Systems Principles, Operating System Review 19(5)*, ACM, New York, Orcas Island. Washington, USA, p. 79-86, 1985.

Briot J., Guerraoui R., « Objets pour la programmation parallèle et répartie : Intérêts, évolutions et tendances », *Techniques et Sciences Informatiques*, vol. 15, n° 6, p. 765-800, June, 1996.

Chandra T. D., Toueg S., « Unreliable Failure Detectors for Reliable Distributed Systems », *Journal for the Association for Computing Machinery (JACM)*, vol. 43, n° 2, p. 225-267, 1996.

Chandy K. M., Lamport L., « Distributed Snapshots : Determining Global States of Distributed Systems », *ACM Transactions on Computer Systems*, vol. 3, n° 1 (February), p. 63-75, 1985.

Chiba S., « A metaobject protocol for C++ », *OOPSLA '95 : Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, ACM Press, New York, NY, USA, p. 285-299, 1995.

Deswarte Y., « Comment peut-on tolérer les intrusions sur internet ? », *Revue de l'Electricité et de l'Electronique*, vol. 8, p. 83-90, September, 2003.

Dieter W. R., Jr. J. E. L., « A User-level Checkpointing Library for POSIX Threads Programs », *29th Int. Symp. on Fault-Tolerant Computing*, Madison, Wisconsin USA, p. 224-227, 1999.

Dieter W. R., Jr. J. E. L., « User-level Checkpointing for LinuxThreads Programs », *2001 USENIX Technical Conf.*, Boston, Massachusetts, USA, 2001.

Elnozahy E., Alvisi L., Johnson D. B., Wang Y.-M., « A Survey of Rollback-Recovery Protocols in Message Passing Systems », *ACM Computing Surveys*, vol. 34, n° 3 (September), p. 375-408, 2002.

ESA, Ariane 5, Flight 501 Failure, Report by the Inquiry Board, Technical report, ESA, Paris, July, 1996.

Espen Skoglund Christian Ceelen J. L., « Transparent Orthogonal Checkpointing Through User-Level Pagers », *9th Int. Workshop on Persistent Object Systems (POS9)*, number 2135 in *Lecture Notes in Computer Science*, System Architecture Group, University of Karlsruhe, Springer Verlag, Lillehammer, Norway, p. 201-214, September, 2000.

Felber P., A Service Approach to Object Groups in CORBA, Thèse de doctorat, Département d'informatique de l'École Polytechnique Fédérale de Lausanne, Lausanne (Suisse), 1998.

Felber P., Garbinato B., Guerraoui R., « The Design of a CORBA Group Communication Service », *Proceedings of the 15th Symp. on Reliable Distributed Systems (SRDS-15)*, Niagara-on-the-Lake, Canada, p. 150-159, 1996.

Ferrari A. J., Chapin S. J., Grimshaw A. S., Process Introspection : A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification, Technical report, University of Virginia (USA), March, 1997. http ://www.cs.virginia.edu/ ajf2j/introspect/.

Fetzer C., Cristian F., « On the Possibility of Consensus in Asynchronous Systems », *1995 Pacific Rim Int'l Symp. on Fault-Tolerant Systems*, Newport Beach, CA, p. 86-91, 1995.

Fischer M. J., Lynch N. A., Paterson M. S., « Impossibility of Distributed Consensus with One Faulty Process », *Journal of the Association for Computing Machinery*, vol. 32, n° 2 (April), p. 374-382, 1985.

Guerraoui R., Garbinato B., Mazouni K. R., « GARF : A Tool for Programming Reliable Distributed Applications », *IEEE Concurrency*, vol. 5, n° 4, p. 32-39, October-December, 1997a.

Guerraoui R., Raynal M., « The Information Structure of Indulgent Consensus », *IEEE Transactions on Computers*, vol. 53, n° 4, p. 453-466, 2004.

Guerraoui R., Rodrigues L., *Introduction to Reliable Distributed Programming*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

Guerraoui R., Schiper A., « Consensus : the Big Misunderstanding », *IEEE Int. Workshop on Future Trends in Distributed Computing Systems (FTDCS'97)*, Tunis, 1997b.

Herlihy M., Shavit N., « The topological structure of asynchronous computability », *J. ACM*, vol. 46, n° 6, p. 858-923, 1999.

Hiltunen M. A., Schlichting R. D., « A Configurable Membership Service », *IEEE Trans. Comput.*, vol. 47, n° 5, p. 573-586, 1998.

Hürsch W., Lopes C. V., Separation of Concerns, technical report n° NU-CCS-95-03, Northeastern University, February, 1995.

Johnson D., Zwaenepoel W., « Sender-Based Message Logging », *17th Int. Symp. on Fault-Tolerant Computing (FTCS)*, p. 14-19, 1987.

Karablieh F., Bazzi R. A., « Heterogeneous Checkpointing for Multithreaded Applications », *21st Symp. on Reliable Distributed Systems (SRDS'02)*, Osaka, Japan, p. 140-149, 2002.

Kiczales G., des Rivieres J., Bobrow D. G., *The Art of the Metaobject Protocol*, MIT Press, 1991.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., « An Overview of AspectJ », *Proceedings of the 15th European Conf. on Object-Oriented Programming*, Springer-Verlag, p. 327-353, 2001.

Killijian M.-O., Fabre J.-C., « Implementing a Reflective Fault-Tolerance CORBA System », *19th IEEE Symp. on Reliable Distributed Systems (SRDS'2000)*, Nürnberg, Germany, p. 154-163, 2000.

Killijian M.-O., Fabre J.-C., Ruiz-García J.-C., Shiba S., « A metaobject protocol for fault-tolerant CORBA applications », *17th IEEE Symp. on Reliable Distributed Systems (SRDS-17)*, West Lafayette (USA), p. 127-134, 1998.

Killijian M.-O., Ruiz-García J.-C., Fabre J.-C., « Using Compile-Time Reflection for Object State Capture », *in* , P. Cointe (ed.), *Reflection*, vol. 1616 of *Lecture Notes in Computer Science*, Springer, p. 150-152, 1999.

Killijian M.-O., Ruiz-García J.-C., Fabre J.-C., « Portable serialization of CORBA objects : a reflective approach », *18th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2002)*, ACM Press, Seattle, Washington, USA, p. 68-82, 2002.

Koo R., Toueg S., « Checkpointing and Rollback-Recovery for Distributed Systems », *IEEE Transactions on Software Engineering*, vol. 13, n° 1, p. 23-31, 1987.

Lamport L., Shostak R., Pease M., « The Byzantine generals problem », *ACM Transactions on Programming Languages and Systems*, vol. 4, n° 3, p. 382-401, July, 1982.

Laprie J.-C., *Le guide de la sûreté de fonctionnement*, Cepadues, 1996.

Li K., Naughton J. F., Plank J. S., « Low-Latency, Concurrent Checkpointing for Parallel Programs », *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, n° 8, p. 874-879, 1994.

LYN, LynxOS User's Guide (Release 4.0), Technical Report n° DOC-0453-01, LynuxWorks inc., San José, CA, USA, 2002.

Maes P., « Concepts and Experiments in Computational Reflection », vol. 22 of *SIGPLAN Notices*, ACM, Orlando, Florida, p. 147-155, October, 1987.

Maffeis S., « Adding Group Communication and Fault-Tolerance to CORBA », *Proceedings of the 1995 USENIX Conf. on Object-Oriented Technologies (COOTS)*, USENIX, Monterey, CA, p. 135-146, June, 1995.

Meyer B., *Object Oriented Software Construction*, seconde edn, Prentice Hall PTR, 1997.

Mishra S., Peterson L. L., , Schlichting R. D., « Modularity in the Design and Implementation of Consul », *Proceedings of the First IEEE Symp. on Autonomous Decentralized Systems (ISADS'93)*, Kawasaki, Japan, p. 376-382, March, 1993.

Napper J., Alvisi L., Vin H., « A Fault-Tolerant Java Virtual Machine », *Proceedings of the Int. Conf. on Dependable Systems and Networks (DSN 2003), DCC Symp.*, p. 425-434, June, 2003.

Narasimhan P., Moser L. E., Melliar-Smith P. M., « The Interception Approach to reliable distributed CORBA Objects », *Proceedings of the 3rd USENIX Conf. on Object-Oriented Technologies and Systems (COOTS)*, USENIX, Portland, OR, USA, p. 245-248, June, 1997.

Netzer R. H. B., Xu J., « Necessary and Sufficient Conditions for Consistent Global Snapshots. », *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, n° 2, p. 165-169, 1995.

OMG, *Common Object Request Broker Architecture : Core Specification (Version 3.0.2 - formal/02-12-02)*, Needham, MA, U.S.A., December, 2002a.

OMG, *Common Object Request Broker Architecture : Core Specification (Version 3.0.2 - formal/02-12-02)*, in OMG (2002a), chapter 23. Fault Tolerant CORBA, December, 2002b.

OMG, *Common Object Request Broker Architecture : Core Specification (Version 3.0.2 - formal/02-12-02)*, in OMG (2002a), chapter 21. Portable Interceptors, December, 2002c.

OMG, The CORBA Persistent State Service, Spécifications, September, 2002d.

Or M. B., « Another advantage of free-choice : Completely asynchronous agreement protocols », *Proceedings of the 2nd annual ACM symposium on Principles of Distributed Computing (PODC 83)*, ACM, Montreal, Canada, p. 27-30, 1983.

Ousterhout J. K., Cherenson A. R., Douglis F., Nelson M. N., Welch B. B., « The Sprite Network Operating System », *Computer*, vol. 21, n° 2, p. 23-36, 1988.

Peterson L. L., Buchholz N. C., Schlichting R. D., « Preserving and Using Context Information in Interprocess Communication », *ACM Transactions on Computer Systems*, vol. 7, n° 3, p. 217-246, 1989.

Powell D., *Delta-4 : A Generic Architecture for Dependable Distributed Computing*, Springer-Verlag, 1991.

Pérennou T., Fabre J.-C., « A Metaobject Architecture for Fault-Tolerant Distributed Systems : the FRIENDS Approach », *IEEE Trans. on Computer, Special Issue on Dependability of Computing Systems*, vol. 47, p. 78-95, 1998.

Randell B., « System structures for software fault tolerance », *IEEE Transaction on Software Engineering 1, 2 (June 1975), 220-232.*, vol. 1, n° 2, p. 220-232, 1975.

Rodriguez M., Technologie d'empaquetage pour la sûreté de fonctionnement des systèmes temps-réel, Thèse de doctorat, Institut National Polytechnique, Toulouse (France), July, 2002.

Rodriguez M., Fabre J.-C., Arlat J., « Formal Specification for Building Robust Real-time Microkernels », *21st IEEE Real-Time Systems Symp.*, Orlando, Florida, USA, 2000.

Russinovich M., Segall Z., Siewiorek D. P., « Application Transparent Fault Management in Fault Tolerant Mach », *Twenty-Third Annual Int. Symp. on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, p. 10-19, 1993.

Smith B. C., « Reflection and Semantics in Lisp », *11th Annual ACM Symp. on Principles of Programming Languages (POPL)*, ACM, Salt Lake City, Utah, p. 23-35, 1984.

Smith P., Hutchinson N. C., « Heterogeneous Process Migration : The Tui System », *Software Practice and Experience*, vol. 28, n° 6, p. 611-639, 1998.

Strom R., Bacon D., Yemini S., « Volatile Logging in n-Fault-Tolerant Distributed Systems », *18th Int. Symp. on Fault Tolerant Computing (FTCS)*, p. 44-48, 1988.

Sun, *JavaTM Remote Method Invocation Specification, Revision 1.9, JavaTM 2 SDK, Standard Edition, v1.4.2*, Santa Clara, CA, 2003.

Tatsubori M., Chiba S., Killijian M.-O., Itano K., « OpenJava : A Class-based Macro System for Java », *in* , W. Cazzola, , R. J. Stroud, , F. Tisato (eds), *Reflection and Software Engineering*, LNCS 1826, Springer-Verlag, p. 119-135, July, 2000.

Taylor D. J., Wright M. L., « Backward Error Recovery in a UNIX Environment », *16th Anual Int. Symp. on Fault- Tolerant Computing (FTCS)*, Vienna (Austria), p. 118-123, 1986.

Taylor D., Wilson G., *In Dependability of Resilient Computers*, Blackwell Scientific Publications, Oxford, chapter The Stratus System Architecture, 1989.

Taïani F., Fabre J.-C., Killijian M.-O., « Principles of Multi-Level Reflection for Fault-Tolerant Architectures », *2002 Pacific Rim Int. Symp. on Dependable Computing (PRDC 2002)*, Tsukuba (Japan), p. 59-66, 2002.

Taïani F., Fabre J.-C., Killijian M.-O., « Towards Implementing Multi-Layer Reflection for Fault-Tolerance », *The Int. Conf. on Dependable Systems and Networks (DSN-2003)*, IEEE Computer Society, San Francisco, CA, June, 2003.

Taïani F., Fabre J.-C., Killijian M.-O., « A multi-level meta-object protocol for fault-tolerance in complex architectures », *The Int. Conf. on Dependable Systems and Networks (DSN-2005)*, IEEE Computer Society, Yokohama, Japan, 2005.

Theimer M. M., Hayes B., « Heterogeneous Process Migration by Recompilation », *the 11th Int. Conf. on Distributed Computing Systems*, p. 18-25, 1991.

van Renesse R., Birman K. P., Hayden M., Vaysburd A., Karr D., « Building Adaptive Systems Using Ensemble », *Software–Practice and Experience*, vol. 28, n° 9, p. 963-979, 1998.

van Renesse R., Birman K. P., Maffeis S., « Horus : A Flexible Group Communication System », *Communications of the ACM (CACM)*, vol. 39, n° 4, p. 76-83, 1996.

Article reçu le 3 août 2004
Version révisée le 24 juin 2005

François Taïani *est Lecturer au département d'informatique de l'Université de Lancaster (UK). Après avoir obtenu une thèse de doctorat en informatique en 2004 pour ses travaux réalisés au LAAS-CNRS (Toulouse) sur la réflexivité multiniveaux, il a séjourné un an au Shannon Laboratory d'AT&T (USA). François Taïani s'intéresse actuellement aux grilles de calcul, à l'utilisation de l'orienté aspect en sûreté de fonctionnement, et aux nouveaux paradigmes de distribution, notamment les réseaux de capteurs et les réseaux logiques (*overlay networks*).*

Marc-Olivier Killijian *est Chargé de Recherche au CNRS, depuis 2001, au sein du LAAS-CNRS (Toulouse). Après une thèse de doctorat en informatique de l'Institut National Polytechnique de Toulouse obtenue en 2000, il a séjourné un an dans le Distributed Systems Group du Trinity College of Dublin en Irlande. Ses domaines actuels de recherche concernent les intergiciels et systèmes opératoires réflexifs, les systèmes mobiles tolérants aux fautes et aux intrusions, le respect de la vie privée, ainsi que les systèmes autonomes critiques.*

Jean-Charles Fabre *est Professeur à l'Institut National Polytechnique de Toulouse, enseignant à l'ENSEEIHT. Depuis une vingtaine d'années, tout d'abord comme Chercheur INRIA, puis Chargé de Recherche et Directeur de Recherche au CNRS, il effectue ses recherches au LAAS-CNRS à Toulouse. Au sein du groupe « Tolérance aux Fautes et Sûreté de Fonctionnement informatique », ses contributions scientifiques portent sur les algorithmes et systèmes distribués, les systèmes tolérant les fautes et les intrusions, les systèmes réflexifs, l'évaluation par injection de fautes et l'empaquetage d'exécutifs commerciaux (des micronoyaux temps-réel aux intergiciels CORBA).*