Systèmes tolérant les fautes à base de support d'exécution réflexifs
Capture en ligne de l'état d'applications

Ludovic Courtès
<ludovic.courtes@laas.fr>

Stage de DEA - Université de Franche-Comté

Stage de fin d'études - Université de Technologie de Belfort-Montbéliard

Printemps 2003

Suiveur UTBM :
Vincent Hilaire

Tuteur LAAS-CNRS :
Jean-Charles Fabre

Avant-propos

Ce rapport présente le travail effectué lors d'un stage au sein du groupe Tolérance aux fautes et Sûreté de Fonctionnement (TSF) du Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS (LAAS-CNRS), à Toulouse. Ce stage marque la fin de mes études d'ingénieur à l'Université de Technologie de Belfort-Montbéliard (UTBM), ainsi que la fin du Diplôme d'Études Approfondies (DEA) Informatique, Automatique et Productique préparé en parallèle à l'Université de Franche-Comté.

Je tiens à remercier Jean Arlat, responsable du groupe TSF, pour m'avoir accueilli dans le groupe, et Jean-Charles Fabre pour m'avoir intégré à son équipe, pour m'avoir guidé pendant ce stage, et pour m'avoir aidé à surmonter les difficultés que j'ai rencontrées. Je remercie François Taïani, futur docteur, pour son suivi, ses nombreuses relectures de ce rapport, ses conseils, ses métaphores et ses périphrases éclairantes, ainsi que Marc-Olivier Killijian, chargé de recherche « alternatif », pour son soutien, ses encouragements, et pour m'avoir aidé à ne pas m'égarer dans le méta-espace. Un grand merci à mes collègues de bureau, Marc Lecointe, Amine Filali et Carlos Aguilar pour leur bonne humeur quotidienne et pour leur capacité à créer une ambiance de travail agréable et sympathique. Enfin, je remercie tous mes autres collègues, stagiaires et thésards, qui ont contribué à faire de ce stage un bon moment.

Ce rapport et les figures qu'il contient ont été réalisés avec le système de mise en page libre *Lout*, <*http://lout.sourceforge.net/*>.

Table des Matières

Liste des figures

Introduction

Ce rapport présente le travail effectué lors d'un stage de DEA effectué au sein du groupe Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF) du LAAS-CNRS, à Toulouse. Ce stage s'inscrit dans le cadre de recherches portant sur la mise en œuvre de systèmes tolérants les fautes adaptatifs. Ces travaux sont fondés sur la notion de support d'exécution réflexif qui offre l'observabilité et la commandabilité nécessaires à la manipulation en ligne de l'état d'un système.

Le travail effectué durant ce stage est axé autour de deux thèmes principaux : d'une part la capture en ligne de l'état d'applications (écrites en langage C ou C++) qui a fait l'objet d'une réalisation pratique, et d'autre part une partie prospective relative aux architectures ouvertes de système d'exploitation et à l'utilisation de capacités réflexives pour mettre en œuvre des mécanismes de tolérance aux fautes.

Après une présentation des principaux concepts en tolérance aux fautes et des applications de la réflexivité aux systèmes sûrs de fonctionnement (chapitre 1), nous aborderons aux chapitres 2 et 3 la mise en œuvre de la capture d'état en ligne d'applications, un mécanisme essentiel pour un système tolérant aux fautes. Le chapitre 2 donne un aperçu des moyens permettant la capture d'état, et le chapitre 3 décrit ensuite la mise en œuvre d'une bibliothèque de capture d'état pour applications C et C++ réalisée durant ce stage. Enfin, nous verrons au chapitre 4, de manière plus générale, l'ensemble des propriétés ainsi que les architectures de systèmes d'exploitation permettant la mise en œuvre de mécanismes de tolérance aux faute.

Chapitre 1. Introduction à la tolérance aux fautes

« *Spécifier, concevoir, réaliser et exploiter des systèmes où la faute est naturelle, prévue, et tolérable.* » [32] .

Ce chapitre expose les principaux concepts liés à la sûreté de fonctionnement et plus particulièrement à la *tolérance aux fautes*, ainsi que différentes techniques. La section 1.2 présente ensuite les concepts liés à la *réflexivité* et leur application à l'architecture de systèmes tolérants aux fautes. L'intérêt de systèmes réflexifs est davantage détaillé dans les chapitres suivants, dans le cas de la mise en œuvre d'un mécanisme de capture d'état portable puis dans le cas de l'architecture de systèmes d'exploitation.

1.1. Concepts de base

1.1.1. Définitions

Les systèmes informatiques étant de plus en plus répandus, une défaillance de l'un d'entre eux peut avoir un impact catastrophique sur la société. En même temps, les systèmes informatiques étant de plus en plus complexes, il devient de plus en plus difficile de les maîtriser et un soin accru doit leur être apporté. La *sûreté de fonctionnement* d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre [32]. Selon les applications les besoins en sûreté de fonctionnement diffèrent. Plusieurs *attributs* ou facettes de la sûreté de fonctionnement peuvent être dégagés [32] :

- la *disponibilité* évalue le fait d'être prêt à rendre le service ;
- la *fiabilité* évalue la continuité du service ;
- la *sécurité-innocuité* est définie par la non-occurrence de conséquences catastrophiques pour l'environnement ;
- la *confidentialité* s'intéresse à la non-divulgation non-autorisée d'information ;
- l'*intégrité* caractérise la non-occurrence d'altérations de l'information ;
- enfin, la *maintenabilité* caractérise l'aptitude aux réparations et aux évolutions du système.

La *défaillance* d'un système survient lorsque le service délivré dévie de l'accomplissement de la fonction du système. Une *erreur* est la partie de l'état du système susceptible d'entraîner une défaillance. Enfin, une *faute* est la cause adjugée ou supposée d'une erreur. Fautes, erreurs et défaillances peuvent donc s'enchaîner comme suit :

$$\textit{faute} \rightarrow \textit{erreur} \rightarrow \textit{défaillance} \rightarrow \textit{faute} \rightarrow \ldots$$

Plus précisément, la défaillance d'un composant peut être vue comme une faute pour le système qui l'englobe, et elle peut à son tour être source d'une erreur dans ledit système, etc.

À leur tour, les défaillances peuvent être rangées dans plusieurs classes. Le service rendu par un système peut défaillir suivant deux axes, l'axe des valeurs (la valeur rendue en réponse à une requête n'est pas correcte) et l'axe temporel (la réponse ne se fait pas dans l'intervalle de temps spécifié).Dans le cas de *défaillances par arrêt*, l'activité du système n'est plus perceptible aux utilisateurs, ce qui peut se traduire soit par un *figement de l'état des sorties du système*, soit par un *silence* (e.g. pas d'envoi de messages dans un système réparti). De plus, la perception des défaillances conduit à distinguer les *défaillances cohérentes* (tous les utilisateurs en ont la même perception) et les *défaillances incohérentes* (les utilisateurs peuvent avoir des perceptions différentes des défaillances). Les fautes peuvent être caractérisées selon plusieurs critères [32] :

- selon leur *cause phénoménologique*, il peut s'agir de fautes physiques ou de fautes dues à l'homme ;
- selon leur *nature*, elles peuvent être accidentelles ou intentionnelles ;
- selon la *phase du cycle de vie du système* où elles ont été introduites, il peut s'agir de fautes de développement (ce que l'on appelle souvent des « *bugs* ») ou de fautes opérationnelles apparaissant durant l'exploitation du système ;
- en fonction de la *situation des fautes*, on parlera de fautes internes lorsqu'elles font partie du système, et de fautes externes lorsqu'elles sont le résultat de l'interaction du système avec son environnement ;
- enfin, selon la *durée de vie des fautes*, on parlera de fautes permanentes ou de fautes temporaires (liées à des conditions ponctuelles et donc présentes pour une durée limitée).

Plusieurs *moyens* permettent d'assurer la sûreté de fonctionnement d'un système. L'*élimination des fautes* cherche à réduire le nombre et la sévérité des fautes. La *prévision des fautes* cherche à estimer la présence et la conséquence des fautes. Enfin, la *tolérance aux fautes*, dont il est question dans la section suivante, est un moyen qui cherche à permettre au système de remplir sa fonction *en dépit de la présence de fautes*.

1.1.2. Tolérance aux fautes

La tolérance aux fautes est un moyen d'assurer la sûreté de fonctionnement d'un système en présence de fautes, qu'il s'agisse de fautes physiques, de fautes de conception, d'erreurs de l'opérateur, ou encore d'intrusions. Dans ce document, nous nous intéressons principalement aux deux premières classes de fautes. La première étape dans la conception d'un système tolérant aux fautes est la mise en œuvre de mécanismes de *détection d'erreurs*. Les formes les plus courantes de détection d'erreurs sont les suivantes [32] :

- le contrôle par *codes détecteurs d'erreur*, qui utilise la redondance de la représentation de l'information (e.g. le *Code à Redondance Cyclique* ou CRC est un code détecteur et même correcteur d'erreur) ;
- la *duplication et comparaison* qui consiste à comparer les résultats fournis par deux unités indépendantes fournissant le même service ;

- le *contrôle de données structurées* qui consiste à vérifier soit la cohérence des données d'un point de vue sémantique, soit la cohérence de la structure des données (e.g. cohérence d'une liste chaînée).

Des exemples de ce type de contrôle au niveau du système d'exploitation sont donnés dans [46] et au chapitre 4.

La deuxième étape de la mise en œuvre d'un système tolérant aux fautes concerne le traitement des erreurs détectées. Une première technique consiste à masquer les fautes en utilisant des *codes correcteurs d'erreurs* [32] : c'est ce que permet l'utilisation de codes tels que SEC (« *Single-Error Correction* ») ou CRC (*Code à Redondance Cyclique*), et c'est aussi l'objectif des systèmes de sauvegarde redondante des données tels que RAID (« *Redundant Array of Inexpensive or Independent Disks* »). Une autre méthode répandue pour tolérer les fautes, une fois détectées, est le *recouvrement d'erreur*. On distingue essentiellement trois techniques de recouvrement d'erreur :

- le *recouvrement d'erreur par poursuite* qui consiste, après avoir détecté une erreur, à rechercher un nouvel état acceptable du système ;

- le *recouvrement d'erreur par compensation* qui nécessite que l'état du système comporte suffisamment de redondance pour permettre de le faire passer dans un état exempt d'erreur ;

- enfin, le *recouvrement d'erreur par reprise*, le plus utilisé dans la tolérance aux fautes logicielles, et qui consiste à capturer régulièrement l'état du système, créant un ensemble de « points de reprise » (ou « *checkpoints* ») possibles, de manière à pouvoir ramener le système » dans un état antérieur après détection d'une erreur.

Ce dernier point fait l'objet des chapitres 2 et 3. Ces modes de recouvrement d'erreur ont donné lieu à des techniques réparties de *réplication de processus* [32, p. 98]. La réplication de processus consiste en l'exécution coordonnée d'un groupe de processus installés sur des processeurs distincts. On distingue alors trois types de réplication différents :

- la *réplication active* est une technique de détection et compensation par laquelle toutes les répliques d'un même processus traitent tous les messages d'entrée ;

- la *réplication passive* est une technique de recouvrement par reprise dans laquelle seule la réplique *primaire* traite les données d'entrée et fournit les résultats ; en l'absence de fautes, les autres répliques (dites répliques *de secours*) ne traitent pas les données d'entrée mais leur état est mis à jour au moyen de points de reprise envoyés par la réplique primaire ;

- la *réplication semi-active* est une technique de recouvrement par laquelle les messages d'entrée sont diffusés à toutes les répliques mais une seule réplique (la réplique *meneuse*) traite tous les messages d'entrée et fournit les messages de sortie ; par contre, contrairement à la réplication active, la réplique meneuse communique avec les suiveuse pour leur indiquer toutes les décisions non-déterministes qu'elle prend.

La technique de réplication passive sera illustrée par la présentation de l'implantation proposée en chapitre 3 (voir notamment la figure 3.9, page 28).

1.2. Utilisation de la réflexivité

Cette section introduit dans un premier temps les apports des concepts liés à la réflexivité dans la conception et l'architecture de systèmes tolérants aux fautes. La section 1.2.2 définit ensuite certains concepts de base en réflexivité. L'utilité des architectures réflexives pour la mise en œuvre de méca-

nismes de tolérance aux fautes est illustrée par la suite aussi bien dans le cas de l'implémentation de mécanismes de capture d'état portable (chapitre 2), que dans le cas de l'introduction de mécanismes divers au niveau du système d'exploitation (chapitre 4).

1.2.1. Motivations

La mise en œuvre de mécanismes de tolérance aux fautes pour une application donnée est une problématique orthogonale au développement des parties *fonctionnelles* de l'application. La mise en œuvre de ces mécanismes peut par ailleurs s'avérer assez complexe, et elle requiert une bonne connaissance du développeur. Pour ces raisons, et aussi pour simplifier la maintenabilité de l'application, l'idéal est que le développeur de l'application puisse s'affranchir des préoccupations liées à la tolérance aux fautes. Les mécanismes de tolérance aux fautes peuvent alors être développés par un spécialiste dans ce domaine, sans avoir cette fois-ci à se préoccuper de l'application elle-même. On parle alors de *séparation des préoccupations* (ou « *separation of concerns* ») entre le développeur de l'application et le développeur des mécanismes de tolérance aux fautes [45]. De cette façon, la tolérance aux fautes devient *transparente* pour le programmeur de l'application.

Pour que cette séparation puisse être respectée, l'application doit fournir des moyens de contrôle et d'observabilité répondant aux besoins des mécanismes de tolérance aux fautes [53]. En effet, les mécanismes de détection et de recouvrement d'erreurs nécessitent tous d'avoir accès à certaines données de l'application mais aussi d'avoir un contrôle sur l'application et sur son déroulement. Ces besoins sont formalisés par la définition de la *réflexivité calculatoire* (ou « *computational reflection* ») donnée par Pattie Maes [38]. La section suivante présente les principaux concepts liés à la réflexivité et leur utilisation.

1.2.2. Concepts

La réflexivité est la capacité d'un système à raisonner à propos de lui-même et à agir sur lui-même [38]. Cette définition élémentaire introduit la notion de moyens d'observation et de modification ou d'extension *a posteriori* du comportement initial du système. Un système réflexif est défini comme un système qui intègre des structures de données représentant certains aspects de lui-même et qui constituent sont *auto-représentation*. Cette auto-représentation ou *méta-modèle* doit pouvoir lui permettre de se poser des questions sur lui-même mais aussi d'être capable d'agir sur lui-même. De cette façon, un tel système a la capacité de s'auto-modifier. Dans un système réflexif, le *méta-modèle* est relié de manière causale aux aspects qu'il représente, de telle sorte que :

(i) le système dispose toujours d'une représentation exacte de lui-même ;

(ii) l'état du système est toujours cohérent avec sa représentation.

On appelle *niveau de base* le système lui-même. Le *méta-niveau* est la partie du système qui observe et modifie le niveau de base en interprétant et en agissant sur le méta-modèle. Ces deux niveaux sont liés par des processus d'interaction illustrés par la figure 1.1 : la *réification* est une information envoyée par le niveau de base au méta-niveau pour indiquer un changement lorsqu'il a subit une modification, l'*intercession* est initiée par le méta-niveau pour communiquer un changement au niveau de base, et l'*introspection* est une demande d'information sur le niveau de base initiée par le méta-niveau. Pour être plus précis, le mécanisme d'intercession peut être divisé en *intercession comportementale* qui vise à modifier le comportement du niveau de base, et en *intercession structurelle* qui vise à modifier l'état du niveau de base [53].

Les langages de programmation ont été l'un des premiers champs d'expérimentation des concepts de réflexivité, même si la réflexivité a depuis été appliqué à bien d'autres champs (voir section 4). Un langage de programmation interprété est dit réflexif si son interpréteur donne accès à des structures de données représentant certains aspects du système, et si cet interpréteur garantie le lien de

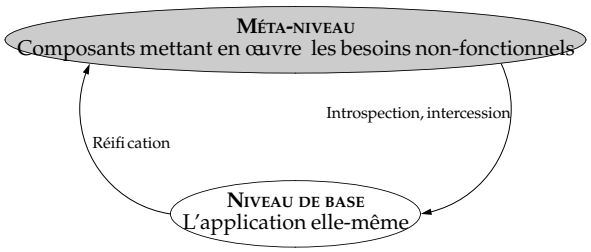


Figure 1.1. Réification, introspection et intercession dans un système réflexif.

causalité entre ces structures de données et le système lui-même [38]. Le fait d'être réflexif rend le langage *ouvert*, dans le sens où une fonctionnalité qui n'est pas présente dans le langage peut toujours être ajoutée. À titre d'exemple, le langage orienté-object 3-KRS présenté par Pattie Maes dans [38] et [37] ne prend pas en charge l'héritage multiple mais permet de rajouter cette fonctionnalité en spécialisant les méthodes d'interprétation des objets. Plus simplement, la primitive `defined?` du langage Scheme (ou son équivalent `boundp` en Common Lisp) qui renvoie un booléen indiquant si une variable est définie ou non est un bon exemple de réification d'informations maintenues par l'interpréteur.

En résumé, le méta-niveau d'un langage interprété peut être vu comme l'ensemble des structures et données qui contrôlent l'interprétation du programme. Dans le cas des langages compilés tels que C++, des compilateurs ouverts tels que OpenC++ [5] permettent d'obtenir une représentation du programme à la compilation. Un *méta-programme* peut utiliser cette représentation et la modifier pour intervenir dans le processus de compilation. Une utilisation de ce type d'outil pour obtenir une représentation d'un programme C++ est discutée plus loin en section 2, page 11.

La section suivante décrit plus spécifiquement comment peuvent être introduits les mécanismes de réflexivité dans un langage orienté-objet.

1.2.3. Protocoles à méta-objets

Dans un modèle objet, la réflexivité est généralement réalisée à travers un *protocole à méta-objets*. Un *méta-objet* correspond au méta-niveau d'un objet : un méta-objet peut donc observer l'objet correspondant par le biais d'introspection et le contrôler par le biais d'intercession. Le *protocole à méta-objets* définit les interactions possibles entre objets et méta-objets, c'est-à-dire l'ensemble des méthodes et réification et d'intercession. Dans un langage réflexif orienté-objet tel que CLOS (*Common Lisp Object System*) [27], GOOPS [19] ou 3-KRS [37, 38], chaque élément du langage est représenté par un méta-objet accessible à l'utilisateur, et chacun de ces méta-objets *encapsule* une représentation ainsi que le comportement de la notion qu'il représente.

Par exemple, dans CLOS et GOOPS, chaque classe est représentée par un objet de type *classe*. Les objets de type *classe* contiennent des informations décrivant la classe qu'ils représentent, telles que son nom, ses attributs, leur valeur par défaut, etc. Ils définissent aussi des méthodes telles que la méthode d'initialisation et la méthode d'allocation des instances (similaire à l'opérateur `new ()` du langage C++) de cette classe. Lorsqu'une instance d'une classe est créée, les méthodes d'allocation puis d'initialisation d'instance de cet objet classe sont appellées. Pour une classe donnée, il est possible de personnaliser le comportement de ces méthodes.

L'objet de type *classe* lui-même est une instance d'un objet représentant la notion de classe et appellé *méta-classe* (en d'autres termes, une méta-classe est la classe d'une classe). La figure 1.2 illustre les relations d'instanciation entre objets et méta-objets en GOOPS (chaque flèche représente une instanciation). Une méta-classe contient des attributs tels que la liste des classes dont hérite la classe qu'elle représente (la « *class precedence list* » ou CPL), un attribut représentant la liste des attributs (en termes CLOS ou GOOPS, on parle de « *slots* ») de la classe qu'elle représente, etc. S'agissant d'un objet

```
; Définition d'une classe et d'une méthode
(define-class <myclass> (<object>)
  (value #:init-value 0))

(define-method (get-value (object <myclass>))
  (slot-ref object 'value))

; Définition d'une méta-classe sans attributs
(define-class <my-metaclass> (<class>))

; Rajoute l'attribut 'new-slot' aux classes qui
; utilisent la métaclasse '<my-metaclass>'
(define-
method (compute-slots (class <my-metaclass>))
  (let ((slots (next-method)))
    (cons '(new-slot #:init-value 0) slots)))

; Création d'une classe
(define-class <myclass2> (<object>)
  (chbouib #:init-value 2)
  (shbweeb #:init-value "bonjour")
  #:metaclass <my-metaclass>)

; Instanciation d'objets
(define object1 (make <myclass>))
(define object2 (make <myclass2>))
```

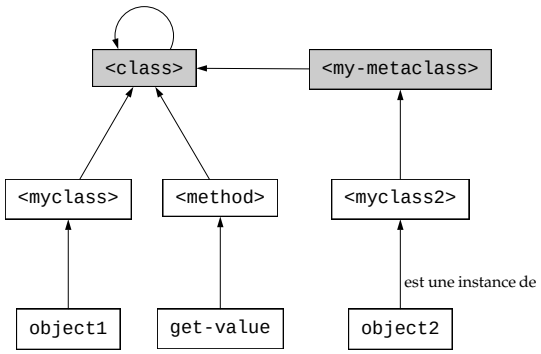


Figure 1.2. Relations d'instanciation dans un programme GOOPS.

comme un autre, une *méta-classe* dispose aussi de méthodes d'allocation et d'initialisation d'instances. Une méta-classe dispose entre autres d'une méthode effectuant le calcul des classes dont hérite une classe, d'une méthode calculant la liste des attributs d'une classe, de méthodes pour la prise en charge d'erreurs telles qu'une tentative d'accès à un attribut inexistant, etc.

Lorsqu'une classe est définie, il est possible de spécifier une méta-classe particulière avec le mot-clef `#:metaclass` suivi de la méta-classe choisie. Ainsi, dans la figure 1.2, on a les relations suivantes :

- `object1` est une instance de `<myclass>` qui hérite de `<object>` ;
- `<myclass>` est une instance de `<class>` ;
- `<class>` est la méta-classe de `object1` ;
- `<myclass2>` est une instance de `<my-metaclass>` ;
- `<my-metaclass>` est la méta-classe de `object2` ;
- `<class>` et `<my-metaclass>` sont toutes les deux des instances de `<class>` (c'est le point de départ de la *méta-hiérarchie*).

La méta-classe `<my-metaclass>` (qui hérite de `<class>`) redéfinit la méthode `compute-slots` qui est appelée lors de la création d'une classe de manière à rajouter automatiquement l'attribut `new-slot` à toutes les classes qui l'utilisent comme méta-classe. Ainsi, la classe `<myclass2>` a automatiquement l'attribut `new-slot` en plus de ceux qu'elle définit. Bien sûr, beaucoup d'autres aspects pourraient être modifiés de la même façon. Notons que dans CLOS comme dans GOOPS, les méthodes sont elles-aussi des objets. La notion de *méta-espace* (« *metaspace* ») introduite notamment dans [56] désigne l'ensemble des méta-objets directs ou indirects d'un objet donné. Dans l'exemple précédent, le méta-espace d'`object2` est constitué des objets `<myclass2>`, `<my-metaclass>` et `<class>`. La section 2.3 donne par la suite des exemples d'utilisation de ces possibilités pour la capture d'état.

La richesse du protocole à méta-objets de ces langages leur procure une puissance et une flexibilité inégalée. C'est pour cette raison que l'utilisation de protocoles à méta-objets tend à s'étendre à d'autres domaines que les langages de programmation, et notamment au domaine des systèmes d'exploitation comme nous le verrons au chapitre 4. Ces expérimentations tendent à montrer que la réflexivité est un concept *transversal* qui peut être appliqué à d'autres domaines que les langages de programmation en offrant des propriétés tout aussi intéressantes.

Chapitre 2. Capture d'état, état de l'art

Pour prévenir les défaillances logicielles ou matérielles d'un système informatique, il est intéressant de pouvoir sauvegarder l'état de l'application critique qui nous intéresse à intervalles réguliers. De nombreux travaux se sont déjà intéressés à la capture d'état d'applications dans des contextes particuliers. Certains de ces travaux s'intéressent à la conception de langages appropriés à la capture de l'état, alors que d'autres cherchent à mettre en œuvre des mécanismes de bas niveau permettant de réaliser des captures *binaires* de l'état de programmes écrits dans des langages compilés indépendamment du langage.

La section 2.1 présente les motivations ainsi que certains principes de base de la capture d'état. Les sections suivantes détaillent différentes techniques de capture d'état ainsi que les avantages et inconvénients de chacune d'entre elles.

2.1. Introduction à la capture d'état

La mémoire de travail physique des ordinateurs, la « mémoire vive », étant volatile, un programme informatique qui s'arrête perd toutes les informations relatives à son état. La mémoire non-volatile d'un ordinateur, c'est-à-dire les supports de stockage tels que les disques durs, est généralement accessible par la notion de *fichier*, dont la manipulation est clairement distincte de l'utilisation de la mémoire volatile[1]. Une application peut créer des fichiers pour y sauvegarder explicitement tout ou partie de son état, de manière à éventuellement pouvoir restaurer cet état de manière explicite lors d'une prochaine exécution. Toutefois, pour sauvegarder des structures de données complexes dans un fichier, l'auteur d'applications devra lui-même définir un format de fichier permettant de décrire aussi fidèlement que possible les données que son programme manipule, et il devra écrire des modules permettant de « mettre à plat » les données dans le fichier (on parle de « *serialization* ») puis de les extraire pour les restaurer (« *deserialization* »). Même si ce travail est facilité par les formats de fichier génériques tel que XML et les outils permettant facilement de les manipuler, il s'avère rapidement fastidieux et représente souvent une partie importante du programme (de l'ordre de 30% du code source d'après Atkinson [2]).

Dans le cadre d'un système sûr de fonctionnement ou à haute disponibilité, il est important qu'une défaillance du système n'entraîne pas la perte des données sur lesquelles travaille l'application. Capturer l'état d'une application critique permet, en cas de faute, de redémarrer rapidement l'application en restaurant son état, de sorte que peu ou pas d'informations auront été perdues. Le simple fait de disposer d'une capture de l'état de l'application permet ensuite de mettre en œuvre différentes stratégies de tolérance aux fautes : réplication passive, réplication semi-active, réplication active (chapitre 1). La possibilité de réaliser régulièrement des captures de l'état d'une application (on parle de « *checkpointing* ») est un point crucial pour un système tolérant aux fautes. Ce mécanisme a donc deux objectifs. D'une part, il donne l'impression au programmeur que les données manipulées par son programme ne sont pas volatiles mais *persistantes* : toutes les données du programme ont une durée de vie illimitée, qu'il s'agisse ou non de fichiers. D'autre part, selon les applications, il peut être nécessaire que le ou les *flux d'exécution* des brins de l'application soient eux-aussi persistants : dans ce cas, chaque brin d'exécution (« *thread* ») de l'application a la possibilité de reprendre son exécution après une défaillance là où elle en était avant. La section 3.6 revient sur cet aspect plus en détail.

[1] Notons toutefois que la possibilité d'associer le contenu d'un fichier à une zone mémoire avec `mmap()` tend à effacer partiellement cet aspect.

Les sections suivantes présentent différents outils logiciels qui permettent de mettre en œuvre, de façon plus ou moins générique, des mécanismes de capture de l'état d'application. Certains d'entre eux sont *transparents*, c'est-à-dire que leur mise en œuvre ne demande que peu, voire pas, de modifications de l'application que l'on souhaite rendre persistante. On parle de *persistance orthogonale* lorsque les mécanismes de persistance sont clairement séparés de la partie fonctionnelle de l'application, respectant ainsi le principe de « séparation des préoccupations » évoqué en section 1.2 [45]. Certains de ces mécanismes ne permettent de réaliser que des sauvegardes des données de l'application, alors que d'autres sont capables d'en sauvegarder également le flux d'exécution. Enfin, alors que certains de ces systèmes ne permettent que de faire une sauvegarde *binaire* de l'état de l'application, c'est-à-dire une sauvegarde qui n'a de sens que sur une plate-forme donnée (architecture matérielle, système d'exploitation, compilateur[1]), d'autres permettent de réaliser des captures d'état *portables* d'une plate-forme à l'autre.

En tolérance aux fautes, les captures d'état portables, c'est-à-dire indépendantes de la plate-forme sur laquelle a été réalisée la capture, sont intéressantes car elles permettent de *diversifier* les plate-formes sur lesquelles fonctionne l'application critique. La redondance de l'application sur des plate-formes différentes permet d'éliminer certaines fautes qui ne peuvent avoir lieu que pour une configuration bien précise [32, p. 100]. C'est pour cette raison que l'outil développé durant le stage qui sera présenté en section 3.2 se donne pour objectif de permettre de façon générique de créer des captures d'état portables dans le cadre d'applications écrites en langage C ou C++.

2.2. Langages persistants

Pour permettre l'écriture d'applications persistantes, une approche consiste à doter les langages de programmation de facilités permettant au programmeur de manipuler de manière transparente des données qui sont en fait persistantes. Le premier langage persistant, PS-Algol [2], était conçu comme une extension d'un langage existant, S-Algol. S-Algol est un langage fortement typé : des vérifications de type peuvent être effectuées à l'exécution, par exemple[2]. Cette fonctionnalité permet aux mécanismes de capture d'état d'être génériques car fournis par le système de prise en charge de l'exécution (« *run-time support system* »). De ce fait, peu de modifications ont dû être apportées au langage et au compilateur. Toutefois, PS-Algol ne fournit pas la persistance de manière transparente : le programmeur doit lui-même explicitement appeler une fonction pour demander l'écriture d'une capture d'état sur disque. Le successeur de PS-Algol, Napier 88 [40], est un langage qui a été spécialement créé dans le but de servir à la création d'un système persistant. Il nécessite donc un apprentissage particulier de la part du programmeur.

Tout comme PS-Algol mais contrairement à Napier 88, le projet *Persistent Java* [3] (ou *PJava*) a pour objectif la mise en œuvre de mécanismes permettant la persistance des données manipulées par le programme avec peu ou pas de modifications à un langage existant, en l'occurrence Java. Un autre objectif de *PJava* est de pouvoir être utilisé avec des programmes existants en Java sans engendrer de perte de performance. L'essentiel des mécanismes requis sont implémentés par la machine virtuelle Java. Le « *ramasse-miettes* » (ou « *garbage collector* ») de celle-ci est responsable de l'écriture des objets « récoltés » vers un support de stockage à long terme.

Le standard CORBA propose une approche permettant la réalisation de captures d'état portables d'applications CORBA grâce à son *Persistent State Service* (PSS) [7]. Le mécanisme proposé n'est pas transparent et repose sur une description faite par le programmeur de l'état qu'il est nécessaire de

[1] Un certain nombre de caractéristiques varient d'une plate-forme à l'autre et notamment la taille des types de bases (e.g. selon la plate-forme, un `int` peut être codé sur 32 ou 64 bits), l'ordre des octets ou « *endianness* », les contraintes d'alignement des données, les conventions d'appel de fonctions, etc.

[2] Cette fonctionnalité existe aussi en C++ pour les classes contenant des méthodes virtuelles et est connue sous le nom de RTTI (*Run-Time Type Identification*).

sauvegarder. Cette description de l'état doit être faite en PSDL (*Persistent State Description Language*) de manière à préciser les attributs de chaque classe qu'il est nécessaire de sauvegarder. À partir d'un fichier de description, un compilateur PSDL produit ensuite le code mettant en œuvre les mécanismes de capture d'état. Toutefois, les spécifications du service [7] précisent que la description PSDL pourrait être générée automatiquement à l'aide d'un outil tel qu'un compilateur ouvert, de manière à rendre la mise en œuvre des mécanismes complètement transparente au programmeur. La section 2.6 reviendra plus en détail sur les compilateurs ouverts.

2.3. Langages réflexifs

Alors que les langages persistants sont conçus entièrement comme des langages (ou des extensions de langages existants) dédiés à l'apport de de la persistance, les langages de programmation *réflexifs* tels que ceux vus en section 1.2, de part leur flexibilité, sont tout-à-fait adaptés à l'ajout de mécanismes non-fonctionnels tels que la persistance de manière orthogonale. Dans [30], G. Kutlu distingue plusieurs opérations qui doivent pouvoir être mises en œuvre par les mécanismes de persistance sur l'application considérée :

- la *transformation* ou *traduction* des données, et notamment des références ou pointeurs, de/vers le support de stockage à long terme ; il s'agit d'une fonctionnalité indispensable qui demande d'avoir des informations sur les données disponibles et sur leur type ;

- accessoirement, afin d'améliorer les performances de la capture d'état et de la restauration, il est utile de pouvoir contrôler les accès en lecture aux objets (de manière à pouvoir éventuellement charger les objets à la demande depuis le support de stockage) ainsi que les accès en écriture (de manière à ne pouvoir capturer l'état que des objets qui ont été modifiés).

Le langage Java a des capacités réflexives limitées : il ne dispose que de fonctions de *réifications* qui permettent simplement de connaître la structure des classes. L'interception des accès aux objets ne peut donc se faire qu'en modifiant le compilateur et/ou la machine virtuelle [30].

En revanche, les langages orientés objets réflexifs basés sur un *protocole à méta-objets* (MOP) procurent une très grande flexibilité (voir section 1.2.3, page 7). Les langages CLOS [27], et son équivalent GOOPS [19] basé sur le langage Scheme, peuvent être étendus par le programmeur sans avoir à modifier le langage ni le compilateur ou l'interpréteur. Comme nous l'avons vu en section 1.2.3 (page 7), le protocole à méta-objets utilisé par l'ensemble du support d'exécution de ces langages permet d'*observer* certains aspects du langage et il peut aussi être modifié et adapté aux besoins de l'utilisateur. La notion de type de donnée est elle-même rendue accessible au programmeur CLOS ou GOOPS puisque les classes sont elles-mêmes représentées par des objets. Pour illustrer cette propriété, la figure 2.1 donne un exemple de programme GOOPS qui utilise la représentation d'une classe pour parcourir la liste des attributs qu'elle définit et sauvegarder leur nom et leur valeur dans une liste.

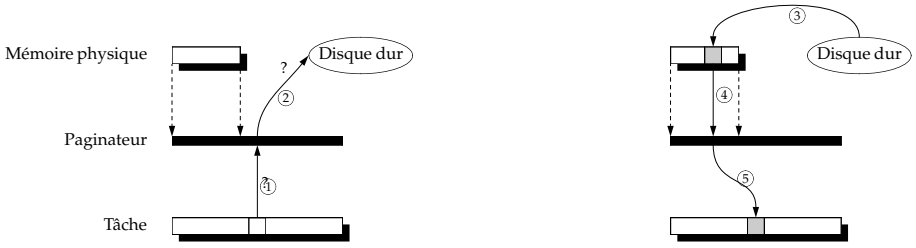
Ces fonctionnalités ont facilité la mise en œuvre de *Persistent CLOS* (PCLOS) [42], un module qui permet de rendre persistant de manière transparente un programme CLOS. PCLOS utilise pleinement l'ouverture du protocole à méta-objets de CLOS pour modifier le comportement de certains éléments de base du langage : une méta-classe `pclos-class` héritant de la classe `class`, par exemple, rajoute des *slots* destinés à contenir des informations relatives à la persistance aux classes qu'elle instancie, elle force les classes qu'elle instancie à hériter d'une classe relative à la persistance, et elle est capable de reconnaître des nouveaux mots-clef permettant de paramétrer la persistance d'un objet.

```
; Renvoie une liste de couples (<nom-du-slot>, <valeur-du-slot>)
(define-method (get-slot-values (object <top>))
  (map
    (lambda (slot)
      (let ((name (slot-definition-name slot)))
        (list name (slot-ref object name))))
    (class-slots (class-of object)))
)

; Exemple
(define-class MyClass ()
  (slot1 #:init-value 0)
  (slot2 #:init-value 1))

(define myobject (make MyClass))

; Affiche la liste des couples (<nom-du-slot>, <valeur-du-slot>)
(display (list 'myobject (get-slot-values myobject)))
```

Figure 2.1. Exemple montrant les capacités réflexives (de *réification*) de GOOPS.

2.4. Systèmes d'exploitation persistants

Un certain nombre de systèmes d'exploitation permettent aux applications qu'ils exécutent d'être persistantes, et ce, de manière totalement transparente. Cette solution se base généralement sur une adaptation du mécanisme de gestion de la *mémoire virtuelle paginée* qui permet d'obtenir de très bonnes performances. Un programme exécuté par un système d'exploitation moderne n'a pas directement accès à la mémoire physique. Au lieu de ça, l'application accède à des adresses en *mémoire virtuelle*, et c'est le système d'exploitation qui se charge de faire la correspondance entre une adresse en mémoire virtuelle et sa position en mémoire physique, lorsqu'elle elle est disponible. En cas de pression mémoire, les *pages mémoires* les moins utilisées sont copiées de la mémoire physique vers une mémoire non-volatile (e.g. un disque dur) ; lorsqu'une page non disponible en mémoire est demandée par une application, le système d'exploitation se charge de ramener la page en question du support de stockage non-volatile à la mémoire physique (figure 2.2). Un système d'exploitation persistant stocke en plus l'association entre une page mémoire et son emplacement dans la mémoire virtuelle d'une tâche de manière à pouvoir réutiliser chaque page mémoire après un redémarrage.



(a) Les fautes de pagination se propagent jusqu'au gestionnaire de la mémoire virtuelle qui récupère la page sur disque.

(b) Une fois la page récupérée en mémoire principle, elle est associée (« *mapped* ») à l'espace d'adressage de la tâche.

Figure 2.2. Mécanisme de gestion de la mémoire virtuelle.

De plus, pour garantir la cohérence des données sauvegardées sur le support de stockage non-volatile, il est nécessaire pour un système persistant que toutes les pages sauvegardées représentent l'état de la mémoire à un moment donné. Après un arrêt brutal du système pendant l'écriture d'une capture, par exemple, l'ensemble des pages mémoires disponibles sur le disque est incohérent. Pour

éviter ce type de problèmes, les systèmes d'exploitation persistants ont généralement recours à un *support de stockage stable* (ou « *stable store* ») tel que décrit dans [49]. Par rapport à un simple pilote de disque dur, un support de stockage stable fournit en plus des opérations de lecture et d'écriture d'un bloc, un opération de *validation* (ou « *checkpoint* ») des données écrites. En l'occurence, les données ne sont validées qu'après écriture de la totalité de l'état du système. Lors du recouvrement, seules les données validées sont prises en considération. La technique sous-jacente décrite dans [49], appellée « *shadowing* » ou « *shadow paging* », consiste à disposer de deux blocs physiques sur disque, un bloc *a* et un bloc *b*, pour chaque bloc logique visible par l'utilisateur. Un tableau spécifi epour chaque bloc logique le bloc physique en cours d'utilisation ; un deuxième tableau permet de déterminer quels blocs logiques ont été modifi ésdepuis la dernière opération de validation. La fi gure2.3 illustre le principe de cet algorithme. Lorsqu'un bloc logique est modifi é, les données sont écrites non pas sur le bloc physique courant mais sur l'autre ; lorsque l'écriture des données est validée, le bloc physique courant devient devient le bloc physique sur lequel les données ont été effectivement écrites. À la validation, un tableau indiquant les blocs physiques courants pour chaque bloc logique est écrit sur le disque de façon atomique.

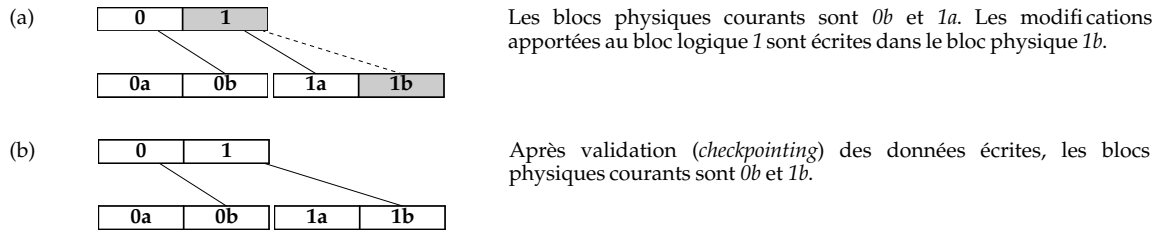


Figure 2.3. Principe de *shadowing* dans un support de stockage stable.

Un certain nombre de systèmes d'exploitation ont été spécialement conçus avec l'objectif de fournir la persistance : Eumel (1979) et son descendant L3 (1988) [34], Grasshopper (1993) [36], KeyKOS (1983) [31] et son descendant EROS (1996) [47]. Dans le cas de Grasshopper, une partie des mécanismes permettant la persistance est présente au niveau du noyau et un certain nombre d'abstractions spécialement conçues permet d'obtenir un fonctionnement proche de celui décrit précédemment. L3, KeyKOS, et EROS ont tous trois été conçus pour être persistants et leurs micro-noyaux intègrent les mécanismes nécessaires.

Outre ces systèmes spécialement conçus dans le but d'atteindre la persistance, différentes expériences ont été menées pour mettre en œuvre la persistance *par-dessus* un système existant : citons notamment *Fault-Tolerant Mach* (FTM) [41] et *Optimistically Recoverable Mach* (ORM) [17] qui sont construits au-dessus du micro-noyau Mach. Ces deux systèmes ont montré certaines limites de ce micro-noyau de première génération. En particulier, Mach intègre un nombre important d'abstractions et de mécanismes qui ne peuvent pas être adaptés à un besoin particulier. Le chapitre 4 reviendra plus en détail sur ces problèmes.

Au contraire, les micro-noyaux dits de seconde génération cherchent à réduire les abstractions et mécanismes fournis par le µ-noyau de sorte à n'imposer aucun choix de conception au concepteur de système d'exploitation client. Selon cette philosophie, une abstraction ne peut rester à l'intérieur du noyau que si la sortir empêcherait la mise en œuvre de fonctionnalités [35]. Dans le cas des noyaux L4 [49] et Fluke [55], l'idée est simplement de *permettre* la mise en œuvre de la persistance au niveau utilisateur. Cette approche permet de conserver une certaine flexibilité et en particulier, elle permet de développer les mécanismes nécessaires à la persistance de manière *orthogonale*, comme une fonctionnalité que l'on peut ajouter au système, et en respectant le principe de *séparation de préoccupations* évoqué au chapitre 1.

En ce sens, la philosophie des micro-noyaux de seconde génération rejoint celle des systèmes réflexifs tels que *Apertos* [56] dont le but est de ne poser aucune limite à leur extensibilité. Le chapitre 4 revient plus en détail sur ces considérations architecturales et sur leur impact sur la mise en œuvre de mécanismes de tolérance aux fautes au niveau du système d'exploitation.

2.5. Bibliothèques de capture d'état

Différentes bibliothèques permettant la capture d'état binaire d'une application ont été développées. La bibliothèque *libckpt* [43], par exemple, est capable de capturer l'image mémoire d'un processus. L'utilisation de services de l'interface de programmation POSIX du système d'exploitation lui permettent d'atteindre de bonnes performances. En particulier, les fonctions ayant trait à la protection de la mémoire virtuelle, telles que `mprotect ()` qui permet de modifier les droits d'accès à une page mémoire, permettent de réaliser une capture d'état *incrémentale*, c'est-à-dire qui ne sauvegarde que les données effectivement modifiées. Toutefois, ce mécanisme de sauvegarde étant totalement transparent à l'application, il devient impossible au programmeur de restaurer explicitement les ressources *externes* à l'application telles que les descripteurs de fichier, les connexions réseau, etc. (voir section 3.3). Cependant, certains types de ressources externes sont pris en charge et restaurés automatiquement par la bibliothèque. Dans le cas des descripteurs de fichier, *libckpt* intercepte les appels systèmes relatifs à la manipulation de descripteurs de fichiers (tels que `open ()`, `read ()`, `write ()`) et *infère* l'état attaché aux descripteurs de fichier, de manière à le restaurer lors du redémarrage de l'application. Notons que la bibliothèque *libft* [22] utilise une approche similaire. Une présentation de moyens d'interception sera donnée au chapitre 4.

2.6. Compilation source-à-source et compilation ouverte

Pour les applications écrites dans un langage compilé non-réflexif tel que C ou C++, une solution pour permettre la capture d'état portable est de faire de la *compilation source-à-source* (ou *traduction de source*) à partir du code source original de l'application, de manière à introduire automatiquement du code prenant en charge la capture d'état (voir figure 2.4). Cet approche est celle du projet PORCH [44] (qui signifie « *Portable Checkpoint Compiler* ») et du « *Process Introspection Project* » [12]. Pour ces deux projets, un compilateur source-à-source dédié à été développé. Il lit les fichiers sources un à un et y ajoute des appels à des fonctions d'une bibliothèque qui, à l'exécution, prend en charge la capture d'état. La structure des types de données définis par l'utilisateur est également analysée par ce compilateur qui génère automatiquement des fonctions de sauvegarde ou de mise-à-plat (*serialization*) pour chacun de ces types. Ceci implique que le compilateur doit être capable d'interpréter toutes les définitions de type qui peuvent être faites dans ce langage, notamment les définitions de structures. Lors du processus de « traduction » du code source, le compilateur source-à-source rajoute du code au début de chaque fonction pour permettre la sauvegarde et la restauration de la pile de manière portable.

Les *compilateurs ouverts* tels que OpenC++ [5] ou OpenJava [54] ont pour objectif de fournir un cadre générique permettant de faire de la compilation source-à-source. OpenC++ définit ainsi un *protocole à méta-objets* (voir section 1.2.3) qui peut être utilisé par un *méta-programme* pour produire un code source modifié. On parle alors de *réflexivité à la compilation* (« *compile-time reflection* »). Concrètement, le méta-programme peut spécialiser un certain nombre de méthodes telles que, par exemple, `CompileMemberFunctionCall ()`, `CompileWriteDataMember ()`, ou encore `CompileVarDeclaration ()` qui seront appelées lorsque le code lu par OpenC++ contient un appel à une fonction membre, l'assignation d'un attribut de classe, ou une déclaration de variable. Ces méthodes peuvent renvoyer un arbre syntaxique modifié. Un compilateur ouvert peut donc être utilisé pour instrumenter le code source de la même façon que les compilateurs source-à-source dédiés présentés précédemment. OpenC++ a ainsi été utilisé pour faire une capture d'état portable des données

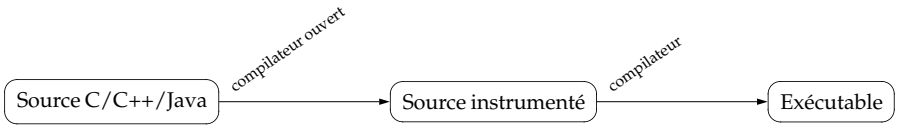


Figure 2.4. Processus d'instrumentation de code à l'aide d'un compilateur source-à-source.

de programmes écrits en C++ [28]. Pour chaque classe, des méthodes de sauvegarde et de restauration de l'état d'instances sont automatiquement générées et exportées *via* une interface CORBA. Le travail présenté dans [28] utilise le format CDR (« *Common Data Representation* ») défi ni par le standard CORBA pour sauvegarder les données d'une manière indépendante de la plate-forme et du langage de programmation. Les captures d'état ainsi créées peuvent être échangées entre une implémentation C++ et une implémentation Java du même ensemble de classes.

L'utilisation d'un compilateur ouvert permet d'envisager la mise en œuvre d'un certain nombre d'optimisations. Une première optimisation consiste à ne sauvegarder que l'état des données qui ont effectivement été modifi ées : en rajoutant du code adéquat autour des assignations de variables de manière à marquer les variables qui ont été modifi ées, il devient possible de faire des captures d'état partielles [29]. Dans le cas d'applications à plusieurs brins d'exécution, le compilateur ouvert peut aussi être utilisé pour instrumenter le code applicatif de manière à suivre les dépendance entre objets et entre brins d'exécutions, comme le propose *libooft* [26].

2.7. Utilisation des informations du compilateur

Les informations de débogage insérées dans le fi chier exécutable par le compilateur[1], contiennent de nombreuses informations donnant une description des types de données utilisés par le programmes, ainsi que des variables défi nies dans chaque fi chier source, des fi chiers d'entêtes qui ont été inclus, etc. Ces informations sont destinées à être utilisées par des débogueurs tels que GDB, le débogueur du projet GNU[2]. Différents standards *de facto* pour ces informations existent et l'un des plus répandus est le format « *stabs* » [39].

De par leur richesse, ces informations sont particulièrement utiles pour les applications telles que la capture d'état portable ou la mise-à-plat (« *serialization* ») de structures de données. Le *Texas Persistent Store* [48], un système dédié à la capture d'état d'applications C++, les utilise pour obtenir une description des structures de données à l'exécution (*Run-Time Type Description* ou RTTD) [24]. Le système n'est capable de sauvegarder que l'état des objets alloués dynamiquement dont il a connaissance grâce à une redéfi nition adéquate de l'opérateur `new ()`. *Texas* utilise les fonctionnalités de protection de l'accès à la mémoire virtuelle offertes par les systèmes de type Unix récents[3] qui lui permettent de restaurer les données persistantes à la demande seulement (« *page-fault-time swizzling* »).

Le système *Tui* [50], dédié à la migration de programmes écrits en langage C, au cours de leur exécution, vers différentes architectures, repose également sur l'interprétation des informations de débogage fournies par le compilateur. Le système utilise une version modifi ée du compilateur ACK (*Amsterdam Compiler Kit*) qui rajoute des informations facilitant la réalisation d'une sauvegarde portable de la pile d'exécution (voir section 3.6, page 34).

Le système qui a été développé durant ce stage utilise la même technique que le projet *Tui*, c'est-à-dire qu'il se base entièrement sur les informations de débogage pour réaliser une capture d'état

[1]Pour GCC, le compilateur C du projet GNU, l'option permettant d'ajouter des informations de débogage est l'option `-g`. Pour explicitement demander d'utiliser des informations de débogage au format « GNU Stabs », utiliser `-gstabs+`.

[2]GDB est un logiciel libre disponible à *<http://www.gnu.org/software/gdb/>*.

[3]Par exemple, l'appel système `mprotect ()` permet de restreindre les droits d'accès à une page de mémoire virtuelle, et un gestionnaire de fautes de pagination (« *page-fault handler* ») peut être installé par l'appel système `signal ()`.

portable. Contrairement à celui-ci, il ne permet pas de réaliser de manière transparente une capture de la pile d'appel. Par contre, sa conception modulaire sépare clairement la partie *réifi cation* des informations données par les informations de débogage du reste du système. Même si l'implémentation actuelle lit des informations de débogage au format *stabs* [39], cette abstraction la rend complètement indépendante du format de débogage. De plus sa mise en œuvre offre davantage de flexibilité tout en étant performante. La conception et la mise en œuvre de cet outil sont détaillées au chapitre 3.

Chapitre 3. Capture de l'état d'applications écrites en langage C ou C++

Cette section s'intéresse à la capture d'état *portable* d'applications écrites en langage C ou C++. C et C++ sont des langages compilés largement répandus mais n'offrant pas la flexibilité de langages réflexifs tels que ceux présentés précédemment. En particulier, C et C++ ne fournissent pas aux applications qui les utilisent une représentation des structures de données qu'elles manipulent ni de la structure du programme, informations essentielles pour mettre en œuvre un mécanisme de capture d'état portable.

Certains travaux présentés au chapitre 2 ont permis la mise au point d'outils permettant de réaliser des captures d'état portables d'applications écrites en C ou C++. Tous ces projets proposent de combler les manques du langage en rendant disponibles ces informations aux programmes. Pour ce faire, beaucoup utilisent des techniques de compilation (section 2.6), d'autres cherchent à tirer parti des informations de débogage introduites par le compilateur (section 2.7).

Dans le cadre de ce stage, un outil utilisant cette dernière approche a été développé en gardant à l'esprit l'approche réflexive et ce de manière aussi flexible que possible, laissant ainsi à l'utilisateur le choix de la façon dont il souhaite l'utiliser. La section 3.2 détaille la conception et la mise en œuvre de cet outil.

3.1. Besoins pour une sauvegarde portable de programmes C

Le langage C est un langage compilé et en tant que tel se montre peu flexible. Aucun mécanisme d'introspection n'est fournit au programmeur : il est impossible de connaître le type d'un objet à partir d'un pointeur, par exemple, et il est encore moins possible d'obtenir une description de types de données et une liste des données existantes (par la suite, on parlera d'une manière générale d'*objets* pour désigner des données de l'application). Le langage C++ dispose pour sa part d'une capacité d'identification des types à l'exécution (RTTI) : la fonction `dynamic_cast<T *>` permet de s'assurer qu'un *cast* vers un type spécifique est valide, et la fonction `typeid ()` permet de comparer le type de deux objets. En revanche, aucune description des types de données n'est disponible, et c'est pour cette raison que le projet *Texas* [48] utilise les informations de débogage (voir section 2.7). Un programme C++ peut par ailleurs redéfinir l'opérateur d'instanciation `new ()`, ce qui peut permettre, par exemple, de garder une trace des objets qui ont été instanciés.

Le langage C est en outre un langage de bas niveau. À ce titre, il permet par exemple d'adresser directement la mémoire (virtuelle) à l'aide d'un pointeur, car aucun contrôle n'est effectué sur la valeur des pointeurs. Il est également possible de faire des opérations arithmétiques sur les pointeurs, telles que des comparaisons d'ordre, qui rendent les programmes dépendants d'une plate-forme donnée. Enfin, les pointeurs sont non typés : il est tout-à-fait possible de faire pointer un pointeur de type pointeur-sur-entier sur une variable de type float, ou encore d'utiliser le type pointeur générique (`void *`), qui peuvent tous deux facilement mener à des erreurs de programmation. Le langage C++, comme le langage Java, introduit la notion de *référence* qui supprime la plupart des problèmes liés aux pointeurs : une référence C++ est nécessairement initialisée (en Java, une référence est soit nulle soit initialisée), des vérifications de type sur les références sont effectuées à la compilation, et enfin il

n'existe pas d'opérations arithmétiques sur les références. Toutefois, C++ est un « sur-ensemble » du langage C et il hérite donc de tous les défauts de ce dernier, et notamment des pointeurs.

Une analyse des informations nécessaires à la sauvegarde portable des données d'un programme écrit en langage C permet d'identifier les points suivants :

- le mécanisme de capture d'état doit avoir accès à une *description des types de données* utilisés par l'application ;
- il doit également avoir accès à une *liste des variables* définies par le programme, avec une indication sur la portée de la variable et la description du type de chacune de ces variables ;
- pour différencier les variables portant le même nom il est nécessaire de connaître le *nom du fichier source* où elles sont définies, et même éventuellement le *numéro de ligne* où elles sont définies dans le cas des variables dont la portée est celle d'un bloc (mot-clef `static`) ;
- pour permettre la sauvegarde des références (pointeurs), un *identifiant unique* indépendant de la plate-forme doit pouvoir être attribué à chaque objet en mémoire ;
- la mémoire allouée dynamiquement étant non typée, il est nécessaire que le programmeur dispose d'un moyen d'*allocation typée de mémoire dynamique* qu'il puisse utiliser à la place des fonctions telles que `malloc ()` ; ceci implique que la persistance des données allouées dynamiquement ne pourra pas se faire de manière totalement transparente au programmeur ; cependant, en pratique, les modification requises pour typer la mémoire dynamique peuvent être assez faibles comme nous le verrons en section 3.3.

Ces besoins ont guidé la conception du système de capture d'état portable décrit en section 3.2.

3.2. Mise-en-œuvre de mécanismes d'introspection et de sauvegarde portable

3.2.1. Aperçu de l'architecture

L'objectif recherché a été de concevoir un outil mettant en œuvre des mécanismes permettant de réaliser des captures d'état portables d'applications écrites en langage C ou C++. L'utilisation des informations de débogage (ou « *stabs* ») produites par le compilateur (voir section 2.7) est apparue comme une technique permettant d'obtenir rapidement et simplement des informations relatives aux types de données et aux données manipulées par un programme.

Une bibliothèque d'*introspection*, appelée *Ego* a été développée : à partir des informations de débogage, elle rend disponible à l'utilisateur les informations sur la structure des types, sur les variables définies dans le programme, et même sur la structure du programme. Elle a été développée en tenant compte des informations requises pour la capture d'état portable et énoncées en section 3.1. La façon dont sont *réifiées* ces informations (i.e. dont elles sont rendues disponibles à l'utilisateur) est définie par le « *méta-modèle* » présenté en section 3.2.2.

Les informations sur les variables et les descriptions de type fournies par *Ego* sont ensuite utilisées par une bibliothèque de capture d'état portable appelée *Pego*[1]. Cette bibliothèque parcourt les données du programme (i.e. les variables *non-locales* définies par le programme) et les sauvegarde de manière portable. Ces données peuvent être sauvegardées dans un fichier ou être envoyées à une réplique de l'application à travers le réseau. La figure 3.1 illustre schématiquement l'architecture de l'ensemble.

[1] *Pego* signifie « *Portable & Persistent Ego* », un égo portable et persistant.

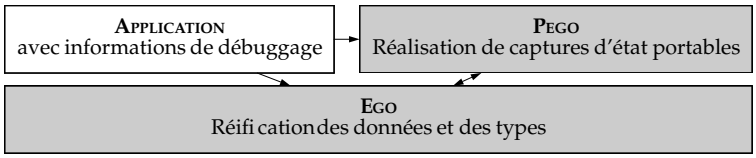


Figure 3.1. Architecture de l'outil de sauvegarde portable *Pego*, basé sur la bibliothèque d'introspection *Ego*.

3.2.2. Définition d'un méta-modèle du langage C

À partir des données produites par le compilateur, *Ego* fournit au programmeur une représentation facilement exploitable des données manipulées par son application. Toutes les données nécessaires à la capture d'état portable sont rendues accessibles. La section 3.2.2.1 présente le principe de base utilisé par *Ego*. Plus de détails sont disponibles dans le manuel de référence [8].

3.2.2.1. Représentation des données

Les « *stabs* » [39] contiennent des informations sur toutes les variables définies par un programme, quelle que soit leur portée, ainsi que sur leur type. À la lecture de ces informations, *Ego* créée des objets représentant les variables du programme, et contenant des informations telles que leur nom, leur portée, leur adresse, un pointeur vers l'objet représentant leur type (voir section 3.2.2.2), un pointeur vers l'objet représentant le fichier où elles ont été définies, ainsi qu'un numéro de ligne indiquant à quelle ligne apparaît la définition dans ce fichier. Par analogie avec les systèmes réflexifs tels que décrits par Pattie Maes [37] et avec les protocoles à méta-objets [27], nous appellerons par la suite ces objets dont le rôle est de décrire d'autres objets des « *méta-objets* » ou « *méta-variables* ».

En langage C, les données allouées dynamiquement (e.g. avec `malloc ()`) ne sont pas typées. Pour que *Ego* puisse « savoir » quelles zones mémoires ont été allouées dynamiquement et quel sont leur taille et leur type, il est nécessaire que le programmeur utilise des fonctions d'allocation et désallocation particulières fournies par *Ego*[1] . De plus, pour l'utilisateur de *Ego* (en l'occurrence, la bibliothèque de capture d'état *Pego*), il est intéressant de pouvoir obtenir directement un « méta-objet » à partir d'une adresse, qu'elle pointe vers une variable statique ou vers une zone allouée dynamiquement. L'abstraction de « *méta-objet générique* » (ou plus précisément, dans *Ego*, de « *Memory Object* » [8]) sert précisément à répondre à ce besoin : à chaque donnée de l'application, qu'elle soit dynamique ou non, est associée un tel méta-objet (voir figure 3.2). Conceptuellement, les méta-objets représentants les variables *héritent* des méta-objets génériques. Pour une variable donnée, il existe donc un lien entre ces deux types de méta-objets.

En pratique, de manière à informer *Ego* de la création d'objets alloués dynamiquement et de leur type, ainsi que de la destruction ou du redimensionnement d'un objet, le développeur doit utiliser un jeu de macros définies par *Ego*. Par exemple, les lignes suivantes

```
struct chbouib *p ;
p = malloc (x * sizeof (struct chbouib)) ;
free (p) ;
```

doivent être remplacées par :

```
struct chbouib *p ;
p = ego_memobj_alloc (&self, struct chbouib, x) ;
```

[1] Le système *Tui* [50] tente de déduire le type de données allouées dynamiquement pour les allocations qui respectent la forme `malloc (size * sizeof (type)) ;` mais cette technique contraint également le programmeur à respecter cette forme particulière.

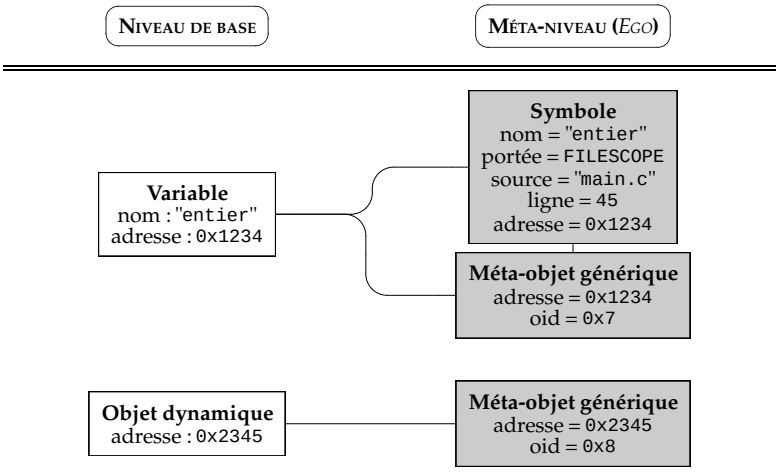


Figure 3.2. Relation entre les données de l'application (niveau de base) et le « méta-niveau » proposé par Ego.

```
ego_memobj_free (&self, p);
```

où `self` est un objet de type `ego_t` représentant l'application. La macro d'allocation mémoire permet à *Ego* de connaître l'adresse et la taille de l'objet alloué. Pour associer un « méta-type » tels qu'ils seront présentés en section suivante, la macro instancie une variable statique du type donné en paramètre. À partir de l'adresse de cette variable (qui ne prend aucune place en mémoire puisqu'elle est non initialisée), une table de hachage des variables est interrogée et permet d'obtenir le méta-objet et le méta-type associés. La modification des méthodes d'allocation dynamique requiert donc peu de travail et peut souvent être effectuée de manière automatique au moyen de substitutions d'expressions régulières. Ceci nous a permis d'adapter rapidement deux applications existantes comme nous le verrons en section 3.3.

3.2.2.2. Informations de type

En langage C, le programmeur a la possibilité de définir des types en fonction d'autres types avec le mot-clef **`typedef`** et les mots-clef **`struct`** et **`union`**. Dans le cas de **`typedef`**, il existe relation entre le type qui est défini et le type qui sert de base à la définition : le type défini peut être *égal* au type de base, il peut être un type *pointeur* sur le type de base, ou encore un type *tableau* d'éléments du type de base. Dans *Ego*, les objets représentant les types de données (on peut parler de « *méta-types* ») reflètent cette relation par la notion de *référence entre types* et de *type de référence* illustrée par la figure 3.3 (les flèches représentent les liens ou « références » entre méta-types, les boîtes à angles arrondis représentent les champs de la structure).

Le cas des structures et unions est particulier : dans le cas où un type représente une structure ou une union, l'objet qui le représente contient une liste d'objets représentant ses champs. Chacun des objets représentant un champ contient des informations telles que le nom du champ, sa position à l'intérieur de la structure, et un pointeur vers l'objet représentant son type.

3.2.2.3. Fichiers source et fichiers d'entête

Comme le montre la section 3.1, pour pouvoir désigner de manière non-équivoque une variable ou un type, il est nécessaire de savoir où elle/il a été défini. Ceci implique qu' un objet représentant un type ou une variable, tels que décrit en sections 3.2.2.1 et 3.2.2.2, doit contenir des informations sur l'endroit (fichier source ou fichier d'entête) où est défini-e le type ou la variable qu'il représente. Or,

```
/* Definition d'une structure et d'un type associe */
typedef struct _person
{
  char *name ;
  int   age ;
} person_t ;

/* Definition d'un type pointeur-sur-la-structure */
typedef person_t *person_ptr_t ;
```

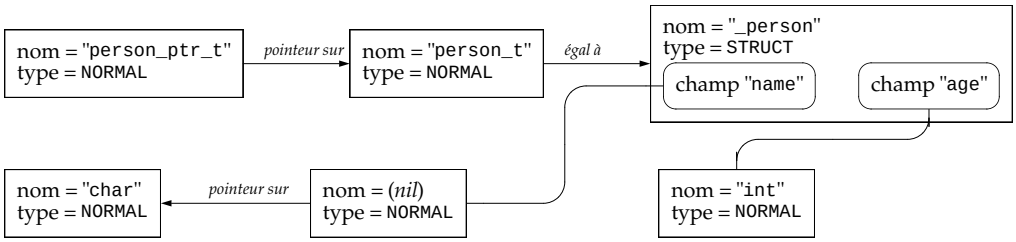


Figure 3.3. Représentations de types de données C par des méta-types dans *Ego* (objets `ctype_t`).

beaucoup de types de base sont définis dans des fichiers d'entête spécifiques au compilateur utilisé et à la plate-forme choisie. Par exemple, le fichier d'entête où est défini un type comme `size_t` varie d'une plate-forme à l'autre. Par conséquent, pour qu'une description de type soit portable d'une plate-forme à l'autre, il est préférable d'utiliser le nom d'un des fichiers sources où ce type est défini plutôt que le nom du fichier d'entête, de sorte à former un couple fichier-type qui soit valable sur toutes les plate-formes.

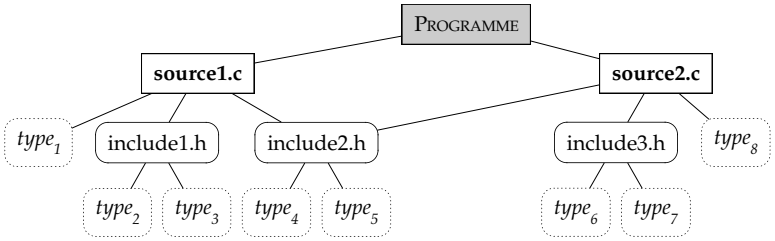


Figure 3.4. Définitions de type et hiérarchie des fichiers sources et fichiers d'entête.

Par conséquent, la bibliothèque d'introspection doit être capable de refléter la hiérarchie d'inclusion des fichiers d'entête. Particulièrement, il est indispensable que l'objet représentant un fichier d'entête contienne à son tour des liens vers les fichiers sources qui l'incluent. La figure 3.4 montre l'arborescence correspondant à un programme constitué de deux fichiers sources, définissant un certain nombre de types (les boîtes en pointillés) dont certains sont locaux ($type_1$, par exemple, est local au fichier `source1.c`), et d'autres sont définis dans des fichiers d'en-tête dont certains peuvent être partagés entre plusieurs fichiers sources (e.g. $type_4$ est défini dans `include2.h` qui est inclus à la fois par `source1.c` et `source2.c`).

3.2.3. Mise-à-plat des données à sauvegarder

La principale tâche de la bibliothèque de capture d'état portable *Pego* est de sauvegarder les données manipulées par un programme de manière linéaire dans un fichier. Or les structures de données qui peuvent être créées en langage C ou C++ peuvent être *imbriquées*, ce qui implique une certaine

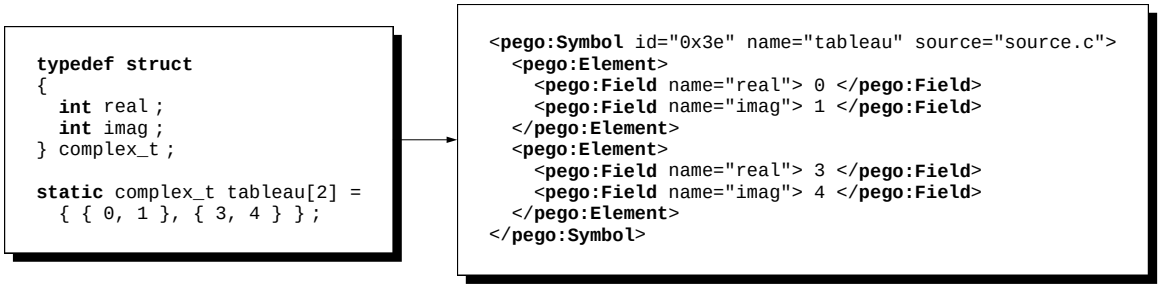


Figure 3.5. Capture d'état de variables en XML.

hiérarchie dans les structures de données : les éléments d'un tableau peuvent être des structures qui à leur tour peuvent contenir des tableaux et des structures, etc. Pour représenter de telles structures de données, il est nécessaire que le format de sauvegarde soit capable de représenter cette hiérarchie. Clairement, le format XML est apparu comme répondant tout-à-fait à ces besoins, puisqu'il permet précisément de stocker des structures de données arborescentes.

Un premier prototype de la bibliothèque de capture d'état portable *Pego* utilisait exclusivement le format XML pour la sauvegarde et le chargement de captures d'état. Les versions suivantes permettent cependant de réaliser des sauvegardes dans d'autres formats, et notamment dans un format binaire spécialement conçu et décrit en section 3.2.5. Des types de nœuds formant l'arborescence du document ont été définis pour représenter d'une part les variables et les objets alloués dynamiquement, et d'autre part les élément constitutifs d'une structure de donnée tels que les éléments d'un tableau ou les champs d'une structure. Chaque structure de données est donc décomposée récursivement en éléments s'il s'agit d'un tableau, et en champs s'il s'agit d'une structure ou d'une union. Le contenu de ces éléments de base est alors soit un pointeur, soit un scalaire (entier, réel, caractère, énumération). Dans le cas d'un scalaire, le contenu du nœud XML correspondant est écrit sous forme d'une simple chaîne de caractères (figure 3.5). Notons que les nœuds représentant des variables ou objets alloués dynamiquement ont tous une propriété *id* qui donne l'identifiant unique qui leur a été attribué par *Ego* (section 3.2.2.1).

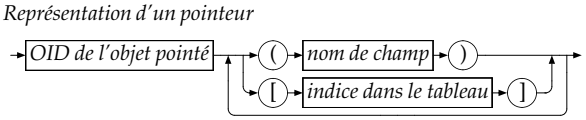


Figure 3.6. Diagramme de syntaxe pour la mise-à-plat des pointeurs (*pointer swizzling*) par *Pego* au format XML.

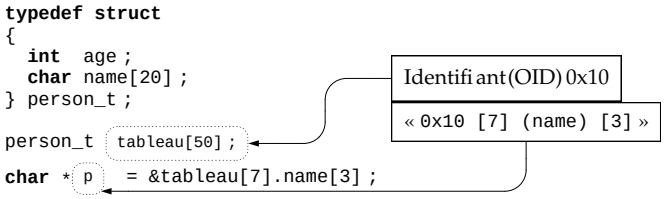


Figure 3.7. Exemple de mise-à-plat d'un pointeur par *Pego*.

Pour les données de type pointeur, *Pego* interroge lors de la sauvegarde *Ego* pour obtenir le « méta-objet » associé aux données désignées par le pointeur à sauvegarder. De cette façon, *Pego* a accès à l'identifiant unique attribué à l'objet pointé ainsi qu'à une description de son type. Le contenu

du nœud XML représentant une valeur de type pointeur commence toujours par l'identifiant de l'objet pointé. Cet identifiant est suivi par le décalage (« *offset* ») à l'intérieur de cet objet, dans le cas où le pointeur pointe sur un champ particulier d'une structure, ou sur un élément particulier d'un tableau. La figure 3.6 donne le diagramme de syntaxe des pointeurs mis-à-plat par *Pego* ; la figure 3.7 donne un exemple concret de sauvegarde d'un pointeur.

3.2.4. Processus de sauvegarde et de restauration

Les données utilisées par une application utilisant *Pego* sont sauvegardées dans un format de fichier hiérarchique tel que XML, comme nous le verrons en section 3.2.5. *Pego* construit une représentation sous forme arborescente du fichier de sauvegarde avant d'écrire le fichier. Cet « arbre de sauvegarde » est constitué de « nœuds de sauvegarde », chacun de ces nœuds pouvant représenter une variable, une donnée dynamique, un tableau d'éléments, un champ de structure, etc. L'écriture de cet arbre de sauvegarde dans un fichier (réalisation d'un « *checkpoint* ») est une étape critique en termes de performance : l'écriture de la capture d'état doit être le moins pénalisant possible pour l'application.

Dans un premier prototype, l'arbre de sauvegarde représentant les données de l'application était construit « à la demande », c'est-à-dire seulement lorsque l'application faisait appel à la fonction de sauvegarde de l'état. Cette approche s'est montrée peu efficace et a été remplacée par la suivante :

- pour chaque variable globale ou `static`, l'ensemble des nœuds de sauvegarde est créé durant la phase d'initialisation ;
- pour les données allouées dynamiquement, le nœud de sauvegarde correspondant n'est créé que lorsque que l'application réalise une capture d'état, et il est réutilisé lors d'une prochaine capture d'état si la donnée est toujours référencée par une variable.

Pour être informé de la création et de la destruction d'objets, *Pego* enregistre des fonctions auprès de *Ego* qui sont appellées lorsque un de ces évènements se produit (on parle de « *hooks* »). De plus, chaque nœud de sauvegarde peut être directement attaché au *memory object* qu'il représente (section 3.2.2.1), ce qui permet de faire directement le lien entre les méta-objets (le *modèle*) et la représentation arborescente du document (la *vue*), comme le montre la figure 3.8. En ce sens, la mise en œuvre de *Pego* est assez proche des patrons de conception *Modèle-Vue-Contrôleur* ou MVC [14]. Cette ensemble de motifs architecturaux est particulièrement répandu dans les applications graphiques où une partie de l'application, le *contrôleur*, fait le lien entre le modèle représentant les données manipulées, et la vue des ces données à l'écran, de sorte qu'une modification de l'un entraîne une mise-à-jour de l'autre. Ce schéma se retrouve dans l'architecture de *Pego*, à ceci près que *Pego* n'a aucun moyen d'être informé de la modification du contenu d'un objet et que, par conséquent, le contenu des nœuds représentant les données du programme doit être entièrement mis à jour lors de la réalisation d'une capture d'état[1].

Pour garder une trace des données qui ont déjà été inclues dans la capture d'état, et aussi pour éviter de sauvegarder plusieurs fois les mêmes objets, une table de hachage contenant les identifiants des objets sauvegardés (les *OID*, section 3.2.2.1) est mise-à-jour à chaque ajout d'un objet à la sauvegarde. La sauvegarde à proprement parler se passe de la façon suivante :

1. on créée une table de hachage vide dont le but est de permettre d'obtenir, à partir d'un *OID*, la liste des objets déjà sauvegardés qui référencent cet objet ; on parle de table des « *références non-résolues* ; »

[1] Des techniques de compilation ouverte, telles que celles présentées en section 2.6, permettraient d'instrumenter le code de manière à informer le méta-niveau de la modification d'un objet du programme [29].

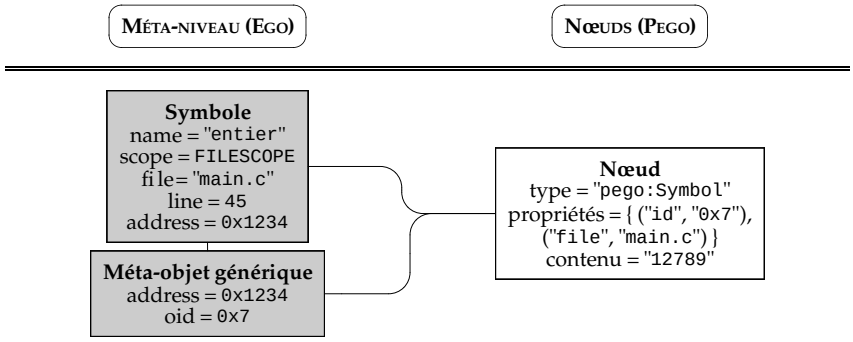


Figure 3.8. Lien entre le *modèle* de l'application (les méta-objets) et la *vue* correspondante du support de stockage (nœuds de l'arbre de capture d'état).

2. on parcourt la liste des variables à portée globale du programme en mettant à jour le contenu des nœuds qui les représentent ;

   – si une variable est de type pointeur, on utilise les services d'*Ego* pour obtenir l'OID de l'objet référencé ; si l'objet qu'elle référence n'a pas déjà été sauvegardé, alors on l'ajoute à la liste des références non-résolues vers ledit objet ; également, la liste des objets référençant l'objet considéré est vidée ;

   – dans le cas d'une variable de type « simple », son contenu est mis à jour ;

3. enfin, la table des références non-résolues est parcourue (à ce stade, elle ne contient que des objets faisant référence à des données allouées dynamiquement ou à des fonctions) ; les objets dynamiques et les fonctions référencées sont donc ajoutés un à un à l'arbre de sauvegarde, jusqu'à ce qu'il ne reste plus aucune référence non résolue.

Notons que le nœud de sauvegarde représentant une fonction ou une donnée dynamique référencée par un pointeur est détaché de l'arbre de sauvegarde à la fin de l'algorithme, sans pour autant être détruit. Lors d'une prochaine capture d'état, ce nœud de sauvegarde sera réutilisé si et seulement si la fonction ou donnée est toujours référencée par une variable. Cette méthode a deux avantages :

- seules les fonctions et seules les données dynamiques effectivement référencées sont sauvegardées ; par conséquent, les « *fuites de mémoire* » ne sont pas persistantes et la taille de la sauvegarde est minimisée ;

- si une même fonction ou donnée dynamique est encore référencée lors d'une prochaine sauvegarde, on dispose toujours de son nœud resté attaché à son méta-objet ; il suffit donc de le mettre à jour et de le rajouter dans l'arbre, ce qui n'implique aucun surcoût.

Enfin, les variables constantes[1] sont inclues dans la sauvegarde car elles sont susceptibles d'être référencées. Toutefois, leur contenu n'est pas sauvegardé puisqu'il est constant. Il peut arriver que certains pointeurs pointent vers des adresses inconnues par *Ego* : c'est notamment le cas si un pointeur n'a pas été initialisé, ou si il pointe vers des données initialisées telles qu'une chaîne de caractères

[1]Les données constantes telles que des variables globales définies avec le mot-clef **`const`** peuvent être inclues dans une section en lecture seule telle que *rodata* (pour « *read-only data* »). Cependant, ceci dépend de la plate-forme et une variable constante peut ne pas être dans une section en lecture seule sur une plate-forme donnée.

allouée statiquement. Dans ce cas, un message d'erreur s'affiche et le pointeur n'est pas sauvegardé. En pratique, cette situation est rarement problématique car un pointeur qui pointe sur une donnée constante de ce type est généralement lui aussi constant, auquel cas sa valeur est inchangée d'une exécution à l'autre.

Le processus de restauration de l'état se déroule de la même manière à ceci près que le parcours final des références non-résolues sert à assigner leur valeur aux variables de type pointeur. Également, le contenu des variables constantes, si il figure dans la capture d'état, n'est pas restauré.

3.2.5. Modules de stockage hiérarchiques

Le format XML, s'il se prête bien à ce type d'expérience notamment du fait de sa structure et de sa lisibilité, s'avère très vite relativement lourd. La taille des fichiers créés peut très vite devenir importante, ce qui demande un temps d'écriture plus long. Également, le temps de conversion des valeurs des variables en chaîne de caractères peut être assez coûteux. La possibilité de créer des fichiers dans un format *binaire* plus compact mais portable s'est donc avérée nécessaire. Cette possibilité ne doit pas empêcher de pouvoir réaliser des sauvegardes au format XML qui peut être très pratique car il est lisible par le programmeur, ce qui permet par exemple d'observer facilement les différences entre deux captures (à l'aide d'un simple outil tel que *diff*).

La version actuelle de *Pego* [9] offre donc la possibilité de choisir le format de lecture et d'écriture des données, par le biais de modules de stockage hiérarchiques. Ces modules doivent fournir une même interface permettant de manipuler un arbre de données à la manière de ce que permet le format XML. La structuration des données de manière arborescente n'a pas été remise en cause puisqu'elle tout-à-fait adéquate. Un module de lecture et écriture de sauvegarde dans un format binaire compact a été développé. Ce format hiérarchique créé spécifiquement pour *Pego* est à rapprocher du format EBML[1], une variante binaire de XML. Les seules données textuelles encore présentes dans ce format sont les noms des variables ainsi que les noms des champs de structure ou d'union. Les entiers sont sauvegardés sur 64 bits de manière indépendante de l'architecture et quelle que soit leur taille originale ; les réels de type **`float`** ou **`double`** étaient initialement sauvegardés sous une forme à virgule fixe constituée de 64 bits pour la partie entière (signée) et de 64 bits pour la partie décimale (non signée). Toutefois, la représentations en virgule fixe devient insuffisante pour des nombres « extrêmement grands » ou « extrêmement petits » : en l'occurrence, il est impossible d'encoder une valeur supérieure à 2^{32} - 1 (environ $4{\times}10^{9}$) ou plus petite (en valeur absolue) que 2^{-64} (environ $5{\times}10^{-20}$), alors que le type **`double`** défini par le standard IEEE 754 peut représenter un réel allant de $1{,}8{\times}10^{308}$ à $2{,}2{\times}10^{-308}$. Pour cette raison, le dernier prototype utilise directement un format à virgule flottante (il s'agit d'un **`long double`** tel que défini par le standard IEEE 854) qui s'avère même plus performant que le format à virgule fixe car aucune conversion n'est requise. Les performances du module de stockage binaire sont sensiblement meilleures que celles du module XML, et la taille des fichiers créés est bien moindre, comme nous le verrons en section 3.5.

En outre, un module de « stockage réseau » utilisant le module binaire a été écrit. Au lieu de lire les données à partir d'un fichier, il attend une connexion sur un port précisé par l'utilisateur et interprète les données envoyées par le client comme des données binaires de capture d'état. De la même façon, au lieu d'être écrites dans un fichier, les données sont envoyées vers une machine et sur un port précisé par l'utilisateur. Ce module permet donc de mettre en place un mécanisme de *réplication passive* tel que décrit au chapitre 1. L'utilisateur ayant la possibilité de choisir un module d'entrée et un module de sortie différents, il est même possible de créer des configurations telles que celle présentée en figure 3.9 : l'application considérée s'exécute sur une première machine et utilise le module réseau (noté « *net* ») pour faire parvenir régulièrement son état à une réplique s'exécutant sur

[1] Voir <*http://embl.sourceforge.net/*>.

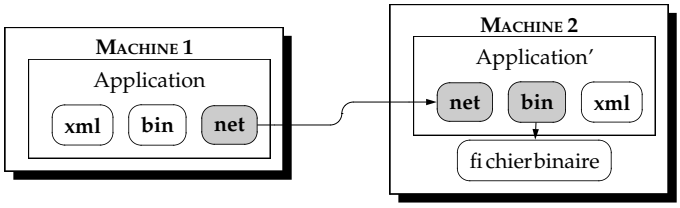


Figure 3.9. Exemple de configuration utilisant le module de stockage réseau.

une autre machine ; à son tour, la réplique peut sauvegarder son état dans un fichier en utilisant, par exemple, le module de stockage binaire.

3.3. Utilisation de *Pego*

Pego ne capture l'état que des données accessibles globalement (variables globales ou `static`, et données allouées dynamiquement). Parmi ces données, certaines peuvent représenter des *ressources externes* : c'est par exemple le cas d'un entier représentant un descripteur de fichier ou une connexion réseau, ou encore d'un identifiant de fenêtre graphique X Window. On qualifie ces ressources d'*externes* car elles sont fournies à l'application par un intervenant externe qui peut être soit une bibliothèque, soit un serveur, soit le système d'exploitation. *Pego* n'ayant aucune connaissance sur la sémantique attribuée aux données du programmes, une variable de type entier, qu'elle représente ou non un descripteur de fichier, est sauvegardée telle quelle. Toutefois, lors de la restauration du programme, ce descripteur de fichier risque d'être invalide[1]. Le programme utilisant *Pego* doit donc *explicitement* réinitialiser les ressources externes qu'il utilise.

La figure 3.10 montre la phase d'initialisation typique d'une application utilisant *Pego*. Il est clair qu'une application où les ressources externes sont clairement séparées des autres données manipulées par le programme pourra facilement bénéficier de *Pego*. Au contraire, une application où les ressources externes sont mélangées aux autres données nécessitera d'être étudiée afin d'identifier les différents types de données. Dans le cas d'applications respectant les patrons de conception *Modèle-Vue-Contrôleur* (déjà vus en section 3.2.4), les ressources externes sont généralement représentées par la *vue* alors que le *modèle* contient les données sur lesquelles travaille l'application (e.g. dans le cas d'une interface graphique, la vue est constituée des éléments graphiques représentant les données du modèle). En d'autres termes, la vue est transitoire alors que le modèle est persistant.

À titre d'exemple, deux applications graphiques libres ont été rendues persistantes et « *migrables* » à l'aide de *Pego*. Il s'agit du jeu *xGalaga* (figure 3.11[2]) entièrement écrit en langage C ANSI, et du jeu *xKobo* écrit en C++, tous deux étant écrits pour le système de fenêtrage client-serveur X Window[3]. Pour les rendre persistants, il a fallu :

- modifier les allocations dynamiques pour qu'elles utilisent les macros d'allocation fournies par *Ego* (voir section 3.2.2.1) ;
- modifier la phase d'initialisation de l'application à la manière de l'exemple de la figure 3.10 pour que seules les ressources externes soient initialisées lors de la restauration ;

[1] Sauf pour les descripteurs 0, 1 et 2 qui représentent toujours l'entrée standard, la sortie standard, et la sortie d'erreur.

[2] Sur la capture d'écrans les différentes répliques de *xGalaga* fonctionnant chacune sur une architecture différente (x86, PowerPC et SPARC), mais toujours avec le système d'exploitation GNU/Linux. Le mot « pause » qui apparaît sur cette figure n'est pas au même niveau sur chaque réplique : cela est dû au fait que le mot « pause » bouge constamment de haut en bas et que donc, après restauration de l'état, les répliques se désynchronisent inévitablement.

[3] Les versions modifiées de *xGalaga* et *xKobo* sont disponibles sur la page internet de *Ego* et *Pego* : *<http://www.laas.fr/~lcourtes/software/ego/>*.

```
int
main (int argc, char *argv[])
{
  pego_t self ;
  int restoring ;

  pego_set_input_backend (&pego_binary_backend) ;

  pego_init (&self, argv[0], "state.ego") ;

  …

  if (restoring)
    pego_load_state ((ego_t *)&self, "state.ego") ;
  else
    init_data () ;

  init_external_resources () ;

  …

  return 0 ;
}
```

Choix du module de stockage d'entrée

Initialisation de `self` qui représente l'application

Chargement d'état à partir de `state.ego`

Initialisation des ressources externes (e.g. descripteurs de fichier, fenêtres X Window, etc.)

Figure 3.10. Exemple d'utilisation de Pego dans une application où les ressources « externes » sont clairement séparées des ressources « locales ».

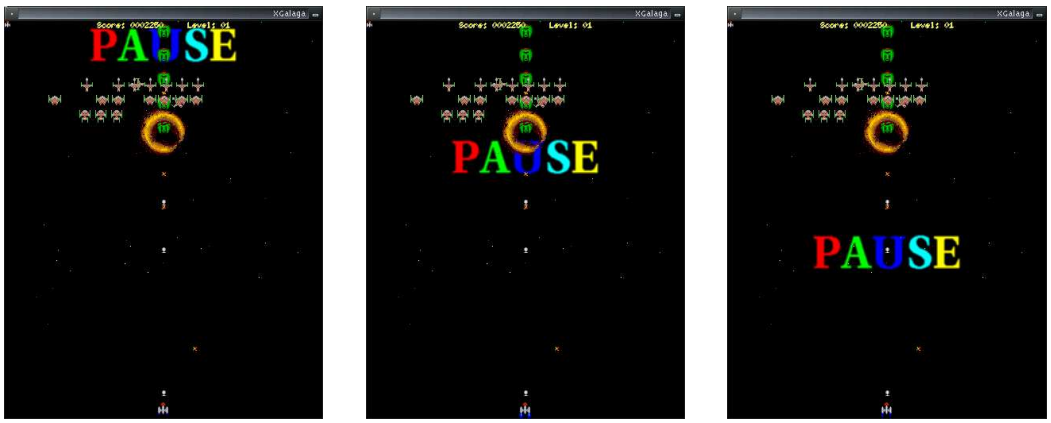


Figure 3.11. Capture d'écran de trois répliques du jeu *xGalaga* instrumenté avec *Pego*.

- et bien sûr, rajouter les appels aux fonctions de capture et de restauration d'état de *Pego*.

Dans le cas de *xGalaga*, cela représente la modification ou l'ajout d'environ 270 lignes sur 15 254, soit 1,7 % du code. Dans le cas de *xKobo*, environ 100 lignes ont été modifiées ou ajoutées sur les 5 129 lignes que compte la version originale, soit 1,9 % du code. *xGalaga* ne sépare pas du tout le modèle de sa représentation à l'écran, ce qui a nécessité une étude permettant de bien différencier les ressources externes qui doivent être réinitialisées du modèle qui devient persistant. Par contre, *xKobo* sépare assez clairement les deux, ce qui a largement facilité la mise en œuvre de la version persistante. Au regard de ces deux expériences, il semble que l'utilisation de *Pego* dans une application « de la vie réelle » soit réalisable avec peu de difficultés, même en ayant une connaissance quasi-nulle de l'application que l'on souhaite rendre persistante.

3.4. Limitations

Prise en charge des ressources externes. Comme le montre la section 3.3, une des limitations intrinsèques de *Pego* est le fait que les ressources externes ne sont pas prises en charge. Une façon simple permettant la capture de l'état d'une ressources externe serait de maintenir dans l'application des structures de données représentant l'état d'une ressource externe. Ces données faisant partie de la capture d'état, elles peuvent être utilisées, lors du recouvrement, pour « re-créer » l'état d'une ressource externe. Dans le cas des fichiers, il serait par exemple aisé de mettre au point une interface d'entrée/sortie persistante (figure 3.12). L'information collectée par cette interface pourrait être réutilisée lors de la restauration de l'état du programme, pour ré-ouvrir les fichiers sous-jacents et les positionner au bon endroit. Toutefois, cela implique que l'application soit modifiée pour utiliser, par exemple, `pego_read ()` et `pego_write ()` au lieu de `read ()` et `write ()`. Une autre technique, qui nécessiterait une prise en charge par le système opératoire, consisterait à importer l'état de la ressource dans l'application juste avant de réaliser la capture d'état. Cette proposition sera discutée au chapitre 4.

Limitations liées à l'utilisation des *stabs*. L'utilisation des informations de débogage produites par le compilateur (section 2.7), comparée à des techniques de compilation ouverte (section 2.6), implique que le travail de réification de la structure des types de données et des variables du programme, ainsi que le travail de l'analyse des types de données, ont lieu pendant l'exécution du programme. La création de « *méta-objets* » par *Ego* a lieu a l'initialisation du programme, et l'analyse des données existantes et des types a lieu lors de la réalisation de la capture d'état. En outre, la représentation du programme créée par *Ego*, ainsi que la « vue » (XML ou autre, voir section 3.2.5) créée par *Pego* occupent une partie de la mémoire du programme pendant toute sa durée d'exécution. Toutefois, la construction d'une représentation des données en mémoire paraît inévitable, y compris si une technique de compilation ouverte est utilisée [28].

Par ailleurs, les *stabs* ne contiennent pas toujours toute l'information nécessaire. Par exemple, dans le cas d'un tableau sans dimension tel que

```
int a[] = { 1, 2, 3, 4 } ;
```

les informations de débogage fournies par GCC 2.95.4 ne contiennent pas d'information sur la taille réelle du tableau. Dans le cas où aucune information sur la taille n'est disponible, *Ego* sait seulement que le tableau contient au moins un élément. Par conséquent, dans le cas où une variable pointerait, par exemple, sur le troisième élément du tableau, il est impossible de savoir si ce pointeur pointe sur un élément du tableau où si il pointe « en dehors » du tableau. Pour résoudre ce problème, il faudrait déclarer le tableau comme suit :

```
int a[4] = { 1, 2, 3, 4 } ;
```

Fort heureusement, les versions récentes de GCC (3.x) incluent automatiquement la taille dans les *stabs* décrivant les tableaux sans dimension.

Limitations intrinsèques du langage. Également, les données de type union ne peuvent pas être sauvegardées de manière convenable par *Pego* car leur signification est souvent très dépendante d'une plate-forme particulière. Considérons le type suivant :

```
union integer
{
  int16_t  int16 ;
  int32_t  int32 ;
}
```

Pour une valeur assignée au champ `int32`, la valeur lue en accédant au champ `int16` diffère selon l'architecture et en particulier selon l'ordre des octets (« *endianness* ») sur cette architecture. L'utilisa-

```
typedef struct
{
  /* Le descripteur de fichier (éventuellement invalide) correspondant */
  int    fd;

  /* Nom du fichier et position */
  const char *name;
  off_t  offset;
} pego_file_t;

extern int pego_read (pego_file_t *file, char *buf, size_t size);

 …
```

Figure 3.12. Exemple d'interface permettant de rendre persistante de l'information relative à un fichier.

tion de types dont la taille diffère selon l'architecture tels que **`int`** ou **`long`** complique davantage la situation. Dans *Tui* [50], l'auteur a modifié le compilateur de manière à connaître le champ auquel le programme a accédé le plus récemment. De cette façon, il ne sauvegarde et restaure que cette valeur, en espérant que ce soit effectivement la plus significative pour le programme.

3.5. Performances

Contexte. Des mesures ont été réalisées de manière à évaluer les performances de l'opération de capture d'état et de celle de recouvrement (rechargement d'un état). Les performances de la capture d'état elle-même sont un point critique : la réalisation d'une capture d'état doit être la plus rapide possible de manière à ce que l'application qui utilise *Pego* puisse sauvegarder son état aussi souvent que possible sans pour autant que cela constitue une gêne pour son utilisateur. L'algorithme présenté en section 3.2.4 présente un certain nombre de particularités susceptibles de participer à l'obtention de bonnes performances, telles que la création d'une représentation des données statiques dès l'initialisation et non pas à la demande, la conservation de la représentation d'une donnée dynamique jusqu'à sa destruction, ou encore le fait de ne pas sauvegarder les données constantes. Le module de stockage (section 3.2.5) joue également un rôle très important dans les performances de la capture d'état et c'est pourquoi les mesures présentées en figures 3.13 et 3.14 ont été effectuées avec le module XML mais aussi le module binaire.

La figure 3.13 présente le temps requis pour effectuer une capture d'état et une restauration d'état pour un programme de multiplication de matrices (noté « *MM* ») d'éléments de type **`double`** avec des tailles de problème variables. Par exemple, « MM 16×1000 » signifie que les matrices sauvegardées étaient une matrice de 16×1000 éléments, une matrice de 1000×16 éléments, et la matrice résultat de 1000×1000 éléments, soit un total de $1{,}032 \times 10^6$ éléments. La multiplication de matrices est un exemple utilisé par plusieurs projets [50, 12, 25] et permet donc de donner une idée de la comparaison des performances de *Pego* par rapport à celles d'autres outils.

Les mesures ont été effectuées sur une machine de type Pentium3 à 1 GHz, dotée de 512 Mo de mémoire vive (RAM), et sur laquelle tourne le système d'exploitation GNU/Linux (avec Linux 2.4.18). Les mesures montrent le temps nécessaire à la capture de ces données (durée d'exécution de la fonction `pego_save_state ()`) d'une part, et d'autre part le temps nécessaire à la restauration de cet état (durée d'exécution de la fonction `pego_load_state ()` et durée de la phase de lecture des *stabs* par *Ego*). Les mesures de temps sont la somme du « temps système » et du « temps utilisateur » du programme renvoyés par `getrusage ()`. Plus précisément, il s'agit d'une moyenne calculée sur 1 000 appels à chacune de ces fonctions. La version de *Pego* utilisée pour ces mesures est la 0.3 et le module de stockage binaire utilise un tampon en écriture de 16 Ko. Par ailleurs, la figure 3.15 donne un aperçu des résultats obtenus avec le module de stockage binaire sur une architecture à base de

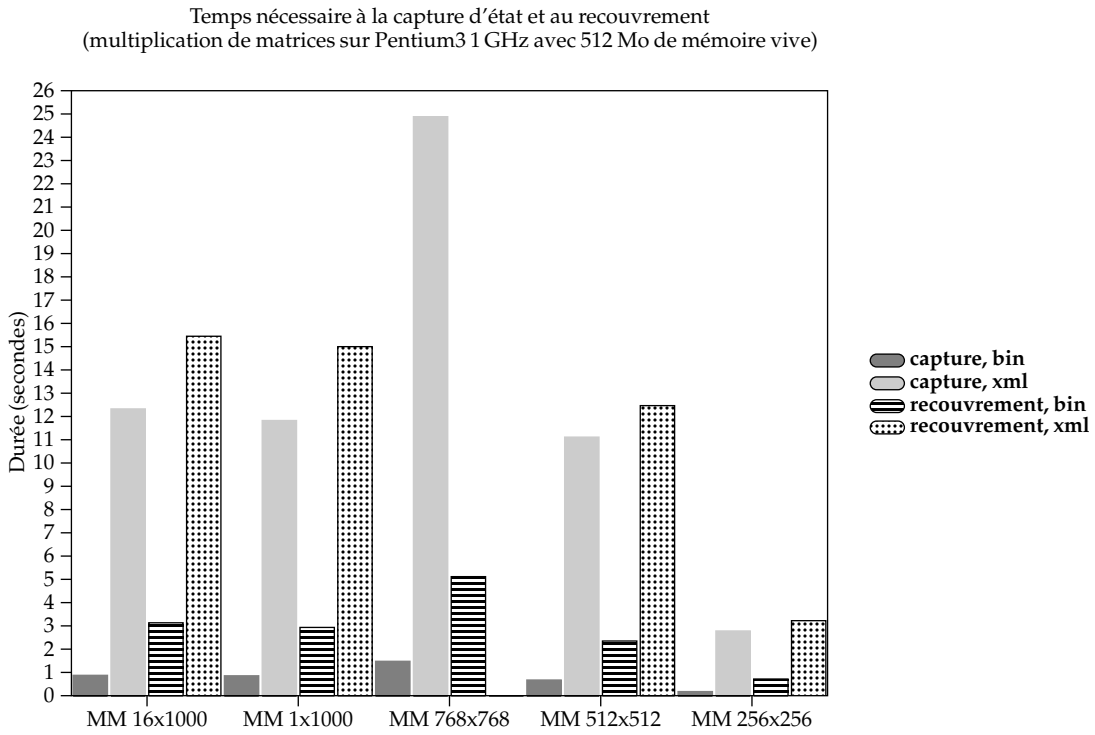


Figure 3.13. Durée de la réalisation d'une capture d'état pour la multiplication de matrices.

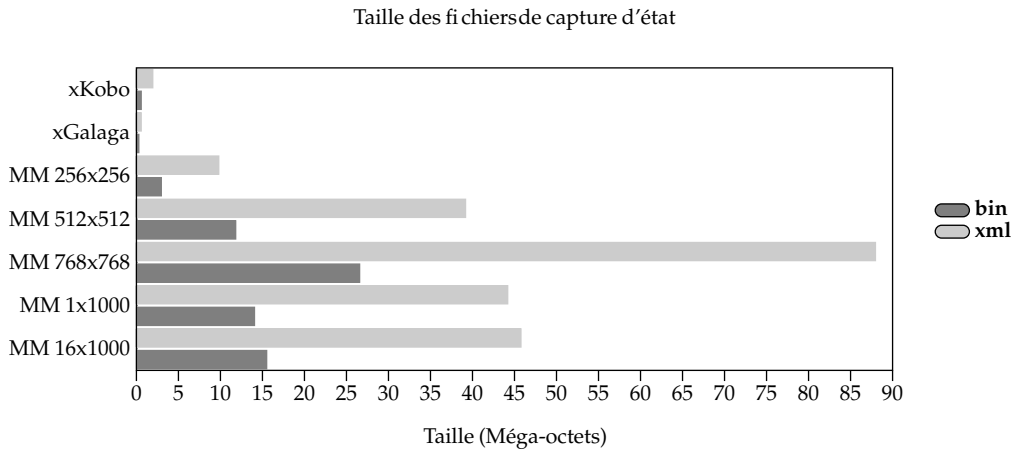


Figure 3.14. Taille du fichier de capture d'état pour la multiplication de matrices.

PowerPC G3 à 267 MHz (92 Mo de mémoire vive), ainsi que sur une SPARCstation 4 équipée d'un processeur à 110 MHz (Fujitsu MB86904) et de 64 Mo de mémoire vive.

Analyse des résultats. Ces mesures montrent que le module de stockage XML n'est pas adapté à des programmes contenant une quantité de données importante : la taille des fichiers qu'il produit est approximativement trois fois celle des fichiers produits par le module binaire (figure 3.14), et le temps de sauvegarde de l'état en XML est environ 14 fois le temps requis par le module binaire (figure 3.13). Notons d'ailleurs que sur la figure 3.13 ne figure pas le temps de restauration dans le cas de matrices de 768×768 avec le module XML : cela est dû au fait que l'application a épuisé les ressources en mémoire avant d'arriver à restaurer son état. Sur des applications de taille moyenne telles que les

jeux *xGalaga* et *xKobo*, l'utilisation du format XML demeure cependant « raisonnable ». Pour le module de stockage binaire, l'utilisation d'un tampon a permis d'améliorer sensiblement les performances par rapport à un prototype précédent qui écrivait directement les données ; de même, l'utilisation de `mmap ()` pour lire les données lors du recouvrement permet d'obtenir de bons résultats par rapport à technique faisant des appels à `read ()`. Il faut noter également que le format binaire de *Pego* sauve les entiers au format *little endian*[1] qui est précisément le format utilisé sur l'architecture x86 sur laquelle ont été réalisées ces mesures. Les performances auraient donc pu être un peu moins bonnes si la machine cible stockait les octets dans un ordre différent du format binaire car une conversion aurait été nécessaire. Concernant le temps nécessaire à la lecture des *stabs* (non représenté mais compris dans les temps de restauration présenté sur la figure 3.13), il s'avère être assez faible. Dans le cas de matrices 16×1000 et pour le stockage binaire, la lecture des *stabs* prend 0,93 seconde sur un total de 3,13 secondes, soit 30% du temps de restauration ; pour le stockage au format XML, elle représente 1,45 seconde sur 15,45 secondes, soit 9% du temps de restauration. Cette différence s'explique par le fait qu'il le temps de lecture du fichier binaire et de restauration de l'état à partir des données qu'il contient est beaucoup plus court que le temps de lecture et de conversion des données à partir de la version XML.

La comparaison de ces résultats avec les performances obtenues avec d'autres outils ne peut souvent pas être effectuée directement compte-tenu des différences entre les outils : certains effectuent des captures d'état non portables [43], d'autres sont spécifiques au langage C++ [28, 48], d'autres effectuent des captures d'état portables mais auxquelles il incluent une capture de la pile d'appel [50, 44, 12], d'autres encore prennent en compte toutes les données nécessaires à une capture d'état d'application à plusieurs brins d'exécution [25]. À cela s'ajoute le fait que tous n'ont pas publié des mesures de performance directement comparables, que les outils présentés ne sont pas tous disponibles librement, et que lorsqu'ils le sont ils ne sont pas nécessairement utilisables.

Malgré tout, parmi les résultats disponibles, nous pensons pouvoir comparer ceux de *Pego* à ceux présentés dans [25]. L'outil présenté en [25] effectue une sauvegarde de la pile de chacun des brins d'exécution mais aussi des données de synchronisation. Toutefois, dans le cas d'une multiplication de matrice, la pile ne contient *a priori* qu'un ou deux niveaux (la multiplication peut être faite dans la fonction `main ()`) et que peu de variables locales voire aucune (les trois compteurs de boucle nécessaires peuvent être déclarés en **`static`** comme c'est le cas dans notre implémentation, ou même en variable locale). Également, parmi les résultats présentés, la multiplication de matrices de 1×1000 éléments est réalisée par un seul brin d'exécution, ce qui rend les mesures comparables à celles de *Pego* pour la même taille de problème. Par chance, le type de machine utilisé dans [25] est le même que celui utilisé pour ces mesures (même processeur, même quantité de mémoire vive, même OS). Pour la version 1×1000, alors que les mesures affichées par Karablieh et *al.* dans [25] sont de l'ordre de 4,25 secondes pour la capture d'état et de 5 secondes pour la restauration, *Pego* réalise la capture en 0,85606 seconde et la restauration en 2,93368 secondes en moyenne (figure 3.13).

Dans [12], Ferrari donne des mesures effectuées sur une machine de type Pentium à 150 MHz et 32 Mo de RAM, ce qui n'est donc pas comparable avec nos mesures. Toutefois, les résultats (incluant la sauvegarde et restauration de la pile) sont de l'ordre de 0,36 secondes pour une multiplication de matrices de 512×512 éléments, créant un fichier de capture de l'ordre de 6,3 Mo. Compte-tenu de la machine qui a été utilisée, ces performances semblent plutôt bonnes. Malheureusement, n'ayant pas de telle machine de test et le code source de ce projet n'étant pas disponible, il a été impossible d'effectuer une comparaison avec nos résultats.

[1] En *little endian*, un entier 32 bits valant 0x04030201 est stocké avec l'octet de poids fort en premier, le deuxième octet de poids fort en deuxième, etc., ce qui donne la valeur 0x01020304 si l'on regarde le contenu de la mémoire en commençant à l'adresse la plus basse.

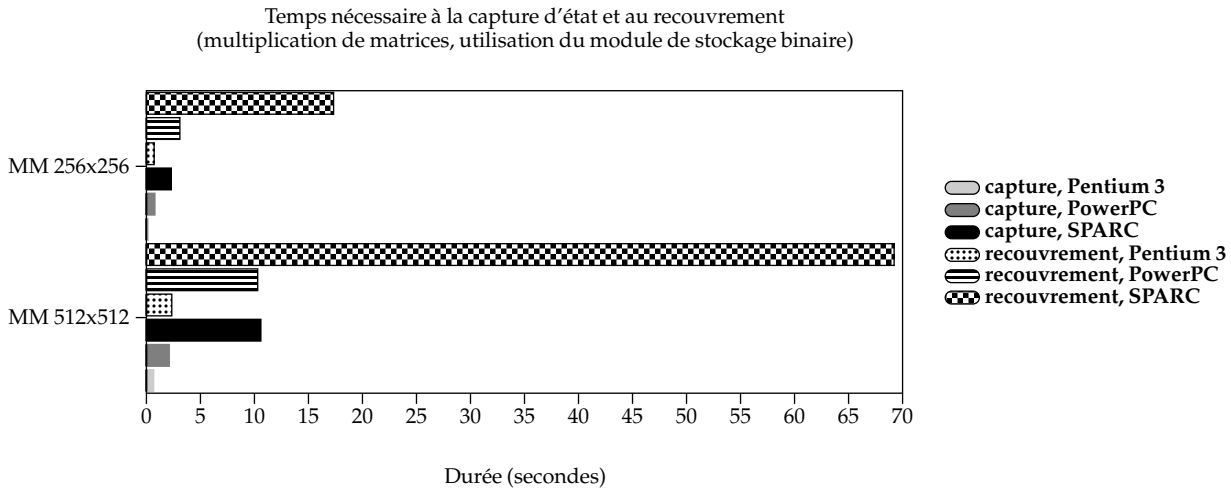


Figure 3.15. Temps de capture d'état et restauration sur différentes architectures.

3.6. Sauvegarde de la pile

Cette section décrit les travaux qui ont été faits pour réaliser une sauvegarde portable de la pile d'appel d'une application C ou C++. Cette fonctionnalité intéressante s'est avérée difficile à mettre en œuvre compte tenu des problèmes techniques rencontrés qui sont essentiellement dûs au fait que l'on a affaire à des mécanismes de très bas niveau.

3.6.1. Problématique

Comme l'explique la section 3.3, deux applications ont été modifiées pour tirer parti des fonctionnalités de *Pego* et devenir persistantes aux yeux de l'utilisateur, bien que leur pile d'appel ne soit pas sauvegardée. Il se trouve que ces deux applications sont constituées essentiellement d'une boucle qui agit sur les données en fonction des évènements qui peuvent se produire. Une variable détermine la situation de l'application (« *écran d'accueil* », « *en cours de jeu* », « *en pause* », etc.), et d'autres variables contiennent les données plus précises sur l'état de l'application (score, niveau de jeu, etc.). Lorsque ces données sont restaurées, elles suffisent à l'application pour poursuivre son exécution. La plupart des applications interactives peuvent, de la même manière, capturer leur état après traitement d'une requête et poursuivre leur exécution après restauration sans nécessiter une sauvegarde de leur pile d'appel.

Toutefois, dans le cadre d'une application telle qu'un serveur à plusieurs brins d'exécution (« *multi-threaded* »), cette technique ne suffit pas. Un tel serveur peut avoir à traiter *en permanence* des requêtes, et on ne peut donc pas se permettre d'attendre un moment où plus aucune requête ne serait en cours de traitement pour effectuer la capture d'état. Par conséquent, la capture d'état d'un tel serveur doit intégrer sont fbt d'exécution de sorte à ce que le traitement des requêtes puisse reprendre exactement là où il en était après restauration [53]. Il est également nécessaire d'obtenir des informations sur les brins d'exécution de l'application et sur l'état relatif à la synchronisation (état des *mutexes*, etc.). Ceci peut être effectué en modifiant le programme de sorte à ce qu'il n'utilise pas directement les fonctions de l'interface de programmation de *threads* POSIX [25], ou bien en interceptant les appels à ces fonctions (cette possibilité sera discutée en section 4.2.2). Les sections suivantes se concentrent uniquement sur la capture portable de la pile d'appel elle-même.

3.6.2. Fonctionnement de la pile et difficultés

La *pile* est une zone de la mémoire servant à stocker une trace des appels de fonction. Pour un processeur donné, l'instruction d'appel de fonction empile automatiquement le pointeur d'instruction courant sur la pile ; lorsque l'instruction de retour de fonction est ensuite exécutée, elle dépile cette valeur et l'installe comme nouveau pointeur d'instruction de manière à reprendre le flux d'exécution là où il en était avant l'appel de fonction. La pile est généralement aussi utilisée à d'autres fins : la fonction appelante ou appelée empile les registres qu'elle utilise et le code généré par les compilateurs l'utilise généralement pour stocker le contenu des variables locales. Le standard System V définit une « *interface applicative binaire* » [51] (ou « *application binary interface* », ABI) qui spécifie un certain nombre de conventions d'utilisation de la pile, avec des suppléments spécifiques à chaque architecture (i386, PowerPC, SPARC, MIPS, etc.) fournissant davantage de détails notamment sur l'utilisation des registres. Ce standard est suivi par la plupart des plate-formes.

La figure 3.16 montre la disposition de pile préconisée par le document générique sur l'ABI System V. La pile croit généralement vers le bas et la partie haute du schéma représente donc les adresses les plus élevées. D'une manière générale, avant un appel de fonction, les arguments sont empilés sur la pile par mots[1]. Les arguments peuvent aussi être passés par registres lorsque leur taille et la disponibilité de ces derniers le permet[2]. L'appel de fonction lui-même entraîne l'empilement de l'adresse de retour et d'un pointeur sur la précédente « trame de pile » ou « *stack frame* », et l'adresse de ce pointeur constitue la trame de pile de la fonction appelée – en d'autres termes, les pointeurs de trames forment une liste chaînée entre les données de pile des fonctions. Enfin, la fonction appelée peut utiliser une partie de la pile pour y stocker des données locales.

La pile d'appel peut donc être parcourue pour obtenir la liste des adresses de retour de fonction appellantes. De plus, les *stabs* contiennent des informations corrélant des adresses dans le code à des numéros de ligne dans le code source. De cette façon, il est possible de savoir à quelle fonction et à quelle ligne dans quel fichier source correspond une adresse de retour donnée, ce qui permet d'obtenir une représentation symbolique de la pile d'appel, à la manière de ce que permettent de faire les débogueurs. En outre, les *stabs* représentant des variables locales indiquent à quel endroit est stockée une variable de pile par rapport au pointeur de pile courant. Toutes ces informations de débogage sont donc suffisantes pour *sauvegarder* une pile d'appel de manière symbolique.

Malheureusement, beaucoup d'informations nécessaires à la *reconstruction* d'une pile d'appel à partir de ces informations symboliques sont absentes des informations de débogage. Par exemple, la taille et le contenu exact d'une trame de pile sont inconnus : on ne sait pas exactement quels registres ont été sauvegardés sur la pile, quelles données y ont été ajoutées. Le projet *Tui* a résolu ce problème en modifiant le compilateur pour qu'il ajoute dans les *stabs* des données sur la taille des trames [50]. De plus, l'utilisation des registres est complètement opaque : le compilateur peut avoir généré du code qui, dans un souci d'optimisation, utilise des registres pour stocker des valeurs temporaires ou même des variables et qu'il serait nécessaire de restaurer. Là encore, *Tui* se base sur une modification du compilateur pour obtenir les informations manquantes.

3.6.3. Mise-en-œuvre

Les informations intégrées à l'exécutable ne suffisant pas, une solution pour réaliser une capture portable de la pile est de modifier le code source original à la manière de ce que font *PORCH* [44] ou encore le compilateur du *Process Introspection Project* [12]. L'idée est la suivante :

- lors de la sauvegarde de l'état, la pile est parcourue et est *reproduite* dans une liste où les adresses de retour sont remplacées par un couple fonction-numéro de ligne (et éventuellement nom du

[1] Sur une architecture 32 bits, un mot fait 32 bits, etc.

[2] C'est souvent le cas sur une machine RISC telle que PowerPC ou SPARC.

Position	Contenu
base + *n*	mot d'argument *n*
	…
base + 2	mot d'argument 0
base + 1	adresse de retour
base (ou « *frame pointer* »)	base de l'appellant
base - 1	*x* mots de données locales: variables locales, données temporaires, etc.
pointeur de pile + *y*	registres sauvegardés

Figure 3.16. Disposition de la pile d'appel (standard System V).

fichier source s'il s'agit d'une fonction locale à un fichier); chaque élément de cette liste contient également la liste des variables locales et une copie de leur valeur ;

- cette liste est donc une *pile persistante* puisqu'elle est automatiquement sauvegardée par *Pego*, comme n'importe qu'elle autre donnée globale ou dynamique ;
- lors du recouvrement, la pile persistante est parcourue : la fonction `main ()` appelle la première fonction de la liste ; du code rajouté au début de cette fonction permet de *sauter à la ligne*, à l'intérieur de cette fonction, où le prochain appel à lieu ; l'appel en question est effectué et la fonction appelée procède de même, jusqu'à ce que la liste ait été entièrement parcourue.

Ce mécanisme a été implémenté à titre expérimental simplement à l'aide de macros. Une macro doit être rajoutée à chaque début de fonction de manière à pouvoir prendre en charge la restauration de la pile lors du recouvrement. Cette technique demande une modification non-négligeable de l'intégralité du code source, même celle-ci pourrait être automatisée à l'aide d'un script.

3.7. Conclusion

Pour permettre la capture d'état portable d'applications écrites en langage C ou C++, une des principales techniques consiste en l'utilisation d'un compilateur ouvert pour insérer dans le code source les fonctionnalités de capture et de restauration d'état [28, 44, 12]. La technique présentée dans ce chapitre utilise l'approche inverse : la phase de compilation sert simplement à intégrer les informations nécessaires au fichier exécutable, ces informations étant *par la suite* utilisées pour la capture d'état. Cette technique permet de séparer le mécanisme de capture d'état de l'application elle-même. L'expérience a montré que de bonnes performances pouvaient être atteintes malgré le fait que ces informations soient traitées à l'exécution. L'implémentation proposée est une bibliothèque qui nécessite d'être utilisée explicitement par l'application ; toutefois, la stratégie de capture d'état pourrait être implémentée *en dehors* de l'application, à condition qu'elle fournisse une interface qui permettre à un programme tierce d'obtenir son état. Comme nous le verrons au chapitre 4, un système d'exploitation à base de micro-noyau permet à toute application de se comporter comme un serveur. En l'occurrence, une application pourrait offrir une interface d'introspection et d'exportation d'état.

Par ailleurs, le format de débogage *stabs* actuellement utilisé par *Ego* manque d'informations, notamment concernant la façon dont est construite la pile d'appel. Une possibilité pour pallier ce manque serait d'étendre le format d'information de débogage pour lui permettre de décrire également la disposition de la pile d'appel comme cela a été proposé dans *Tui* [50]. Une autre solution serait de développer un préprocesseur ou d'utiliser un compilateur ouvert pour rajouter des informations relatives à la localisation des appels de fonction dans une section dédiée du format binaire et dans

un format *ad hoc*. Cependant, cette solution ne règle pas la question de l'utilisation des registres qui est faite par le compilateur. Ce dernier problème nécessite soit une modification du compilateur, soit l'ajout de restrictions sur les endroits où peuvent être effectuées les captures d'état [50], mais il est toutefois inexistant lorsque l'on considère une transformation source à source. Une approche « hybride » utilisant à la fois les informations produites par le compilateur et des mécanismes ajoutés par une transformation source à source permettrait sans doute de bénéficier des avantages de chacune des deux approches.

Chapitre 4. Mécanismes de tolérance aux fautes pour les systèmes d'exploitation

« *La perfection n'est pas lorsqu'il n'y a plus rien à ajouter, mais lorsque qu'il n'y a plus rien à enlever.* »
– Antoine de Saint Exupéry, *Le Petit Prince*.

Le précédent chapitre a montré comment un système de capture d'état simple et une stratégie de réplication associée peuvent être introduits au niveau applicatif, sous forme d'une bibliothèque. Il a aussi montré qu'étendre la portée de l'état sauvegardé aux ressources externes ne peut être effectué au niveau applicatif que difficilement et au coût de modifications importantes de l'application (section 3.4). Différents mécanismes de tolérance aux fautes nécessitent d'être inclus dans le système d'exploitation lui-même pour lever ce type de limitation.

Ce chapitre présente dans un premier temps les besoins en tolérance aux fautes au niveau du système d'exploitation, en transposant l'expression de ces besoins en termes réflexifs (section 4.1). La section 4.2.1 présente ensuite différentes architectures de système d'exploitation et montre comment chacune d'elle peut être utilisée pour répondre à ces besoins. Une implémentation transparente de mécanismes de sûreté de fonctionnement a été réalisée pendant ce stage dans le cadre du projet DSoS[1]. Des exemples tirés de cette expérience sont utilisés pour illustrer les besoins d'intégration de mécanismes de tolérance aux fautes au niveau exécutif.

4.1. Objectifs

4.1.1. Besoins pour l'intégration de mécanismes de tolérance aux fautes

L'ensemble du logiciel permettant de faire fonctionner un système informatique est souvent organisé sous forme de couches : la couche la plus haute symbolise l'application, tandis que la couche la plus basse symbolise le système d'exploitation. Cette architecture par couches – en pratique, une couche est souvent mise en œuvre par une bibliothèque de fonctions – permet de rendre disponibles à l'application des abstractions de haut niveau. En contrepartie, l'empilement de couches d'abstractions souvent *opaques* se traduit par un appauvrissement de l'information disponible aux couches les plus hautes [52]. Pour cette raison, certains mécanismes ne peuvent être mis en œuvre qu'au niveau système. En outre, intégrer ces mécanismes au niveau du système garantit qu'ils seront (presque) *transparents* pour le développeur [10].

Un certain nombre de propriétés sont souhaitables pour les mécanismes de tolérance aux fautes intégrés au système : en plus de leur *transparence* vis-à-vis du développeur d'applications, ils doivent être *réutilisables*, *indépendants* l'un de l'autre, et *composables* [10]. Pour être réutilisables et pour faciliter

[1] DSoS signifie « *Dependable System of Systems* », c'est-à-dire *sytème de systèmes sûr de fonctionnement*. Il s'agit d'un projet européen financé par la Commission Européenne dans le cadre du programme « *Information Society Technologies* » (IST-1999-11585), débuté le 1er avril 2000 et achevé le 31 mars 2003. Voir <*http://www.laas.research.ec.org/dsos/*> et [15] pour plus d'informations.

la maintenabilité de l'ensemble, l'intégration de ces composants doit respecter le principe de *séparation des préoccupations* déjà vu au chapitre 1. Il s'agit donc de définir un ensemble d' interfaces par lesquelles les mécanismes de tolérance aux fautes pourront interagir avec le système d'exploitation de manière à pouvoir l'*observer* et à pouvoir *modifier* son comportement. Là-encore, les concepts de la réflexivité aident à la conception de telles interfaces d'une manière générique et qui pourrait bénéficier également à d'autres mécanismes telles que la persistance, la migration de processus, etc. La section suivante cherche à définir les réels besoins en réflexivité en se basant sur des exemples concrets de mécanismes de tolérance aux fautes.

4.1.2. Vue réflexive des besoins

De nombreux mécanismes de tolérance aux fautes ont besoin de pouvoir *observer* le système d'exploitation et d'avoir un certain *contrôle* sur son comportement. Parmi ces mécanismes, on peut citer quelques exemples sur lesquels nous reviendrons ensuite :

- la *vérification de propriétés* de logique temporelle concernant le fonctionnement du système d'exploitation telle que proposée par Frédéric Salles [46];
- la *capture de l'état d'applications* qui ne peut être entièrement résolue au niveau applicatif comme le montre la section 3.4 ;
- la *réplication de serveurs à plusieurs brins d'exécution* telle qu'elle a été étudiée par François Taïani [53]; elle nécessite de connaître et de contrôler les décisions de synchronisation prises par le système, ainsi que les évènements de réception et d'envoi de messages perçus par le système;
- pour améliorer la confidentialité et/ou l'intégrité des données échangées entre applications, il peut être nécessaire d'*étendre ou d'adapter certaines fonctionnalités du système;* Brendan Gowing a identifié principalement quatre types d'adaptation dynamique du système qui peuvent être requises [18] :

  1. l'*adaptation* par sélection d'un algorithme ou d'une implémentation particulière d'un service parmi un choix prédéfini;
  2. l'*extension par remplacement* d'un service du système; un système d'exploitation persistant (section 2.4), par exemple, peut être implémenté en fournissant une implémentation spécifique du service de gestion de la mémoire virtuelle [49];
  3. l'*extension par modification* d'un objet ou service par extension de son interface;
  4. l'*extension par introduction* d'un nouveau service.

La détection d'erreur sur les *mutexes* proposée par Frédéric Salles [46] nous permet d'illustrer l'exemple de vérifications de propriétés. Cette détection d'erreur consiste en la vérification, pour une mutex m, du prédicat suivant :

$$P(m) - V(m) = C(m) + Q(m) \tag{4.1.1}$$

où $P(m)$ et $V(m)$ représentent respectivement le nombre de clients « entrés » et « sortis » de la mutex, $Q(m)$ représente le nombre de brins d'exécution en attente sur la mutex, et $C(m)$ vaut 1 en cas de contention (i.e. la mutex est prise) et 0 sinon. Pour avoir accès à toutes ces données tout en étant séparé du système d'exploitation lui-même, le mécanisme de vérification doit avoir la possibilité d'utiliser une *interface d'introspection* fournie par le système. La section 4.2.3 reviendra plus en détail sur une mise en œuvre possible de cette interface.

Pour répliquer une application à plusieurs brins d'exécution, cette capacité d'introspection des primitives de synchronisation est elle aussi requise, de manière à pouvoir reproduire les même décisions de synchronisation sur chaque réplique. De plus, pour qu'une décision de synchronisation puisse être imposée par le mécanisme de réplication sur une réplique, le système doit disposer d'une interface d'*intercession comportementale* [53]. S'il s'agit d'un serveur, il est en outre nécessaire de réifier les évènements de réception et d'envoi de messages, de manière à savoir quels messages nécessiteraient d'être renvoyés en cas de défaillance.

Comme le montre la section 3.4, la capture de l'état des ressources externes utilisées par une application ne peut être traitée entièrement au niveau applicatif. Parmi ces ressources externes, beaucoup sont mises à disposition par le système d'exploitation : c'est le cas des descripteurs de fichier, ou encore des *sockets* réseau. Pour que l'état de ces ressources puisse aussi être capturé et restauré, il est également nécessaire pour le système d'exploitation de disposer d'une interface d'*introspection* et d'une interface d'*intercession*. Ces interfaces devraient permettre soit de journaliser les opérations effectuées sur ces ressources de manière à pouvoir les « rejouer » lors de la restauration, soit de pouvoir obtenir l'état de ces ressources de manières à pouvoir le restaurer par la suite. Parmi ces « ressources externes » typiques manipulées par une application sur un système de type Unix, on trouve des données telles que les variables d'environnement connues de l'application lorsqu'elle s'exécute, ou encore les signaux perçus par l'application.

Enfin, certains services du système d'exploitation peuvent nécessiter d'être *adaptés* ou *spécialisés* pour correspondre aux besoins d'une application en termes de sûreté de fonctionnement. Par exemple, une application peut avoir besoin de s'assurer de l'intégrité des données qu'elle échange, ou bien de leur confidentialité. Ces besoins nécessitent de pouvoir *étendre* le service de communication de manière à ce que celui-ci puisse prendre en charge les besoins particuliers de l'application. De la même façon, rendre persistantes les applications exécutées par le système nécessite une spécialisation du service de gestion de la mémoire virtuelle et une extension des possibilités offertes par le pilote de disque dur, comme nous l'avons vu en section 2.4 (page 14).

À travers ces quelques exemples, une architecture *ouverte* ou *réflexive* de système d'exploitation parait tout-à-fait adaptée. Après une brève présentation de différentes architectures de système d'exploitation, la section suivante se propose d'étudier comment ces capacités réflexives peuvent être mises-en-œuvre sur chacune d'elle, en soulignant les avantages et défauts de chacune.

4.2. Techniques et architectures de système

Cette section présente dans un premier temps les principales familles d'architectures de systèmes d'exploitation ainsi que les motivations qui ont amené à leur conception et leur avantages et inconvénients. Par la suite, en se basant sur les besoins identifiés dans la section précédente, nous étudierons l'adéquation de chacune de ces architectures à la mise en œuvre de mécanismes en respectant le principe de séparation des préoccupations.

4.2.1. Architectures de systèmes d'exploitation

Noyaux monolithiques. Les systèmes d'exploitation à base de noyau monolithique sont actuellement la catégorie la plus répandue. C'est notamment l'architecture de la plupart des systèmes de type Unix tels que GNU/Linux. L'idée de *noyau* se base sur une fonctionnalité offerte par la plupart des processeurs modernes : ceux-ci distinguent un *mode utilisateur* où seules certaines instructions peuvent être exécutées et où seules certaines opérations peuvent être effectuées, du *mode noyau* ou *superviseur* où toutes les instructions et opérations peuvent être exécutées. Un mécanisme d'interruption logicielle ou « *kerneltrap* » est fournit en plus pour permettre le passage du mode utilisateur au mode noyau et réaliser un *appel système* (ou « *syscall* »). En pratique, un noyau monolithique est donc un programme fonctionnant en mode noyau et qui fournit l'ensemble des services dont peut avoir besoin

l'utilisateur. Une application, puisqu'elle fonctionne en mode utilisateur, accède à ces services par le biais d'appels système. Un noyau monolithique intègre une grande quantité de services : pilotes de périphérique, gestion de la mémoire virtuelle, pile réseau, systèmes de fi chiers,etc. Par conséquent, un système d'exploitation à base de noyau monolithique est souvent peu modulaire, peu adaptables et assez fermé puisque son interface avec l'utilisateur se limite à celles des services offerts. Un tel noyau est aussi peu sûr car la défaillance d'un composant du noyau entraîne inévatiblement la défaillance du noyau entier, et donc du système.

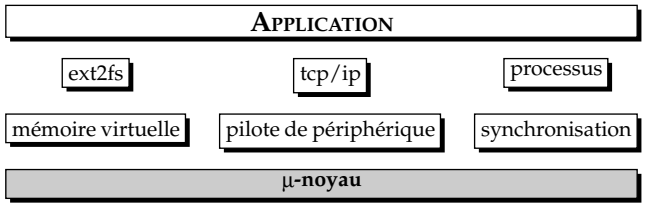


Figure 4.1. Architecture d'un système d'exploitation multi-serveur.

Systèmes multi-serveurs à base de micro-noyaux. Pour pallier à ces problèmes, une proposition consiste à réduire autant que possible les services fournis par le noyau afi n d'obtenir un noyau épuré et simple, appelé *micro-noyau* (ou µ-noyau). L'ensemble des services ayant été retirés du noyau peuvent alors être fournis par un ensemble d'applications fonctionnant en mode utilisateur appelées *serveurs*. Puisque les serveurs fonctionnent, comme dans le cas de processus Unix, dans des espaces d'adressage séparés, la défaillance d'un serveur n'a pas d'effet sur les autres. De plus, cette architecture rend l'adaptation dynamique du système possible : l'utilisateur peut ajouter ou retirer dynamiquement des serveurs du système, ou même faire cohabiter plusieurs implémentations différentes d'un même service. Parmi les systèmes d'exploitation appliquant ce principe, le système libre GNU (ou « GNU/Hurd »), qui utilise le µ-noyau Mach, est l'un des plus aboutis [4] puisqu'il offre une interface POSIX quasiment complète qui lui permet de faire fonctionner un grand nombre d'applications écrites pour systèmes de type Unix. Le système *SawMill* [16] cherche à expérimenter ces principes au-dessus du µ-noyau de seconde génération L4.

La communication entre les applications et les serveurs se fait alors par un mécanisme de *communication inter-processus* ou IPC. Le service d'IPC est fourni par le noyau, *via* un appel système,

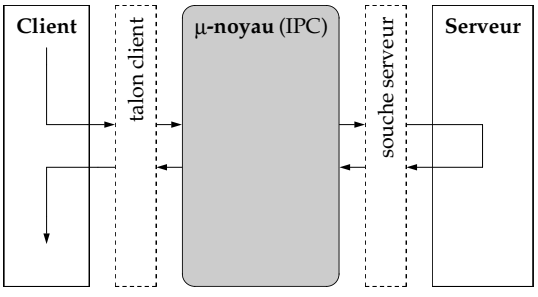


Figure 4.2. Mécanisme d'appel de procédures distantes (RPC).

qui se charge de transmettre l'information à la tâche destinataire. Au-dessus du mécanisme basique d'IPC est généralement construit un système d'*appel de procédures distantes* (ou RPC pour « *Remote Procedure Call* ») dont l'objectif est de masquer au programmeur le fait que la fonction appelée par le programme est exécutée par une autre processus. Pour ce faire, une interface décrivant les RPC

qui peuvent être appelées doit être écrite ; un « générateur d'interfaces » produit ensuite à partir de cette description un « talon client » et une « souche serveur » (on parle de « *skeleton* » pour le client et de « *stub* » pour le serveur) qui prennent en charge la transformation d'un appel de fonction en un envoi de données par IPC et l'opération inverse. Le mécanisme de RPC est illustré par la figure 4.2 où les flèches représentent le flux d'exécution d'un client invoquant une procédure distante tel qu'il est perçu par le développeur.

L'idée d'une telle architecture n'est pas neuve mais elle a souffert des faibles performances des premières implémentations de µ-noyaux, dits µ-noyaux de permière génération, telles que Mach ou Chorus [35]. En effet, dans une telle architecture, la performance du mécanisme de communication inter-processus est critique, et ces premiers µ-noyaux la rendaient trop coûteuse. Ce problème semble toutefois résolu par les µ-noyaux « de seconde génération » tels que L4 [35]. En termes de flexibilité, ces µ-noyaux de secondes générations semblent également tenir leurs promesses : le noyau lui-même ne contient qu'un jeu réduit d'abstractions de bas niveau (e.g espaces d'adressages, brins d'exécution, IPC) mais *il n'impose aucune manière de les utiliser*.

Le système d'exploitation *Apertos* [56] est conçu autour d'un µ-noyau mais en appliquant à l'ensemble du système une approche orientée-objet réflexive. Il atteint une séparation des fonctionnalités avec une granularité plus fine que celle généralement proposée dans le cas d'un système multi-serveur. Dans *Apertos*, un objet est considéré comme une entité active, c'est-à-dire comme un ensemble de segments de données ainsi qu'un fil d'exécution. Un ensemble de méta-objets constituant son *méta-espace* fournissent une représentation de l'objet : des méta-objets *Segment* représentent les segments de données de l'objet (code, données, pile, etc.), et un objet *Context* représente l'activité de l'objet, c'est-à-dire son contexte d'exécution [57]. À leur tour, les objets *Segment* contiennent dans leur méta-espace des objets représentant les pages mémoires dans lesquelles ils sont stockés, et ainsi de suite. *Apertos* permet de faire migrer dynamiquement un objet d'un méta-espace à un autre – c'est-à-dire de changer l'environnement d'exécution d'un objet. Son µ-noyau, appelé *MetaCore*, n'intègre qu'un petit nombre de primitives basiques, à la manière de L4 [35].

Architecture à composants. L'approche « orienté-composant » cherche à rendre la composition d'un système très flexible, et avec une granularité très fine. L'idée est de n'imposer aucune des archi-

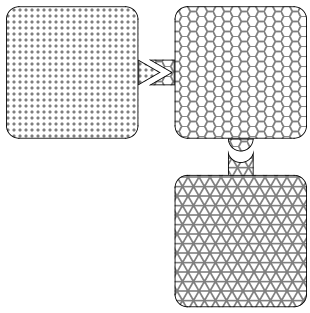


Figure 4.3. Architecture à composants.

tectures précédemment présentées. Le concepteur du système d'exploitation peut utiliser comme il l'entend un ensemble de *composants*, chacun fournissant un *service* accessible au moyen d'une *interface*. Ces composants peuvent être liés entre eux de manière flexible, et deux composants offrant la même interface peuvent être interchangés (figure 4.3). Les dépendances entre composants, en termes d'interfaces « produites » et d'interfaces « consommées » sont visibles et explicites. Dans le cadre de l'architecture à composants THINK [11], il est possible de choisir le type de lien reliant deux composants parmi un simple lien de type appel de fonction, un lien de type appel système, un lien de type RPC, ou encore un lien de type RPC à travers le réseau. Cette technique de composition et de liens flexibles entre composants étant appliquée à la totalité des services que peut offrir un système d'exploitation

offre une grande liberté à l'utilisateur dans la configuration du système. L'optique du système d'exploitation *Go!* [33] va plus loin : une architecture à composants doit permettre d'abolir la barrière créée par l'interface rigide des appels systèmes, en supprimant l'idée même de noyau. Dans cette approche, tous les composants fonctionnent en mode noyau ; la protection mémoire est assurée par une fonctionnalité particulière des processeurs (le mécanisme de *segmentation* de la mémoire virtuelle[1]), et une vérification des instructions contenues dans un composant est effectuée lors du chargement afin de vérifier qu'il ne contient pas d'instructions privilégiées.

Les sections suivantes cherchent à évaluer les possibilités offertes par chacune de ces architectures pour l'insertion de mécanismes en respectant le principe de séparation des préoccupations.

4.2.2. Réification

Interposition. Comme nous l'avons vu en section 4.1, il est souvent nécessaire de pouvoir *réifier* certains évènements du système aux mécanismes de tolérance aux fautes. Par exemple, pour connaître le nombre fois où ont été appelées les primitives de verrouillage ou de déverrouillage d'une *mutex*, il est nécessaire de disposer d'un moyen permettant d'être informé de l'appel de ces fonctions. De la même façon, pour pouvoir journaliser les accès à un fichier faits par une application dont on souhaite capturer l'état, il est nécessaire de réifier les appels aux fonctions de manipulation de fichier.

Dans le cadre d'un système de type Unix, une technique permettant d'observer ces évènements est l'utilisation d'une *bibliothèque d'interposition*. Cette idée repose sur le fait que les appels de fonction que l'on souhaite observer sont des appels de fonctions contenues dans une bibliothèque dynamique partagée. Dans un système de type Unix, l'édition de liens dynamiques, c'est-à-dire des liens entre un exécutable et les bibliothèque partagées qu'il utilise, est faite lors du lancement du dit exécutable. Le fichier exécutable contient une table indiquant le nom des bibliothèques dynamiques qu'il souhaite utiliser, ainsi que le nom des fonctions qu'il souhaite appeler ; l'éditeur de liens dynamiques `ld.so` utilise ces informations pour faire le lien entre le nom d'une fonction et son adresse. Or, il est possible de demander à l'éditeur de liens de charger, à l'exécution d'un programme, une bibliothèque partagée *avant* les autres[2]. Si cette bibliothèque fournit des fonctions déjà présentes dans une autre bibliothèque dynamique, elles les remplacent. De cette façon, il est possible d'*interposer* de manière transparente à l'application une bibliothèque entre l'application et la « vraie » bibliothèque.

Les applications de cette technique sont nombreuses. Dans le cadre du projet DSoS [15], une bibliothèque d'interposition interceptant les appels aux primitives de synchronisation a été écrite, ce qui permettait d'obtenir les valeurs de $P(m)$ et $V(m)$ du prédicat vu en section 4.1. L'obtention des valeurs $C(m)$ et $Q(m)$ requises pour la vérification du prédicat a nécessité la mise en place de mécanismes d'introspection qui seront discutés dans la section suivante. Cette bibliothèque intègre en outre un mécanisme de vérification de l'intégrité des données échangées par le biais de *sockets*. Pour ce faire, la bibliothèque *intercepte* les appels aux fonctions de gestion des *sockets* (les appels `socket ()` et `accept ()` qui permettent de créer une *socket*, les appels à `recv ()`, `send ()`, `read ()`, `write ()`, etc.). Lors de l'émission de données, elle calcule un condensé des données à envoyer (une CRC 32 bits) et rajoute au message un entête comprenant la CRC ainsi que le nombre d'octets sur lesquels elle porte. À la réception de données, la bibliothèque vérifie cet entête, calcule une CRC des données reçues, et s'assure que la CRC calculée est égale à la CRC reçue. En cas de corruption, elle renvoie à l'application le code d'erreur standard `EAGAIN` qui signifie que l'application doit retenter l'opération.

---

[1] Malheureusement pour *Go!*, cette possibilité est appelée à disparaître des processeurs (notamment IA64). Le seul mécanisme de protection de la mémoire qui sera alors fourni sera la pagination, déjà largement utilisée, mais dont la granularité est plus faible (la taille d'une page mémoire est de 4 Ko sur IA32) et donc inadaptée à la protection inter-composants.

[2] La variable d'environnement `LD_PRELOAD` indique le nom d'une bibliothèque dynamique que l'éditeur de liens dynamiques chargera *avant* les autres à la prochaine exécution d'un programme.

Serveur proxy. Dans le cadre d'un système multi-serveurs, il est possible d'*empiler* des serveurs fournissant une même interface de sorte à *étendre* les fonctionnalités offertes à l'utilisateur. On appelle *serveur proxy* un serveur qui s'interpose entre l'application et le serveur « original ». L'ajout de fonctionnalités pour la vérification de l'intégrité de données décrit ci-dessus peut-être mis en œuvre par un serveur proxy : il suffit que ce serveur implémente la même interface réseau et effectue les modifications et vérifications aux bons endroits. La journalisation de certaines interactions d'une application avec le système d'exploitation peut aussi être prise en charge par un ou plusieurs serveurs proxy. De plus, un serveur pouvant avoir plusieurs interfaces, il est possible de lui rajouter une interface qui, par exemple, donnerait accès aux données journalisées. Le système d'exploitation libre GNU/Hurd [4] permet de choisir, par simple modification d'une variable d'environnement, le serveur à utiliser pour un service donné.

L'utilisation d'un serveur proxy s'avère plus élégante, mais aussi plus flexible et plus puissante que la technique d'utilisation d'une bibliothèque d'interposition. Une des principales limitations est que certaines fonctionnalités de bas niveau ne sont pas directement accessibles aux applications par le biais d'une bibliothèque partagée. C'est le cas de la gestion de la mémoire virtuelle, de l'allocation de la mémoire physique. L'utilisation d'une bibliothèque d'interposition ne permet donc pas de spécialiser ces fonctionnalités, ni d'observer les interactions de l'application avec elles.

Cas d'un système à composants. Dans un système d'exploitation à base de composants tel que THINK [11], il est possible de modifier dynamiquement le lien entre deux composants. Un composant donné peut être remplacé dynamiquement par un composant fournissant la même interface. Il est également possible d'interposer un composant entre deux autres de manière à pouvoir observer les interactions entre ces deux composants. Un composant est comparable à un objet dans un langage réflexif tel que CLOS ou GOOPS (voir chapitre 1) dans le sens où le mécanisme d'invocation de méthode est ouvert et peut-être observé ou modifié de manière transparente au composant.

4.2.3. Introspection

Un certain nombre de mécanismes de tolérance aux fautes ont besoin de pouvoir accéder à certaines informations internes au système. C'est notamment le cas d'un mécanisme de vérifications de propriétés tel que celui évoqué précédemment. La capture d'état d'applications nécessite aussi l'accès à des structures de données représentant l'état de « ressources externes » de l'application, comme cela a été vu au chapitre 3. Dans cette section, nous nous intéressons aux mécanismes d'introspection qui sont fournis ou peuvent être mis en place dans chacune des architectures précédemment présentées.

Dans le cadre du projet DSoS [15], le mécanisme de vérification de propriétés sur les mutexes a été mis en œuvre au moyen d'une bibliothèque d'interposition fonctionnant sur GNU/Linux. Toutefois, comme nous l'avons vu en section 4.2.2, certaines informations requises dans le calcul du prédicat 4.1.1 ne peuvent être obtenues par simple interposition. Par exemple, la file des processus en attente sur une mutex est gérée à l'intérieur du noyau. Pour connaître la taille de la file d'attente d'une mutex, il a donc fallu instrumenter le noyau Linux pour permettre au mécanisme de vérification du prédicat, *via* un appel système, d'obtenir cette information. En pratique, cela a nécessité une modification du noyau Linux et un redémarrage du système. Un noyau monolithique ne se prête donc pas facilement à ce type d'extension. La possibilité de pouvoir exporter et importer l'état d'une ressource fournie par le système d'exploitation à une application serait souhaitable pour le mécanisme de capture d'état (voir section 3.4, page 30). Toutefois, une extension l'interface POSIX pour permettre, par exemple, l'exportation de l'état associé à un descripteur de fichier serait très intrusive.

En revanche, dans le cadre d'un système d'exploitation constitué de serveurs distincts, il est possible de concevoir une interface d'introspection, en plus de l'interface de « base », pour chaque serveur. Toutefois, rien n'oblige le concepteur d'un tel système d'exploitation à appliquer ce principe à l'ensemble du système. Il est cependant possible que certaines interfaces soient automatiquement héritées par chaque serveur ou application du système. À titre d'exemple, chaque processus

fonctionnant sur le système d'exploitation GNU/Hurd se comporte comme un serveur pouvant être interrogé par le biais d'une interface d'introspection basique. Cette interface [23] permet d'interroger tout processus pour, entre autres, connaître et modifier l'ensemble de ses variables d'environnement, connaître et modifier sa table de descripteurs de fichiers, ou encore obtenir une description de la raison pour laquelle un de ses brins d'exécution est en attente. Le système de fichier `/proc` disponible sur la plupart des systèmes de type Unix permet également d'observer ce type d'information, mais généralement sans pouvoir les modifier; il peut en revanche être utilisé pour suivre l'exécution d'un programme [1].

Des techniques de compilation ouverte telles que celles vues en section 2.6 pourraient être appliquées à la compilation d'interfaces de serveurs pour automatiquement généré, par exemple, une interface de réification. IDL^4 [21] est un compilateur d'interfaces en langage IDL tel que défini par le standard CORBA pour L4 qui a la particularité d'être ouvert [20], et donc de permettre à l'utilisateur d'intervenir sur le processus de compilation, à la manière d'OpenC++ [5].

Les abstractions de bas niveau fournies par le µ-noyau doivent cependant être traitées différemments. Dans [46], Frédéric Salles propose une architecture de *micro-noyau réflexif* mise en œuvre au dessus du µ-noyau de première génération Chorus/ClassiX. Une méta-interface du noyau permet de réifier à un module du noyau les invocations de méthodes du noyau, et une série de méta-interfaces pour les abstractions fournies par le noyau a été définie. La méta-interface des objets *mutex* permet, par exemple, d'obtenir les informations nécessaires à la vérification du prédicat 4.1.1.

La capture d'état des programmes exécutés par un système d'exploitation persistant est l'une des applications qui nécessite des mécanismes d'introspection au niveau du système d'exploitation. Comme nous l'avons vu en section 2.4 (page 14), le µ-noyau de première génération Mach se prête mal à l'intégration de mécanismes de capture d'état. Une des principales raisons à cela est que Mach intègre un nombre trop important de mécanismes et d'abstractions. Par exemple, les identifiants de *ports* (l'abstraction représentant un canal de communication entre deux applications) sont alloués automatiquement par Mach ; de plus, les primitives de communication offertes par Mach sont asynchrones et Mach gère donc lui-même une file de messages pour chaque port. Cet exemple est une des limites qui a dû être contournée pour mettre en place un mécanisme de capture d'état au-dessus de Mach [17]. Au contraire, le µ-noyau L4 n'offre qu'un mécanisme simple de communication interprocessus synchrone, sans abstraction représentant un canal de communication [49].

Le µ-noyau *Fluke* [13, 55] permet d'exporter et d'importer l'état des *objets noyaux* qu'il met à la disposition des applications (les « *fobbs* »). Contrairement au micro-noyau réflexif basé sur Chorus [46], la représentation des objets du noyau est accessible aux applications fonctionnant en mode utilisateur. Ce mécanisme rend possible la mise en œuvre d'un service au niveau utilisateur capable de capturer l'état d'une application en incluant l'état des ressources µ-noyau qu'elle utilise. *Apertos* [57], pour sa part, permet à un objet d'accéder à l'objet *Context* représentant son activité et qui est le seul méta-objet encapsulé par le noyau. Cette possibilité s'apparente aux *blocs de contrôle d'exécution* (« *thread control blocks* » ou TCB), structures de données représentant le contexte d'exécution d'un brin, que L4 permet de stocker dans l'espace d'adressage de l'application [49].

L'approche de L4 est donc d'encapsuler aussi peu d'état que possible dans le µ-noyau, tandis que celle de *Fluke* est de permettre l'exportation de l'état des objets gérés par le µ-noyau. Ces deux approches permettent de régler le problème d'observabilité du noyau. Comme nous le verrons dans la section suivante, ces approches peuvent aussi être appliquées au niveau des services et processus du système.

4.3. Conclusion et perspectives

Pour permettre le respect du principe de séparation des préoccupations, tout en permettant la mise en œuvre de toute extension ou adaptation possible, un système logiciel doit être construit

de manière *ouverte* (il doit être « *open-ended* »). De cette façon, les mécanismes internes du système doivent pouvoir être adaptés. Aucun choix de conception ne doit donc être imposé et aucune possibilité ne doit être écartée par le système. Cette propriété est un moyen pour parvenir à la mise en œuvre, de manière transparente à l'utilisateur du système, de mécanismes *orthogonaux* aux préoccupations de l'utilisateur – persistance, réplication, vérification de propriétés, distribution, etc. Les notions de *réflexivité* et de *protocole à méta-objets* vues au chapitre 1 modélisent tout-à-fait ce type d'ouverture.

Dans le cas d'un système d'exploitation, l'objectif est de pouvoir modifier l'environnement d'exécution, éventuellement au cours de son exécution. Les travaux tels que *Apertos* [56] ou *Tigger* [58] proposent une application de ces principes à des fins d'adaptabilité. Dans le même temps, l'approche à composants, tout comme l'approche multi-serveur, ont montré qu'elles permettaient de construire des systèmes plus modulaires où les interactions entre applications et serveurs sont clairement définies par des interfaces. L'approche multi-serveurs apporte également la possibilité de reconfigurer dynamiquement le système.

Toutefois, la séparation des fonctionnalités du système sous forme de serveurs n'apporte en elle-même que certaines des propriétés réflexives qui viennent d'être présentées. En particulier, des *capacités d'introspection* d'une part, et des fonctionnalités aidant à la *reconfiguration dynamique* des services utilisés par une application donnée d'autre part, ne font pas partie intégrante des fonctionnalités d'un tel système. Des interfaces d'introspection peuvent cependant être définies, comme le montrent celles proposées par le GNU Hurd et présentées en section 4.2.3. Au lieu de définir des interfaces d'introspection *ad hoc*, un langage de description d'état tel que PSDL [7] (voir section 2.2, page 12) peut être utilisé pour automatiquement générer une interface d'introspection permettant d'importer et d'exporter l'état des objets fournis par des serveurs. Cette approche diffère de la description d'interfaces pure où seules les opérations qu'un client peut effectuer sur un objet lui sont visibles. La description d'état à la manière de PSDL [7] a d'ailleurs été intégrée au langage de description d'interfaces à partir de la version 3 standard CORBA [6, chapitre 5]; elle permet à des applications CORBA de se passer des objets *par valeur*, c'est-à-dire de s'échanger une représentation de l'état d'un objet. Ajouter une description d'état aux descriptions d'interfaces des serveurs d'un système d'exploitation permettrait de créer, à l'échelle des processus et serveurs d'un système, une représentation orientée-objet réflexive proche de celle fournie par les langages de programmation tels que CLOS [27] et GOOPS [19]. Pour reprendre l'exemple de la capture d'état de ressources externes telles que des fichiers (section 3.4), une interface de serveur de fichiers intégrant une description d'état d'un objet de type fichier permettrait d'importer l'état des fichiers ouverts lors de la capture d'état, et de l'exporter lors de la restauration (figure 4.4). Les méthodes d'importation et d'exportation d'état pour un type d'objet donné peuvent être générées automatiquement d'après la description d'état, à la manière de ce qui est proposé dans [28].

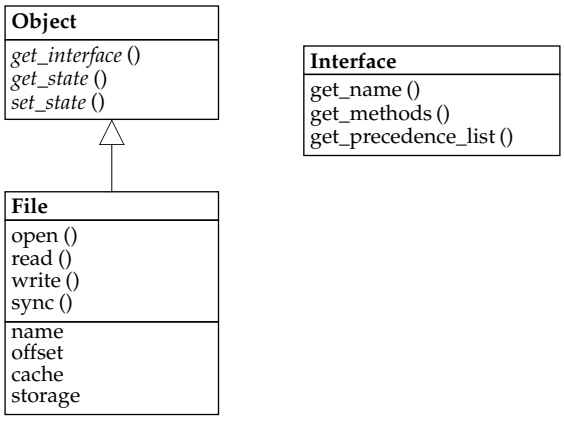


Figure 4.4. Interface d'import/export de l'états d'objets distants (UML).

D'autre part, un système d'exploitation multi-serveur pourrait profiter des avancées faites dans le domaine de l'informatique répartie, et notamment de certains services définis par la norme CORBA. Parmi les services standardisés par la norme, l'*Interface Repository* et l'*Implementation Repository* qui permettent respectivement d'obtenir une description d'une interface et une implémentation de cette interface, sont particulièrement intéressants [6]. Leur utilisation permet à une application de connaître l'ensemble des implémentations d'un service qui sont disponibles et de choisir parmi elles.

En termes d'ouverture et d'extensibilité, les langages réflexifs d'une part et les techniques mettant en œuvre certains concepts empruntés aux langages réflexifs en informatique répartie (i.e. CORBA) nous semblent être deux sources d'inspiration intéressantes pour l'architecture de systèmes d'exploitation. Le respect des principes architecturaux issus de la réflexivité et des protocoles à méta-objets doivent permettre de rendre plus adaptables les systèmes. On facilite ainsi la mise en œuvre de composants de tolérance aux fautes réutilisables et plus faciles à maintenir, leurs préoccupations ayant été clairement identifiées.

Chapitre 5. Conclusion

Le développement de systèmes tolérants aux fautes adaptatifs nécessite, entre autres, la capture en ligne de l'état des composants d'une application. Cette problématique couvre différents aspects : d'une part la capture de l'état au niveau applicatif, et d'autre part la capture de l'état du support d'exécution qui intègre notamment le support d'exécution du langage et le système d'exploitation.

Le travail effectué durant ce stage s'est d'abord focalisé sur le premier aspect, c'est-à-dire sur la capture de l'état au niveau applicatif. L'état d'une entité au niveau applicatif comprend donc l'état de ses structures de données persistantes, mais aussi celles générées par le processus de compilation, notamment la pile d'exécution. Ces caractéristiques sont capturées par la bibliothèque écrite pendant ce stage. L'approche adoptée est basée sur l'analyse des informations descriptives introduites par le compilateur dans l'exécutable d'un programme C ou C++. Nous avons ainsi vu que cette technique permet de lever certains manques du langages en fournissant à l'application des moyens d'introspection. Cette possibilité est un pré-requis à la capture d'état portable, c'est-à-dire indépendante de l'architecture, des données de l'application. Elle a permis la mise en œuvre d'un mécanisme de capture d'état portable performant.

Cependant, la capture en ligne de l'état d'applications ne se limite pas à la capture de l'état disponible au niveau applicatif. L'environnement d'exécution d'une application est constitué de différents éléments susceptibles de contenir une partie de l'état de l'application. En particulier, le système d'exploitation lui-même contient une partie de l'état des programmes qu'il exécute. Pour cette raison, le système d'exploitation doit offrir des moyens d'*introspection* permettant d'exhiber cet état. Ces moyens sont nécessaires à d'autres mécanismes de tolérance aux fautes, ainsi que des moyens permettant l'*adaptation* et la *reconfi gurationdynamique* du système. Pour répondre à ces besoins, l'application des concepts liés à la réflexivité à la conception de l'ensemble du système d'exploitation est une approche prometteuse.

Références

[1] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, Chris J. Scheiman. Extending the Operating System at the User-Level : the Ufo Global File System. dans *Annual Technical Conference on UNIX and Advanced Computing Systems (USENIX'97)*. University of California, Santa Barbara (USA), janvier 1997. URL *http://www.cs.ucsb.edu/~berto/*.

[2] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., Morrison, R.. PS-algol : A Language for Persistent Programming. dans *Proc. 10th Australian National Computer Conference, Melbourne, Australia*, pages 70-79, 1983. URL *http://www.dcs.st-and.ac.uk/rsch/publications/ABC+83b.shtml*.

[3] M. P. Atkinson, M. J. Jordan, L. Daynè and S. Spence. Design Issues for Persistent Java : A Type-Safe, Object-Oriented, Orthogonally Persistent System. dans *Proceedings of the 7th Workshop on Persistent Object Systems (POS'96), Cape May (NJ), USA*, pages 33-47, 1996.

[4] Thomas Bushnell, BSG. Towards a New Strategy of OS Design. *GNU's Bulletin, Cambridge, MA (USA)* (janvier 1994). URL *http://www.gnu.org/software/hurd/hurd-paper.html*.

[5] Shigeru Chiba. A Metaobject Protocol for C++. dans *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1995.*, pages 285-299, octobre 1995. URL *http://www.csg.is.titech.ac.jp/~chiba/openc++.html*. An overview of the compile-time reflection provided by OpenC++.

[6] Common Object Request Broker Architecture : Core Specification. Spécifications (décembre 2002), Object Management Group, Inc.. URL *http://www.omg.org/technology/corba/corba3releaseinfo.htm*.

[7] The CORBA Persistent State Service. Spécifications (septembre 2002), Object Management Group, Inc.. URL *http://www.omg.org/technology/documents/formal/persistent.htm*.

[8] Ludovic Courtès. The Ego Reference Manual, LAAS-CNRS, juin 2003. URL *http://www.laas.fr/~lcourtes/software/ego/*.

[9] Ludovic Courtès. The Pego Reference Manual, LAAS-CNRS, juillet 2003. URL *http://www.laas.fr/~lcourtes/software/pego/*.

[10] Jean-Charles Fabre, Tanguy Pérennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems : the FRIENDS Approach. dans *IEEE Transactions on Computers*, pages 78-95. LAAS-CNRS, 1998. URL *http://www.newcastle.research.ec.org/deva/trs/*.

[11] J. Fassino, J. Stefani, J. Lawall, G. Muller. THINK : A software framework for component-based operating system kernels. dans *Usenix Annual Technical Conference, Monterey (USA)*, 2002. URL *http://think.objectweb.org/*.

[12] Adam J. Ferrari, Stephen J. Chapin, Andrew S. Grimshaw. Process Introspection : A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification. Rapport technique (mars 1997), University of Virginia (USA). URL *http://www.cs.virginia.edu/~ajf2j/introspect/*.

[13] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, Stephen Clawson. Microkernels Meet Recursive Virtual Machines. dans *Proceedings of OSDI '96*. University of Utah, Salt Lake City, UT 84112 (USA), octobre 1996. URL *http://www.cs.utah.edu/flux/fluke/html/*. Présentation de l'architecture à base de µ-noyau *Fluke*.

[14] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2001.

[15] M.C. Gaudel, V.Issarny, C.Jones, H.Kopetz, E.Marsden, N.Moffat, M.Paulitsch, D.Powell, B.Randell, A.Romanovsky, R.J.Stroud, F.Taiani. Dependable System of Systems : Final Version of Conceptual Model. Rapport LAAS n°02441, projet IST-1999-11585 (octobre 2002). URL *http://www.laas.research.ec.org/dsos/*.

[16] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, L. Reuther. The SawMill Multiserver Approach. dans *9th SIGOPS European Workshop, Kolding, Denmark*. University of Karslruhe, IBM T. J. Watson Research Center, septembre 2000. URL *http://l4ka.org/publications/*.

[17] Arthur Goldberg, Ajei Gopal, Kong Li, Rob Strom, David F. Bacon. Transparent Recovery of Mach Applications. Research Paper (1990). A propos de *Optimistically Recoverable Mach*.

[18] Brendan Gowing, Vinny Cahill. Making Meta-Object Protocols Practical for Operating Systems. dans *4th International Workshop on Object Orientation in Operating Systems (IWOOS)*, pages 52-55. Distributed Systems Group, Trinity College, Dublin 2, Irlande, 1995. URL *http://www.cs.tcd.ie/publications/tech-reports/tr-index.95.html*.

[19] Christian Lynbech, Mikael Djurfeldt, Neil Jerram. GOOPS Manual, 1999. URL *http://www.gnu.org/software/guile/docs/goops/*. Manuel de référence de GOOPS, l'extension orienté-objet de Guile, l'interpréteur Scheme du projet GNU : <*http://www.gnu.org/software/guile/*>

[20] Andreas Haeberlen. Using Platform-Specific Optimizations in Stub-Code Generation. Study thesis (juillet 2002), University of Karslruhe, Allemagne. URL *http://www.uni-karlsruhe.de/~Andreas.Haeberlen/*.

[21] Andreas Haeberlen. IDL4 User's Manual. Manuel utilisateur (avril 2003), University of Karslruhe, Allemagne. URL *http://l4ka.org/projects/idl4/*.

[22] Yennun Huang, Chandra Kintala. Software Fault Tolerance in the Application Layer. dans *23rd Intl. Symposium on Fault Tolerant Computing (FTCS-23), Toulouse, France*. AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974., 1995. Etudes de composants applicatifs Unix pour la tolérance aux fautes (*libft*, *watchd*, *REPL*).

[23] Interfaces des serveurs du GNU Hurd, Free Software Foundation. URL *http://savannah.gnu.org/cgi-bin/viewcvs/~checkout~/hurd/hurd/hurd/*.

[24] Sheetal V. Kakkad, Mark S. Johnstone, Paul R. Wilson. Portable Run-Time Type Description for Conventional Compilers. dans *International Symposium on Memory Management, Vancouver, British Columbia, Canada*. University of Texas (USA), octobre 1998. URL *http://www.cs.utexas.edu/users/oops/papers.html#texas*. Description du *Run-Time Type Description* (RTTD) utilisé dans Texas.

[25] Feras Karablieh, Rida A. Bazzi. Heterogeneous Checkpointing for Multithreaded Applications. dans *21st IEEE Symposium on Reliable Distributed Systems, Osaka University, Suita, Japon*, pages 140-149, octobre 2002.

[26] Mangesh Kasbekar, Chandramouli Narayanan, Chita R. Das. Selective Checkpointing and Rollbacks in Multi-Threaded Object-Oriented Environment. dans *IEEE Transactions on Reliability*. Pennsylvania State Universisty (USA), décembre 1999. Implémentation de *libooft*.

[27] Gregor Kikczales, Jim des Rivières, Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press (ISBN 0 262 61074 4). Massachussets Institute of Technology, 1991.

[28] Marc-Olivier Killijian, Juan-Carlos Ruiz-Garcia, Jean-Charles Fabre. Portable Serialization of CORBA Objects : A Reflective Approach. dans *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, November 4-8, 2002, Seattle, Washington*. LAAS-CNRS, novembre 2002. URL *http://www.laas.fr/~mkilliji/publications.html*. Utilisation d'OpenC++ et de CORBA.

[29] Marc-Olivier Killijian, Jean-Charles Fabre, Juan-Carlos Ruiz-Garcia. Using Compile-Time Reflection for Object State Capture. dans *Reflection'99, Saint-Malo (France)*, pages 150-152. LAAS-CNRS, 1999. URL *http://www.laas.fr/%7Emkilliji/publications.html*.

[30] Gökhan Kutlu, J. Eliot B. Moss. Exploiting Reflection to Add Persistence and Query Optimization to a Statically Typed Object-Oriented Language. dans *Eighth International Workshop on Persistent Object Systems : Design, Implementation and Use, Tiburon, California (USA)*. University of Massachussets, Amherst, MA (USA), 1998. URL *http://vis-www.cs.umass.edu/~kutlu/publications/pos98.html*.

[31] Charles R. Landau. The Checkpoint Mechanism in KeyKOS. dans *Second International Workshop on Object Orientation in Operating Systems (IWOOOS'92)*. Key Logic, California (USA), septembre 1992. URL *http://www.cis.upenn.edu/~KeyKOS/*.

[32] Jean-Claude Laprie, Jean Arlat, H. Blanquart, Alain Costes, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, H. Guillermain, Mohamed Kaâniche, Karama Kanoun, C. Mazet, David Powell, C. Rabéjac, Pascale Thévenod. *Guide de la sûreté de fonctionnement*. Cépaduès (ISBN : 2.85428.382.1). Laboratoire d'Ingénierie de la Sûreté de Fonctionnement (LIS : Matra-Marconi Space, LAAS-CNRS, Technicatome), 1995-1996.

[33] Greg Law, Julie McCann. A New Protection Model for Component-Based Operating Systems. dans *IEEE IPCCC*. City University, London (UK), 2000. URL *http://goos.sourceforge.net/*.

[34] Jochen Liedtke. A Persistent System in Real Use – Experiences of the First 13 Years. dans *International Workshop on Object-Orientation in Operating Systems (I-WOOOS'93), Asheville, North Carolina*. German National Research Center for Computer Science, décembre 1993. URL *http://os.inf.tu-dresden.de/L4/doc.html*. Expérience des systèmes d'exploitation persistants Eumel et L3.

[35] Jochen Liedtke. On µ-Kernel Construction. dans *15th ACM Symposium on Operating System Principles (SOSP), Copper Mountain Resort, CO (USA)*. GMD – German National Research Center for Information Technology, décembre 1995. URL *http://l4ka.org/publications/*.

[36] Anders Lindström, Rex di Bona, Alan Dearle, Stephen Norris, John Rosenberg, Francis Vaughan. Persistence in the Grasshopper Kernel. dans *Proceedings of the Eighteenth Australasian Computer Science Conference, ACSC-18, ed. Ramamohanarao Kotagiri, Glenelg, South Australia*, pages 329-338. University of Sydney, University of Adelaide, février 1995. URL *http://os.dcs.st-and.ac.uk/GH/*.

[37] Pattie Maes. Issues in Computational Reflection . dans P. Maes, D. Nardi, *Meta-level Architectures and Reflection* , pages 21-35. ISBN : 0 444 70343 8. North-Holland, 1986-1987.

[38] Pattie Maes. Concepts and Experiments in Computational Reflection . dans *2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 147-155, 1987.

[39] Julia Menapace, Jim Kingdon, David MacKenzie. The "Stabs" Debug Format. Manuel de référence (1992). URL *http://sources.redhat.com/gdb/download/onlinedocs/stabs.html*.

[40] R. Morrison and R. C. H. Connor and G. N. C. Kirby and D. S. Munro and M. P. Atkinson and Q. I. Cutts and A. L. Brown and A. Dearle. The Napier88 Persistent Programming Language and Environment. dans *Fully Integrated Data Environments*, pages 98-154. Springer, M. P. Atkinson and R. Welland, 1999.

[41] Gilles Muller, Mireille Hue, Nadine Peyrouze. Performance of Consistent Checkpointing in a Modular Operating System : Results of the FTM Experiment. dans *First Dependable Computing Conference, Berlin, Germany*. IRISA/INRIA, Bull Research, octobre 1994.

[42] Andreas Paepcke. PCLOS : A Flexible Implementation of CLOS Persistence. dans *European Conference on Object-Oriented Programming (ECOOP'88), Oslo, Norway*, pages 374-389. Springer-Verlag. Hewlett-Packard Laboratories, août 1988. URL *http://www.ifs.uni-linz.ac.at/~ecoop/cd/tocs/t0322.htm*.

[43] James S. Plank, Micah Beck, Gerry Kingsley. Libckpt : Transparent Checkpointing Under Unix. dans *Usenix Conference Proceedings*. University of Tennessee, Knoxville (USA), janvier 1995.

[44] Balkrishna Ramkumar, Volker Strumpen. Portable Checkpointing for Heterogeneous Architectures. dans *27th International Symposium on Fault-Tolerant Computing (FTCS'97) - Digest of Papers, Seattle, WA (USA)*, pages 58-67, juin 1997. URL *http://theory.lcs.mit.edu/~porch/*. À propos de la conception et mise-en-œuvre de PORCH (*Portable Checkpoint Compiler*)

[45] Juan Carlos Ruiz, Marc-Olivier Killijian, Jean-Charles Fabre, Pascale Thévenod-Fosse. Reflective fault-tolerant systems : from experience to challenges. Rapport technique (février 2002), LAAS-CNRS.

[46] Frédéric Salles. *Sûreté de fonctionnement des logiciels exécutifs à base de micro-noyau : analyse des modes de défaillance et confinementdes erreurs*. Thèse de Doctorat, LAAS-CNRS, avril 1999.

[47] Jonathan S. Shapiro, Jonathan Adams. Design Evolution of the EROS Single-Level Store. dans *2002 USENIX Annual Technical Conference*. Johns Hopkins University, University of Pennsylvania, mai 2002. URL *http://www.eros-os.org/devel/00Devel.html*.

[48] Vivek Singhal, Sheetal Kakkad, Paul Wilson. Texas : An Efficient,Portable Persistent Store. dans *Persistent Object Systems : Proceedings of the Fifth International Workshop on Persistent Object Systems, San Miniato, Italy*, pages 11-33. University of Texas (USA), septembre 1992. URL *http://www.cs.utexas.edu/users/oops/papers.html#texas*.

[49] Espen Skoglund, Christian Ceelen, Jochen Liedtke. Transparent Orthogonal Checkpointing Through User-Level Pagers. dans *9th International Workshop on Persistent Object Systems (POS9), Lillehammer, Norway*. System Architecture Group, University of Karlsruhe, septembre 2000. URL *http://www.l4ka.org/publications/*.

[50] Peter Smith, Norman C. Hutchinson. Heterogeneous Process Migration : The Tui System. Rapport technique (février 1996), University of British Columbia, Vancouver (Canada). URL *http://www.cs.ubc.ca/spider/psmith/tui.html*. Utilise les *stabs* (symboles de débogage produits par le compilateur) pour en déduire les informations sur les variables et leur type.

[51] System V Application Binary Interface, The Santa Cruz Operation, Inc., 1990-1996. URL *http://www.caldera.com/developers/devspecs/*.

[52] François Taïani, Jean-Charles Fabre, Marc-Olivier Killijian. Principles of Multi-Level Reflection for Fault-Tolerant Architectures. dans *Pacific Rim International Symposium on Dependable Computing 2002 (PRDC'2002), Tsukuba (Japon)*. LAAS-CNRS, 31077 Toulouse, France, décembre 2002. URL *http://www.laas.fr/~ftaiani/*.

[53] François Taïani, Jean-Charles Fabre, Marc-Olivier Killijian. Towards Implementing Multi-Layer Reflection for Fault-Tolerance. dans *Proceedings of the International Conference on Dependable Systems and Networks (DSN), San Francisco, USA*. LAAS-CNRS, juin 2003. URL *http://www.laas.fr/~ftaiani/*.

[54] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, Kozo Itano. OpenJava : A Class-Based Macro System for Java. dans *Lecture Notes in Computer Science 1826, Reflection and Software Engineering*, pages 117-133. Springer-Verlag, 2000. URL *http://www.csg.is.titech.ac.jp/openjava/*.

[55] Patrick Tullmann, Jay Lepreau, Bryan Ford, Mike Hibler. User-level Checkpointing Through Exportable Kernel State. Research Paper (1996), University of Utah (Etats-Unis). URL *http://www.cs.utah.edu/projects/flux/* . A propos de l'exportation et importation d'objets du noyau par les applications au-dessus du µ-noyau Fluke.

[56] Yasuhiko Yokote. The Apertos Reflective Operating System : The Concept and Its Implementation. dans *Proceedings of the 1992 International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*. Sony Computer Science Laboratory, Inc., juin 1992. URL *http://www.csl.sony.co.jp/project/Apertos/*.

[57] Yasuhiko Yokote. Kernel Structuring for Object-Oriented Operating Systems : The Apertos Approach. dans *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS)*. Sony Computer Science Laboratory, Inc., novembre 1993. URL *http://www.csl.sony.co.jp/project/Apertos/*.

[58] Chris Zimmermann, Vinny Cahill. It's Your Choice - On the Design and Implementation of a Flexible Metalevel Architecture. dans *3rd International Conference on ConfigurableDistributed Systems (ICCDS'96)*. Distributed Systems Group, Trinity College, Dublin 2, Irlande, mai 1996. URL *http://www.cs.tcd.ie/publications/tech-reports/tr-index.96.html*.

Glossaire

IPC (*INTER-PROCESS COMMUNICATION*) *page 42*

Moyen de communication inter-processus offert aux applications par le (micro-)noyau d'un système d'exploitation.

MÉTA-CLASSE *page 7*

Un méta-objet particulier qui représente la notion de classe dans un système orienté-objet.

MÉTA-ESPACE *page 7*

La notion de méta-espace d'un objet décrit l'ensemble des méta-objets qui représentent les mécanismes et abstractions du système réflexif utilisés par cet objet.

MÉTA-NIVEAU *page 6*

Partie d'un système réflexif qui observe et agit sur le niveau de base (i.e. la partie fonctionnelle) du système.

MÉTA-OBJET *page 7*

Dans un système réflexif orienté-objet, un méta-objet est un objet qui représente et contrôle un aspect des fonctionnalités du système.

PROTOCOLE À MÉTA-OBJECTS (MOP) *page 7*

Dans un système réflexif orienté-objet, ensemble des règles décrivant les interactions entre les objets et les méta-objets du système.

RPC (*REMOTE PROCEDURE CALL*) *page 42*

Mécanisme basé sur un moyen de communication inter-processus qui permet l'invocation de procédures distantes (i.e. dans un autre processus) de manière transparente au développeur (i.e. comme un appel de fonction local).

RÉFLEXIVITÉ *page 6*

Capacité d'un système à *raisonner* à propos de lui-même et à *agir* sur lui-même.

STABS *page 17*

Les « *stabs* » sont un format d'informations de débogage très répandu et notamment utilisé par GCC sur la plupart des plate-formes.

Résumé

Ce rapport présente le travail effectué lors d'un stage de DEA au sein de l'équipe Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF) du LAAS-CNRS. Le travail présenté s'axe principalement autour de deux thèmes qui sont la capture d'état portable et les architectures ouvertes de systèmes d'exploitation. Après une étude de différentes techniques de capture d'état, une bibliothèque de capture d'état pour applications écrites en langage C ou C++, appelée *Pego*, est présentée. Cette bibliothèque permet, en utilisant les informations de débogage produites par le compilateur, de capturer l'état d'applications de manière indépendante de l'architecture, et d'une façon peu intrusive pour l'application. D'autre part, différentes architectures de système d'exploitation sont étudiées, ainsi que leur impact sur l'intégration de mécanismes de tolérance aux fautes dans de tels systèmes. Cette étude met l'accent sur le principe de *réflexivité* qui présente de nombreux avantages dans l'architecture de systèmes tolérants aux fautes.

Mots-clef : tolérance aux fautes, capture d'état portable, *stabs*, séparation des préoccupations, réflexivité, protocole à méta-objets, système d'exploitation, micro-noyau.

Abstract

This document presents some of the work done in the Dependable Computing and Fault Tolerance group (TSF) of LAAS-CNRS during a six-month internship. The two main topics discussed in this paper are portable state capture and checkpointing techniques on one hand, and operating system support for fault-tolerance mechanisms on the other. Checkpointing techniques are presented and evaluated. The design and implementation of a portable state capture library, *Pego*, for C and C++ programs is discussed. This library allows to save application state in an architecture-independent way with little tuning of the application code. It relies on the type and data descriptions available in the compiler-generated debugging information (so-called *stabs*). The next part of the document gives an overview of the kind of OS support required for implementing fault-tolerance mechanisms. It includes a survey of existing OS architectures and an evaluation of their suitability for the mechanisms we are interested in. A *reflective* approach to OS architecture is advocated since it has proved to provide the necessary extensibility and *open-endedness*.

Keywords : fault-tolerance, portable state capture and checkpointing, *stabs*, separation of concerns, reflection, meta-object protocols, operating system, micro-kernel.